JBoss AOP - User Guide

The Case For Aspects

ISBN:
Publication date:

JBoss AOP - User Guide: The Case For Aspects

Preface

Aspect-Oriented Programming (AOP) is a new paradigm that allows you to organize and layer your software applications in ways that are impossible with traditional object-oriented approaches. Aspects allow you to transparently glue functionality together so that you can have a more layered design. AOP allows you to intercept any event in a Java program and trigger functionality based on those events. Mixins allow you to introduce multiple inheritance to Java so that you can provide APIs for your aspects. Combined with JDK 5.0 annotations, it allows you to extend the Java language with new syntax.

JBoss AOP is a 100% Pure Java aspected oriented framework usable in any programming environment or as tightly integrated with our application server.

This document walks you through how AOP can be used to build your applications. A large number of real-world examples are given in each section to illustrate various ideas. The book is broken up in the following manner:

What is an Aspect?
: The section gives an overview on exactly what Aspect-Oriented Programming is.

Terms
: Defines some basic AOP terms that you will need when reading this document.

Building Aspects
: The chapter walks through some examples of building an aspect. One example shown is how to detect JDBC connection leakages in your code. Another example is using the Observer/Observable pattern Another is doing dependency injection.

Aspect-Oriented Annotations
: This chapter walks through how you can use JDK 5.0 annotations and AOP to actually extend the Java language with new syntax. Numerous real-world examples are given.

Mixins and Introductions
: This chapter walks through how you can use AOP introductions and mixins to have aspects with APIs, or how to extend a POJO's functionality transparently. Specific examples are an Asynchronous framework and making a POJO into a JMX MBean.

Dynamic AOP
: This chapter walks through how you can use the dynamic per-instance API that each aspectized class has to define aspects on a per instance basis rather than for every instance of a class.

Integration with Pointcuts
: This chapter steps away from writing aspects, but rather talks about how you can publish pointcuts that can be used by users/customers to integrate with your products.

Testing with AOP
: This chapter shows how you can use AOP to test your applications in a easier way.

If you have questions, use the user forum linked on the JBoss AOP website. We also provide a tracking links for tracking bug reports and feature requests. If you are interested in the development of JBoss AOP, post a message on the forum. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

Commercial development support, production support and training for JBoss AOP is available through JBoss Inc. (see http://www.jboss.org/). JBoss AOP is a project of the JBoss Professional Open Source product suite.

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
   long line that \
   does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

What Is Aspect-Oriented Programming?

1. What is it?

An aspect is a common feature that's typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can't find a way to express this structure in code with traditional object-oriented techniques.

For example, metrics is one common aspect. To generate useful logs from your application, you have to (often liberally) sprinkle informative messages throughout your code. However, metrics is something that your class or object model really shouldn't be concerned about. After all, metrics is irrelevant to your actual application: it doesn't represent a customer or an account, and it doesn't realize a business rule. It's simply orthogonal.

In AOP, a feature like metrics is called a crosscutting concern, as it's a behavior that "cuts" across multiple points in your object models, yet is distinctly different. As a development methodology, AOP recommends that you abstract and encapsulate crosscutting concerns.

For example, let's say you wanted to add code to an application to measure the amount of time it would take to invoke a particular method. In plain Java, the code would look something like the following.

```
public class BankAccountDAO
{
  public void withdraw(double amount)
  {
    long startTime = System.currentTimeMillis();
    try
    {
      // Actual method body...
    }
    finally
    {
      long endTime = System.currentTimeMillis() - startTime;
      System.out.println("withdraw took: " + endTime);
    }
  }
}
```

While this code works, there are a few problems with this approach:

1. It's extremely difficult to turn metrics on and off, as you have to manually add the code in the try>/finally block to each and every method or constructor you want to benchmark.

2. The profiling code really doesn't belong sprinkled throughout your application code. It makes your code bloated and harder to read, as you have to enclose the timings within a try/finally block.

3. If you wanted to expand this functionality to include a method or failure count, or even to register these statistics to a more sophisticated reporting mechanism, you'd have to modify a lot of different files (again).

This approach to metrics is very difficult to maintain, expand, and extend, because it's dispersed throughout your entire code base. And this is just a tiny example! In many cases, OOP may not always be the best way to add metrics to a class.

Aspect-oriented programming gives you a way to encapsulate this type of behavior functionality. It allows you to add behavior such as metrics "around" your code. For example, AOP provides you with programmatic control to specify that you want calls to BankAccountDAO to go through a metrics aspect before executing the actual body of that code.

2. Creating Aspects in JBoss AOP

In short, all AOP frameworks define two things: a way to implement crosscutting concerns, and a programmatic construct -- a programming language or a set of tags -- to specify how you want to apply those snippets of code.

Let's take a look at how JBoss AOP, its cross-cutting concerns, and how you can implement a metrics aspect in JBoss.

The first step in creating a metrics aspect in JBoss AOP is to encapsulate the metrics feature in its own Java class. Listing Two extracts the try/finally block in Listing One's BankAccountDAO.withdraw() method into Metrics, an implementation of a JBoss AOP Interceptor class.

Listing Two: Implementing metrics in a JBoss AOP Interceptor

```
01. public class Metrics implements
 org.jboss.aop.advice.Interceptor
02. {
03.   public Object invoke(Invocation invocation) throws Throwable
04.   {
05.     long startTime = System.currentTimeMillis();
06.     try
07.     {
08.       return invocation.invokeNext();
09.     }
```

```
10.     finally
11.     {
12.       long endTime = System.currentTimeMillis() - startTime;
13.       java.lang.reflect.Method m =
 ((MethodInvocation)invocation).method;
14.       System.out.println("method " + m.toString() + " time: " +
 endTime + "ms");
15.     }
16.   }
17. }
```

Under JBoss AOP, the Metrics class wraps withdraw(): when calling code invokes withdraw(), the AOP framework breaks the method call into its parts and encapsulates those parts into an Invocation object. The framework then calls any aspects that sit between the calling code and the actual method body.

When the AOP framework is done dissecting the method call, it calls Metric's invoke method at line 3. Line 8 wraps and delegates to the actual method and uses an enclosing try/finally block to perform the timings. Line 13 obtains contextual information about the method call from the Invocation object, while line 14 displays the method name and the calculated metrics.

Having the metrics code within its own object allows us to easily expand and capture additional measurements later on. Now that metrics are encapsulated into an aspect, let's see how to apply it.

3. Applying Aspects in JBoss AOP

To apply an aspect, you define when to execute the aspect code. Those points in execution are called pointcuts. An analogy to a pointcut is a regular expression. Where a regular expression matches strings, a pointcut expression matches events/points within your application. For example, a valid pointcut definition would be "for all calls to the JDBC method executeQuery(), call the aspect that verifies SQL syntax."

An entry point could be a field access, or a method or constructor call. An event could be an exception being thrown. Some AOP implementations use languages akin to queries to specify pointcuts. Others use tags. JBoss AOP uses both. Listing Three shows how to define a pointcut for the metrics example.

Listing Three: Defining a pointcut in JBoss AOP

```
1. <bind pointcut="public void
 com.mc.BankAccountDAO->withdraw(double amount)">
2.       <interceptor class="com.mc.Metrics"/>
3. </bind >

4. <bind pointcut="* com.mc.billing.*->*(..)">
```

```
5.       <interceptor class="com.mc.Metrics"/>
6. </bind >
```

Lines 1-3 define a pointcut that applies the metrics aspect to the specific method BankAccountDAO.withdraw(). Lines 4-6 define a general pointcut that applies the metrics aspect to all methods in all classes in the com.mc.billing package.

There is also an optional annotation mapping if you do not like XML. See our Reference Guide for more information.

JBoss AOP has a rich set of pointcut expressions that you can use to define various points/events in your Java application so that you can apply your aspects. You can attach your aspects to a specific Java class in your application or you can use more complex compositional pointcuts to specify a wide range of classes within one expression.

With AOP, as this example shows, you're able to pull together crosscutting behavior into one object and apply it easily and simply, without polluting and bloating your code with features that ultimately don't belong mingled with business logic. Instead, common crosscutting concerns can be maintained and extended in one place.

Notice too that the code within the BankAccountDAO class has no idea that it's being profiled. This is what aspect-oriented programmers deem orthogonal concerns. Profiling is an orthogonal concern. In the OOP code snippet in Listing One, profiling was part of the application code. With AOP, you can remove that code. A modern promise of middleware is transparency, and AOP (pardon the pun) clearly delivers.

