

# JBOSS IDE: Aop Plugin

## Features of JBoss IDE's Aop Plug-in

1.0

---

# Table of Contents

1. Introduction .....	1
1.1. Document Purpose .....	1
1.2. What is JBoss AOP? .....	1
1.3. How can the IDE help me? .....	1
2. Getting Started .....	2
2.1. Installation .....	2
2.2. Creating a new AOP Project .....	2
2.3. Exploring the Views .....	3
2.3.1. Project Structure .....	3
2.3.2. Aspect Manager View .....	3
2.3.3. The Advised Members View .....	4
3. Creating Aop Elements .....	5
3.1. Setup .....	5
3.2. Creating a named pointcut .....	5
3.3. Binding the Pointcut .....	7
3.4. Creating Typedefs .....	9
3.5. Creating Introductions .....	9
3.5.1. Setup .....	9
3.5.2. Creating the Introduction .....	9
4. The Aop Editor .....	12
4.1. Setup .....	12
4.2. Type matchers .....	12
4.3. Advised Members Markers .....	13
4.4. Adding / deleting methods .....	13
4.5. Changing Annotations .....	14
4.5.1. Making it match .....	15
5. Running your AOP app .....	16

---

# 1

## Introduction

### 1.1. Document Purpose

The purpose of this document is to outline how the JBoss Aop IDE eclipse plug-in can help increase productivity when writing Aop applications with eclipse. A workable knowledge of the basics of AOP generally, and JBoss AOP specifically, is assumed.

### 1.2. What is JBoss AOP?

The project page for JBoss Aop is available at <http://www.jboss.org/products/aop>, along with much documentation.

### 1.3. How can the IDE help me?

The AOP IDE attempts to make creating, building, and testing aop applications much easier. It provides most of its assistance visually, through both of its views (Aspect Manager View, and Advised Members View), as well as markers in editors. An assortment of dialogs and actions attempt to make creating type expressions or method expressions easier as well.

The plug-in also provides an incremental builder so that testing your application can proceed as painlessly as possible.

---

# 2

## Getting Started

### 2.1. Installation

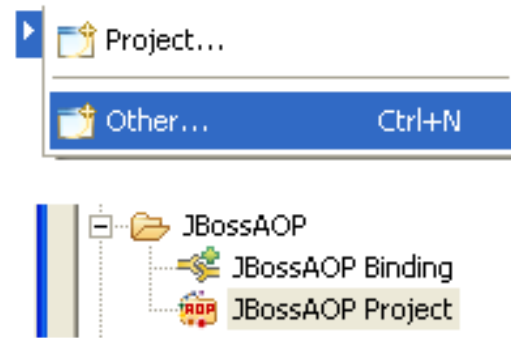
You install the JBoss AOP IDE in the same way as any other Eclipse plugin.

- Make sure you have Eclipse 3.1.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):
  - If you have JBoss-IDE installed, or want to use all the other (non-AOP) features of JBoss-IDE:
    - Check the "JBoss-IDE AOP Standalone" checkbox.
    - In the feature list you should check the "JBoss-IDE AOP Standalone 1.0" checkbox.
    - 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.

### 2.2. Creating a new AOP Project

The first step is to create your project. Select `file->new->other...`

From the wizard that opens, expand the JBoss AOP group, and select `JBossAOP Project`. From there, the remainder of the project creation process is the same as for any generic Java project.



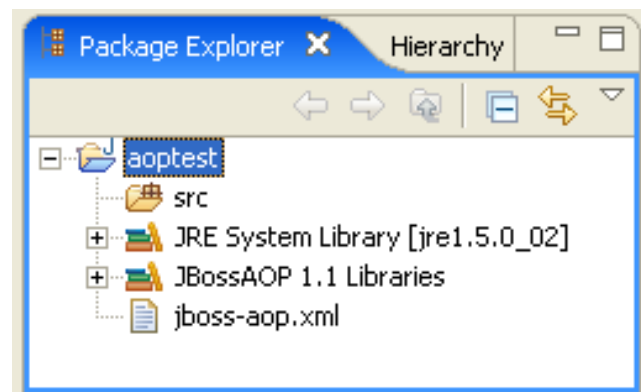
## 2.3. Exploring the Views

The first thing to do is open the two views that the AOP plugin has available for you. To do this, select `window->Show View->Other...` and expand the JBoss AOP category. Select both views and click ok.



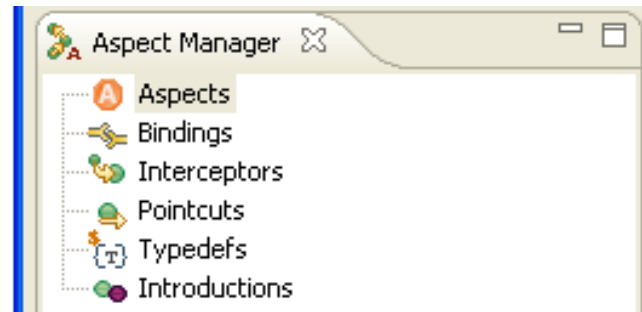
### 2.3.1. Project Structure

The Aop Project creation wizard has done a few things already for you. First, it has included all of the JBoss AOP jar files. Secondly, it has created a descriptor file for you, which will store most of the aop functionality such as pointcut expressions, typedefs, introductions, and all other things available to you via the descriptor file.



### 2.3.2. Aspect Manager View

The Aspect Manager View is the birds'-eye view of the model that underlies the entire aop project. For the most part, all of its data is gotten from the xml descriptor directly, and not from any other model elements.

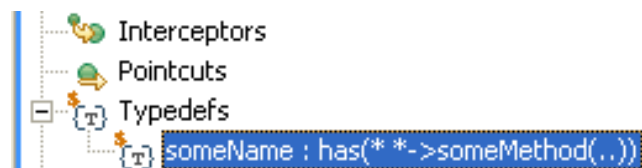


Currently, not all elements represented in this view have actions associated with them to assist in creating or developing those elements. However, if you modify the descriptor file directly, the view will be updated.

