Chapter No. 5
"Design with AOP"
In this package, you will find:

A Biography of the author of the book
A preview chapter from the book, Chapter NO.5 "Design with AOP"
A synopsis of the book’s content
Information on where to buy this book

About the Author

Massimiliano Dessì is an experienced Java developer who started developing JEE applications in 2000. In 2004 he discovered the Spring Framework 1.0, and since then he has been one of its most enthusiastic users.

Massimiliano is specialized in design and development of enterprise Web-based applications, such as portals, content management systems and banking applications. JEE technology and applied agile methodologies like eXtreme Programming are his core skills. He currently works as a Software Architect and Engineer for Sourcesense (www.sourcesense.com), one of the leading European Open Source System Integrators. He have a strong background as a community supporter and opensource software contributor. He's also an active technical writer, author of various articles, publications, and reviews availables on http://www.jugsardegna.org/vgwiki/jsp/Wiki?MassimilianoDessi and on http://wiki.java.net/bin/view/People/MassimilianoDessi.

Massimiliano also speaks regualrly at Users Groups conferences (including Java Users Groups, Spring Framework User Group, Javaday, and Linux Users Groups).

He is one of the founders of Java User Group Sardinia (http://www.jugsardegna.org), as well as the founder of "Spring Framework Italian User Group", "Jetspeed Italian user Group" and "Groovy Italian User Group".

He maintains a personal weblog at: http://jroller.com/page/desmax.

Massimiliano lives in Cagliari, Sardinia with his family.
Spring 2.5
Aspect-Oriented Programming

In software engineering, mostly low-level languages were used for many years, which were closer to the computer machine code than to human language. In the 70s, Brian Kernighan and Dennis Ritchie created the language C. It was quite similar to human language, making it easier and faster to write code, while keeping a high level of abstraction. This allowed the realization of concepts and ideas, which was not possible for the previous languages as they were forced to focus on the processor's language. Later, Smalltalk and C++ permitted the shaping of concepts and ideas through objects, providing a new way to structure applications and write programs. With the object-oriented languages, any system could be created with increasing complexity in a more manageable way, thanks to the modeling of entities in the form of types and the collaboration between them. In some cases, object-oriented programming introduces or causes inefficiencies, and aspect-oriented programming helps in filling these gaps. The aim of Aspect-Oriented Programming (AOP) is not to replace Object-Oriented Programming (OOP), but to complement it, allowing you to create clearer and better structured programs. Gregor Kiczales, one of the founders of AOP, said (an extract from http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf) "We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in tangled code that is excessively difficult to develop and maintain." Neither aspect-oriented programming nor object-oriented programming can make up for a bad design: The first assumption is that a software system is well-designed. There is no solution for a badly designed system, and also none for a badly implemented system. There is only one good strategy: to change it. The difference between a good and a bad design is the capacity to evolve and adapt to new requirements without being twisted. Object-oriented programming, supported by aspect-oriented programming, helps designers and developers in this direction.

What This Book Covers
Chapter 1 introduces the ideas that led to Aspect-Oriented Programming. An overview of main concepts of AOP is used to describe components and features provided by Spring AOP, while a set of concise yet clear examples lets the reader discover what can actually be done with AOP.
Chapter 2 describes in detail the fundamentals of AOP in Spring, presenting interfaces and classes introduced in early 1.x versions of the framework. This chapter shows how to use AOP programmatically, to let the reader discover the basis of Spring AOP and the components that implement Aspect-Oriented Programming in Spring.

Chapter 3 explains how the weaving of AOP components is done using the proxy pattern and JDK or CGLIB implementations. It describes the purpose of proxies and how to use them effectively. Some practical examples show how to use the proxies programmatically, with annotations and with XML; they explain the ProxyFactoryBean and how to make the programmer's work easier with AutoProxy. The chapter describes also some smart techniques on target sources.

Chapter 4 explains how Spring AOP is supported by AspectJ. Configuration activity is made simpler, more flexible and more powerful, thanks to annotations and the syntax of AspectJ on pointcuts (without which those costructors would not be available). All examples show how to use AspectJ with both annotations and XML. The chapter contains practical recipes for specific cases, such as the injection of dependencies on domain objects, the management of aspects' priority, the use of different life cycles for Aspects and how to use Load Time Weaving. The chapter ends with some strategies on how to choose different AOP approaches to fulfıl specific requirements.