Just as important, orthogonal behavior could be bolted on after development. In Listing One, monitoring and profiling must be added at development time. With AOP, a developer or an administrator can (easily) add monitoring and metrics as needed without touching the code. This is a very subtle but significant part of AOP, as this separation (obliviousness, some may say) allows aspects to be layered on top of or below the code that they cut across. A layered design allows features to be added or removed at will. For instance, perhaps you snap on metrics only when you're doing some benchmarks, but remove it for production. With AOP, this can be done without editing, recompiling, or repackaging the code.

Terms

1. Overview

The section defines some basic terms that will be used throughout this guide.

Joinpoint
: A joinpoint is any point in your java program. The call of a method. The execution of a constructor the access of a field. All these are joinpoints. You could also think of a joinpoint as a particular Java event. Where an event is a method call, constructor call, field access etc...

Invocation
: An Invocation is a JBoss AOP class that encapsulates what a joinpiont is at runtime. It could contain information like which method is being called, the arguments of the method, etc...

Advice
: An advice is a method that is called when a particular joinpoint is executed, i.e., the behavior that is triggered when a method is called. It could also be thought of as the code that does the interception. Another analogy is that an advice is an "event handler".

Pointcut
: Pointcuts are AOP's expression language. Just as a regular expression matches strings, a pointcut expression matches a particular joinpoint.

Introductions
: An introduction modifies the type and structure of a Java class. It can be used to force an existing class to implement an interface or to add an annotation to anything.

Aspect
: An Aspect is a plain Java class that encapsulates any number of advices, pointcut definitions, mixins, or any other JBoss AOP construct.

Interceptor
: An interceptor is an Aspect with only one advice named "invoke". It is a specific interface that you can implement if you want your code to be checked by forcing your class to implement an interface. It also will be portable and can be reused in other JBoss environments like EJBs and JMX MBeans.

Building Aspects

The last chapter had a basic boring introduction to aspects with the lame, commonly used example of metrics. AOP can be applied in a much broader sense than the overused examples of tracing and security and this chapter looks into other more compelling examples of using basic AOP.

1. Identifying Aspects

Aspect-Oriented programming is not a replacement for object-oriented programming, but rather a compliment to it. AOPers generally say that OOP solves 90% of problems and AOP solves the 10% of problems that OOP isn't good at. This section of the docbook will expand over time, but let's discuss some ways in which you can identify whether or not AOP is a good solution for a particular problem.

Cross-cutting Concerns
: The metrics example in the previous chapter is an example of a cross-cutting concern in its purest form. Sometimes you see structure in your code that can't be expressed as an object because it completly wraps around the method you are invoking. If the behavior in question is something that you want to be able to extend and maintain within its own structure then it may be a candidate for aspectizing.

Layering Based on Deployment
: Another place where AOP may be very useful is to layer your applications. Sometimes you want to model a particular service or object that has many configuration options yet you don't want to bloat your service with unmaintainable code. AOP provides a nice way to layer such complex services. JBoss AOP provides a XML configurable mechanism to configure such aspects per deployment. A good example of this is a caching service that might have different locking policies. It is easier to encapsulate such locking policies as aspects so that the base caching code doesn't get polluted with locking concerns. This makes the code easier to maintain.

Transparency
: Many times you want to write plain Java code that focuses solely on business or application logic and do not want to introduce any concerns like middleware. AOP allows you to apply things like middleware transparently to your code without polluting your code. Some examples include the transaction demarcation and role-based security features in the JBoss AOP Aspect Library.

2. Exception Handling

Metrics and tracing are simple examples of building aspects. Another great simple example is to use AOP for exception handling. For example, SQLException is an

exception that contains error information like invalid sql statement or deadlock that is similar per database vendor, but is expressed as different error codes and string messages. You can use AOP to intercept statement execution, catch SQLException errors, and turn them into typed exceptions that application code can handle independent of database vendor. So let's code an example of this.

```
public class InvalidSQlException extends SQLException
{
   InvalidSQLException(SQLException ex)
   {
       super(ex.getMessage(), ex.getSQLState(), ex.getErrorCode());
   }
}
```

What we'll do is write an aspect that wraps calls to all `java.sql.Statement` execute methods and turn them into typed exceptions like the example above. We'll leave some code up to your imagination since such an aspect would be quite long to deal with every error code of every database vendor, but hopefully you can get the idea here.

```
public class SQLExceptionAspect
{
   public Object handleSqlException(Invocation invocation) throws
 Throwable
   {
      try
      {
         return invocation.invokeNext();
      }
      catch (SQLException ex)
      {
         if (isVendorInvalidSqlErrorCode(ex.getErrorCode())) throw
 new InvalidSQLException(ex);
         if (isVendorDeadlockErrorCode(ex.getErrorCode()) throw new
 SQLDeadlockException(ex);
         ... and so on ...
      }
   }
   ... impl of isVendor methods ...
}
```

Now that the aspect is defined we use a `call` pointcut expression to intercept all the execute methods of `java.sql.Statement`.

```
<aspect class="SQLExceptionAspect" scope="PER_VM"/>
<bind pointcut="call(*
 $instanceof{java.sql.Statement}->execute*(..))">
   <advice name="handleSqlException" aspect="SQLExceptionAspect"/>
```

```
</bind>
```

Aspect-Oriented Annotations

Annotations are a new feature of JDK 5.0 that allow you to attach metadata to any Java construct. They allow you to define metadata in a typesafe way and apply it to a class, method, constructor, field, or parameter. For those of you familiar with XDoclet, annotations will be very intuitive to you in that you are used to declaring tags to generate code. The main difference between the two is that annotations are a typed part of the Java language, while XDoclet tags can be mistyped and are harder to create. In a nutshell, JDK 5.0 annotations allow you to define new Java syntax.

AOP provides a unique way of encapsulating behavior and applying it transparently to your application code. If you combine it with annotations, you basically have a very structured, simple way of extending the Java language. The annotation is the syntax, and the aspect provides the functionality for that aspect. This chapter walks through detailed examples on how you can use AOP and annotations to turn your frameworks into Java language features.

1. Methods and Annotations

Let's take a look at how you can use method annotations with AOP. Using annotations and AOP together and applying this to a method is very analogous to using Java's synchronized keyword with a method. When you tag a method as synchronized, you are telling the JVM that you want that method to behave in a special way when it is invoked. Annotations allow you to define new keywords that you want to have trigger your own special custom behavior. AOP gives you the ability to encapsulate this behavior and weave it into the execution of the method.

Let's say we want to add a new syntax that will allow us to fire `void` methods in the background, in another thread, if they are tagged as `@Oneway`. Using this new syntax would look like this:

```
import org.jboss.aspects.Oneway;

public class Foo
{
   @Oneway public static void someMethod() {...}

   public static void main(String[] args)
   {
      someMethod(); // executes in background
   }
}
```

When `someMethod()` is invoked within main, it will run asynchronously so that the code in main is free to do ther tasks in parallel.

To implement this functionality, the first thing that must be done is to define the new Java syntax for our `@Oneway` tag within an annotation.

```
package org.jboss.aspects;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
public @interface Oneway {}
```

Simple enough. The `@Target` tag allows you to narrow down where the annotation is allowed to be applied. In this case, our @Oneway annotation can only be applied to a method. Remember, this is all pure 100 percent Java that is available in J2SE 5.0.

The next thing we have to do is to define an aspect class that will encapsulate our `@Oneway` behavior.

```
package org.jboss.aspects;

public OnewayAspect
{
   private static class Task implements Runnable
   {
      private MethodInvocation invocation;

      public Task(MethodInvocation invocation)
      {
        this.invocation = invocation;
      }
      public void run()
      {
        try { invocation.invokeNext(); }
        catch (Throwable ignore) { }
      }
   }


   public Object oneway(MethodInvocation invocation) throws
 Throwable
   {
      MethodInvocation copy = invocation.copy();
      Thread t = new Thread(new Task(copy));
      t.setDaemon(false);
      t.start();
      return null;
```

```
   }
}
```

The aspect is simple enough. The oneway() method copies the invocation, creates a thread, fires off the complete invocation in the background, and returns. We could imagine a more sophisticated example using some of the new Executors within the J2SE 5.0 java.util.concurrent package, but hopefully this code illustrates how you could build on this example to implement more complete implementations.

