



# CHAPTER 2

---

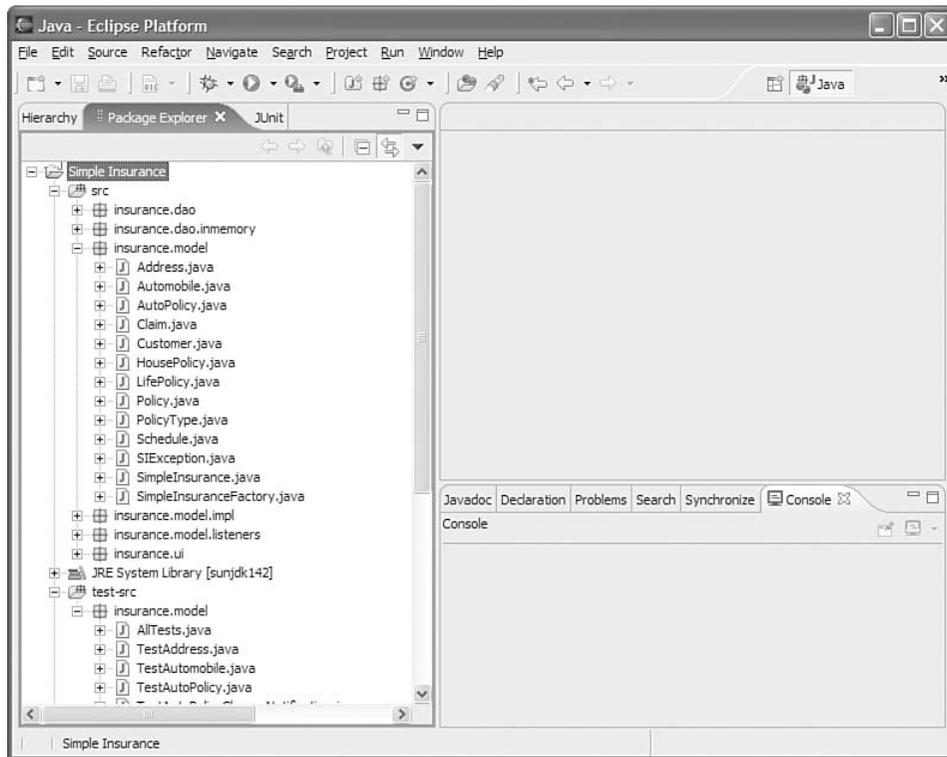
## *First Steps in AJDT*

If you've been following along, then at this stage you should have a working Eclipse environment with AJDT installed. What we need now is an application we can use to do some AspectJ development. This chapter introduces a simple insurance application that we can use for this purpose. We show you how to convert the simple insurance Java project into an AspectJ project, discover all the places in the code to do with informing listeners whenever an insurance policy object is updated, and refactor all those code fragments into an aspect to give a modular implementation.

### **2.1 A Simple Insurance Application**

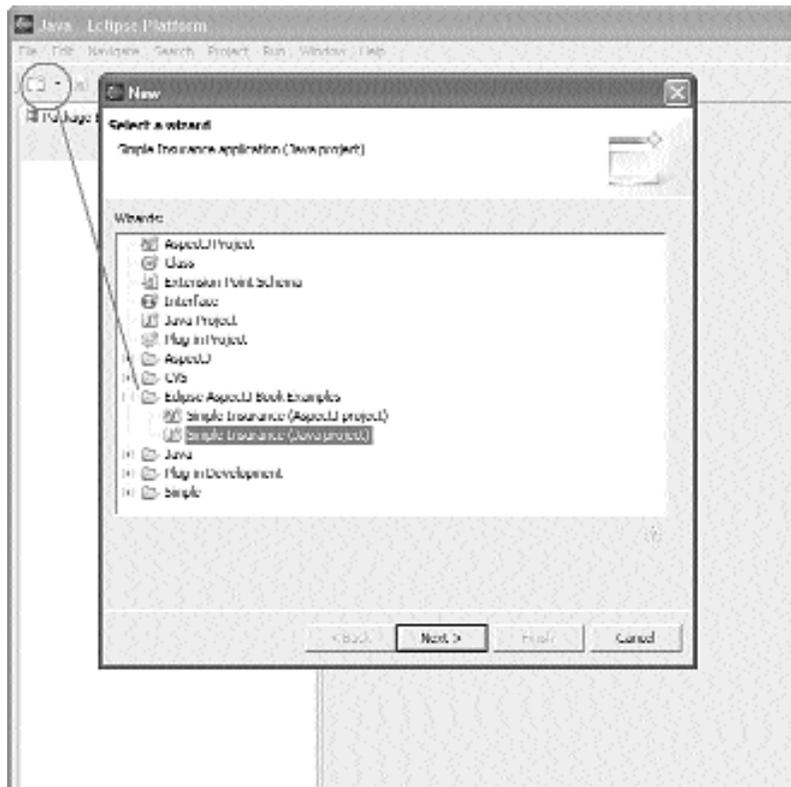
Simple Insurers Inc. are considering going into business as no-frills, bargain-basement insurers. They have been developing a simple insurance application to underpin their new business venture and keep track of customers and policies. It is still early days, and the application is far from complete, but it implements just enough functionality to cover the first few user stories and get feedback from the internal customers to help plan the next iterations.

Figure 2.1 shows the Simple Insurance application project in Eclipse. It is just a regular Eclipse Java project at this stage. Notice in the Package Explorer (the view on the left side of the Eclipse window) that we have two source folders in the project, one named `src` and one named `test-src`. The `src` source folder contains the main application classes, and the `test-src` source folder contains the JUnit test cases.



