**What Does Spring Provide, in Terms of AOP?**

*"The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications."*   
*The Spring Framework Reference Documentation*

In order to meet this aim, the Spring framework currently supports a selection of AOP concepts, ranging from pointcuts to advice. This article will show you how to use the following AOP concepts as they are implemented in the Spring framework:

* **Advice**: How to declare before, afterReturning and afterThrowing advice as beans.
* **Pointcuts**: How to declare static pointcut logic to tie everything together in the XML Spring Bean Configuration files.
* **Advisors**: The way to associate pointcut definitions with advice beans.

**Setting the Scene: An Example Simple Application**

*"In general, Spring isn't prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach."*  
*The Spring Framework Reference Documentation*

The first step in trying out the AOP capabilities of the Spring framework is to create a simple Java application. The IBusinessLogicinterface and BusinessLogic class provide the simple building blocks for a bean within the Spring framework. Although the interface is not necessarily required by the logic of our simple application, it is a good practice that is promoted by the Spring framework.

public interface IBusinessLogic

{

public void foo();

}

public class BusinessLogic

implements IBusinessLogic

{

public void foo()

{

System.out.println(

"Inside BusinessLogic.foo()");

}

}

A MainApplication class can be written to exercise the BusinessLogic bean's public methods.

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.FileSystemXmlApplicationContext;

public class MainApplication

{

public static void main(String [] args)

{

// Read the configuration file

ApplicationContext ctx =

new FileSystemXmlApplicationContext(

"springconfig.xml");

//Instantiate an object

IBusinessLogic testObject =

(IBusinessLogic) ctx.getBean("businesslogicbean");

// Execute the public

// method of the bean

testObject.foo();

}

}

There's nothing too worrying in the BusinessLogic class or its associated interface. However, the MainApplication class does something interesting in how it initializes the BusinessLogic object. By using the ctx.getBean("businesslogicbean") call, theMainApplication hands off the loading and management of bean instance of the BusinessLogic class to the Spring framework.

Allowing Spring to control the initialization of the BusinessLogic bean gives the Spring runtime the opportunity to do all of the management tasks required of a J2EE system for the bean before it is handed back to the application. The configuration of the Spring runtime can then decide which tasks and modules are to be applied to the bean. This configuration information is provided by an XML file similar to the one shown below.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC

"-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-- Bean configuration -->

<bean id="businesslogicbean"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces">

<value>IBusinessLogic</value>

</property>

<property name="target">

<ref local="beanTarget"/>

</property>

</bean>

<!-- Bean Classes -->

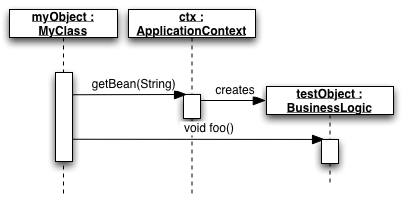
<bean id="beanTarget"

class="BusinessLogic"/>

</beans>

This configuration file, saved as *springconfig.xml*, specifies that a bean is to be loaded whose interface matches IBusinessLogic. The bean is then tied to the implementation class BusinessLogic. This may seem like a lot of effort just to get a simple bean loaded and to call a single method, but remember that this configuration file is an integral piece of the puzzle that enables the Spring framework to transparently apply its components to your application.

Figure 1 shows the basic sequence diagram when the MainApplication is executed as it stands, with no aspects applied.

  
*Figure 1. Sequence diagram showing when no aspects are applied to the BusinessLogic bean*

See the [references](http://www.onjava.com/pub/a/onjava/2004/07/14/springaop.html?page=2#references) at the end of this article for the source code to this simple Spring application.

**Applying a Method Tracing Aspect**

Possibly the most basic stalwart of aspects is the Method Tracing aspect. This passive aspect is about as simple an aspect as you will find, and so is the best place to start when investigating a new implementation of AOP.

A Method Tracing aspect captures calls to, and returns from, methods being traced within a target application and displays this information in some way. In AO, the before and after types of advice are used to capture these types of join points as they can be triggered before and after a method call join point. Using the Spring framework, the before advice for the Method Tracing aspect is declared in the TracingBeforeAdvice class.

import java.lang.reflect.Method;

import org.springframework.aop. MethodBeforeAdvice;

public class TracingBeforeAdvice

implements MethodBeforeAdvice

{

public void before(Method m,

Object[] args,

Object target)

throws Throwable

{

System.out.println(

"Hello world! (by " +

this.getClass().getName() +

")");

}

}

Similarly, the after advice can be declared in the TracingAfterAdvice class.

