Extracted from:

No Fluff, Just Stuff Anthology

The 2006 Edition

This PDF file contains pages extracted from No Fluff, Just Stuff Anthology, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragmaticprogrammer.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2005 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Chapter 5

Spring AOP

by Stuart Halloway

Stuart is a founding partner at Relevance, LLC.

Prior to founding Relevance, Stuart was chief architect at Near-Time and the chief technical officer at DevelopMentor. Stuart is the author of Component Development for the Java Platform, part of the DevelopMentor book series and available for free online.

Stuart writes regularly, including a long-running column for the Java Developer Connection and articles for JavaPro magazine and InformIT. Stuart regularly speaks at industry events including the No Fluff, Just Stuff Java symposiums, Pragmatic Studio: Ajax, and JavaOne. Prior to DevelopMentor, Stuart worked as a lead engineer and project manager, shipping successful projects for Prentice Hall, National Geographic, and Duke University's Humanities Computing Facility. He received his B.S. and M.P.P. from Duke University in 1990 and 1994, respectively.

Aspect-oriented programming (*AOP*) is the unsung other core of Spring. Although dependency injection (*DI*) deserves its place as the core of Spring, the combination of DI and AOP is more powerful than the sum of its parts. This chapter introduces Spring AOP in the belief that AOP should be a standard part of all modern Spring development.

We will introduce the core concepts of *AOP*, show how they work hand-in-hand with *DI*, and present sample code to demonstrate each point along the way. At the conclusion, we will look at changes in Spring 2 that may make *AOP* both easier to use and more powerful.

5.1 From Java to Dependency Injection to AOP

Java is not just an application platform—it is a platform for writing *components*, reusable modules of software. Probably the key feature in Java that encourages reuse is the use of interface. An interface, unlike a class, implies no specific implementation. Instead, an interface is simply a list of required methods. When you program in terms of interfaces, you can easily introduce new components. No recompilation is required, as long as the new components continue to honor existing interfaces. To take a simple example, consider the MessageSource interface:

```
package di;

public interface MessageSource {
  public String getMessage();
}
```

Clients that need a MessageSource can program against this interface and retrieve messages. Clients do not need any knowledge of (or dependency on!) specific MessageSource interfaces. Maybe the messages come from files, from email, or from extraterrestrials. It simply does not matter. This is shown in Figure 5.1, on the next page.

Unfortunately, the use of the interface keyword is not enough to maintain a clean separation between components. For a variety of reasons, components can become *tightly coupled*, with unnecessary dependencies on internal details of other components. This defeats the intention of programming against interfaces! In Java, the following factors typically introduce tight coupling:

- Variables that are typed to concrete types.
- Calls to new.

Click Here to purchase this book now.

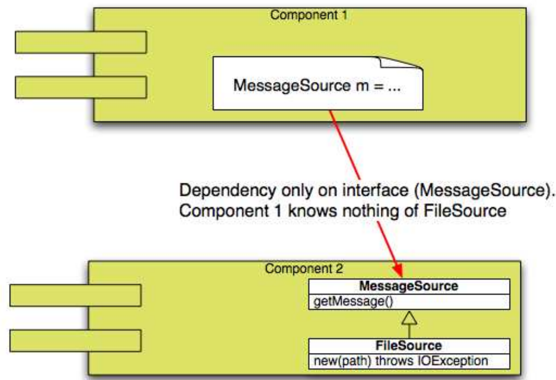

Figure 5.1: Dependency only on MessageSource

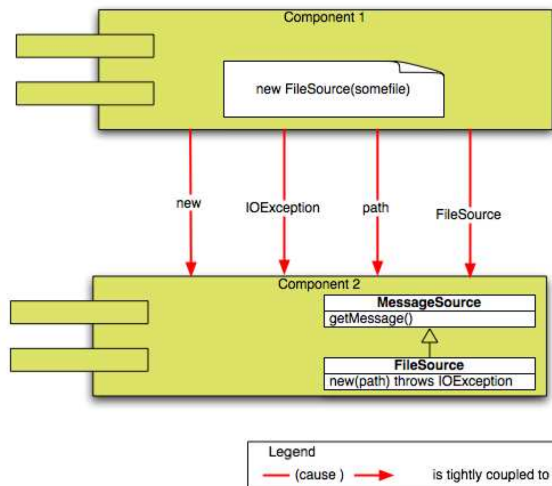

Figure 5.2: Tight coupling

- Checked exceptions.
- Statically typed argument lists for object creation. Constructors often (but not always) cause this problem.