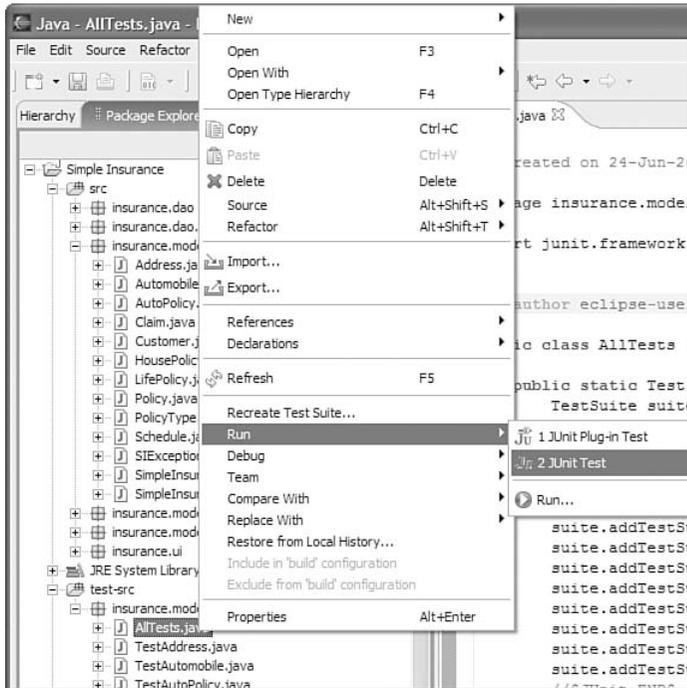
**Figure 2.1** The Simple Insurance project in Eclipse.

If you installed the Eclipse AspectJ Examples plug-in following the instructions in Chapter 1, you can create the Simple Insurance project in your own workspace by clicking the **New** icon and selecting the Simple Insurance project from the Eclipse AspectJ Examples category (see Figure 2.2).

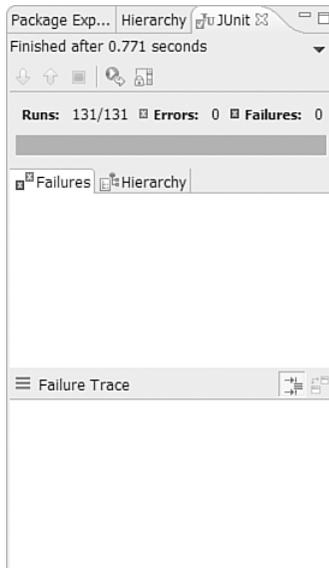


**Figure 2.2** Installing the Simple Insurance project into your workspace.

It is a good idea at this point to run the test suite and make sure that nothing is amiss. We will be working primarily with the `insurance.model` and `insurance.model.impl` packages where the domain classes can be found. As shown in Figure 2.3, expand the `insurance.model` package node inside the `test-src` source folder and select the file `AllTests.java`. From the context menu (right-click) select **Run > JUnit Test**. If all goes well, you should be able to click the **JUnit** tab to bring the JUnit view to the foreground, and see a successful result as shown in Figure 2.4.

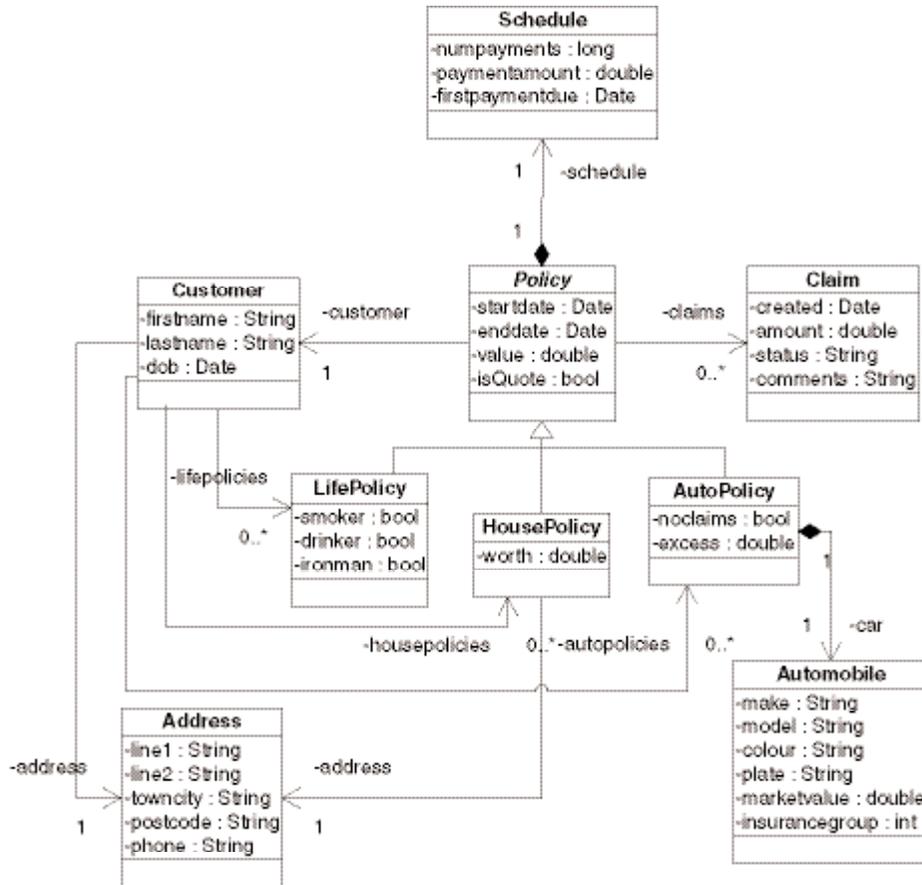


**Figure 2.3** Launching the test suite.



**Figure 2.4** A passing test suite—the JUnit view.

Figure 2.5 shows an overview of the classes in the Simple Insurers Inc. domain model (the `insurance.model` package).