The Aspect Manager View represents its data in a tree structure, so when pointcuts, bindings, and other elements are found, they will be displayed. Currently, if you right-click on several elements (Bindings, Pointcuts, Typedefs, Introductions), a menu item will appear offering to assist in the creation of those elements. These will be explored more later on.

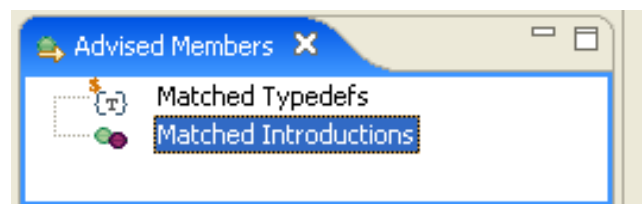
This view is updated automatically when the descriptor file is changed. If we were to change the descriptor xml file to contain the string below, we would see a change in the Aspect Manager view.

```
<aop>
  <typedef expr="has(* *->someMethod(..)"
    name="someName" />
</aop>
```



### 2.3.3. The Advised Members View

The Advised Members view shows data not directly available through the descriptor file, but rather represents how the descriptor data will be interpreted when your application is built and run.



Currently in the view, we see a category for Typedefs, and a category for Introductions. More data, however, will be visible when such data is available. Whenever you switch from one file in your project to another, this view will automatically update, and show which member elements inside that java file are advised by some pointcut, as well as what interceptors are currently advising it.

Underneath the Typedefs and Introductions categories will be a list of which expressions of those types match the currently active java file's main type (primary class). This will be demonstrated later.

## Creating Aop Elements

### 3.1. Setup

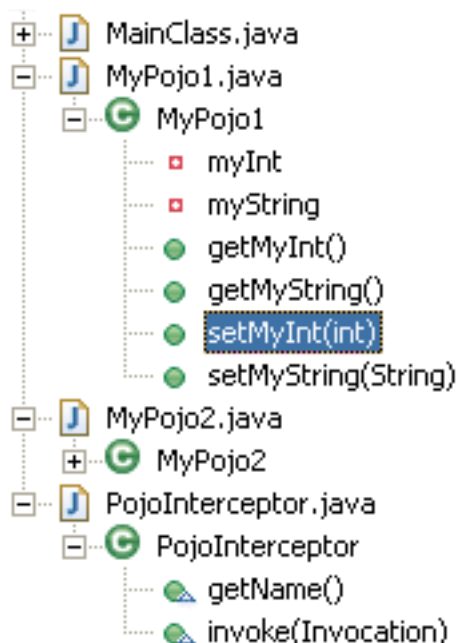
For the setup of the following examples, I've created two different POJO's.

The first, `MyPojo1`, has two member variables: `myString` and `myInt`, along with getters and setters for both.

The second POJO class, `MyPojo2`, contains only one member variable, `myString`, along with both a getter and a setter.

I have also created a simple Interceptor called `PojoInterceptor`, which implements the `Interceptor` interface (two methods: `getName`, and `invoke(Invocation arg0)`).

When expanded, the package explorer view should resemble the image to the right.



### 3.2. Creating a named pointcut

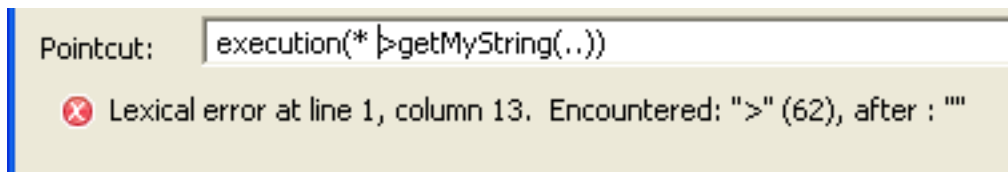
A pointcut expression is an expression that is used to refer to one or many joinpoints distributed throughout the classes in your Java AOP Project. The expression language is defined in the AOP tutorials and other resources and includes support for wildcards which can help you create an expression that matches a specific group of classes.

To turn one specific java field or method into a named pointcut, you can right-click on the java element in either the `Outline` or `Package Explorer` views, and select `JBoss AOP->Convert to Pointcut...`. There, you'll be able to name it.

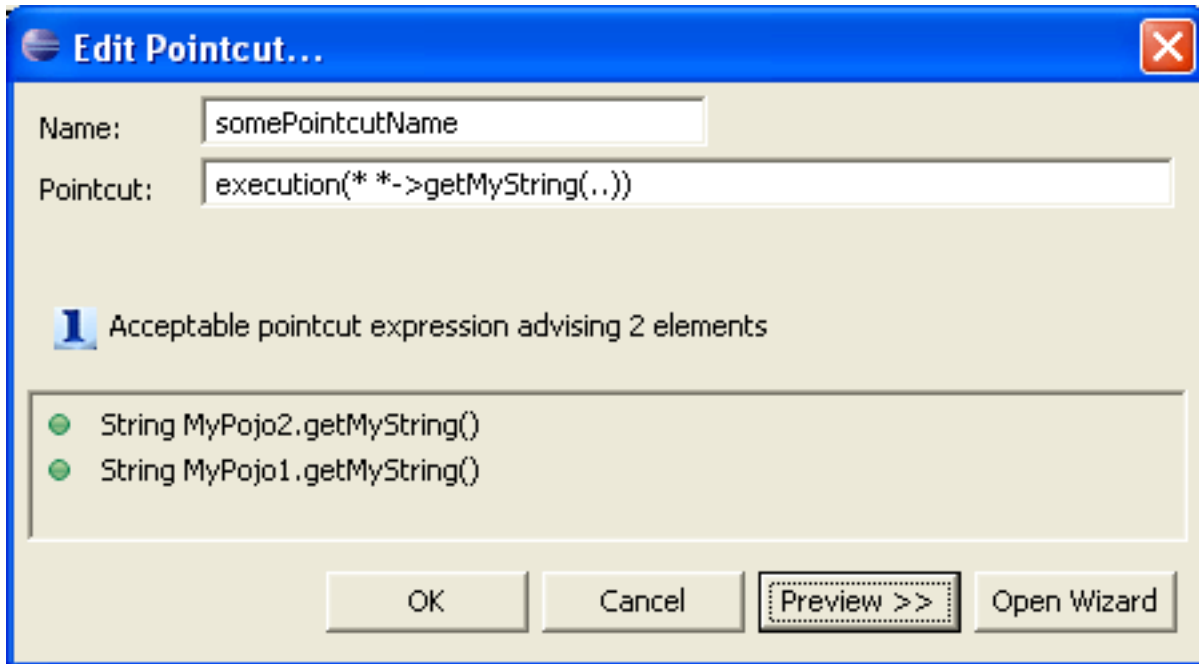
To create a named pointcut with a more generic wizard, you should right-click on the `Pointcut` element in the `Aspect Manager View`. Then, select `Create Pointcut`.