The last thing that must be done is to specify the pointcut expression that will trigger the application of the `OnewayAspect` when the `@Oneway` annotation is declared on a method.

```
<aop>
   <aspect class="org.jboss.aspects.OnewayAspect"/>

   <bind pointcut="execution(void *->@org.jboss.Oneway(..))">
      <advice name="oneway"
              aspect="org.jboss.aspects.OnewayAspect"/>
   </bind>
</aop>
```

The pointcut expression states that any void method that is tagged as `@Oneway` should have the `OnewayAspect.oneway()` method executed before it itself executes. With the annotation, aspect, and pointcut expression now defined, the @Oneway syntax is now usable in your application. A simple, clean, easy way of extending the Java language!

2. Fields and Annotations

Let's look at how you could use field annotations and AOP. Using annotations and AOP, you can can actually change how a field is stored by an object or as a static member of a class. What we want to accomplish in this example is that when you tag a field (static or member) as @ThreadBased, its value will behave as though it were stored in a java.lang.ThreadLocal. Sure, you could use a ThreadLocal variable directly, but the problem with ThreadLocal is that it is untyped and you have to use "verbose" (okay, they're not that verbose) get() and set() methods. So what we'll do here is create a typed ThreadLocal field. Basically, we'll create a new Java field type called the @Threadbased variable.

Using this new type would look like this:

```
import org.jboss.aspects.Threadbased;
```

```
public class Foo
{
   @Threadbased private int counter;
}
```

To implement this functionality, we must first define the annotation.

```
package org.jboss.aspects;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.FIELD})
public @interface Threadbased {}
```

Simple enough. The @Target tag allows you to narrow down where the annotation is allowed to be applied. In this case, our @Threadbased annotation can only be applied to fields.

The next thing to do is to define the aspect that will encapsulate our ThreadLocal behavior.

```
package org.jboss.aspects;

import org.jboss.aop.joinpoint.*;
import java.lang.reflect.Field;

public class ThreadbasedAspect
{
   private ThreadLocal threadbased = new ThreadLocal();

   public Object access(FieldReadInvocation invocation)
       throws Throwable
   {
      // just in case we have a primitive,
      // we can't return null
      if (threadbased.get() == null)
          return invocation.invokeNext();
      return threadbased.get();
   }

   public Object access(FieldWriteInvocation invocation)
       throws Throwable
   {
      threadbased.set(invocation.getValue());
      return null;
```

```
    }
}
```

ThreadbasedAspect encapsulates the access to a Java field. It has a dedicated ThreadLocal variable within it to track threadlocal changes to a particular field. It also has separate access() methods that are invoked depending upon whether a get or set of the field is called. These methods delegate to the ThreadLocal to obtain the current value of the field.

Finally, we must define a pointcut expression that will trigger the application of the ThreadbasedAspect when the @Threadbased annotation is specified on a particular field.

```
<aop>
   <aspect class="org.jboss.aspects.ThreadbasedAspect"
           scope="PER_JOINPOINT"/>
   <bind pointcut="field(* *->@org.jboss.aspects.Threadbased)">
      <advice name="access"
              aspect="org.jboss.aspects.ThreadbasedAspect"/>
   </bind>
</aop>
```

Just in case we have multiple @Threadbased variables defined in one class, we want an instance of ThreadbasedAspect to be allocated per field for static fields. For member fields, we want an instance of ThreadbasedAspect to be allocated per field, per object instance. To facilitate this behavior, the aspect definition scopes the instance of when and where the aspect class will be allocated by setting it to PER_JOINPOINT. If we didn't do this scoping, JBoss AOP would only allocate one instance of ThreadbasedAspect and different fields would be sharing the same instance of the ThreadLocal -- something that we don't want.

Well that's it. A clean, easy way of extending Java to specify a new special type. Note: This particular aspect comes bundled with JBoss AOP.

3. Dependency Injection

Another interesting place where field annotations and AOP can be used is with dependency injection. Dependency injection is about objects declaring what information, configuration, or service references they need, and having the runtime automagically inject those dependencies rather than having your code do explicit lookups on a registry service. In J2EE-land, getting access to a javax.transaction.TransactionManager service is not standardized and is actually different per vendor implementation. Many framework developers need to use the TransactionManager to implement custom transactional services. The use of AOP with field annotations is a great way to provide this dependency injection

and to abstract away the details of how a TransactionManager is referenced by components that need it. Let's define an aspect that will inject a reference to a TransactionManager into the value of a field.

First, we must again define our annotation.

```
package org.jboss.aspects;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.FIELD})
public @interface Inject {}
```

Next we will define the aspect class that will encapsulate the resolving of the TransactionManager. This aspect will be specific to the JBoss application server, but you could define different implementations per vendor.

```
package org.jboss.aspects;

import org.jboss.aop.joinpoint.*;
import java.lang.reflect.Field;
import javax.transaction.TransactionManager;
import org.jboss.tm.TxManager;

public InjectTMAspect
{
   private TransactionManager tm = TxManager.getInstance();

   public Object access(FieldReadInvocation invocation)
       throws Throwable
   {
      return tm;
   }

   public Object access(FieldWriteInvocation invocation)
       throws Throwable
   {
      throw new RuntimeException(
          "Setting an @Injected variable is illegal");
   }
}
```

Finally, we have to define the XML binding that will trigger the application of the InjectTMAspect when the @Inject tag is applied to a field. The pointcut expression basically states that for any field of type TransactionManager and tagged as @Inject, apply the InjectTMAspect.

```
<aop>
  <aspect class="org.jboss.aspects.InjectTMAspect"/>

  <bind pointcut="field(javax.transaction.TransactionManager
 *->@org.jboss.aspects.Inject)">
     <advice name="access"
             aspect="org.jboss.aspects.InjectTMAspect"/>
  </bind>
</aop>
```

Now that the annotation, aspect class, and XML binding have been defined, we can use it within our code.

```
import javax.transaction.TransactionManager;
import org.jboss.aspects.Inject;

public class MyTransactionalCache
{
   @Inject private TransactionManager tm;
...
}
```

Mixins and Introductions

When people thing of AOP, they usually think of interception, pointcut expressions, aspects, and advices. AOP isn't just about those things. Another important feature in JBoss AOP is the ability to introduce an interface to an existing Java class in a transparent way. You can force a class to implement an interface and even specify an additional class called a mixin that implements that interface. Very similar to C++'s multiple inheritance. Now, why would you want to use introductions/mixins? That's what this chapter is all about.

1. Introducing Introductions

The first thing we'll show here is how to force an existing Java class to implement any interface you want. The JBoss AOP tutorial is a good place to start for an example, so let's grab the code from the introductions tutorial.

The first example we'll show is how to take an existing non-serializable class and make it serializable. This use case may be usable if there's a thirdparty library you don't have the source for, or you want to control whether your class is serializable or not based on how you deploy your application.

```
public class POJO
{
  private String field;
}
```

To take this class and make it serializable is very simple. Just the following XML is required:

```
<introduction class="POJO">
   <interfaces>java.io.Serializable</interfaces>
</introduction>
```

The above XML just states that the AOP framework is to apply the `java.io.Serializable` interface to the `POJO` class. You can have one or more interfaces specified with the `interfaces` element. These interfaces are comma delimited.

If the introduced interfaces have methods not implemented by the class, then the AOP framework with add an implementation of these methods to the class. The methods will delegate to the AOP framework and must be handled/serviced by an interceptor or advice otherwise a `NullPointerException` will result.

2. Mixin It Up

Introducing interfaces only is quite limited when the introduced interfaces have methods that the class doesn't implement as you have to write a lot of generically inclined code that handle these types of method calls within an advice or interceptor. This is where mixins come in. The AOP framework allows you to define a mixin class that implements the introduced interface(s). An instance of this mixin class will be allocated the first time you invoke a method of the introduced interface.

Again, let's steal from the introductions tutorial. We'll take an existing class, force it to implement the `java.io.Externalizable` interface and provide a class that implements `Externalizable`

```
public class POJO
{
  private String field;
}
```

To take this class and make it externalizable is very simple. Just the following XML is required:

```
<introduction class="POJO">
   <mixin>
      <interfaces>
         java.io.Externalizable
      </interfaces>
      <class>ExternalizableMixin</class>
      <construction>new ExternalizableMixin(this)</construction>
   </mixin>
</introduction>
```

The above XML just states that the AOP framework is to apply the `java.io.Externalizable` interface to the `POJO` class. You can have one or more interfaces specified with the `interfaces` element. These interfaces are comma delimited.