Chapter 5 describes the design alternatives that can be implemented using AOP. These alternatives are solutions for common requirements: concurrency, caching, and security. Using AOP, they can be achieved in a very elegant and easy way, being at the same time totally transparent for the system where they are applied.

Chapter 6 introduces Domain-Driven Development as a alternative way to design applications. The prototype example presented in this chapter is a typical Three-Layer application, where DDD is used for design and AOP is used to inject the dependencies on domain objects. iBatis is used for persistence to the database.

Chapter 7 completes the prototype application started in Chapter 6, showing the application layer and the user interface. The latter is implemented with Spring MVC using annotations. Integration and unit tests are used to verify the correctness of the classes; DBUnit is used to test persistence classes, while some Mock classes are used to test the UI. The chapter contains the configurations for the prototype infrastructure, including autentication and authorization with Spring Security and the JUnit 4.5 test suite.

Chapter 8 describes the development tools needed to include Spring AOP and AspectJ in the Eclipse IDE. The reader can find here detailed instructions on how to configure Eclipse with the plug-Ins for Spring and for the AspectJ Development Tool, and how to install the PostgreSQL database and the Apache Tomcat servlet engine. All installation procedures are described for the three main operating systems: Ubuntu Linux, Apple Mac OS X, and Microsoft Windows XP.

For More Information:
In this chapter, we're going to examine some design decisions that are important for building better applications. In these design decisions, the AOP plays a significant role because it provides smart solutions to common crosscutting problems.

We will look at the following AOP design solutions:

- Concurrency with AOP
- Transparent caching with AOP
- Security with AOP

Designing and implementing an enterprise Java application means not only dealing with the application core business and architecture, but also with some typical enterprise requirements.

We have to define how the application manages concurrency so that the application is robust and does not suffer too badly from an increase in the number of requests.

We have to define the caching strategies for the application because we don't want CPU- or data-intensive operations to be executed over and over.

We have to define roles and profiles, applying security policies and restricting access to application parts, because different kinds of users will probably have different rights and permissions. All these issues require writing additional code that clutters our application business code and reduces its modularity and maintainability.

But we have a choice. We can design our enterprise Java application keeping AOP in mind. This will help us to concentrate on our actual business code, taking away all the infrastructure issues that can otherwise be expressed as crosscutting concerns.

This chapter will introduce such issues, and will show how to design and implement solutions to them with Spring 2.5 AOP support.

For More Information:
Concurrency with AOP

For many developers, concurrency remains a mystery.

Concurrency is the system's ability to act with several requests simultaneously, such a way that threads don't corrupt the state of objects when they gain access at the same time.

A number of good books have been written on this subject, such as Concurrent Programming in Java and Java Concurrency in Practice. They deserve much attention, since concurrency is an aspect that's hard to understand, and not immediately visible to developers. Problems in the area of concurrency are hard to reproduce. However, it's important to keep concurrency in mind to assure that the application is robust regardless of the number of users it will serve.

If we don't take into account concurrency and document when and how the problems of concurrency are considered, we will build an application taking some risks by supposing that the CPU will never simultaneously schedule processes on parts of our application that are not thread-safe.

To ensure the building of robust and scalable systems, we use proper patterns. There are JDK packages just for concurrency. They are in the java.util.concurrent package, a result of JSR-166.

One of these patterns is the read-write lock pattern, which consists of is the interface java.util.concurrent.locks.ReadWriteLock and some implementations, one of which is ReentrantReadWriteLock.

The goal of ReadWriteLock is to allow the reading of an object from a virtually endless number of threads, while only one thread at a time can modify it. In this way, the state of the object can never be corrupted because threads reading the object's state will always read up-to-date data, and the thread modifying the state of the object in question will be able to act without the possibility of the object's state being corrupted. Another necessary feature is that the result of a thread's action can be visible to the other threads. The behavior is the same as we could have achieved using synchronized, but when using a read-write lock we are explicitly synchronizing the actions, whereas with synchronized synchronization is implicit.

Now let's see an example of ReadWriteLock on the BankAccountThreadSafe object.