In the resulting dialog box, if you fill in name with some string value, and expression with `execution(*->getMyString(..)`, and then click the preview box, the dialog should display back to you which classes and which methods that expression matches.

Directly below, as you type, and if your expression is not syntactically correct, an error message will be displayed to try to point out where the error is.



If the syntax is correct, however, the preview button becomes enabled and the dialog box will present the matching java elements.



Should the user need more assistance in creating a pointcut expression, they can click on the `wizard` button for assistance. The wizard will help you create new expressions from scratch, but is not currently able to modify an expression that is already created.

The wizard consists of a scrolling composite, which can have the number of rows changed via two buttons. The pulldown for each row consists of every acceptable expression type, in this case for pointcuts. Method expressions, Type expressions, Field expressions, and others are all acceptable to create pointcut expressions. More documentation on this topic can be found on the JBoss AOP main site.

Also included in the pulldown is the possibility to reference `OTHER` named pointcuts.

*What is NOT included in the wizard* is the ability to add parenthesis or the negation symbol (!). However, these can be manually added to the expression afterwards in the main expression textbox.

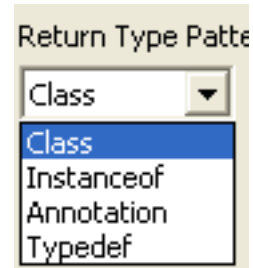
From the pulldown, you can select a joinpoint that references a type, a field, a method or a constructor. Once you've chosen from the pulldown, you can click `Modify` to complete the remaining pieces of the expression.

Once modifying your expression's details, you'll be faced with a large number of controls. Most of the text fields provide content assistance and completion (by pressing `ctrl+i` to invoke it.) If modifying a method expression, which is the one with the most controls, you'll find sections for designating information about a matching element's

return type, containing class type, a pattern for the method's name, as well as areas to designate parameters and exceptions.

*NOTE: Many of the fields provide content assistance. Pressing `ctrl+space` will invoke this feature*

Return type and Class type both use a generic Type composite, and its pulldown looks like the one to the right, offering class, instanceof, annotation, or typedef as options. (Typedefs are named expressions that represent a class type, not a method or field.) The parameters composite also has these four options, but it accepts several if the user requires.



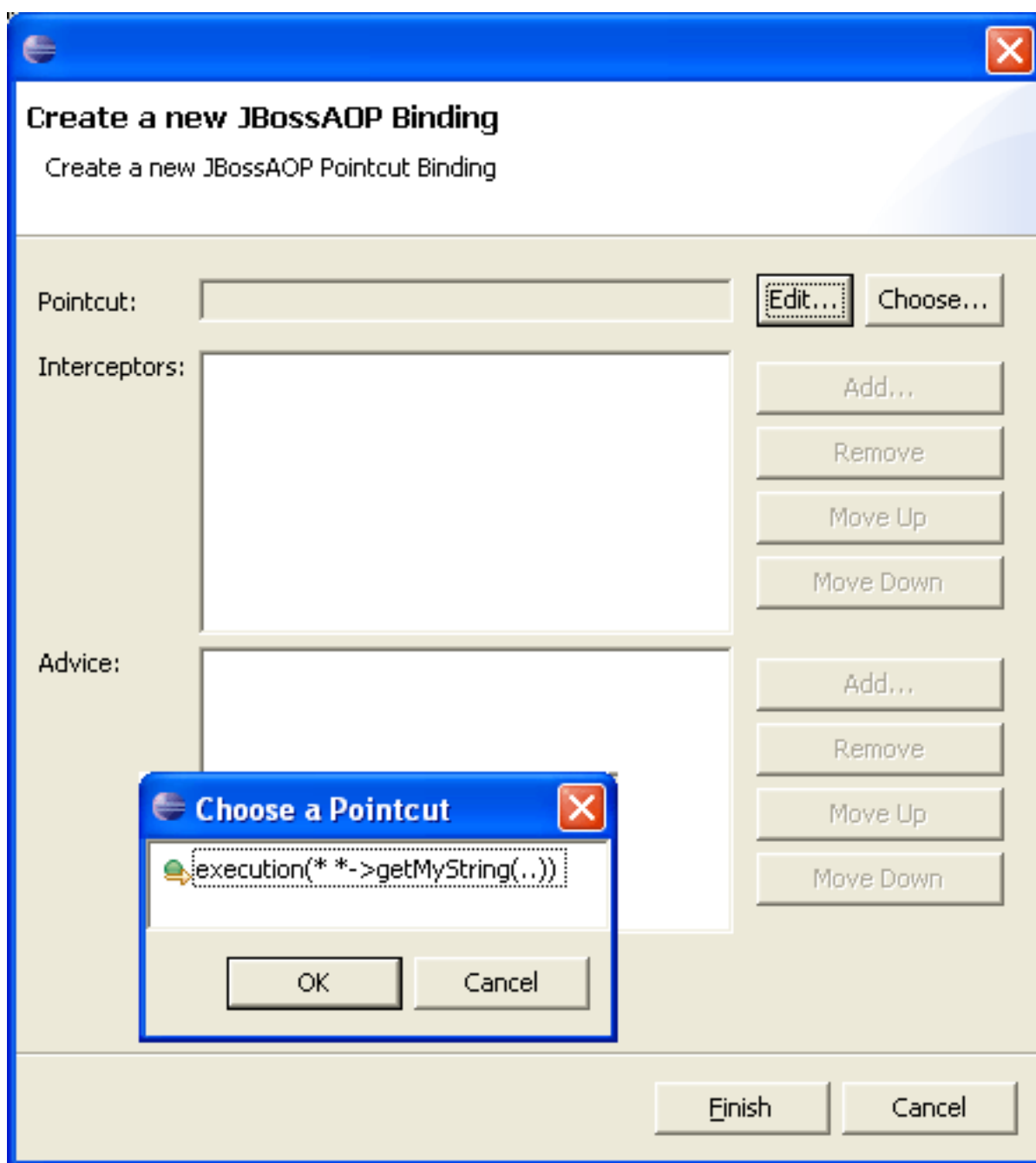
Upon selecting `typedefs` from the pulldown, the textbox at the right will turn into another dropdown listing all declared typedefs. If none are declared, the list will be empty.