The `class` element defines the mixin class that will implement the externalizable interface and handle serialization of the `POJO` class.

The `construction` element allows you to specify Java code that will be used to initialize the mixin class when it is created. JBoss AOP will create a field within the `POJO` class that will hold the instance of the mixin. This field will be initialized with the Java code you provide in the `construction` element. The `this` pointer in the construction above pertains to the `POJO` class instance.

Finally, you need to implement the mixin class that will handle externalization.

```
public class ExternalizableMixin implements java.io.Externalizable
```

```
{
   POJO pojo;

   public ExternalizableMixin(POJO pojo)
   {
      this.pojo = pojo;
   }

   public void readExternal(ObjectInput in) throws IOException,
 ClassNotFoundException
   {
      pojo.stuff2 = in.readUTF();
   }
   public void writeExternal(ObjectOutput out) throws IOException
   {
      out.writeUTF(pojo.stuff2);
   }
}
```

2.1. Multiple Inheritance

One thing that should be noted about mixins is that they allow you to do true multiple inheritance within Java. Yes, it is not explicit multiple inheritance as you must define the inheritance mappings in XML or via an annotation binding, but it does allow you to take advantage of something C++ has had for years. Many thought leaders argue against the use of multiple inheritance, but when used correctly and purely as an abstract mixin, it can be a very useful tool in application design.

3. Aspects with APIs

The serialization and externalization examples show previously in this chapter are kinda lame. Sure, you can use introductions/mixins for multiple inheritance, or to do nice tricks like forcing an existing class to be serializable. A more compelling use case is needed to justify this particular feature. This is what this section is about.

The best use cases for introductions/mixins comes when you have an aspect that requires an API for the user to interact with. Many aspects have runtime APIs so that the application developer can interact with a particular aspect to set configuration or to gain added behavior. A good example of an aspect with an API is if we expand on the `@Oneway` example in the "Aspect-Oriented Annotations" section of this user guide. `@Oneway` allows you to tag a method as oneway and when you invoke that method it runs in the background. The problem with this example is that you can only run `void` methods in the background and cannot interact asynchronously with methods that return a value. You have no way of obtaining the return value of an asynchronous call. Let's walk through an example of taking the oneway aspect and adding a runtime API for that aspect so that application developers can obtain method return values asynchronously.

The end goal of this example is to allow an application developer to tag a method as `@Asynchronous` have the method run in the background, but to provide an API so that the developer can obtain the value of a method return asynchronously. What we'll use here is an introduction and mixin to provide an API to obtain a `java.util.concurrent.Future` instance (from JDK 5.0 concurrent package) that will allow us to get access to the asynchronous method's return value.

Using the `@Asynchronous` annotation

```
public class POJO
{
   @Asynchronous int someMethod() { ... }
}
```

This is the interface we want to introduce to any class that has a method tagged as `@Asynchronous`

```
public interface AsynchronousFacade
{
   java.util.concurrent.Future getLastFuture();
}
```

So, the user would interact with this asynchronous aspect in the following way.

```
{
   POJO pojo = new POJO();
   AsynchronousFacade facade = (AsynchronousFacade)pojo;
...
   pojo.someMethod(); // invokes in background
   Future future = facade.getLastFuture();
... do other work...
   // go back and get result. block until it returns.
   int result = (Integer)future.get();
}
```

The first thing we need to do is define the mixin that will provide `Futures`. This mixin should also have a private interface so that the asynchronous aspect has a way to set the current invocation's future after it spawns the method invocation to the background. The mixin will be very very simple. It will basically expose a `java.lang.ThreadLocal` so that the Future can be set and acquired.

```
public class AsynchMixin implements AsynchronousFacade,
 FutureProvider
{
    private ThreadLocal currentFuture = new ThreadLocal();

    public Future getLastFuture()
```

```
    {
       return (Future)currentFuture.get();
    }

    public void setFuture(Future future)
    {
       currentFuture.set(future);
    }
}
```

The `FutureProvider` is an additional interface introduction that the aspect will use to set the future when after it spawns the task in the background.

```
public interface FutureProvider
{
   public void setFuture(Future future);
}
```

Next, let's look at the aspect that will implement the asynchronous behavior. The aspect is made up of an advice that will create a `java.util.concurrent.Callable` instance so that the current method invocation will run in the background.

```
public class AsynchAspect
{
   ExecutorService executor = Executors.newCachedThreadPool();

   public Object invokeAsynch(MethodInvocation invocation) throws
 Throwable
   {
      final Invocation copy = invocation.copy();
      Future future = executor.submit( new Callable()
      {
         public Object call()
         {
            try
            {
               return copy.invokeNext();
            }
            catch (Throwable throwable)
            {
               return throwable;
            }
         }
      });
      FutureProvider provider =
 (FutureProvider)invocation.getTargetObject();
      provider.setFuture(future);

      return nullOrZero(invocation.getMethod().getReturnType());
```

```
   }

   private Object nullOrZero(Class type)
   {
      if (type.equals(long.class)) return 0;
      //... other types ...
      return null;
   }
}
```

The `invokeAsynch` advice first copies the invocation. A copy copies the entire state of the invocation objec and remembers exactly in the interceptor/advice chain to continue on when the method is spawned off into a separate thread. The copy allows the current Java call stack to return while allowing the copy to live in a separate thread and continue down the interceptor stack towards the actual method call.

After creating a callable and running the method in a separate thread, the advice gets the target object from the invocation, and typecasts it to `FutureProvider` so that it can make the future available to the app developer.

So the mixin and aspect are written. The next thing to do is to define an advice binding so that when a method is tagged as asynchronous, the asynch advice will be triggered, and the method will run in the background.

```
         <aspect class="AsynchAspect" scope="PER_VM"/>
<bind pointcut="execution(!static * *->@Asynchronous(..))">
   <advice name="invokeAsynch" aspect="AsynchAspect"/>
</bind>
```

After defining the aspect binding, we then come to the introduction definition itself. We want the introduction to be added to any class that has any method tagged as `@Asynchronous`. The JBoss AOP pointcut expression language has a keyword `has` to allow for this type of matching. Let's look at the introduction binding.

```
<introduction expr="has(!static * *->@Asynchronous(..))">
   <mixin>
      <interfaces>AsynchronousFacade, FutureProvider</interfaces>
      <class>AsynchMixin</class>
      <construction>new AsynchMixin()</construction>
   </mixin>
</introduction>
```

The example is now complete. Introductions/mixins aren't solely limited to pseudo-multiple inheritance and the asynch aspect is a great example of an aspect with a runtime API.

Dynamic AOP

1. Hot Deployment

Any joinpoint that has been aspectized by the `aopc` compiler or by a load time transformation is set up to be able to have advices/interceptors added or removed from it at runtime. This is JBoss AOP's first definition of Dynamic AOP. Using the `prepare` action allows you to aspectize any joinpoint in your application so that advices/interceptors can be applied later at runtime. The over head of such a massaging of the bytecode is very minimal as it is just an extra boolean expression. The benefits for search an architecture allow you to do things like deploy and undeploy metrics or statistic gathering on a needed basis. If you are using AOP for testing (See "Testing with AOP"), it allows you to deploy/undeploy testing aspects as you run your automated tests on your live system.

2. Per Instance AOP

JBoss AOP has the ability to apply interceptors on a per instance basis rather than having interceptors be applied entirely to the class. This is very useful when you have instances of an object that need to behave differently in different circumstances.

A perfect example of this is JBoss Cache AOP (TreeCacheAOP). It uses AOP to `prepare` classes so that field access may be intercepted. When an object is inserted into the cache, TreeCacheAOP adds field interceptors for that particular instance so that it can do automatic replication across a cluster or to automatically provide transaction properties to the object's state. When the object is removed from cache, the field interceptors are removed from that particular instance.

Integration With Pointcuts

This docbook has talked a lot about building aspects either with regular aspects, annotations, and introductions. This chapter takes a step back and doesn't talk about building aspects, but rather how you can use plain old pointcuts in your application to provide logical integration points.