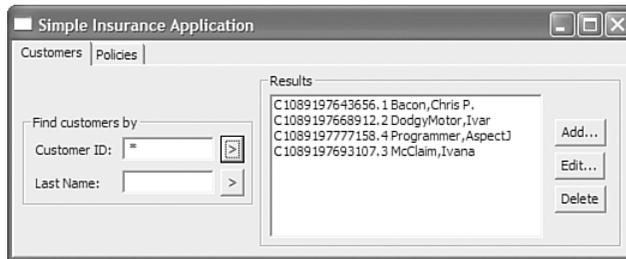


**Figure 2.5** Simple Insurers Inc. domain model.

Simple Insurers Inc. will be offering three kinds of insurance policies when they first launch: life insurance, car insurance, and home insurance. Policies are taken out by customers, who pay for their insurance in accordance with some payment schedule. The model also has a facility to record claims made against policies, although worryingly there is no implementation yet to actually pay out on claims.

Simple Insurers Inc. will initially use telesales to market their insurance products, and agents at their company headquarters will have available to them a

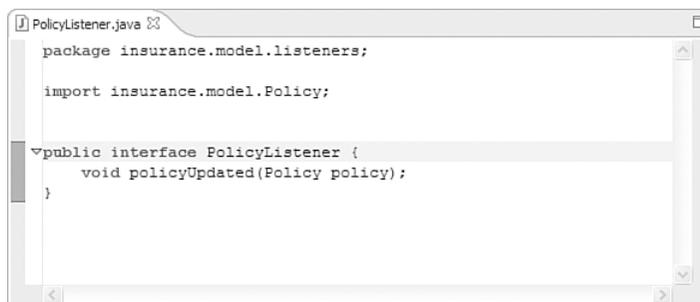
simple desktop application with which they can create, view, and update information on customers and policies. The user interface for this application is implemented in the `insurance.ui` package. You can launch the application from the Eclipse workbench by selecting the `SimpleInsuranceApp.java` file in the Package Explorer view, and then choosing **Run > Java Application** from the context menu. Figure 2.6 shows how the application looks when running.



**Figure 2.6** The Simple Insurers Inc. application.

## 2.2 Tracking Policy Updates

The user interface is connected to the model using a simple model-view-controller design. There is a simple `PolicyListener` interface that clients can implement, and after registering themselves with a policy, they will receive a `policyUpdated()` notification whenever the policy is updated. Figure 2.7 shows the `PolicyListener` interface in the editor. Figures 2.8 and 2.9 show excerpts from the `PolicyImpl` class. Notice in Figure 2.8 the calls to `notifyListeners` after updating the state of the `Policy` on lines 86 and 101.



**Figure 2.7** `PolicyListener`.

```

117  /**
118   * @param endDate The endDate to set.
119   */
120  public void setEndDate(Date endDate) {
121      this.endDate = endDate;
122      notifyListeners();
123  }
124
125  /**
126   * @return Returns the endDate.
127   */
128  public Date getEndDate() {
129      return endDate;
130  }
131
132  /**
133   * @param value The value to set.
134   */
135  public void setValue(double value) {
136      this.value = value;
137      notifyListeners();
138  }
139
140  /**
141   * @return Returns the value.
142   */
143  public double getValue() {
144      return value;

```

**Figure 2.8** Change notifications.

Figure 2.9 shows the code that keeps track of registered listeners and performs the actual notification.

```

144  /**
145   * @param claim
146   */
147  public void removeClaim(Claim claim) {
148      if (claim == null) {
149          throw new IllegalArgumentException("Received a null claim!");
150      }
151      this.claims.remove(claim);
152      notifyListeners();
153  }
154
155  public void addPolicyListener(PolicyListener l) {
156      this.listeners.add(l);
157  }
158
159  public boolean removePolicyListener(PolicyListener l) {
160      return this.listeners.remove(l);
161  }
162
163  protected void notifyListeners() {
164      for (Iterator iter = listeners.iterator(); iter.hasNext();) {
165          PolicyListener l = (PolicyListener) iter.next();
166          l.policyUpdated(this);
167      }
168  }

```

**Figure 2.9** Managing the listeners.

The various types of insurance policy all inherit the basic capability of managing a set of listeners from their parent `PolicyImpl` class, but each subclass has to be sure to call the `notifyListeners` method whenever it updates any of its own state. For example, the `AutoPolicyImpl` class keeps track of a no-claims bonus. If we look into the update method for the no-claims bonus, we will see the code shown in Listing 2.1, with a call to `notifyListeners()` duly made after updating the state.

**Listing 2.1** Excerpt from *AutoPolicyImpl* Class

---

```
public void setNoClaims(boolean noClaims) {
    this.noClaims = noClaims;
    notifyListeners();
}
```

---

So far this strategy has been working okay, but now the business has asked us to consider adding support for pet insurance policies, too. Rather than keep adding more and more calls to `notifyListeners()` spread throughout the hierarchy, is there something better we can do? What we're looking at is a 1-to-*n* implementation: We have one simple requirement, to notify all registered listeners whenever a policy is updated, but an implementation that is scattered in *n* places throughout the policy hierarchy. To quote from the eXtreme programming discipline: "Refactor mercilessly to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. Make sure everything is expressed once and only once. In the end it takes less time to produce a system that is well groomed."<sup>1</sup> We said in the introduction that aspect-oriented programming offered us a new kind of a module, known as an *aspect*, that could help solve problems such as this and turn it back into a one-to-one mapping. In other words, we should be able to modularize the change notification policy inside an aspect. Instead of adding yet more scattered calls to `notifyListeners` in the new `PetPolicyImpl` class, perhaps we should consider refactoring.