Before the read operation that needs to be safe, we set the read lock. After the read operation, we release the read lock.
Before the write operation that needs to be safe, we set the write lock. After a state modification, we release the write lock.

```java
package org.springaop.chapter.five.concurrent;
import java.util.Date;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public final class BankAccountThreadSafe {

    public BankAccountThreadSafe(Integer id) {
        this.id = id;
        balance = new Float(0);
        startDate = new Date();
    }

    public BankAccountThreadSafe(Integer id, Float balance) {
        this.id = id;
        this.balance = balance;
        startDate = new Date();
    }

    public BankAccountThreadSafe(Integer id, Float balance, Date start) {
        this.id = id;
        this.balance = balance;
        this.startDate = start;
    }

    public boolean debitOperation(Float debit) {
        wLock.lock();
        try {
            float balance = getBalance();
            if (balance < debit) {
                return false;
            } else {
                setBalance(balance - debit);
                return true;
            }
        } finally {
```

[163]
    wLock.unlock();
}
}

public void creditOperation(Float credit) {
    wLock.lock();
    try {
        setBalance(getBalance() + credit);
    } finally {
        wLock.unlock();
    }
}

private void setBalance(Float balance) {
    wLock.lock();
    try {
        balance = balance;
    } finally {
        wLock.unlock();
    }
}

public Float getBalance() {
    rLock.lock();
    try {
        return balance;
    } finally {
        rLock.unlock();
    }
}

public Integer getId() {
    return id;
}

public Date getStartDate() {
    return (Date) startDate.clone();
}

...
private Float balance;
private final Integer id;
private final Date startDate;
private final ReadWriteLock lock = new ReentrantReadWriteLock();
private final Lock rLock = lock.readLock();
private final Lock wLock = lock.writeLock();
}

BankAccountThreadSafe is a class that doesn't allow a bank account to be
overdrawn (that is, have a negative balance), and it's an example of a thread-safe
class. The final fields are set in the constructors, hence implicitly thread-safe.
The balance field, on the other hand, is managed in a thread-safe way by the
setBalance, getBalance, creditOperation, and debitOperation methods.

In other words, this class is correctly programmed, concurrency-wise. The problem is
that wherever we would like to have those characteristics, we have to write the same
code (especially the finally block containing the lock's release).

We can solve that by writing an aspect that carries out that task for us.

- A state modification is execution(void
  com.mycompany.BankAccount.set*(*))
- A safe read is execution(* com.mycompany.BankAccount.getBalance())
Design with AOP

```java
public void safeRead(){
    @Pointcut("execution(* org.springframework.chapter.five.concurrent.BankAccount.*(..))")
    public void stateModification(){
    }
    @Pointcut("execution(* org.springframework.chapter.five.concurrent.BankAccount.getId())")
    public void getId(){
    }
    @Pointcut("execution(* org.springframework.chapter.five.concurrent.BankAccount.getStartDate())")
    public void getStartDate(){
        //advises/>
    @Before("safeRead()")
    public void beforeSafeRead() {
        rLock.lock();
    }
    @After("safeRead()")
    public void afterSafeRead() {
        rLock.unlock();
    }
    @Before("stateModification()")
    public void beforeSafeWrite() {
        wLock.lock();
    }
    @After("stateModification()")
    public void afterSafeWrite() {
        wLock.unlock();
    }
    private final ReadWriteLock lock = new ReentrantReadWriteLock();
    private final Lock rLock = lock.readLock();
    private final Lock wLock = lock.writeLock();
}
```
The BankAccountAspect class applies the crosscutting functionality. In this case, the functionality is calling the lock and unlock methods on the ReadLock and the WriteLock. The before methods apply the locks with the @Before annotation, while the after methods release the locks as if they were in the final block, with the @After annotation that is always executed (an after-finally advice).

In this way the BankAccount class can become much easier, clearer, and briefer. It doesn't need any indication that it can be executed in a thread-safe manner.

```java
package org.springaop.chapter.five.concurrent;
import java.util.Date;
public class BankAccount {
    public BankAccount(Integer id) {
        this.id = id;
        this.balance = new Float(0);
        this.startDate = new Date();
    }
    public BankAccount(Integer id, Float balance) {
        this.id = id;
        this.balance = balance;
        this.startDate = new Date();
    }
    public BankAccount(Integer id, Float balance, Date start) {
        this.id = id;
        this.balance = balance;
        this.startDate = start;
    }
    public boolean debitOperation(Float debit) {
        float balance = getBalance();
        if (balance < debit) {
            return false;
        } else {
            setBalance(balance - debit);
            return true;
        }
    }
    public void creditOperation(Float credit) {
        setBalance(getBalance() + credit);
    }
```
private void setBalance(Float balance) {
    this.balance = balance;
}

public Float getBalance() {
    return balance;
}

public Integer getId() {
    return id;
}

public Date getStartDate() {
    return (Date) startDate.clone();
}

private Float balance;
private final Integer id;
private final Date startDate;

Another good design choice, together with the use of ReadWriteLock when necessary, is using objects that once built are immutable, and therefore, not corruptible and can be easily shared between threads.