The sections for parameters and exceptions is currently implemented in a non-intuitive way, and bears a bit of explaining. For a matching element to have ANY parameters, in the provided text box, type `..` and click `add`. For the matching element to have NO parameters, leave the section blank. The exceptions are implemented differently, and leaving it blank will allow the method to throw exceptions of ANY type. Adding exception types to this box only requires the method to have the specified exceptions, not contain ONLY those elements.

I welcome you to test out this wizard and use the preview button often to make sure you are getting the desired results.

### 3.3. Binding the Pointcut

We now have a named pointcut, but it does not perform any action. It only references some group of methods, fields, or types. The next step is to bind this expression along with some combination of interceptors or advice. To begin, we right-click on the `Bindings` item in the `Aspect Manager View`, and select `Create new Binding`



Creating a Binding

**Figure 3.1.**

When the dialog appears, you can do one of two things. If you already have a pointcut you'd like to use, you can click on the `Choose...` button and select your pointcut from the resultant list of pointcuts you have already defined. If you'd like to craft your own again, you can select `edit`, which will display the pointcut creation dialog as before with the single exception that the pointcut will not be named.

Once a pointcut is created or selected, you can add interceptors or aspect advice to the lists, and apply those elements to your pointcut. We can select `add` for the interceptors, and the interceptor we defined should be at the top of the list. You are welcome, of course, to use from any of the provided interceptors from the JBoss AOP Aspect

Library.

Once you accept this binding and click `OK`, the binding will be added to the xml descriptor file, and the model will be updated. Markers will be added to matching methods, fields, and types. These markers will be described in a later chapter.

## 3.4. Creating Typedefs

Creating Typedefs closely resembles the process for creating named pointcut expressions. Every typedef must be named. They cannot refer to methods or fields; only classes.

To create a named typedef, you should right-click on the `Typedefs` element in the `Aspect Manager View`. Then, select `Create Typedef`

The resulting dialog should look very familiar. The only difference will be that the pulldowns will contain fewer types of expressions and will not have the option of selecting a named pointcut.

The dialog works the same way as the last, with both the wizard and preview boxes. When you are satisfied with your typedef, you can select `OK`, which will save the typedef to the descriptor file. The internal model will be updated. Matching classes will be marked with an image in their editor window to delineate them as matched.

## 3.5. Creating Introductions

An Introduction is a way to force some class you already have to implement an interface it does not already implement. Introductions can optionally have mixins, which will provide the implementation for those interfaces. If no mixins are supplied, a default implementation is provided by JBoss AOP (return nulls, etc).