Tight coupling is shown in Figure 5.2 .

Spring solves tight coupling with *dependency injection* (*DI*). With *DI*, application code takes the form of *Plain Old Java Objects* (*POJOs*). These POJOs hold interface-based references to the other objects they depend on, but they take no active role in acquiring these objects. Instead, these dependencies are managed via a configuration file and injected automatically by the container (Spring). For example, consider:

dependency injection

Plain Old Java Objects

```
package di;

public interface MessageRenderer {
  public void render();
}
```

and

```
package di;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class DemoCI {
  public static void main(String[] args) {
    XmlBeanFactory bf =
        new XmlBeanFactory(new FileSystemResource("config/demo_ci.xml"));
    MessageRenderer renderer =
        (MessageRenderer) bf.getBean("renderer");
    renderer.render();
  }
}
```

The application DemoDI and the bean it calls are connected only by the MessageRenderer interface. Thanks to *DI*, there are no subtle forms of tight coupling to trouble us later. This is shown in Figure 5.3, on the next page.

Spring is best known for DI, which eliminates a lot of unnecessary tight coupling. But what about *necessary* coupling? You will often find a large set of dependencies among components, even with DI. Sometimes these dependencies are not clearly captured and localized in a single source file. Such dependencies are called *cross-cutting concerns*. Here are a few examples:

cross-cutting concerns

- Since Java has single inheritance for implementation, it may be difficult to model entities that seem to need multiple inheritance. Secondary categorization hierarchies may have their code spread among many classes.

CLICK HERE to purchase this book now.

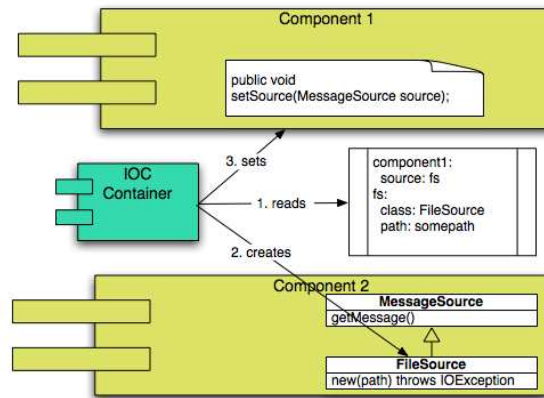

Figure 5.3: Dependency injection

- Generic services, such as persistence, transactions, validation, and auditing, are typically spread across many classes.

For an example of cross-cutting concerns, consider Figure 5.4, on the following page. This shows two totally different application domains (superheroes and software projects). Within those domains, there are several classes whose instances have names. In the superhero domain, these classes all share a common base, but in software projects, they do not. Imagine that you make an application- or organization-wide change to how names are validated. This change might be easy to describe: "Our new web forms tool needs us to allow null values, which were formerly forbidden." However, the implementation change will be spread across several source code files in different application repositories. Name validation is a cross-cutting concern.

Aspect-oriented programming (*AOP*) allows us to localize the handling of cross-cutting concerns. Rather than having name validation spread across many classes, a name validation aspect captures everything in one place, and the run-time *weaves* the necessary code into affected classes, as shown in Figure 5.5, on the next page.

Aspect-oriented programming

AOP and DI are complementary and synergistic. *DI* helps you eliminate unnecessary dependencies, and *AOP* helps you to localize and manage real dependencies. Together, *DI* and *AOP* are the core of Spring.

CLICK HERE to purchase this book now.