**Transparent caching with AOP**

Often, the objects that compose applications perform the same operations with the same arguments and obtain the same results. Sometimes, these operations are costly in terms of CPU usage, or may be there is a lot of I/O going on while executing those operations.

To get better results in terms of speed and resources used, it's suggested to use a cache. We can store in it the results corresponding to the methods' invocations as key-value pairs: method and arguments as key and return object as value.

Once you decide to use a cache you're just halfway. In fact, you must decide which part of the application is going to use the cache. Let's think about a web application backed by a database. Such a web application usually involves **Data Access Objects** (DAOs), which access the relational database. Such objects are usually a bottleneck in the application as there is a lot of I/O going on. In other words, a cache can be used there.
The cache can also be used by the business layer that has already aggregated and elaborated data retrieved from repositories, or it can be used by the presentation layer putting formatted presentation templates in the cache, or even by the authentication system that keeps roles according to an authenticated username.

There are almost no limits as to how you can optimize an application and make it faster. The only price you pay is having RAM to dedicate the objects that are to be kept in memory, besides paying attention to the rules on how to manage the life of the objects in cache.

After these preliminary remarks, using a cache could seem common and obvious. A cache essentially acts as a hash into which key-value pairs are put. The keys are used to retrieve objects from the cache. Caching usually has configuration parameters that allow you to change its behavior.

Now let's have a look at an example with ehcache (http://ehcache.sourceforge.net). First of all let's configure it with the name methodCache so that we have at the most 1000 objects. The objects are inactive for a maximum of five minutes, with a maximum life of 10 minutes. If the objects count is over 1000, ehcache saves them on the filesystem, in java.io.tmpdir.

```xml
<ehcache>
  ...
  <diskStore path="java.io.tmpdir"/>
  ...
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    overflowToDisk="true"
    diskPersistent="false"
    diskExpiryThreadIntervalSeconds="120"
  />
  ...
  <cache name="methodCache"
    maxElementsInMemory="1000"
    eternal="false"
    overflowToDisk="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
  />
</ehcache>
```
Now let's create a CacheAspect. Let's define the cacheObject to which the ProceedingJoinPoint is passed. Let's recover an unambiguous key from the ProceedingJoinPoint with the method getCacheKey. We will use this key to put the objects into the cache and to recover them.

Once we have obtained the key, we ask to cache the Element with the instruction cache.get(cacheKey). The Element has to be evaluated because it may be null if the cache didn't find an Element with the passed cacheKey.

If the Element is null, advice invokes the method proceed(), and puts in the cache the Element with the key corresponding to the invocation. Otherwise, if the Element recovered from the cache is not null, the method isn't invoked on the target class, and the value taken from the cache is returned to the caller.

```java
package org.springaop.chapter.five.cache;
import it.springaop.utils.Constants;
import net.sf.ehcache.Cache;
import net.sf.ehcache.Element;
import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;

public class CacheAspect {
    public Object cacheObject(ProceedingJoinPoint pjp) throws Throwable {
        Object result;
        String cacheKey = getCacheKey(pjp);
        Element element = (Element) cache.get(cacheKey);
        logger.info(new StringBuilder("CacheAspect invoke:").append("get:").
                  append(cacheKey).append(" value:").append(element).toString());
        if (element == null) {
            result = pjp.proceed();
            element = new Element(cacheKey, result);
            cache.put(element);
            logger.info(new StringBuilder("\n put:").append(cacheKey).append(" value:").append(result).toString());
        }
    }
}
```
return element.getValue();
}

public void flush() {
    cache.flush();
}

private String getCacheKey(ProceedingJoinPoint pjp) {
    String targetName = pjp.getTarget().getClass().getSimpleName();
    String methodName = pjp.getSignature().getName();
    Object[] arguments = pjp.getArgs();
    StringBuilder sb = new StringBuilder();
    sb.append(targetName).append(".").append(methodName);
    if ((arguments != null) && (arguments.length != 0)) {
        for (int i = 0; i < arguments.length; i++) {
            sb.append(".").append(arguments[i]);
        }
    }
    return sb.toString();
}

public void setCache(Cache cache) {
    this.cache = cache;
}

private Cache cache;
private Logger logger = Logger.getLogger(Constants.LOG_NAME);