### 3.5.1. Setup

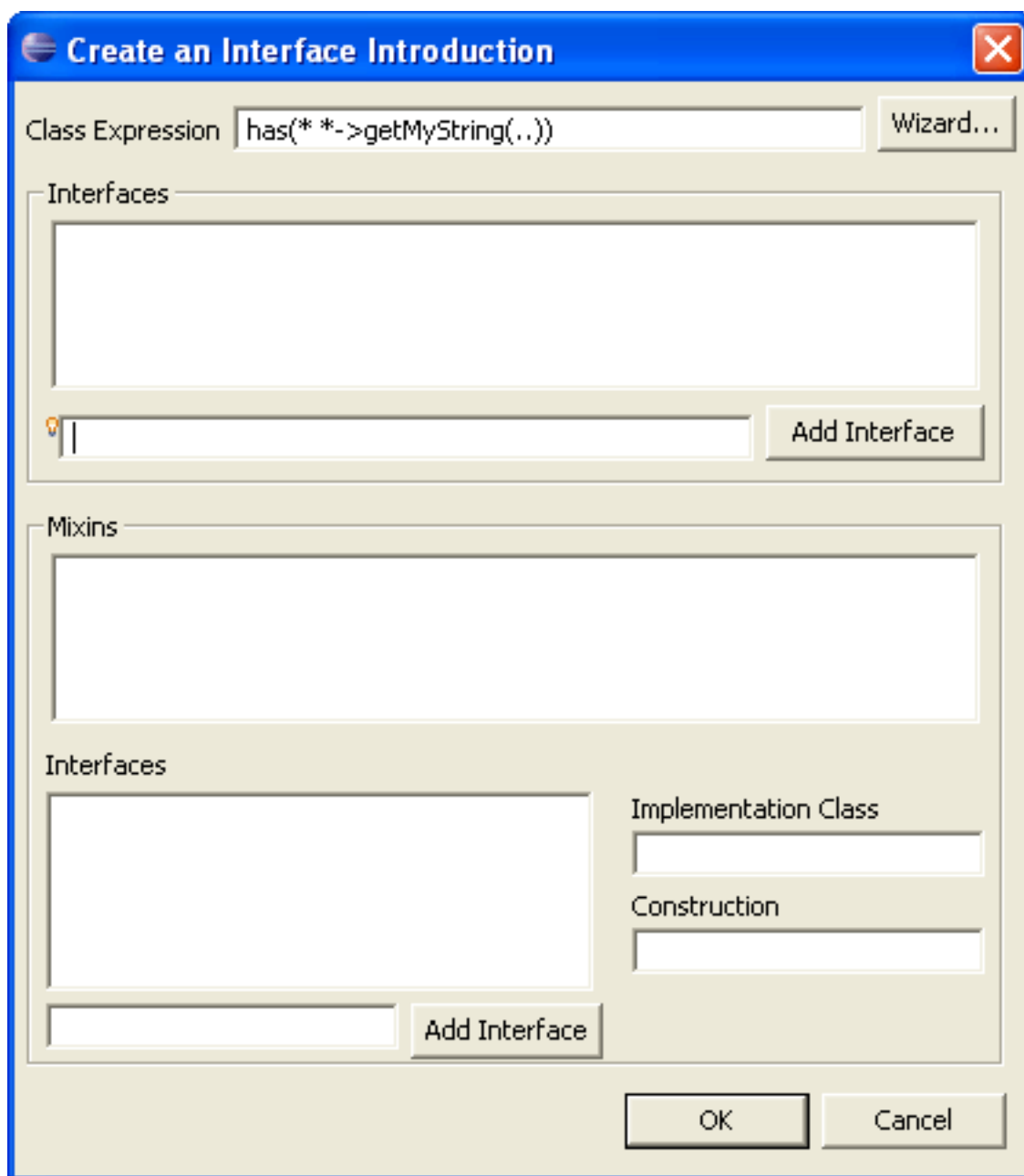
Before we start, I created a new Interface called `IntroPojoInterface` with one method. The method had the following signature:

```
public void IntroductionMethod();
```

I also created an implementing class called `IntroPojoMixin` which had a method by the same signature.

### 3.5.2. Creating the Introduction

To create an introduction, you should right-click on the `Introductions` element in the `Aspect Manager View`. Then, select `Create Introduction`. The dialog should look like the one shown below.



Creating an Introduction

**Figure 3.2.**

*NOTE: Many of the fields provide content assistance. Pressing `ctrl+space` will invoke this feature*

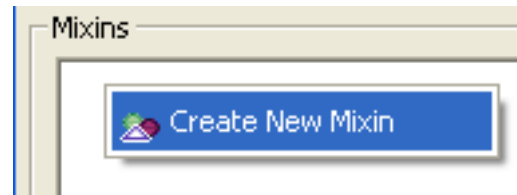
The top line is the class expression which describes what group of classes this introduction is going to refer to. In the image shown, any class which contains the method `getMyString`, with any combination (or none) of parameters, will be affected by this Introduction.

The next section is where you can add interfaces. A default implementation of EVERY method in this list of interfaces will be injected into any class that matches the above type expression.

Following that is the mixin section. Here, you can add a list of mixins, each of which contains a similar list of interfaces, but also provides a field to designate a class as the implementation of that group of interfaces.

To create a mixin, right-click on the list area and select `Create New Mixin`. A mixin will appear in the list.

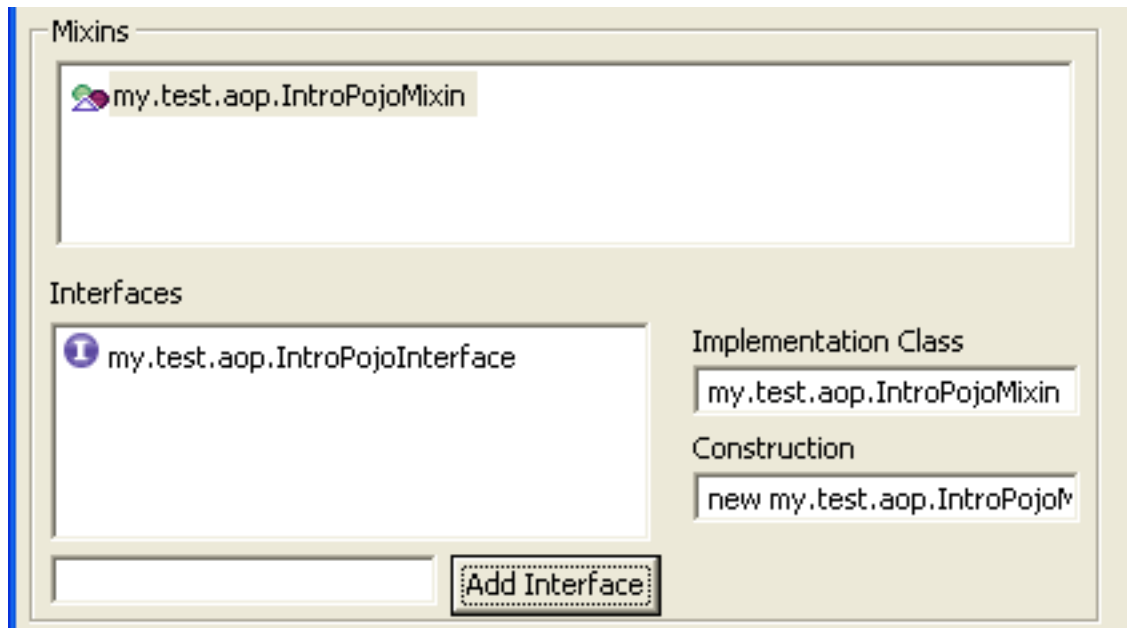
To **MODIFY** this mixin, click on it. The lists and text fields below will fill automatically with the values from the mixin. Any modifications to these fields will instantly update the mixin that is selected.