---

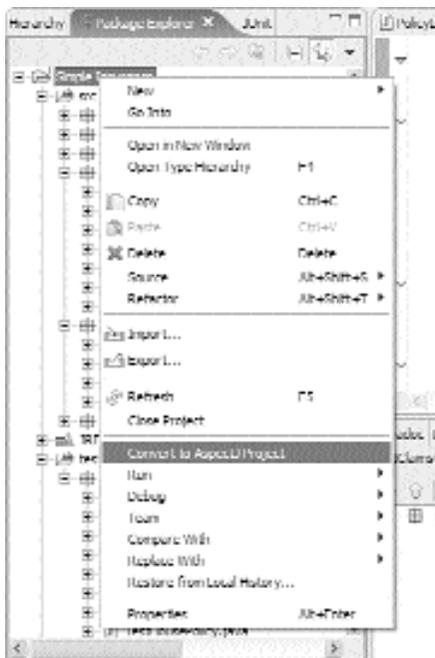
1. <http://www.extremeprogramming.org/rules/refactor.html>

## 2.3 Creating an AspectJ Project

The first thing we need to do before we can start using AspectJ in the Simple Insurance application is to convert the Simple Insurance project from a plain Java project into an AspectJ project. Converting the project to an AspectJ project enables us to use the AspectJ language to implement the application, and the AspectJ compiler to build it.

### 2.3.1 Converting an Existing Java Project

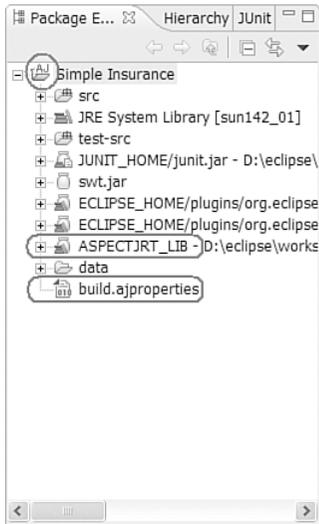
Converting an existing Java project to an AspectJ project is simple. Select the project in the Package Explorer, and choose **Convert to AspectJ Project** from the context menu (see Figure 2.10). The AJDT Preferences Configuration Wizard may pop up during the conversion process; refer to the next section to learn more about it.



**Figure 2.10** Converting from a Java project to an AspectJ project.

After performing the conversion, you will notice a couple of changes in the Package Explorer, as highlighted in Figure 2.11. First, the project icon has changed from the Java project icon (a folder with a J decoration, as shown in

Figure 2.10), to the AspectJ project icon (a folder with an AJ decoration). Second, a new jar file has been added to the project’s build path, using the Eclipse path variable `ASPECTJRT_LIB`. This path variable is defined by the AJDT plugins to point to the AspectJ runtime library Jar file (`aspectjrt.jar`) shipped with AJDT. The AspectJ runtime library is a small library (about 35K) that all AspectJ programs need when they execute. This is the only runtime requirement for AspectJ—that `aspectjrt.jar` be somewhere on the class path—the AspectJ compiler produces regular Java class files that can be executed on any *Java Virtual Machine* (JVM) and look just like any other Java application to the JVM. Third, a new file called `build.ajproperties` has been created in the project. This file stores configuration information about the building of the project, which we look at in detail later.



**Figure 2.11** An AspectJ project.

If you ever want to turn an AspectJ project back into a regular Java project, the process is equally simple: select the project in the Package Explorer and choose **Remove AspectJ Nature** (see Figure 2.12). See the sidebar “Eclipse Builders and Natures” if you are curious as to what goes on when a project is converted to an AspectJ project and back again.



Figure 2.12 Converting an AspectJ project back into a Java project.

### Eclipse Builders and Natures

Projects in Eclipse have one or more project natures associated with them. A project nature tells the rest of the world what kind of capabilities the project has. Java projects have the Java nature, and tools that work with Java programs can offer their workbench contributions for working with those projects. AspectJ projects have both the Java nature *and* the AspectJ nature, so Java tools should continue working, but they also benefit from the AspectJ specific tools too. Builders are closely associated with natures. Normally when a nature is added to a project, it installs a project builder. Projects can have multiple builders, just as they can have multiple natures. Java projects use the Java builder (the Eclipse Java compiler), but AspectJ projects use the AspectJ builder. When you convert a Java project to an AspectJ project, the AspectJ nature is *added* to the project, and the Java builder is *replaced* with the AspectJ builder (any other builders defined for the project are left alone). When you remove the AspectJ nature from a project, the AspectJ nature is removed, and the AspectJ builder is replaced with the Java builder. Of course, if you have AspectJ-specific artifacts in your projects, you might get build errors when the Java builder encounters them.

### 2.3.2 Configuring the Workbench for AspectJ Development

The very first time you activate the AspectJ tools inside a workspace (for example, by converting a project to an AspectJ project, or by creating a new AspectJ project) you will see the AJDT Preferences Configuration Wizard appear (see Figure 2.13).



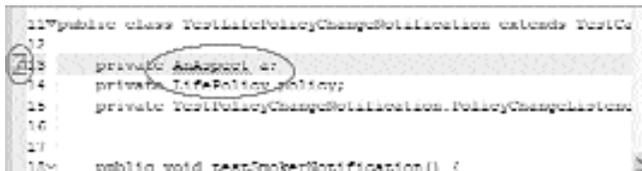
**Figure 2.13** AJDT Preferences Configuration Wizard.

This wizard offers to make two configuration customizations for you that will make working with AspectJ inside Eclipse a much more pleasant experience:

**Make the AspectJ editor default for “.java”** The standard Java editor that is associated with .java files does not understand the extra keywords that AspectJ introduces, and neither does the Outline view associated with it understand aspect constructs. Choosing this customization makes the AspectJ editor be associated with .java files by default. The AspectJ editor is an extension of the Java editor offering all of the Java editor features, plus the capability to understand AspectJ programs.