Here is applicationContext.xml:

<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-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
    <bean id="rockerCacheAspect" class="org.springaop.chapter.five.
    cache.CacheAspect">
        <property name="cache">
            <bean id="bandCache" parent="cache">
                <property name="cacheName" value="methodCache" />
            </bean>
        </property>
    </bean>
</beans>

For More Information:
The idea about the caching aspect is to avoid repetition in our code base and have a consistent strategy for identifying objects (for example using the hash code of an object) so as to prevent objects from ending up in the cache twice.

Employing an around advice, we can use the cache to make the method invocations return the cached result of a previous invocation of the same method in a totally transparent way. In fact, to the methods of the classes defined in the interception rules in pointcuts will be given back the return values drawn from the cache or, if these are not present, they will be invoked and inserted in the cache. In this way, the classes and methods don't have any knowledge of obtaining values retrieved from the cache.

Let's define the pointcut that intercepts the methods of the class DummyClass.

```xml
<brans 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-2.5.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">
... </beans>

<!-- Pointcuts -->
<aop:pointcut id="readOperation" expression="execution(* org.springaop.chapter.five.cache.DummyClass.get*(..))"/>
```
<aop:pointcut id="exitOperation" expression="execution(void org.springaop.chapter.five.cache.DummyClass.exit())" />

<!-- Aspects -->
<aop:aspect id="dummyCacheAspect" ref="rockerCacheAspect">
  <aop:around pointcut-ref="readOperation" method="cacheObject" />
  <aop:after pointcut-ref="exitOperation" method="flush" />
</aop:aspect>
</aop:config>

Class DummyClass used to check the cache's working:

package org.springaop.chapter.five.cache;

public class DummyClass {
  public String getFooFighters()
  {
    return "My hero";
  }

  public String getHives(String year)
  {
    if(year.equals("2004"))
    {
      return "Walk idiot walk !";}
    else{
      return "Abra Cadaver";
    }
  }

  public String getDandyWarhols()
  {
    return "Ride";
  }

  public void exit()
  {
    System.out.println("The end.");
  }
}

Here is ApplicationContext.xml complete:

<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/aop
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"/>
Design with AOP

http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd

<bean id="dummy" class="org.springframework.chapter.five.cache.DummyClass"/>

<aop:config>
 <!-- Pointcuts -->
 <aop:pointcut id="readOperation"
 expression="execution(* org.springframework.chapter.five.cache.DummyClass.get*(..))" />
 <aop:pointcut id="exitOperation"
 expression="execution(void org.springframework.chapter.five.cache.DummyClass.exit())" />

 <!-- Aspects -->
 <aop:aspect id="dummyCacheAspect" ref="rockerCacheAspect">
   <aop:around pointcut-ref="readOperation" method="cacheObject"/>
   <aop:after pointcut-ref="exitOperation" method="flush"/>
 </aop:aspect>
</aop:config>

<bean id="rockerCacheAspect" class="org.springframework.chapter.five.cache.CacheAspect">
 <property name="cache">
   <bean id="bandCache" parent="cache">
     <property name="cacheName" value="methodCache"/>
   </bean>
 </property>
</bean>

<!-- CACHE config -->
<bean id="cache" abstract="true"
 class="org.springframework.cache.ehcache.EhCacheFactoryBean">
 <property name="cacheManager" ref="cacheManager"/>
</bean>

<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
 <property name="configLocation" value="classpath:org/springaop/chapter/five/cache/ehcache.xml"/>
</bean>
</beans>
Chapter 5

ApplicationContext contains the following beans:

1. The dummy bean, used to test the cache’s working.
2. The readOperation and exitOperation pointcuts.
3. The dummyCacheAspect aspect, with around and after advices.
4. The rockerCacheAspect, which is the implementation of the aspect class that contains the logic of recovery from and insertion into the cache.
5. The cache bean, which is an EhCacheFactoryBean.
6. The cacheManager bean, which is an EhCacheManagerFactoryBean.