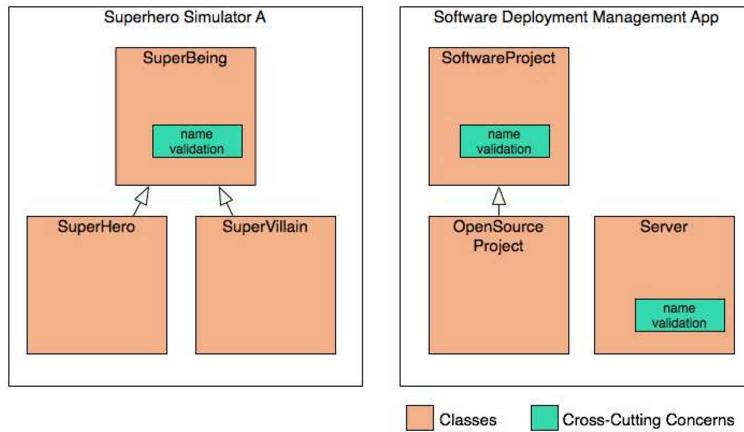

Figure 5.4: Cross-cutting concerns

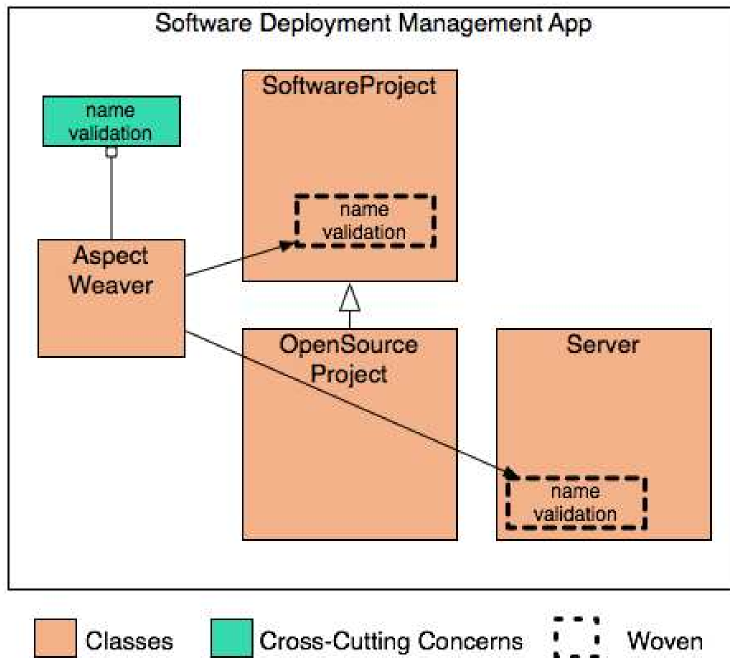

Figure 5.5: Aspect-oriented programming

CLICK HERE to purchase this book now.

5.2 Aspect Terminology

In order to read the code examples that follow, you will need to know the following key AOP terms:

Term	Definition
Advice	Code that is woven in (to a pointcut)
Joinpoint	Point in the execution of an application
Pointcut	Combination of join points used to place advice
Aspect	Advice + pointcut
Introduction	Special case of advice: add entirely new fields/methods

Spring provides several approaches for wiring all this together. The sections *Advice* and *Pointcuts* will show you how to wire aspects in code, and then the section *Aspect Dependency Injection* will show you how to do the same things declaratively, using *DI* for your aspects. The examples shown here are representative but by no means exhaustive. Note that Spring offers more than shown here, including a number of conveniences for common tasks, plus support for many less common tasks.

5.3 Advice

Advice is code that is to be woven into existing classes. In Spring, it is possible to use advice without a pointcut, in which case the advice is woven into all methods on a bean. An example will make this clear. Consider the Superhero class:

```
package aop;

import java.lang.reflect.Method;

public class Superhero {
  private String name;

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

We want to modify Superhero so that a null name will be rejected. The NullBlocker class will do the trick:

```
package aop;

import org.springframework.aop.MethodBeforeAdvice;

import java.lang.reflect.Method;

public class NullBlocker implements MethodBeforeAdvice {
  public void before(Method method, Object[] objects, Object object)
      throws Throwable {
    if ((method.getName().startsWith("set")) && (objects[0] == null)) {
      throw new IllegalArgumentException("null passed to " + method.getName());
    }
  }
}
```

NullBlocker implements the class MethodBeforeAdvice, which means that it should be executed before any methods on a bean. Its sole method, before(), has a generic signature (using Method and Object arguments), because we cannot know in advance what kind of bean the NullBlocker will be applied to.

To weave this all together, we can use Spring's ProxyFactory, as shown in the following test code:

```
package aop;

import org.springframework.aop.framework.ProxyFactory;
import util.TestBase;

public class TestBeforeAdvice extends TestBase {
  public void testBeforeAdvice() {
    Superhero h = new Superhero();
    ProxyFactory pf = new ProxyFactory();
    pf.addAdvice(new NullBlocker());
    pf.setTarget(h);
    Superhero proxy = (Superhero) pf.getProxy();
    proxy.setName("Spiderman");
    assertEquals("Spiderman", proxy.getName());
    assertThrows(IllegalArgumentException.class, proxy, "setName", (Object)null);
  }
}
```

TestBeforeAdvice demonstrates the basic steps for wiring together an aspect in Spring:

1. Create a ProxyFactory.
2. Call setTarget() to associate a POJO.
3. Call getProxy() to return an instance that has the POJO functionality, but with the Aspect woven in.

CLICK HERE to purchase this book now.

In TestBeforeAdvice, the call to the method assertThrows shows that the Superhero instance is now protected by a NullBlocker.

Many people wonder, “Wouldn’t it be easier to just add the validation to Superhero directly?” If all superheroes need this validation and no other types need it, then maybe. But if the concern is cross-cutting, then others need the validation as well, and the aspect approach simplifies the code. TestBeforeAdvice2 shows the NullBlocker being used in a totally different application domain:

```
package aop;

import util.TestBase;
import org.springframework.aop.framework.ProxyFactory;

public class TestBeforeAdvice2 extends TestBase {
  public void testBeforeAdvice() {
    OpenSourceProject osp = new OpenSourceProject();
    ProxyFactory pf = new ProxyFactory();
    pf.setTarget(osp);
    pf.addAdvice(new NullBlocker());
    OpenSourceProject proxy = (OpenSourceProject) pf.getProxy();
    proxy.setName("Spring");
    assertEquals("Spring", proxy.getName());
    assertThrows(IllegalArgumentException.class, proxy, "setName", (Object) null);
  }
}
```

The advice shown in the previous two examples is called *before advice*, because it runs before the method. Before advice has access to method parameters and can prevent the execution of the method by throwing an exception. There are several other types of advice, shown in Figure 5.6, on the following page.

The interaction of the various types of advice with a method call on a Java object are shown in Figure 5.7, on the next page.

Around advice is the most powerful, because it has complete access to a method both before and after execution. This allows potent modification to existing programs. Consider the Recorder class. Recorder does a bit of gymnastics with getters and setters to keep a record of all setters called on an object. With a bean aspected by Recorder, you can get a list of past values for any bean property.

CLICK HERE to purchase this book now.

Type of Advice	**AOP Alliance Interface (Spring Class)**	**Example Usage**
Around	MethodInterceptor	Recording
Before	BeforeAdvice	Validation
Throws	ThrowsAdvice	Remapping exception types
After	AfterReturning	Remapping return types
Introduction	IntroductionInterceptor, DelegatingIntroduction-Interceptor	Mixins

Figure 5.6: The different types of advice

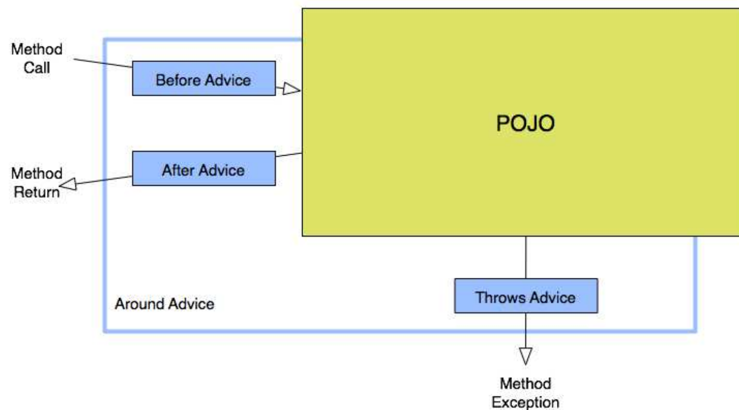

Figure 5.7: Advice

CLICK HERE to purchase this book now.

```
package aop;

import org.aopalliance.intercept.*;
import java.lang.reflect.Method;
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

public class Recorder implements MethodInterceptor {
  private List history = new CopyOnWriteArrayList();
  public Iterator historyIterator() {
    return history.iterator();
  }
  public Method getterForSetter(Class cls, Method m) {
    String name = m.getName();
    if (name.startsWith("set")) {
      try {
        return cls.getMethod(name.replaceFirst("set", "get"));
      } catch (NoSuchMethodException e) { ; }
    }
    return null;
  }
  public Object invoke(MethodInvocation mi)
      throws Throwable {
    Method method = mi.getMethod();
    Method getter = getterForSetter(mi.getThis().getClass(), method);
    Object result = null;
    if (getter != null) {
      Object before = getter.invoke(mi.getThis());
      result = mi.proceed();
      if (before != getter.invoke(mi.getThis())) {
        history.add(new Record(method.getName(), before));
      }
    } else {
      result = mi.proceed();
    }
    return result;
  }
  public static class Record {
    private final String method;
    private final Object value;
    public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;

      final Record record = (Record) o;
      if (!method.equals(record.method)) return false;
      if (value != null ? !value.equals(record.value) : record.value != null)
        return false;

      return true;
    }
```

CLICK HERE to purchase this book now.