**Disable analyzing of annotations while typing** The Java Development Tools include an eager parser that analyzes your program as you type, giving you early indications of problems you will have when you compile the code. This eager parser does not understand AspectJ; hence it incorrectly reports errors (the infamous “red squiggles”) in your AspectJ program, which can be distracting. Figure 2.14 shows an example of the problem.

The eager parser does not recognize the type `AnAspect` (because aspects are not present in the type-space that it uses for name resolution), and so highlights it as a problem when in fact the AspectJ compiler will build the source file perfectly fine. Selecting this customization deactivates the eager parsing. The AJDT development team is working on an update to the tools support that enables the eager parser to understand AspectJ constructs, too. When this new version is available, it will be possible to turn the “analyze annotations while typing” option back on again for AspectJ programs.



**Figure 2.14** Don't get the red squiggles; disable “analyze annotations while typing.”

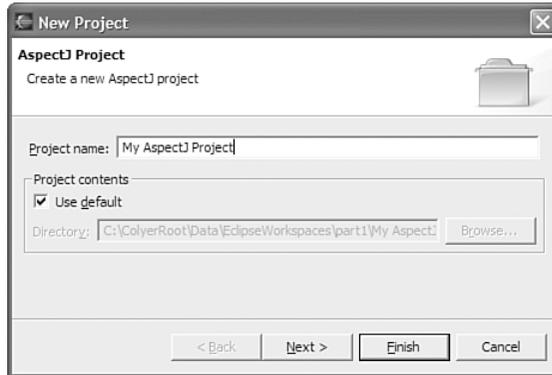
### 2.3.3 Creating a New AspectJ Project

Of course, it is also possible to create a new AspectJ project from scratch without converting an existing Java project. To do this, you use the New AspectJ Project Wizard. You can reach the wizard via the Eclipse workbench menus by selecting **File > New > Project**, and then **AspectJ Project**. Alternatively, use the **New** icon on the toolbar and select **AspectJ Project** from there (see Figure 2.15).



**Figure 2.15** Launching the New AspectJ Project Wizard.

When the wizard launches, the dialog box shown in Figure 2.16 displays. It behaves exactly like the New Java Project wizard, except that the project it creates at the end is an AspectJ one.



**Figure 2.16** The New AspectJ Project Wizard.

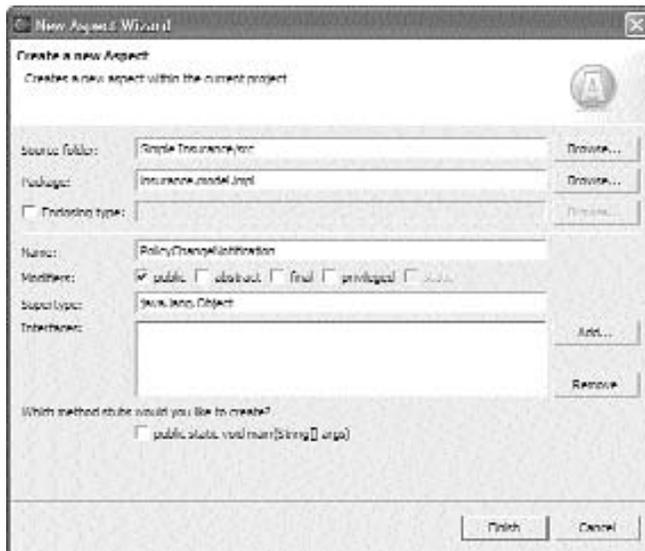
## 2.4 Creating the PolicyChangeNotification Aspect

The Simple Insurance project is now ready for use with AspectJ. It was automatically built by the AspectJ compiler when we converted it to an AspectJ project, so now would be a good time to run the test suite again and verify that the AspectJ compiler has indeed built the project correctly. The tests should all pass correctly.

We are ready to begin the refactoring. Recall that we want to replace the calls to `notifyListeners` that are scattered throughout the policy class hierarchy with a modular implementation in an aspect. Just as to create a new class we would use the New Class Wizard, so to create a new aspect we use the New Aspect Wizard.

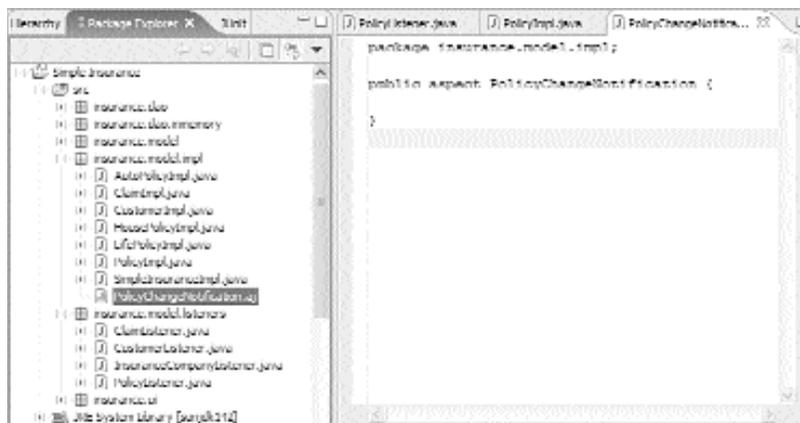
### 2.4.1 The New Aspect Wizard

We want to create an aspect in the `insurance.model.impl` package where the policy implementation classes are defined. Select the `insurance.model.impl` package in the Package Explorer, and from the context menu choose **New > Aspect**. This launches the New Aspect Wizard, as shown in Figure 2.17. (You can also get to this wizard from the **File > New** workbench menu, and the **New** icon on the toolbar.)