1. Integration

What you've seen by reading this docbook and the "Reference Manual" on JBoss AOP is that AOP provides a rich pointcut expression language that allows you to intercept various points in the Java language. If you think about it, the pointcut language allows you to turn any point in your Java language into an event. An event that can be caught and handled by any piece of code.

After productizing and shipping an application, sometimes users want to integrate with such "events". They want to be able to hook into different places of your application so that they can trigger things specific to their particular deployment of your product. Using object-oriented techniques to provide these hooks to users would require special gluecode every time a user request like this was made. Also, as more and more of these hooks are exposed through object-orientation, it becomes harder and harder to redesign, refactor, or change APIs as the user base is tightly coupled to existing code.

This is were AOP and pointcuts come in. Instead of writing sometimes complicated gluecode, or refactoring the application to provide such user-request integration points, the application can provide pointcut expressions the user base can use to integrate their specific integration use cases. The application provides logical names to code points as pointcut expressions. The pointcut expression can change over time as the application code is redesigned and/or refactored, but the logical name of the join point/event/integration point stays the same and user hooks don't have to change either. Let's look at an example:

```
public class BankAccount
{
   public void withdraw(double amount) {...}
}
```

Let's say the user of this bank account ERP system wanted to have an email sent to the account holder whenever a successful withdraw was made. The ERP system could provide the hook as a pointcut and then the user can write an email aspect that binds with this pointcut.

```
<pointcut name="WITHDRAW" expr="execution(public void
 BankAccount->withdraw(double))"/>
```

The `BankAccount` class would be instrumented with AOP hooks. The overhead is quite tiny as only an additional boolean expression is needed to instrument this kind of hook. If the class or method name changes, the user integration code is unaffected as they bind their email hook to the logical pointcut name.

JBoss currently provides integration points in its EJB and MBean layers in such the same way. Recently, BEA Weblogic published AspectJ style pointcuts into the Weblogic runtime so that users could integrate using AspectJ. As AOP becomes more popular you'll see more and more software products offering integration points through pointcut expressions.

Testing with AOP

In the previous sections we talked more about using AOP to build and design applications and services. This chapter focuses on how you can use AOP to test your applications.

1. Testing Exception Handling

The sign of a well design application is how gracefully it can handle errors. To be able to handle errors gracefully in all circumstances though requires lots and lots of testing. You have to test that your application is catching and handling exceptions carefully. Sometimes its hard to produce error conditions because your code is interacting with a third party library, or third party service like a database or something. You can write complex mock objects in these scenarios, but let's see how you can create these error conditions in an easier and more flexible way with JBoss AOP.

The example scenario we'll give is an application that needs to be tested on whether or not it handles an Oracle database deadlock exception gracefully. What we'll do is write an advice that intercepts calls to `java.sql.Statement` execute methods and always throw a SQLException with the appropriate deadlock error code.

```
public class SQLDeadlockExceptionInjector
{
   public Object throwDeadlock(Invocation invocation) throws
 Throwable
   {
      throw new SQLException("Oracle Deadlock", "RETRY",
 ORACLE_DEADLOCK_CODE);
   }
}
```

What's great about the JBoss AOP approach to testing exception handling is that you can use it on a live system and change how your tests run by deploying and/or undeploying certain aspects at runtime during your automatic testing phase. Let's apply this aspect.

```
<aspect class="SQLDeadlockExceptionInjector/>
<bind pointcut="call(*
 $instanceof{java.sql.Statement}->execute*(..))">
   <advice name="throwDeadlock"
 aspect="SQLDeadlockExceptionInjector"/>
</bind>
```

So, the above binding will throw a deadlock exception every time an execute method of a Statement is invoked. This example is a bit limited though. Maybe not all

code paths can handle deadlock exceptions, or they should not handle deadlock exceptions and just rollback and such. The pointcut expression language allows you to do more fine grain application of this particular exception aspect. Let's say that only our BankAccount class is designed to successfully recover from a Oracle deadlock exception. We can change the pointcut expression to be as follows:

```
<bind pointcut="call(*
 $instanceof{java.sql.Statement}->execute*(..)) AND /

 within(BankAccount)">
   <advice name="throwDeadlock"
 aspect="SQLDeadlockExceptionInjector"/>
</bind>
```

The difference in this expression is the `within`. It is saying that any call to execute methods that are within the BankAccount class. We can even get more fine grained than that. We can even specify which methods within BankAccount the exception aspect should be applied to.

```
<bind pointcut="call(*
 $instanceof{java.sql.Statement}->execute*(..)) AND /
                                                  withincode(void
 BankAccount->withdraw(double))">
   <advice name="throwDeadlock"
 aspect="SQLDeadlockExceptionInjector"/>
</bind>
```

In the above listing the `withincode` keyword specificies to match the calling of any execute method that is invoked within the `BankAccount.withdraw()` method.

AOP gives you a lot of flexibilty in testing error conditions, JBoss AOP in particular. Because JBoss AOP allows you to hotdeploy (deploy/undeploy) aspects at runtime it is very easy to integrate these types of tests into a live system instead of having to go through the pain of writing complex mock objects and running your applications outside of the application server environment.

2. Injecting Mock Objects

This section was taken from Staale Pedersen's article at `http://folk.uio.no/staalep/aop/testing.html`. Thanks Staale for putting together some ideas on how you can use JBoss AOP for use in unit testing.

The use of unit testing has increased tremendously lately, and many developers have seen the increase in quality and speed that comes from having a comprehensive unit-test suite. As the use of unit testing has increased, so have the number of situations where writing test are troublesome or maybe impossible. A

common problem with writing tests is that it can require large amount of setup code. Testing code that rely on a remote system or data access from file/db/net can be almost impossible to implement. But with the help of JBoss AOP and mock objects this is no longer any problem.

In this example we will examine a common situation where writing unit tests is difficult, but desirable. For simplicity we will use POJO's, but the example can easily be translated for a large J2EE application.

2.1. Required Knowledge

This article focuses on unit testing with JUnit using Mock Maker and of course JBoss AOP. Knowledge of JUnit and JBoss AOP is required, Mock Maker is used, but thoroughly knowledge is not required. The example source code is compiled with Ant, env JUNIT_HOME must be set (mock maker and JBoss AOP jars are included in the example source).

2.2. The Problem

The situation is common, we have a Bank application that manages Customers which can have one or more BankAccounts. The Bank has different business methods to calculate interest, accept loans, etc. (in production code this would be large and complex methods.) We want to write tests for the Bank's business methods to make sure they work as intended and that we don't introduce bugs if we refactor, extend, modify etc. The Bank has three business methods.

```
package bank;

import java.util.ArrayList;

import customer.*;

public class BankBusiness {

  private BankAccountDAO bankAccountDAO;

  public BankBusiness() {
    try {
      bankAccountDAO =
 BankAccountDAOFactory.getBankAccountDAOSerializer();
    }
    catch(Exception e) {
      System.out.println(e.getMessage());
    }
  }

  public boolean creditCheck(Customer c, double amount) {
    return (getSumOfAllAccounts(c) < amount * 0.4);
```

```
  }

  public double calculateInterest(BankAccount account) {
    if(account.getBalance() < 1000)
      return 0.01;
    else if(account.getBalance() < 10000)
      return 0.02;
    else if(account.getBalance() < 100000)
      return 0.03;
    else if(account.getBalance() < 1000000)
      return 0.05;
    else
      return 0.06;
  }

  public double getSumOfAllAccounts(Customer c) {
    double sum = 0;
    if(c.getAccounts().size() < 1)
      return sum;
    else {
      for(int i=0; i < c.getAccounts().size(); i++) {
        BankAccount a =
          bankAccountDAO.getBankAccount( ((Long)
 c.getAccounts().get(i)).longValue());
        if(a != null)
          sum += a.getBalance();
      }
    }
    return sum;
  }

}
```

`calculateInterest(BankAccount b)` can easily be tested since it is only dependent on the object it recieves as a parameter.

`creditCheck(Customer c, double amount)` and `getSumOfAllAccounts(Customer)` are more complicated since they are data dependent. It uses a DAO layer to fetch all BankAccounts for a specified customer. The test should not be dependent of the DAO implementation since its goal is to check the business logic, not the DAO layer. In this example the DAO implementation is a simple serializer, but it could easily be an Entity beans, Hibernate, etc..