```
    public int hashCode() {
      int result;
      result = method.hashCode();
      result = 29 * result + (value != null ? value.hashCode() : 0);
      return result;
    }

    public Record(String method, Object value) {
      this.method = method;
      this.value = value;
    }
  }
}
```

The TestRecorder class demonstrates the Recorder in action.

```
package aop;

import util.TestBase;
import org.springframework.aop.framework.ProxyFactory;
import static aop.Recorder.*;

public class TestRecorder extends TestBase {
  public void testRecorder() {
    ProxyFactory pf = new ProxyFactory();
    Recorder r = new Recorder();
    pf.addAdvice(r);
    pf.setTarget(new Superhero());
    Superhero hero = (Superhero) pf.getProxy();
    hero.setName("Spiderman");
    assertIteratorEquals(r.historyIterator(), new Record("setName", null));
    hero.setName("Batman");
    assertIteratorEquals(r.historyIterator(),
                         new Record("setName", null),
                         new Record("setName", "Spiderman"));
  }
}
```

Although this Recorder is pretty simple, the implications are powerful. Similar code could be used to audit all access, record all state changes into persistence media, implement undo and redo, or perform a variety of other services around managing object state.

Recorder is written as around advice because it needs access to the method before it runs (to cache the old value of a property) and after it runs (to check whether the property changed). The other forms of advice (before, after, and throws) provide subsets of around behavior. It is considered good style to use the weakest advice that can get the job done. Don't use around advice unless none of the simpler types will do.

CLICK HERE to purchase this book now.

5.4 Pointcuts

The advice examples were applied to all methods on a bean. Often, it is necessary to limit the application of advice to some methods and not others. Spring AOP provides several ways to create such pointcuts:

- You can configure static pointcuts based on the names of methods or classes, by simple name matching or by regular expression.
- You can configure dynamic pointcuts that examine the runtime call stack to determine where to apply advice. This is used less often, since examining the call stack is expensive.
- The entire pointcut architecture is built around interfaces (but of course!), so you can define your own pointcuts if Spring's provided ones do not match your needs.

The basic steps to using pointcuts in Spring are as follows:

1. Start with one or more POJOs.
2. Write some advice.
3. Write some pointcuts.
4. Combine pointcuts and advice into an advisor (advisor is another name for Aspect).
5. Use the ProxyFactory to weave the advisor and the POJOs together.

As an example, let's start with a slightly larger POJO, PersonName:

```
package aop;

public class PersonName {
  private String lastName;
  private String firstName;
  private String middleInitial;
  public String getLastName() {
    return lastName;
  }
  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getMiddleInitial() {
    return middleInitial;
  }
```

CLICK HERE to purchase this book now.

```
  public void setMiddleInitial(String middleInitial) {
    this.middleInitial = middleInitial;
  }
}
```

We want to make sure that PersonName's fields are not set to null. The NullBlocker we wrote earlier can be reused here. But this time there is a twist: first names and last names cannot be null, but middle initials can. The TestNameMatchMethodPointcut class demonstrates how:

```
package aop;

import util.TestBase;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.*;
import org.springframework.aop.Advisor;

public class TestNameMatchMethodPointcut extends TestBase {
  public void testNameMatchMethodPointcut() {
    ProxyFactory pf = new ProxyFactory();
    NameMatchMethodPointcut pc = new NameMatchMethodPointcut();
    pc.setMappedNames(new String[]{"setLastName", "setFirstName"});
    Advisor adv = new DefaultPointcutAdvisor(pc, new NullBlocker());
    pf.addAdvisor(adv);
    pf.setTarget(new PersonName());
    PersonName pn = (PersonName) pf.getProxy();
    assertThrows(IllegalArgumentException.class, pn, "setFirstName", (Object) null);
    assertThrows(IllegalArgumentException.class, pn, "setLastName", (Object) null);
    pn.setMiddleInitial(null);
  }
}
```

NameMatchMethodPointcut is part of Spring, and it matches an array of names. There are several methods for setting the names to match; here we call setMappedNames(). The constructor for DefaultPointcutAdvisor then combines our pointcut and advice into an advisor. Calling addAdvisor() configures the ProxyFactory, and then we use getProxy() to retrieve our aspected PersonName.

For more complex situations, you can use JdkRegexpMethodPointcut to match classes and methods by regular expression, as demonstrated by the method TestJdkRegexpMethodPointcut:

```
package aop;

import util.TestBase;
import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.*;
import org.springframework.aop.Advisor;
```

CLICK HERE to purchase this book now.