**Figure 2.17** The New Aspect Wizard.

Notice that the New Aspect Wizard is very similar to the New Class Wizard. The source folder and package fields are prefilled because we launched the wizard from the context menu of the package in the Package Explorer. We just need to provide a name for the aspect and then click **Finish**. Because the aspect is going to encapsulate the implementation of policy-change notification, we have called it `PolicyChangeNotification`. Upon completion of the wizard, the aspect is created and opened in the editor, as shown in Figure 2.18.

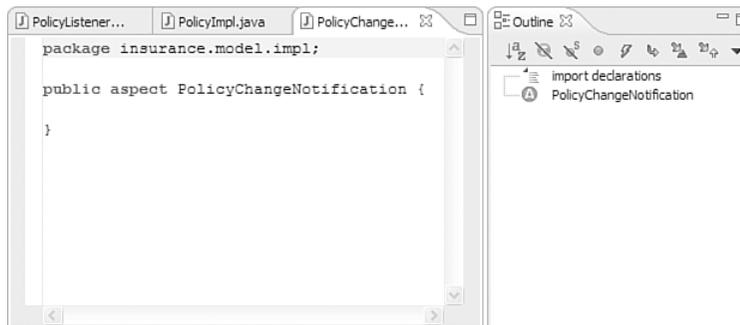


**Figure 2.18** A skeletal aspect.

In the Package Explorer, you can see that the wizard has created a new source file called `PolicyChangeNotification.aj`. AJDT uses the `.aj` extension for source files containing aspects, but you can configure it to use `.java` instead if you prefer. See the sidebar “Choice of File Suffix: `.java` or `.aj`”

In the editor, you can see the basic form of an aspect. Notice that it looks just like a class definition, except that it uses the `aspect` keyword in place of the keyword `class`. In AspectJ, aspects are first class entities in the program just as classes are. There are a lot of similarities between aspects and classes—in fact pretty much anything you can declare in a class you can declare in an aspect, too, but aspects can also do things that classes cannot do. We look at some of those things in Section 2.5.

If you have the Outline view open (**Window > Show View > Outline**), you will see an outline like that shown in Figure 2.19. This shows us that the source file contains no import declarations (the package statement and any import declarations work in exactly the same way for an aspect as they do for a class), and a single aspect with no members.



**Figure 2.19** The Outline view.

Chapter 9 provides full details on aspects in AspectJ, but you already know a surprising amount because of your familiarity with classes in Java. You certainly know enough to continue following the examples in this chapter, so let’s move on and make our `PolicyChangeNotification` aspect do something useful.

**Choice of File Suffix: .java or .aj**

The AspectJ compiler doesn't care whether you name your source files with the `.aj` extension or with the `.java` extension—it treats them both equally. So you could name all your source files with the `.aj` extension, or all with the `.java` extension, or with any combination of the two, and it wouldn't make any difference. It does make a difference, however, if you convert an AspectJ project back to a Java project for some reason. The Java compiler ignores files with an `.aj` extension, which can make the move easier (but you're still going to have to cope with the fact that all the functions your aspects were implementing are now missing). However, using the `.aj` extension for files containing regular Java classes has the consequence that the types defined in them are not visible to Eclipse's Java model. By default therefore, AJDT creates new classes in source files with a `.java` extension, and new aspects in source files with an `.aj` extension. If you want to change this behavior, you can do so via the AspectJ project Properties page accessible via the **Properties** option in the context menu when a project is selected.

## 2.5 Stating the Notification Policy

How will we know whether our refactoring has succeeded? All the tests will pass, of course, but they do that now. The problem we want to address is the duplication and scattering of calls to `notifyListeners` throughout the policy hierarchy. If our refactoring is a success, there will be no calls to `notifyListeners` left in the policy hierarchy, and they will all have been replaced by a single call in the `PolicyChangeNotification` aspect. AspectJ enables us to capture requirements such as this directly in the code, and in this section you learn how. To do that, we need to introduce you to a couple of new concepts: join points and pointcuts.

### 2.5.1 Introducing Join Points

Programs live to execute, and when they execute, stuff happens. Methods get called, objects get initialized, fields are accessed and updated, constructors are executed, and so on. AspectJ calls these events that happen when a program is running *join points*. If program execution was a spectator sport, and you were commentating on the 2006 world championships, join points are the things that you would highlight in your commentary:

```
Bob: Looks like we're in for a good clean run today.
Jim: Yes Bob, the classes are being fetched from disk now; I
can't wait for this one to get started.
Bob: We're off! The main class is loaded and the static
initializer just ran.
```

```

Jim: A traditional opening, executing the main method now
...
Bob: Hmmmm, accessing the "out" field on the old System
class—think he's going for the System.out.println routine
Jim?
Jim: It does look that way, Bob. Ah, yes, look—calling the
println method, and that's a nice String there in the
arguments.
Bob: So, the println method is executing, you can almost
feel the tension. Here it comes ... there's an H, an E, ...
Jim: It is, it is, it's "Hello, World!". Very nicely done.
Safely back from the call to println.
Bob: Seems like the execution of the main method is just
about done.
Jim: They think it's all over ... It is now.

```

AspectJ supports join points for calls made to methods and constructors, for the actual execution of methods and constructors, for the initialization of classes and objects, for field accesses and updates, for the handling of exceptions, and a few more besides. Chapter 5 provides a more thorough introduction to join points.

Join points by themselves aren't all that exciting. Stuff has always happened when programs are executed. What is different in AOP and in AspectJ is that the programmer has a way to refer to join points from within the program. A way to say things such as, “Whenever you update the state of a policy object, notify all of its listeners.” To the AspectJ compiler, that sentence looks a little bit more like this: “Whenever a join point that meets these conditions occurs (we'll call that a `policyStateUpdate` shall we?), call the `notifyListeners` method on the policy object.”