2.3. Mock Objects

Mock objects are objects that implement no logic of their own and are used to replace the parts of the system with which the unit test interacts. In our case it is the

DAO layer we would like to mock. We could write our own mock implementation, but mock maker does a very good job of autogenerating the mock for us.

```
package bank;
/**
 * @mock
 */
public interface BankAccountDAO {

  public void saveBankAccount(BankAccount b) throws Exception;

  public BankAccount getBankAccount(long a);

  public void removeBankAccount(BankAccount b) throws Exception;
}
```

With the `@mock` tag in the header mock maker generates the mock. In the example the ant target `ant generate-mocks` generates the mock implementation of BankAccount. Now we need to replace the DAO call to return our mock objects instead of the DAO implementation.

2.4. AOP with Mocks

- and intercepting a method invocation is just what aop does best. Our jboss-aop.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
   <bind pointcut="execution(public static bank.BankAccountDAO
 bank.BankAccountDAOFactory->getBankAccountDAO*())">
       <interceptor class="bank.BankAccountDAOInterceptor"/>
   </bind>
</aop>
```

The pointcut expression intercepts the factorycall bank.BankAccountDAOFactory.getBankAccountDAO*() and calls the interceptor bank.BankAccountDAOInterceptor.

```
package bank;

import org.jboss.aop.joinpoint.Invocation;
import org.jboss.aop.joinpoint.MethodInvocation;
import org.jboss.aop.advice.Interceptor;

import util.MockService;

public class BankAccountDAOInterceptor implements Interceptor {
```

```
   public String getName() { return "BankAccountDAOInterceptor"; }

   public Object invoke(Invocation invocation) throws Throwable {
     try {
       MockService mockService = MockService.getInstance();

       Object mock = mockService.getMockForInterface(
  "BankAccountDAO");
       if(mock == null) {
         System.out.println("ERROR: BankAccountDAOInterceptor didnt
  find class!");
         // this will probably fail, but its the sainest thing to do
         return  invocation.invokeNext();
       }

       return mock;
     }
     finally {
     }
   }
}
```

Instead of returning `invocation.invokeNext()`, we ignore the invocation stack since we want to replace the invocation call with a mock implementation. The interceptor receives the invocation and get an instance of the singleton MockService. The use of MockService may not be clear, but we want the test to instanciate the mock objects. That way, the test can easily modify the input to the methods we want to test. The test creates an object of the mock and put it into the MockService with the interface name as the key. The Interceptor then tries to get the mock from MockService and return it.

```
package util;

import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class MockService {

    private static MockService instance = new MockService();

    private Map mockReferences = null;

    protected MockService() {
        mockReferences = new Hashtable();
    }
```

```
    public static MockService getInstance() {
        return instance;
    }

    public void addMock(String c, Object mock) {
        mockReferences.put(c, mock);
    }

    public Object getMockForInterface(String myKey) {
      Set keys = mockReferences.keySet();

      for (Iterator iter = keys.iterator(); iter.hasNext();) {
        String key = (String) iter.next();
        if(myKey.equals(key)) {
          return mockReferences.get(key);
        }
      }
      return null;
    }

}
```

Everyting is now in place to write the test. Note that much of this setup code is written once and it will be reused by all similar tests. Then the test: BankBusinessTestCase

```
package bank;

import junit.framework.TestCase;

import customer.Customer;
import util.MockService;

public class BankBusinessTestCase extends TestCase {

  private MockBankAccountDAO mock;
  private Customer customer;

  public BankBusinessTestCase(String name) {
    super( name);
  }

  public void setUp() {
    mock = new MockBankAccountDAO();

    BankAccount account = new BankAccount( 10);
    account.setBalance( 100);
    BankAccount account2 = new BankAccount( 11);
    account2.setBalance( 500);
```

```
    mock.setupGetBankAccount( account);
    mock.setupGetBankAccount( account2);

    MockService mockService = MockService.getInstance();
    mockService.addMock( "BankAccountDAO", mock);

    customer = new Customer("John", "Doe");
    customer.addAccount( new Long(10));
    customer.addAccount( new Long(11));
  }

  public void testSumOfAllAccounts() {
    BankBusiness business = new BankBusiness();
    double sum = business.getSumOfAllAccounts( customer);
    assertEquals((double) 600, sum);
    System.out.println("SUM: "+sum);
  }
}
```

To compile and run the test we call ant compile test. Output from the test:

```
test-bankbusiness:
    [junit] .SUM: 600.0
    [junit] Time: 0,23
    [junit] OK (1 test)
```

The testresult was exactly what we expected.

With the the use of AOP we can test every aspect of our code. This example show the limits of object-oriented programming (OOP) compared to AOP. It must be pointed out that it is possible to write these tests without AOP, but it would require to edit production code just to make the tests pass.

The approach in this example can easily be used to mock SessionBeans instead of a DAO layer. Theoretically, we can test all of the business methods in a large J2EE application outside the container. This would greatly increase quality and speed during software development.

JBoss AOP IDE

1. The AOP IDE

JBoss AOP comes with an Eclipse plugin that helps you define interceptors to an eclipse project via a GUI, and to run the application from within Eclipse. This is a new project, and expect the feature set to grow quickly!

2. Installing

You install the JBoss AOP IDE in the same way as any other Eclipse plugin.

- Make sure you have Eclipse 3.0.x installed, and start it up.
- Select Help > Software Updates > Find and Install in the Eclipse workbench.
- In the wizard that opens, click on the "Search for new features to install" radio button, and click Next.
- On the next page you will need to add a new update site for JBossIDE. Click the "New Remote Site.." button.
- Type in "JBossIDE" for the name, and "http://jboss.sourceforge.net/jbosside/updates" for the URL, and click OK.
- You should see a new site in the list now called JBossIDE. click the "+" sign next to it to show the platforms available.
- Now, depending if you just want to install the AOP IDE (if you don't know what JBoss-IDE is, go for this set of options):
  - Check the "JBoss-IDE AOP Standalone" checkbox.
  - In the feature list you should check the "JBoss-IDE AOP Standalone 1.0" checkbox.

  If you have JBoss-IDE installed, or want to use all the other (non-AOP) features of JBoss-IDE:

  - If you don't have JBossIDE installed, check the "JBoss-IDE 1.4/Eclipse 3.0" checkbox.
  - Check the "JBoss-IDE AOP Extension" checkbox.
  - In the feature list you should check the "JBoss-IDE AOP Extension 1.0" checkbox, and the JBoss-IDE (1.4.0) checkbox if you don't have JBossIDE installed.

- At this point you should only need to accept the license agreement(s) and wait for the install process to finish.

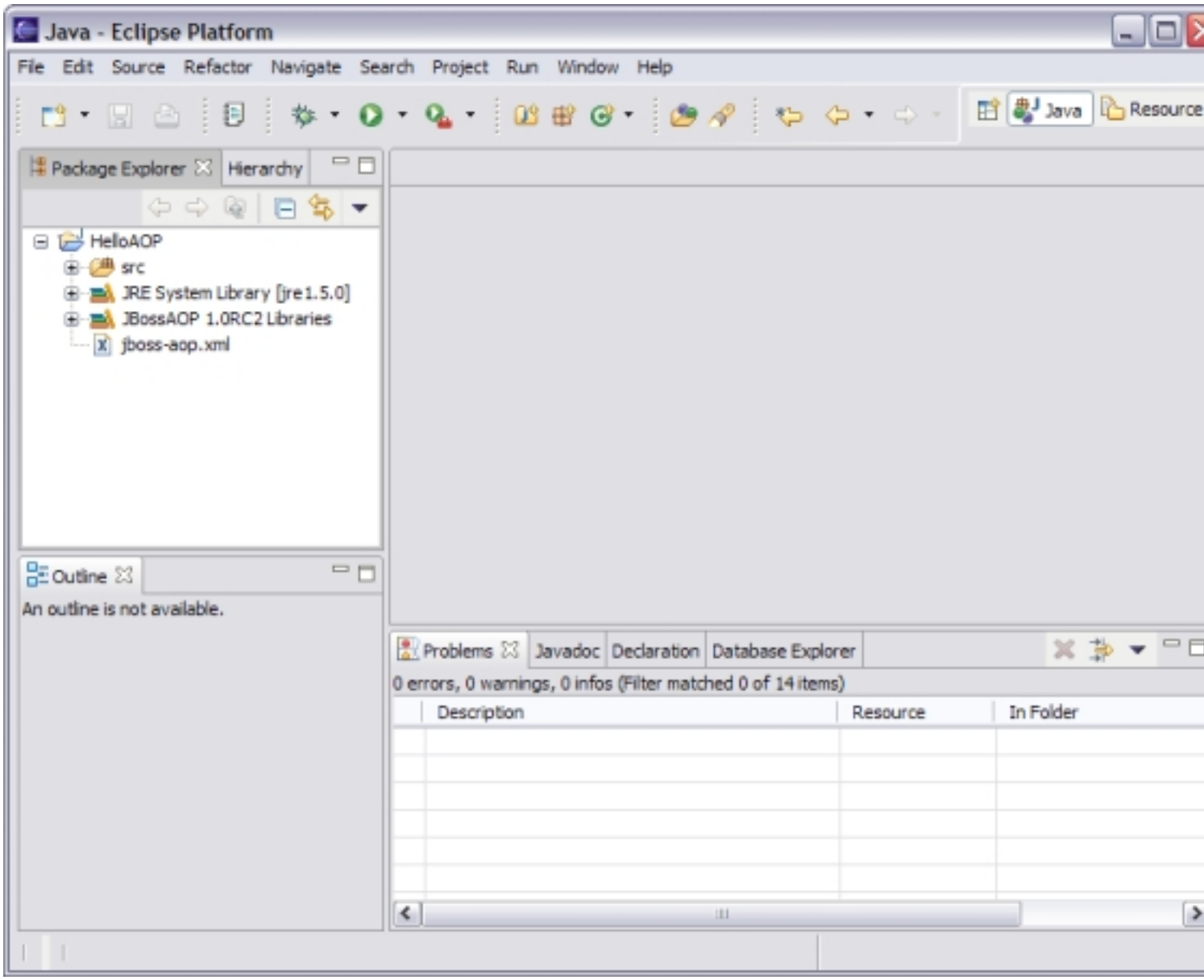
3. Tutorial

This tutorial is meant to guide you through creating a new AOP project in eclipse using the AOP extension to JBossIDE. It assumes that you have some working knowledge of AOP, and Java.. and possibly some minimal experience dealing with eclipse as well.

3.1. Create Project

- From eclipse's main menu, you can click on the File Menu, and under it, New > Project...
- Double click on JBoss AOP Project under the JBossAOP folder
- In the Project Name text box, let's enter `HelloAOP`.
- `Use Default` should be fine for the project location. (If you want to use an external location, make sure there are no spaces in the path.)
- Click `Finish`

At this point, your eclipse workbench should look something like this:



3.2. Create Class

Next step is to create a normal Java class.

- Right click on the "src" directory in the Package Explorer and in the menu, click New > Class.
- The only thing you should need to change is the Name of the class. Enter `HelloAOP` without quotes into the Name textbox, and click `Finish`

Modify the code for your class so it loks like