**Figure 3.3.**

The `construction` field is updated with the class you provide above it, but you can change the construction field on your own as well. What that field represents is what java code will be called to create your mixin implementation class. If your implementation class requires no parameters, then the generic `new yourclass()` will suffice. The other most common construction text would be to pass in `this` as the first parameter.

Of course, this all depends on your implementations.



What your completed mixin should look like.

**Figure 3.4.**

When your introduction and their mixins are finalized, clicking `ok` and closing the dialog will update the aop descriptor file. The internal model will be updated. Matching classes will be marked with an image in their editor window to delineate them as matched.

# 4

## The Aop Editor

The AOP Editor is a small extension to the generic java editor class for eclipse. The main difference is that it does provide markers on the left side to visually clue the user in to what java elements (methods, types, fields, etc) match the myriad of expressions you have created for your pointcuts, typedefs, and introductions.

### 4.1. Setup

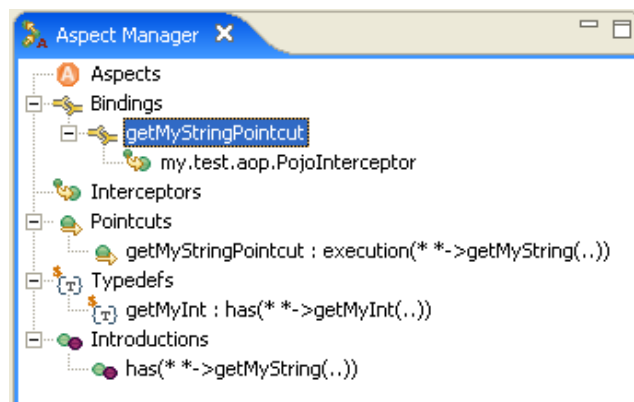
I've created some elements to begin, as seen to the right.

A typedef with the expression `has( * *->getMyInt( .. )`

An introduction with the expression `has( * *->getMyString( .. )`

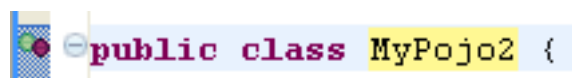
A named pointcut with the expression `execution( * *->getMyString( .. )`

A binding that uses that pointcut. The binding binds our `PojoInterceptor` as advice.



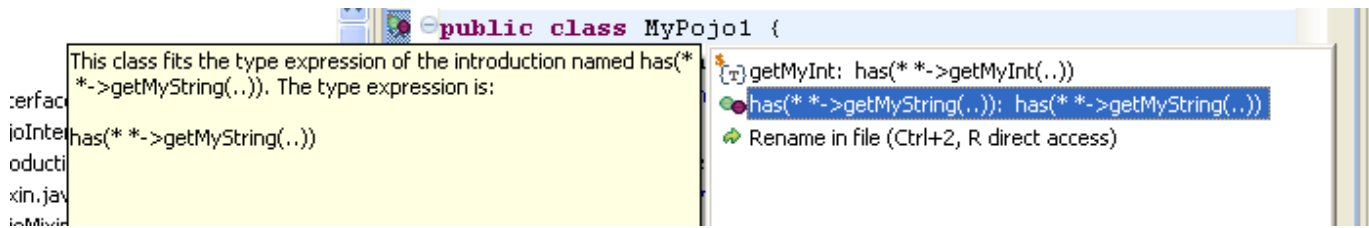
### 4.2. Type matchers

Currently, introductions and typedefs are the only implemented features who's expressions reference only class types.



To the upper-right, we see our `pojo2` class declaration. `MyPojo2` only matches the introduction, and does not match the typedef. Below, we see our `MyPojo1` definition, which matches both our typedef and the introduction. Because the image markers overlap, it can become difficult to see everything that references or matches this class type.

To remedy this situation, we've also included "advice", which can be accessed through a hotkey combination. If you place your cursor on the class declaration (on the words `MyPojo1`) and press `ctrl+1`, this information will be displayed. It is possible this feature will prove less beneficial than it at first seemed, and could be removed, instead presenting the information in the `Advised Members View`. This remains to be seen, based on user feedback.

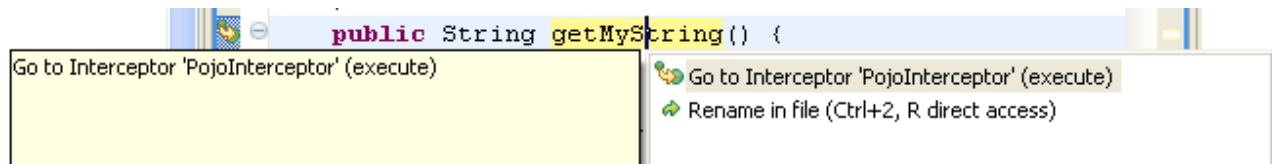


"Advice" offered for class types

Figure 4.1.

### 4.3. Advised Members Markers

The other type of markers we have are for member elements of a class source file that are advised by some type of advice. `Advice` is defined as Interceptors or Aspect methods. An example of this is shown below.