Here is the test class:

```java
package org.springaop.chapter.five.cache;

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

public class CacheTest {
    public static void main(String[] args) {
        String[] paths = {"org/springaop/chapter/five/cache/applicationContext.xml"};

        ApplicationContext ctx = new ClassPathXmlApplicationContext(paths);
        DummyClass dummy = (DummyClass) ctx.getBean("dummy");
        dummy.getFooFighters();
        dummy.getHives("2004");
        dummy.getDandyWarhols();
        dummy.getFooFighters();
        dummy.getHives("2004");
        dummy.getDandyWarhols();
        dummy.exit();
    }
}
```
Design with AOP

The result will be:

As we can see from the log, the first invocations of methods are always followed by put because the cache didn't contain the results of invocations.

On the second calls, the values are instead recovered by the cache with key className.method.arguments. The number of hits, the date of creation, and the last access are shown as well.

Security with AOP

Security is one of the most important elements of an application. The word "security" covers two concepts:

- Authentication is the verification's process of a principal's identity; a principal is typically a user. A principal in order to be authenticated provides a credential that is the password.
- Authorization, on the other hand, is the process of granting authorities, which are usually roles, to an authenticated user.

Once a user is authenticated and has roles, he or she can work on the application and perform the actions permitted by an access control list, which according to the user's roles allows certain operations.
Before **Spring Security**, the rules of who can do what were usually implemented using custom code and an in-house framework, or using JAAS. Usually, the first type of implementation was a consequence of the second type's difficulty. Unfortunately, though custom-type security fits its purposes, it lacks in its main aim. This is because it's safer to employ a much-used framework that is constantly updated and corrects security problems, rather than having an in-house framework that might be barely tested. Beside these considerations, which should be carefully take into account, defining and applying security rules without AOP means causing code tangling and code scattering.

In fact, AOP applied to security solves most of the common practical problems concerning security. In order to solve them we use Spring Security 2.0.x (formerly Acegi Security System for Spring), configuring it properly to carry out most of the work according to the application's needs. We will see its configuration in Chapter 7. Now let's look just at some parts where AOP intervenes in its configuration.

For now, we will not deal with the authentications and roles attribution. Instead, we will start from the point at which the decision is taken to authorize a user and to provide him or her with roles to access a certain resource. Taking an actual decision whether or not to allow the user (based on its roles) gain access to the secure resource is the responsibility of the access decision manager.

An access decision manager implements the **AccessDecisionManager** interface, and in order to carry out his or her job, the manager needs a group of voters which implement the **AccessDecisionVoter** interface.

The **AccessDecisionManagers** provided by Spring are:

- **AffirmativeBased**: At least one voter votes to grant access
- **ConsensusBased**: A consensus of voters votes to grant access
- **UnanimousBased**: All voters vote to abstain or grant access

If none of them is specified, we employ **AffirmativeBased** with two voters, **RoleVoter** and **AuthenticatedVoter**. A voter can vote to grant, deny, or abstain.

- **RoleVoter**: This bases its vote on role. If the user has the required role by the required resource, it votes **ACCESS_GRANTED**. But if the resource doesn't have a specified role, it votes **ACCESS_ABSTAIN**. If the resource has a role the user doesn't have, then it votes **ACCESS_DENIED**.
- **AuthenticatedVoter**: This votes on the strength of the user's authentication. A user can be authenticated with:
  - **IS_AUTHENTICATED_FULLY**
  - **IS_AUTHENTICATED_REMEMBERED**
Design with AOP

° IS AUTHENTICATED_ANONYMOUSLY
° AuthenticatedVoter votes ACCESS_GRANTED if the authentication level is higher than the level requested by the resource. The highest one is IS_AUTHENTICATED_FULLY.