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class TracingAfterAdvice

implements AfterReturningAdvice

{

public void afterReturning(Object object,

Method m,

Object[] args,

Object target)

throws Throwable

{

System.out.println(

"Hello world! (by " +

this.getClass().getName() +

")");

}

}

Both classes represent a specific piece of advice by implementing the appropriate advice interface from the Spring framework. Each type of advice specifies that either the before(..) or afterReturning(..) methods are implemented to enable the Spring runtime to notify the advice when an appropriate join point has been reached. It is worth noting that the TracingAfterAdvice actually extends from AfterReturningAdvice, which means that the advice will only be run if the join point that is encountered returns without an exception.

In order to attach the advice to the appropriate join points in your application you must make a few amendments to the*springconfig.xml*.

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE beans PUBLIC

"-//SPRING//DTD BEAN//EN"

"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

<!-- Bean configuration -->

<bean id="businesslogicbean"

class="org.springframework.aop.framework.ProxyFactoryBean">

<property name="proxyInterfaces">

<value>IBusinessLogic</value>

</property>

<property name="target">

<ref local="beanTarget"/>

</property>

<property name="interceptorNames">

<list>

<value>theTracingBeforeAdvisor</value>

<value>theTracingAfterAdvisor</value>

</list>

</property>

</bean>

<!-- Bean Classes -->

<bean id="beanTarget"

class="BusinessLogic"/>

<!-- Advisor pointcut definition for before advice -->

<bean id="theTracingBeforeAdvisor"

class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice">

<ref local="theTracingBeforeAdvice"/>

</property>

<property name="pattern">

<value>.\*</value>

</property>

</bean>

<!-- Advisor pointcut definition for after advice -->

<bean id="theTracingAfterAdvisor"

class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">

<property name="advice">

<ref local="theTracingAfterAdvice"/>

</property>

<property name="pattern">

<value>.\*</value>

</property>

</bean<

<!-- Advice classes -->

<bean id="theTracingBeforeAdvice"

class="TracingBeforeAdvice"/>

<bean id="theTracingAfterAdvice"

class="TracingAfterAdvice"/>

</beans>

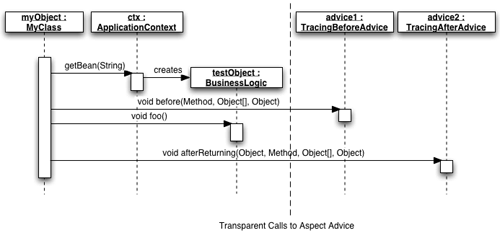
The theTracingBeforeAdvisor and theTracingAfterAdvisor advisors are added to the previously declared businesslogicbean. Each advisor potentially intercepts all join points on the beans to which they are attached. The advisors are beans themselves, and their sole job is to tie together pointcut definitions and advice beans. The pointcut definitions in this example are regular expressions specifying the join points of interest within the static object hierarchy.

Since the org.springframework.aop.support.RegexpMethodPointcutAdvisor pointcut advisor is being used in this example, the pointcut logic is specified using a regular expression. The regular expression is used to identify the join points on the public interface to the IBusinessLogic interface. Some simple examples of regular expressions that could have been used to specify different collections of join points on the IBusinessLogic interface are:

* **<value>.\*</value>**: This expression selects all join points on the bean or beans to which the advisor has been attached.
* **<value>./IBusinessLogic/.foo</value>**: This expression selects join points on the foo() method on the IBusinessLogicinterface only. *This will only select join points on the IBusinessLogic interface if it is one of the beans to which the advisor has been attached.*

The final bean declarations in the *springconfig.xml* file specify the classes that implement the advice beans.

Now that the correct configuration for the tracing aspects has been specified, the next time the MainApplication is executed, the aspects will be woven in during the initialization, and all methods within the BusinessLogic bean will be traced, as shown in Figure 2.

[](http://www.onjava.com/onjava/2004/07/14/graphics/fig2_full.png)  
*Figure 2. Sequence diagram showing the Method Tracing aspect being applied to the BusinessLogic bean (click for full-size image)*

The source code for the Method Tracing aspect and example application is available for download in the [references](http://www.onjava.com/pub/a/onjava/2004/07/14/springaop.html?page=2#references) section at the end of this article.