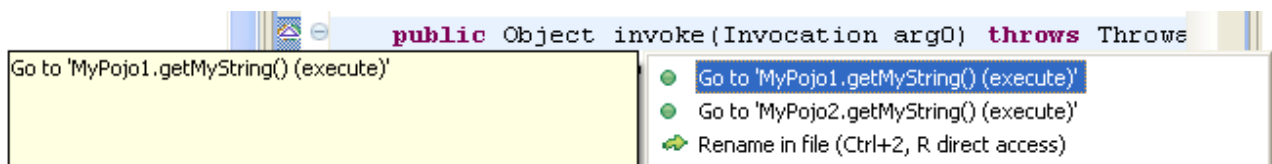


"Advice" offered for elements advised by interceptors / aspects

Figure 4.2.

By selecting the `goto` option, an editor with the source for that Interceptor or Aspect will be opened. There, also, there will be a marker that can bring you to any of members that that advice advises as shown below.

An image marker is also available, however it is not currently shown because there is already an image marker designating the `invoke` method as an implementer method of the `Interceptor` interface.



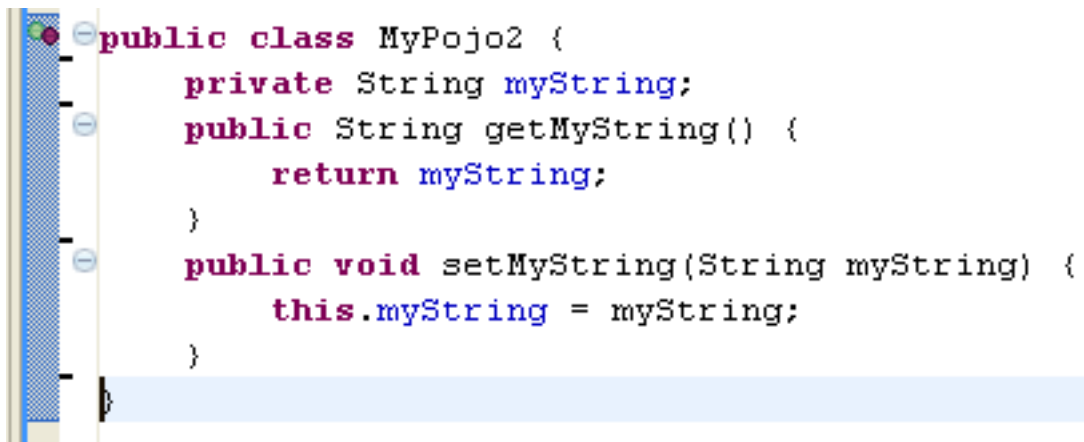
"Advice" offered to navigate to java methods / fields

Figure 4.3.

### 4.4. Adding / deleting methods

Adding or removing methods from classes could change which expressions they match. (Pointcut, typedef, introduction, etc). So whenever a delta can be calculated, whether the classes match is recalculated and markers are

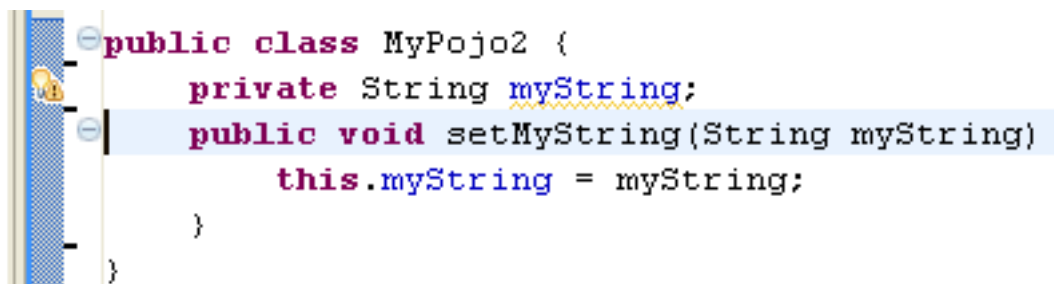
either shown (if a matching method is added) or removed (if removed). A before-and-after series is posted below to demonstrate this.



```
public class MyPojo2 {
    private String myString;
    public String getMyString() {
        return myString;
    }
    public void setMyString(String myString) {
        this.myString = myString;
    }
}
```

The before view. Class matches the introduction.

Figure 4.4.



```
public class MyPojo2 {
    private String myString;
    public void setMyString(String myString) {
        this.myString = myString;
    }
}
```

The after view. Class does not match the introduction.

Figure 4.5.

## 4.5. Changing Annotations

Similar behavior can be achieved by changing a class's annotation, as demonstrated below. First, I created a new Pojo with the following code:

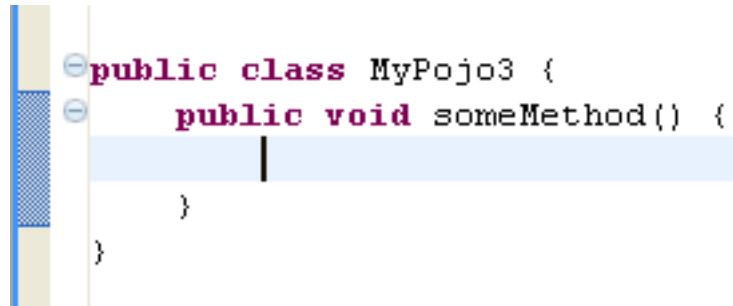
```
public class MyPojo3 {
    public void someMethod() {
    }
}
```

I've added the following typedef expression: `has(* *->@my.test.aop.AnnotInterface(...)`

Along with that, I created an empty interface by that name.