## 2.5.2 Writing a Pointcut

The way that we specify which join points we are interested in is to write a *pointcut*. Pointcuts are like filters; as the program executes, a stream of join points occur, and it is the job of a pointcut to pick out the ones that it is interested in.<sup>2</sup> The join points we are interested in right now are those that represent a call to the `notifyListeners()` method. When we make such a call, we are notifying the listeners. Here is what the pointcut declaration looks like in AspectJ:

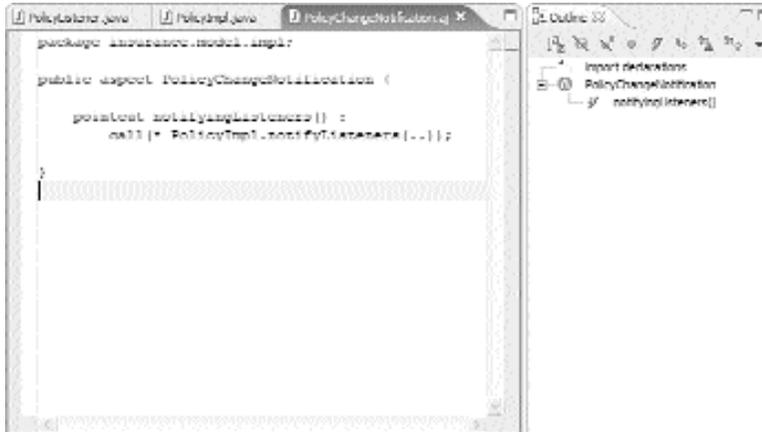
```

pointcut notifyingListeners() :
    call(* PolicyImpl.notifyListeners(..));

```

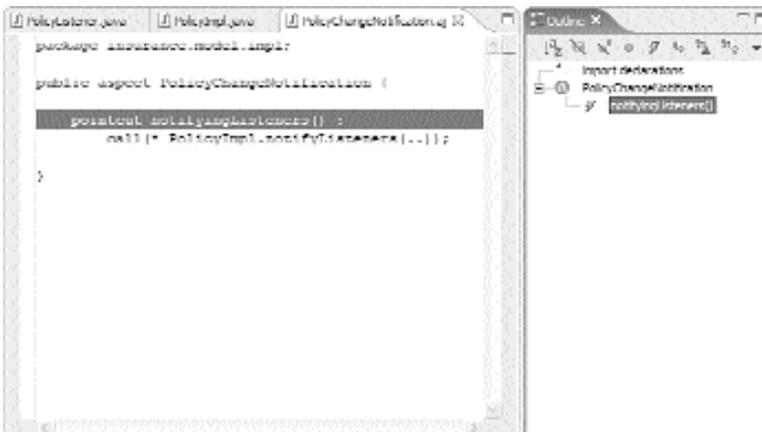
2. This is just a conceptual model of how it works; the actual implementation as generated by the AspectJ compiler is much more efficient than this.

Figure 2.20 shows the result of typing this into the aspect editor buffer and pressing **Ctrl+S** to save.



**Figure 2.20** A pointcut declaration.

The syntax highlighting shows us that `pointcut` and `call` are keywords in the AspectJ language. Also note that the Outline view has updated to show the pointcut as a member of the aspect (just like a field or a method is a member of a class). We can use the Outline view to navigate in the editor buffer by selecting the nodes in the tree. For example, if we click the `notifyingListeners()` node in the Outline view, the selection in the editor is changed to the pointcut declaration, as shown in Figure 2.21.



**Figure 2.21** Using the Outline view to navigate in the editor buffer.

The pointcut declaration declares a new pointcut called `notifyingListeners` (the name that appears in the Outline view). After a colon (:), comes the definition of which join points the `notifyingListeners` pointcut should match. In this case we are interested in join points that represent a call to certain methods. Inside the parentheses that follow the `call` keyword, we get to say which particular method calls we want to match: calls to the `notifyListeners` method defined in the `PolicyImpl` class. The first asterisk (\*) in the expression is a wildcard that says we don't care what the return type of the method is, and the two periods (..) inside the `notifyListeners(..)` say that we don't care what arguments it takes. We know both of these things: The `notifyListeners` method takes no parameters and returns void, but they are not pertinent to our pointcut—if the definition of the `notifyListeners` method were to change to take a parameter, or to return the number of listeners notified, we would still want the pointcut to match. By not specifying the details we don't care about, we make our program more robust in the face of change.

Chapter 6 explains everything you ever wanted to know about pointcuts and more; all that matters for the time being is that you get a feel for what a pointcut declaration looks like and what it does.

### 2.5.3 Using *Declare Warning*

This is all very nice, but the aspect still doesn't actually *do* anything. At the start of this section, we said that if our refactoring is a success, there will be no calls to `notifyListeners` left in the policy hierarchy, and they will all have been replaced by a single call in the `PolicyChangeNotification` aspect. So any call to the `notifyListeners` method that does not occur within the `PolicyChangeNotification` aspect breaks the modularity that we are trying to achieve. It would be useful at this point if we could find all such places. We can use an AspectJ construct known as *declare warning* to do just that. Let's declare it to be a compile time warning if anyone other than the `PolicyChangeNotification` aspect starts `notifyingListeners`. We can write it like this:

```
declare warning :
    notifyingListeners() && !within(PolicyChangeNotification)
    : "Only the PolicyChangeNotification aspect should be
        notifying listeners";
```

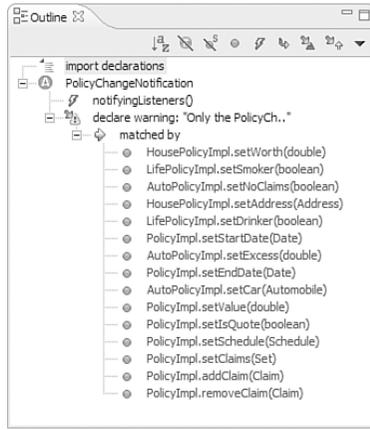
Figure 2.22 shows what happens when we type this into the editor buffer and press **Ctrl+S** to save.