```
public class HelloAOP {
```

```
  public void callMe ()
  {
   System.out.println("AOP!");
  }

  public static void main (String args[])
  {
   new HelloAOP().callMe();
  }
 }
```

3.3. Create Interceptor

Next we want to create an interceptor to the class.

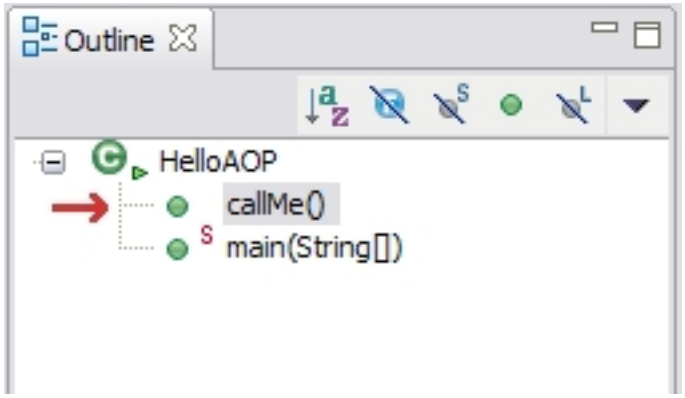
- Right click on the "src" directory in the Package Explorer and in the menu, click New > Class. In the resulting dialog:
  - Name the class `HelloAOPInterceptor`
  - Add `org.jboss.aop.advice.Interceptor` to the list of interceptors.

Then modify the class so it looks like:

```
    import org.jboss.aop.advice.Interceptor;
    import org.jboss.aop.joinpoint.Invocation;