### 4.5.1. Making it match

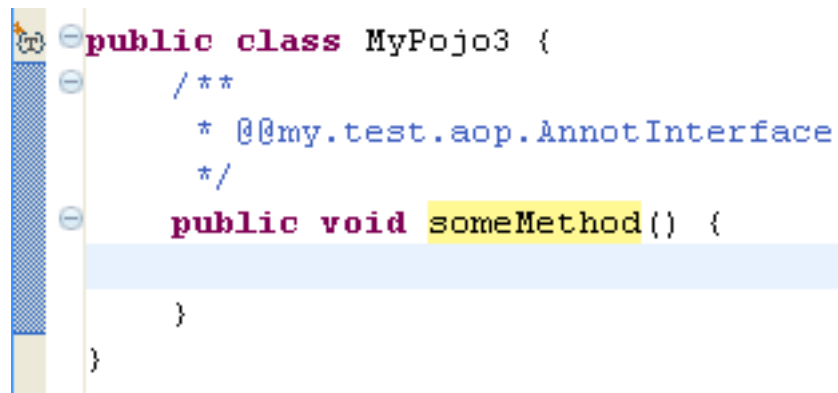
To make it match, just add the annotation as shown below. The example shown is an example of JBoss AOP's 1.4-style annotations. In the standalone AOP library, you must pre-compile classes with this style of annotations. In the IDE, it is done for you.



```
public class MyPojo3 {  
    public void someMethod() {  
    }  
}
```

The before view. Class matches the typedef.

Figure 4.6.



```
public class MyPojo3 {  
    /**  
     * @my.test.aop.AnnotInterface  
     */  
    public void someMethod() {  
    }  
}
```

The after view. Class does not match the typedef.

Figure 4.7.

# 5

## Running your AOP app

Right-click on your project, and select `run->run as...`

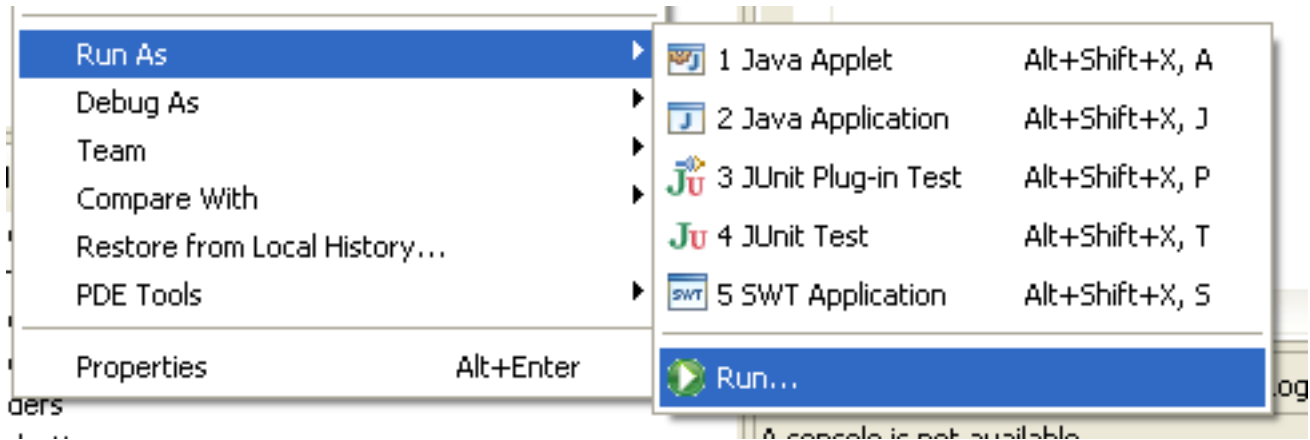


Figure 5.1.

Right-click on `JBoss AOP Application` and select `new`. Make `SURE` the project and main class are set properly.

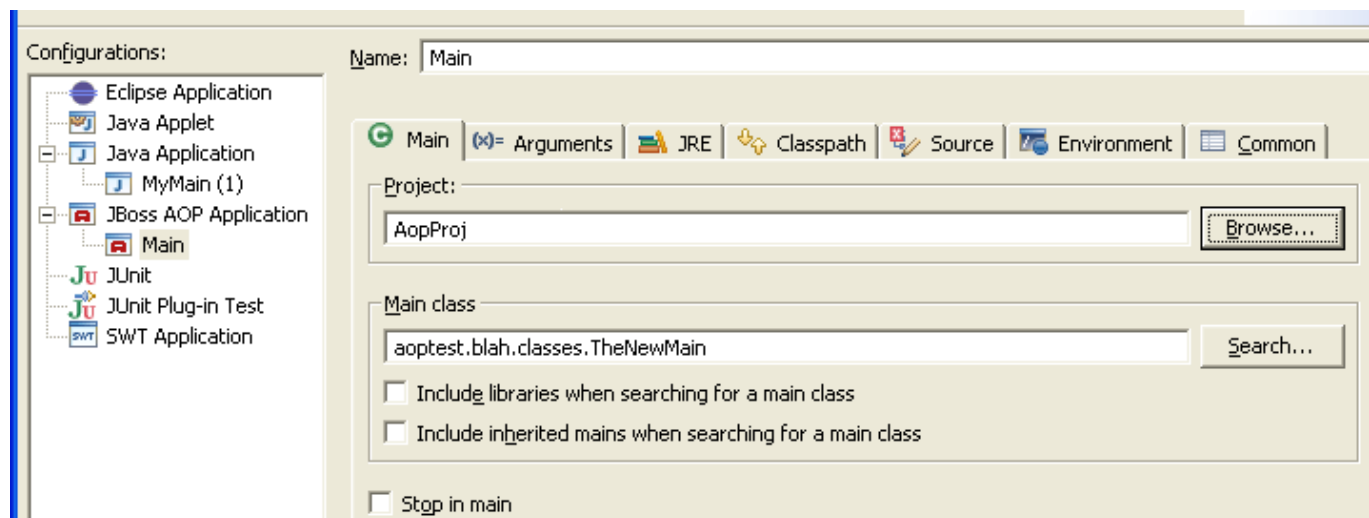


Figure 5.2.

Apply the changes and run your application