In the following XML, we see the declaration of the access decision manager:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-2.0.4.xsd
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="accessDecisionManager" class="org.springframework.
security.vote.AffirmativeBased">
    <property name="decisionVoters">
      <list>
        <bean class="org.springframework.security.vote.RoleVoter" />
        <bean class="org.springframework.security.vote.AuthenticatedVoter" />
      </list>
    </property>
  </bean>
</beans>
```

Once we have defined the AccessDecisionManager, we can use AOP to decide the roles that are necessary to call the several beans’ methods.

We can employ three strategies:

1. Securing methods with security interceptors.
2. Securing methods with pointcuts.

In order to employ interceptors, the aspectjrt.jar and aspectjweaver.jar JARs must be in the classpath. For the annotations, there must be the JAR spring-security-core-tiger-2.0.x.jar.

For More Information:
Securing methods with security interceptors

With security interceptors we can define the roles necessary to execute methods on the bean.

Let’s have the interface:

```java
public interface FooService {
    public Integer getBalance(Integer idAccount);
    public void setBalanceAccount(Integer id, Integer balance);
    public boolean suspendAccount(Integer id);
}
```

`FooService` is implemented by `FooServiceImpl`, that is configured as follows:

```xml
<bean id="accessDecisionManager" class="org.springframework.security.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean class="org.springframework.security.vote.RoleVoter"/>
            <bean class="org.springframework.security.vote.AuthenticatedVoter"/>
        </list>
    </property>
</bean>

<bean class="org.springaop.chapter.five.security.FooServiceImpl">
    <security:intercept-methods access-decision-manager-ref="accessDecisionManager">
        <security:protect method="org.springaop.chapter.five.security.FooService.getBalance" access="ROLE_USER"/>
        <security:protect method="org.springaop.chapter.five.security.FooService.setBalanceAccount" access="ROLE_USER"/>
        <security:protect method="org.springaop.chapter.five.security.FooService.suspendAccount" access="ROLE_USER"/>
    </security:intercept-methods>
</bean>
```

For More Information:

We have defined some roles (separated by a comma) that can execute those methods we want to be executed by a user that has a particular role. This choice permits us to define roles on methods directly on beans, but makes the configuration files too long.

### Securing methods with pointcuts

With this strategy, it's possible to define the roles required for the different pointcuts with AspectJ syntax that we define with the tag global-method-security.

We use the same rules on the same methods of the interface **FooService**.

```xml
<global-method-security
  access-decision-manager-ref="accessDecisionManager">
  <protect-pointcut
    expression="execution(* org.springaop.chapter.five.security.
    FooService.getBalance(..))"
    access="ROLE_USER" />
  <protect-pointcut
    expression="execution(* org.springaop.chapter.five.security.
    FooService.set*(..))"
    access="ROLE_ACCOUNTING,ROLE_ADMIN" />
  <protect-pointcut
    expression="execution(* org.springaop.chapter.five.security.
    FooService.suspendAccount(..))"
    access="ROLE_ADMIN" />
</global-method-security>
```

When using pointcuts, we don't have to use interceptors if they can be in conflict with the execution of methods that we have defined in the configuration. This modality of configuration is consistent with AspectJ syntax for the definition of pointcuts, making the modality of configuration of aspects homogeneous.

Compared to interceptors, the configuration is less prolix and dispersive as it concentrates the methods that can be invoked with the different roles at one point.
Securing methods with annotations
We can use annotations to define which roles can be executed by the methods of our classes. We will use the same rules on the same methods of the interface FooService.

```java
package org.springaop.chapter.five.security;
import org.springframework.security.annotation.Secured;
public class FooServiceImplWithAnnotations implements FooService{
    @Secured("ROLE_USER")
    public Integer getBalance(Integer idAccount) {
        Integer result = 0;
        // do something
        return result;
    }
    @Secured( { "ROLE_ACCOUNTING", "ROLE_ADMIN" })
    public void setBalanceAccount(Integer id, Integer balance) {
        // do something
    }
    @Secured("ROLE_ADMIN")
    public boolean suspendAccount(Integer id) {
        boolean result = false;
        // do something
        return result;
    }
}
```

With this strategy, we define the roles within the class, needed for the execution of methods. With this choice we don't have to configure any XML, but we lose the possibility of seeing the roles for the methods present defined in a single place.

Summary
In this chapter we've seen how to implement crosscutting functionalities, such as concurrency control, the employment of a cache, and security management. Without AOP those functionalities would be scattered across the application, with the same code duplicated in different modules. With AOP we can have cleaner code, much more concise and easier to maintain and debug. We’ve seen how it is possible to implement these functionalities without having tangled or scattered code, implementing functionalities with aspects and advices.

Other important functionalities are transactions management, application logging, exception management, layout, and others. They will be illustrated in the next two chapters in a complete and working application.
**Where to buy this book**


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.