```
public class TestJdkRegexpMethodPointcut extends TestBase {
  public void testIt() {
    ProxyFactory pf = new ProxyFactory();
    JdkRegexpMethodPointcut pc = new JdkRegexpMethodPointcut();
    pc.setPattern("aop.*set.*Name");
    Advisor adv = new DefaultPointcutAdvisor(pc, new NullBlocker());
    pf.addAdvisor(adv);
    pf.setTarget(new PersonName());
    PersonName pn = (PersonName) pf.getProxy();
    assertThrows(IllegalArgumentException.class,
        pn, "setFirstName", (Object) null);
    assertThrows(IllegalArgumentException.class,
        pn, "setLastName", (Object) null);
    pn.setMiddleInitial(null);
  }
}
```

Note that this code is almost the same as the previous example, except for the line that creates the pointcut. Beware that with regular expression pointcuts, the regular expression must match the fully qualified class name plus the method name.

5.5 Aspect Dependency Injection

As we have seen, creating an aspected bean is a matter of wiring POJOs, advice, and pointcuts together. Most Spring applications will not do this wiring in application code, instead preferring to configure aspects via dependency injection. The TestDeclarativeAop class demonstrates creating a PersonName bean with aspects injected:

```
package aop;

import util.TestBase;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class TestDeclarativeAop extends TestBase {
  public void testIt() {
    ApplicationContext ac =
        new FileSystemXmlApplicationContext("config/test_declarative_aop.xml");
    PersonName pn = (PersonName) ac.getBean("personName");
    assertThrows("IllegalArgument", pn, "setFirstName", (Object) null);
    assertThrows("IllegalArgument", pn, "setLastName", (Object) null);
    assertDoesNotThrow(pn, "setMiddleInitial", (Object) null);
  }
}
```

But hold on a minute—there is nothing in TestDeclarativeAop that says anything about aspects! The fact that the PersonName bean uses (or

CLICK HERE to purchase this book now.

does not use) aspects is an implementation detail. In fact, if you could see aspects in TestDeclarativeAop, then we would have an unnecessary dependency on aspects, which is exactly the kind of thing *DI* helps us avoid. To see that Spring AOP is being used, you would have to look at the bean configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="personName"
        class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <bean class="aop.PersonName"/>
    </property>
    <property name="interceptorNames">
      <list>
        <value>nullNameBlocker</value>
      </list>
    </property>
  </bean>
  <bean id="nullNameBlocker"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="advice">
      <bean class="aop.NullBlocker"/>
    </property>
    <property name="pointcut">
      <bean class="org.springframework.aop.support.JdkRegexpMethodPointcut">
        <property name="pattern">
          <value>aop.*set.*Name</value>
        </property>
      </bean>
    </property>
  </bean>
</beans>
```

Here you can see that personName is not declared as a PersonName at all. Instead, personName is a Spring ProxyFactory. The target property then specifies the actual PersonName POJO, and the interceptorNames property contains a list of advisors. The personName bean has only one advisor, the nullNameBlocker.

If you examine its bean properties, you will see that it contains advice (a NullBlocker) and a pointcut (a JdkRegexpMethodPointcut with an appropriate pattern). With this configuration file, the TestDeclarativeAop application is the *DI* equivalent of the TestJdkRegexpMethodPointcut app shown previously.

Note that the target and pointcut properties use nested beans. Another

Click Here to purchase this book now.

alternative would have been to make these top-level named beans in their own right, and then refer to them by reference. The choice of nested beans is an important decision, because it indicates our intent that these beans are not to be used alone but only in the context of the top-level beans personName and nullNameBlocker. Keeping the list of top-level beans small makes it easy for clients to find the beans they need, without wading through all the subsidiary support beans.