    public class HelloAOPInterceptor implements Interceptor {

     public String getName() {
      return "HelloAOPInterceptor";
     }

       //We renamed the arg0 parameter to invocation
     public Object invoke(Invocation invocation) throws Throwable {
      System.out.print("Hello, ");
          //Here we invoke the next in the chain
      return invocation.invokeNext();
     }
    }
```

3.4. Applying the Interceptor

In order to apply your Interceptor to the callMe() method, we'll first need to switch back to the `HelloAOP.java` editor. Once the editor is active, you should be able to see the callMe() method in the Outline view (If you cannot see the outline view, go to Window > Show View > Outline).



Right click on this method, and click JBoss AOP > Apply Interceptor(s)... A dialog should open, with a list of available Interceptors. Click on `HelloAOPInterceptor`, and click `Finish`.

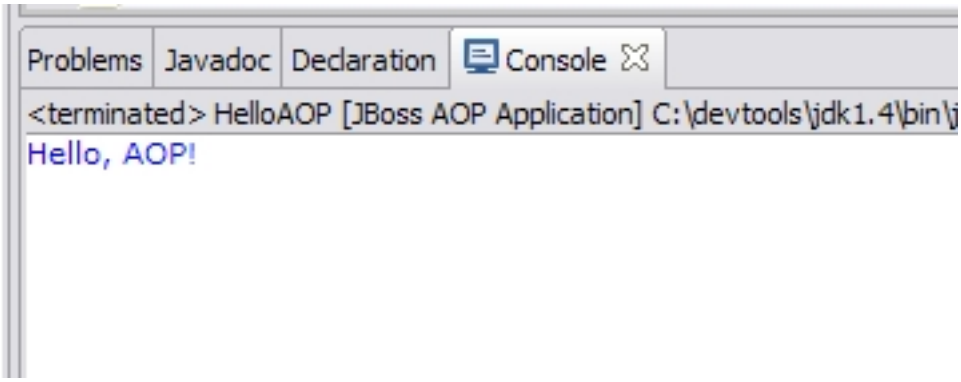
You should see in your Package Explorer that the file "jboss-aop.xml" now exists under your project root.

3.5. Running

Now all that's left is running the application! Similar to running a normal Java Application from Eclipse, you must create a Run Configuration for your project.

- From the Run menu of eclipse, and choose "Run..."
- In the dialog that opens, you should see a few choices in a list on the left. Double click on "JBoss AOP Application".
- Once it is finished loading, you should have a new Run Configuration under JBoss AOP Application called "Hello AOP".
- Click the "Run" button

The Eclipse console should now say: `Hello, AOP!`, where the `Hello,` bit has been added by the interceptor.

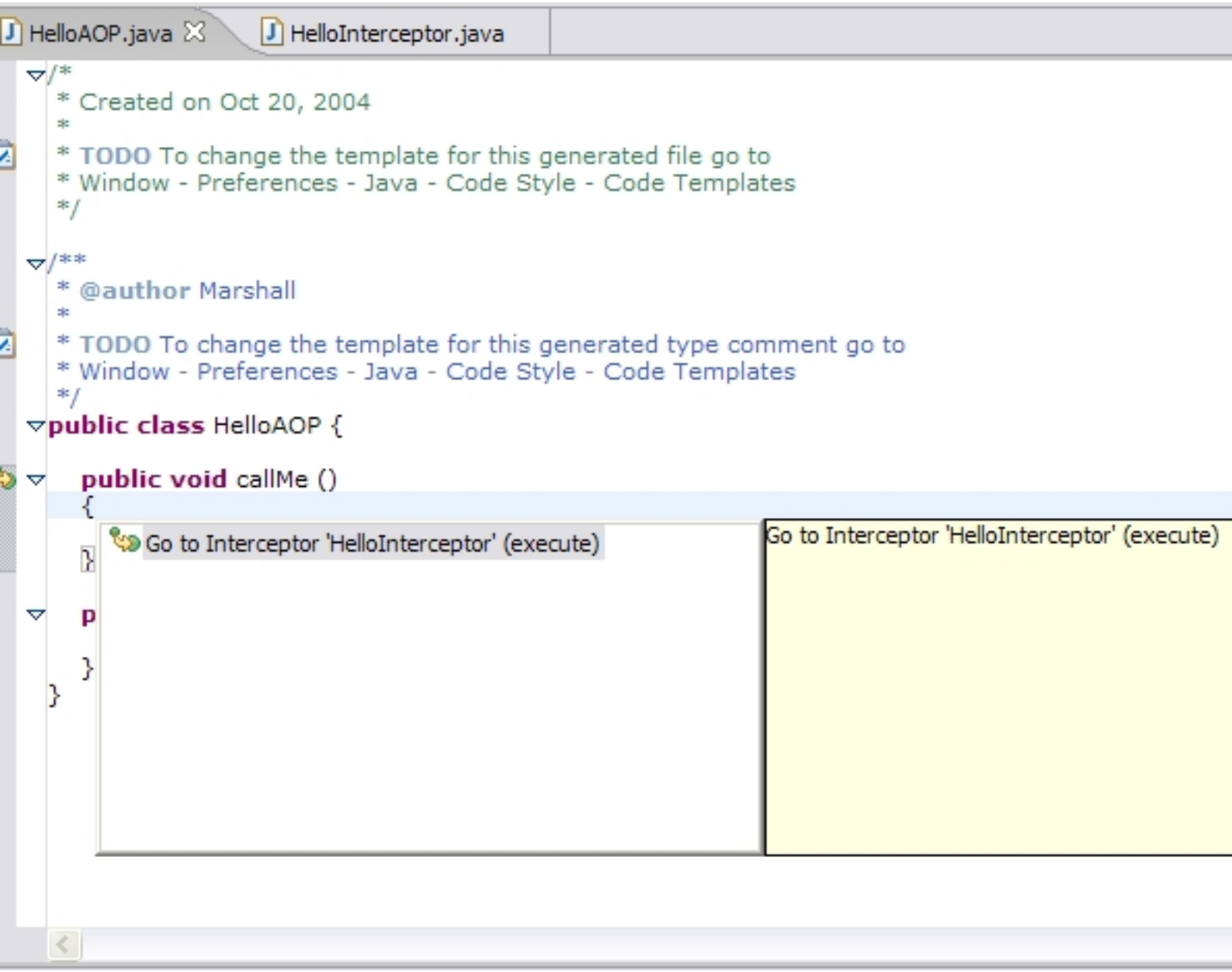


3.6. Navigation

In the real world, when developing AOP application across a development team, you can expect it will be hard to understand when and where aspects are applied in your codebase. JBoss-IDE/AOP has a few different strategies for notifying developers when an aspect is applied to a certain part of code.

3.6.1. Advised Markers

A marker in eclipse is a small icon that appears on the left side of the editor. Most developers are familiar with the Java Error and Bookmark markers. The AOP IDE provides markers for methods and fields which are intercepted. To further facilitate this marking, anytime the developer presses Ctrl + 1 (the default key combination for the Eclipse Quick Fix functionality)), a list of interceptors and advice will be given for that method or field. This makes navigation between methods and their interceptors extremeley easy!

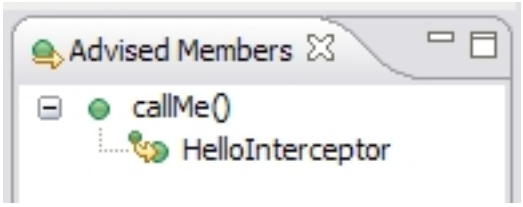


3.6.2. The Advised Members View

The Advised Members view gives the developer an overview of every single method and field in the current class that is advised by an Aspect or Interceptor. Let's have a look.

- From the Eclipse main menu, click on Window > Show View > Other...
- In the window that opens, you should see a folder called "JBoss AOP". Press the "+" to expand it.
- Double click on "Advised Members"

Once you've done this, you should now make sure you are currently editing the `HelloAOP` class we created in the last tutorial. Once you have that class open in an editor, you should see something similar to this in the Advised Members view:



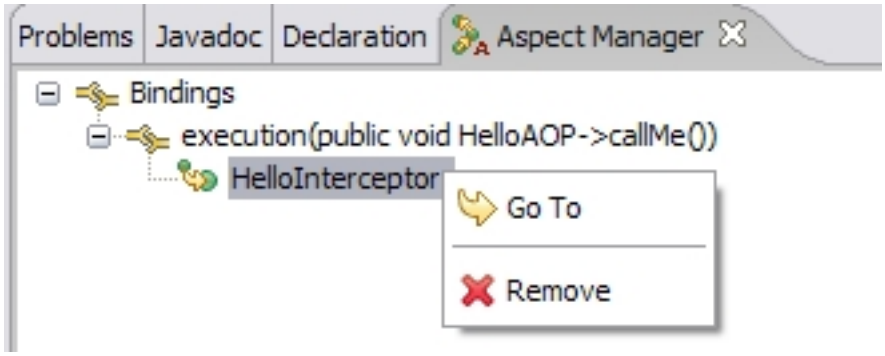
Here we see that the method "callMe()" is intercepted by the interceptor `HelloInterceptor`. Double clicking on `HelloInterceptor` will take you straight to it. This view is similar to the Outline view, except it only shows members in your class which are intercepted.

3.6.3. The Aspect Manager View

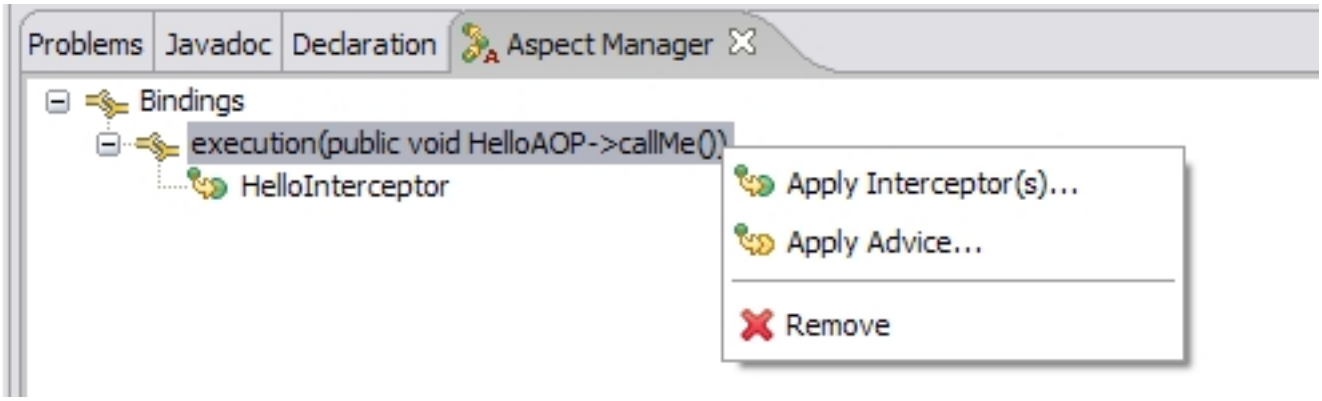
The Aspect Manager View is a graphical representation of the AOP descriptor file (jboss-aop.xml). It allows you to remove an Interceptor or advice from a pointcut, as well as apply new Interceptors and Advice to existing pointcuts.

- From the Eclipse main menu, click on Window > Show View > Other...
- In the window that opens, you should see a folder called "JBoss AOP". Press the "+" to expand it.
- Double click on "Aspect Manager"

Under Bindings, you'll notice that a pointcut is already defined that matches our "callMe()" method, and our `HelloInterceptor` is directly under it. Right Click on `HelloInterceptor` will provide you with this menu:



You can remove the interceptor, or jump to it directly in code. If you right click on the binding (pointcut) itself, you'll be able to apply more interceptors and advice just like when right clicking on a field or method in the outline view. You can also remove the entire binding altogether (which subsequently removes all child interceptors and advice, be warned)


Problems
Javadoc
Declaration
Aspect Manager
Bindings
execution(public void HelloAOP->callMe())
HelloInterceptor
Apply Interceptor(s)...
Apply Advice...
Remove