There is an even more important reason for making target (the raw PersonName POJO) a nested bean. If the raw PersonName could be accessed directly, its first and last names could be set to null. On the other hand, the pointcut might reasonably be used elsewhere, so the choice between top-level and nested bean for pointcut is not nearly so clear-cut. As a rule of thumb, beans that require their aspects in order to function correctly should be nested inside their ProxyFactory.

5.6 Spring 2

Note: The following is based on the most recent build available at the time of this writing: the March 6, 2006, Spring 2 M3 daily build. Some details may change.

Spring 2 makes several important enhancements in the area of aspects:

- Spring 2 uses *XML* Schema (*xsd*) instead of Document Type Declarations (*DTD*s) to describe the bean configuration file. This, in turn, allows a new AOP-specific namespace to be introduced. The new *AOP* namespace provides a cleaner, terser syntax for aspects.
- Spring 2 allows the use of *AspectJ*'s pointcut language, which is much more powerful (and complex) than Spring *AOP*.

  AspectJ
- Spring 2 allows Aspects to be POJOs, instead of forcing them to have dependencies on interfaces such as MethodBeforeAdvice and MethodInterceptor.

In addition to all this new goodness, the Spring team is determined to provide an easy transition path from Spring *AOP* up to AspectJ. To this end, all the 1.*x* features continue to work.

Better still, the new *AOP* namespace uses Aspect terminology directly, instead of referring to Spring-specific classes. This lets you plug in different implementations with no other change to configuration. New applications can use the new configuration schema to start simple with

Spring *AOP* and then power up to the AspectJ pointcut language with a minimum of fuss. The TestDeclarativeAop application demonstrates AOP, Spring 2–style:

```java
package aop2;

import org.springframework.context.support.FileSystemXmlApplicationContext;
import org.springframework.context.ApplicationContext;
import aop.PersonName;
import util.TestBase;

public class TestDeclarativeAop extends TestBase {
  public void testIt() {
    ApplicationContext ctxt =
        new FileSystemXmlApplicationContext("config/test_declarative_aop_2.xml");
    PersonName pn = (PersonName) ctxt.getBean("personName");
    assertThrows("IllegalArgument", pn, "setFirstName", (Object) null);
    assertThrows("IllegalArgument", pn, "setLastName", (Object) null);
    assertDoesNotThrow(pn, "setMiddleInitial", (Object) null);
  }
}
```

By now it should come as no surprise that this client shows no awareness that it depends on beans that use aspects (or Spring 2, for that matter). To look under the hood, check out the configuration file:

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

  <aop:config>
    <aop:aspect id="nullBlocker" ref="paramValidator">
      <aop:pointcut id="pc"
                    expression="execution(* set*Name(..)) and args(name)"/>
      <aop:before pointcut-ref="pc"
                  method="blockNull"
                  arg-names="name "/>
    </aop:aspect>
  </aop:config>
  <bean id="personName"     class="aop.PersonName"/>
  <bean id="paramValidator" class="aop2.ParamValidator"/>
</beans>
```

Here things are quite a bit different. First you see a bunch of schema and namespace goo added as attributes on the <*beans*> element. This is verbose but boilerplate. Then the fun begins. The aop:config element

CLICK HERE to purchase this book now.

configures our aspect. Much of this is self-explanatory. Here are a few key points:

- The aspect specifies not only an aspect instance paramValidator, but also the exact method and arguments that will be called (the method and arg-names() attributes). This is a necessary good. Since paramValidator is now a *POJO*, we can choose to invoke any method we want as before advice.
- The expression attribute uses the AspectJ pointcut syntax. That's a topic for a whole book, but suffice it to say that this matches calls of the form setXXXName() and makes the argument available to the aspect.

Now let's look at ParamValidator:

```
package aop2;

public class ParamValidator {
  public void blockNull(String name) {
    if (name == null) {
      throw new IllegalArgumentException("Cannot be null");
    }
  }
}
```

The important thing here is what's not here. ParamValidator is just a *POJO*—no aspects or Spring-specific interfaces. Since we can invoke any method on this POJO as advice, we have chosen the meaningful name blockNull. (This hints that in the future we might hang all sorts of other advice methods on this same class.)

The sum of these new features is very exciting. The new configuration syntax and *POJO* support make aspects more friendly and approachable and may encourage wider adoption. On the other end of the scale, the AspectJ integration makes Spring *AOP* much more powerful and ready to tackle more complex problems. With Spring 2, aspects have completed their move from "esoteric power tool" to a regular part of the development process for all applications.

5.7 In Conclusion

Spring represents the tireless application of common sense and field-tested knowledge to Java development. The two cores of Spring are dependency injection (*DI*) and aspect-oriented programming (*AOP*).

CLICK HERE to purchase this book now.

With *DI*, you rid your code of unnecessary dependencies—it becomes focused, testable, and domain-oriented. With *AOP*, you can elegantly manage the necessary dependencies in your application. Your application source files can each clearly focus on a single task and avoid degeneration to an unmaintainable tangle of cross-cutting concerns.

5.8 Resources

Sample Code Download

The Spring Exploration application sample code used in this article is available online at http://www.codecite.com/project/spring_xt. Some of the examples were inspired by *Pro Spring* [HM05], which is recommended follow-up reading.

Running the Code

The code was tested against the following classpath on an M3 build of Spring 2. All the code before the Spring 2 section was also tested on Spring 1.2.6. You would use the same JAR files on 1.2.6, minus the "asm" files.

Classpath

```
eclipse/plugins/org.junit_3.8.1/junit.jar
spring-framework-2.0-m3/dist/spring.jar
spring-framework-2.0-m3/lib/asm/asm-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-analysis-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-attrs-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-commons-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-tree-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-util-2.2.1.jar
spring-framework-2.0-m3/lib/asm/asm-xml-2.2.1.jar
spring-framework-2.0-m3/lib/aspectj/aspectjweaver.jar
spring-framework-2.0-m3/lib/cglib/cglib-nodep-2.1_3.jar
spring-framework-2.0-m3/lib/jakarta-commons/commons-logging.jar
spring-framework-2.0-m3/lib/log4j/log4j-1.2.13.jar
```

Recommended Books

- *Pro Spring* [HM05] by Rob Harrop and Jan Machacek
- *AspectJ in Action* [Lad03] by Ramnivas Laddad

CLICK HERE to purchase this book now.

Utility Code

Utility code that is called by the examples in this article is included here for reference:

```
package util;

import junit.framework.TestCase;

import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
import java.util.*;

public class TestBase extends TestCase {
  static final HashMap exTypes = new HashMap();
  static {
    exTypes.put("IllegalArgument", IllegalArgumentException.class);
  }
  private Method getMethod(Object o, String methName, Object... args) {
    Class cls = o.getClass();
    Method[] methods = cls.getMethods();
    for (int i = 0; i &lt; methods.length; i++) {
      Method method = methods[i];
      if (method.getName().equals(methName)) {
        return method;
      }
    }
    throw new IllegalArgumentException(String.format("%s has no %s",
                                                      cls, methName));
  }
  public void assertIteratorEquals(Iterator it, Object... value) {
    int count=0;
    while (it.hasNext()) {
      assertEquals(value[count], it.next());
      count++;
    }
    assertEquals(count, value.length);
  }
  public Object assertDoesNotThrow(Object obj, String meth,
                                   Object... args) {
    Method method = getMethod(obj, meth, args);
    try {
      return method.invoke(obj, args);
    } catch (IllegalAccessException e) {
      throw new Error(e);
    } catch (InvocationTargetException e) {
      throw new Error(e.getTargetException());
    }
  }
  public void assertThrows(String exPrefix, Object obj, String meth,
                           Object... args) {
```

CLICK HERE to purchase this book now.

```
      Class cls = (Class) exTypes.get(exPrefix);
      if (cls == null) {
        throw new Error("Unknown exPrefix " + exPrefix +
                        ", add to exTypes");
      }
      assertThrows(cls, obj, meth, args);
    }
    public void assertThrows(Class exClass, Object obj, String meth,
                             Object... args) {
      Method method = getMethod(obj, meth, args);
      try {
        method.invoke(obj, args);
      } catch (InvocationTargetException ite){
        Throwable t = ite.getTargetException();
        if (!exClass.isAssignableFrom(t.getClass())) {
          throw new Error(t);
        }
        return;
      } catch (IllegalAccessException e) {
        fail("Unexpected " + e);
      }
      fail("Expected " + exClass);
    }
  }
```

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by working practitioners. The titles continue the well-known Pragmatic Programmer style, and continue to garner awards and rave reviews. As software development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers and their managers stay on top of their game.

Visit Us Online

No Fluff Just Stuff 2006
`pragmaticprogrammer.com/titles/nfjs06`
Web home for this book, where you can find and submit errata, send us feedback, and find other related resources.

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/nfjs06.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	http://www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com