**Figure 2.22** Declare warning in the editor buffer.

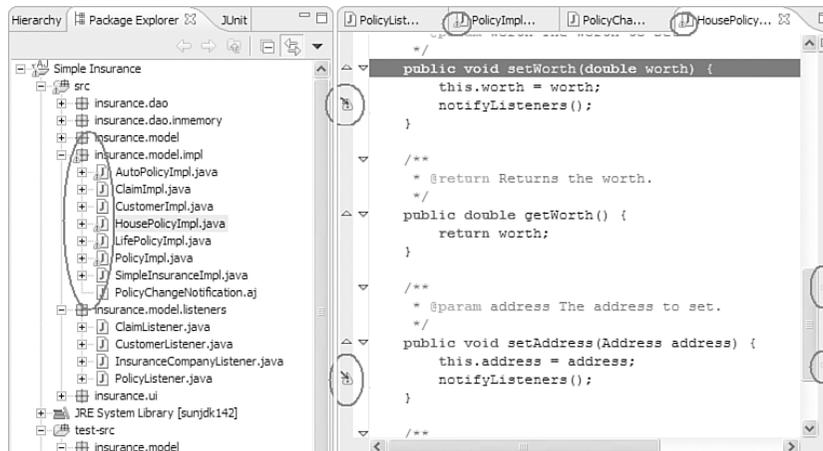
We can see again from the syntax highlighting that `declare warning` is a keyword (pair of keywords to be precise) in the AspectJ language. The general form of the statement is this: “Declare it to be a compile-time warning, if any join point matching the following pointcut expression occurs, and this is the warning message I’d like you to use.” In this particular case: “Declare it to be a compile-time warning, if any join point occurs that matches the `notifyingListeners` pointcut and is not within the `PolicyChangeNotification` aspect. At such points, issue a compile-time warning with the message ‘Only the `PolicyChangeNotification` aspect should be notifying listeners.’”

If we turn our attention to the Outline view, we can see that something very interesting has happened, as shown in Figure 2.23.



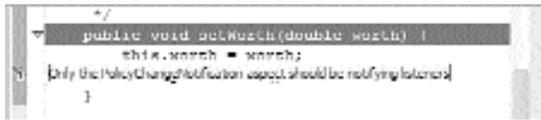
**Figure 2.23** Declare warning matches in the Outline view.

First of all, you can see that the `declare warning` appears in the Outline view as another kind of member within the aspect. There's also a plus sign (+) next to the `declare warning` node, indicating that there is content beneath it. If you expand this node you see `matched by`, and if you expand that node you see a list of all the places that are violating our policy of not calling `notifyListeners` outside of the `PolicyChangeNotification` aspect. There are 15 of them, and these nodes are actually hyperlinks that will take you directly to the offending statements in the program source code if you click them. Let's click the first entry in the list, `HousePolicyImpl.setWorth`. The editor opens on the `HousePolicyImpl` class, at the point where the call to `notifyListeners` is made, as shown in Figure 2.24.



**Figure 2.24** Showing warnings in the workbench.

In the gutter of the editor buffer, next to the call to `notifyListeners`, you can see a warning icon. There are also warning decorations on the file icons for all the source files in which warnings have been found, as you can see in the Package Explorer, and in the titles of the files at the top of the Editor window. The gutter to the right of the editor buffer gives an overview of the whole source file. It shows us that there are two warnings in the file, one on the line we are looking at, and one farther down in the source file. Hovering over the warning in the editor brings up the tool tip shown in Figure 2.25.



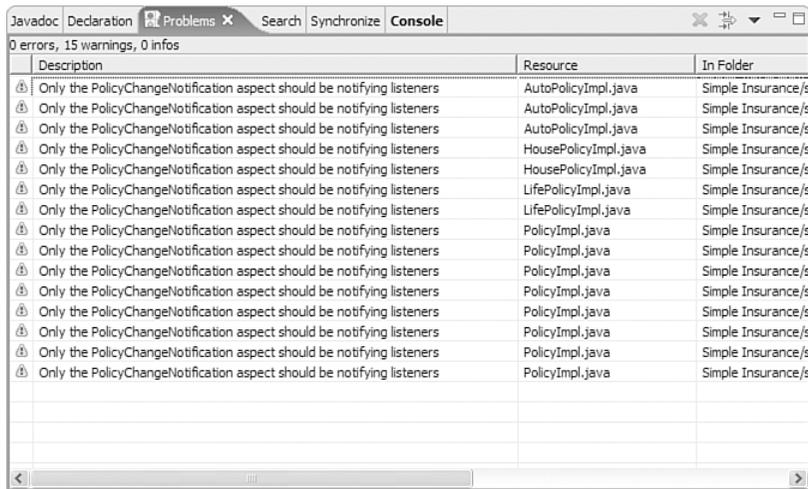
**Figure 2.25** Hover help for declared warnings.

You can clearly see the text of our declare warning—a powerful way to get a message across to a programmer who inadvertently violates the intended encapsulation of change notification in the aspect. Having navigated to the `HousePolicyImpl.java` file using the Outline view, we can click the back button on the toolbar (see Figure 2.26) to go back to the `PolicyChangeNotification` aspect again. In this way, we can easily navigate back and forth.



**Figure 2.26** Navigating back to the aspect.

Just in case you weren't getting the message that there are violations of the `PolicyChangeNotification` aspect's change notification rule by now, take a look at the Problems view as shown in Figure 2.27. You will see that a compiler warning message has been created for each join point that matches the pointcut we associated with the "declare warning" statement. These are just like any other compiler warning, and you can double-click them to navigate to the source of the problem.



**Figure 2.27** We've got problems!

When we have successfully completed the refactoring, none of these warnings should remain.

### How Can You Match Join Points at Compile Time?

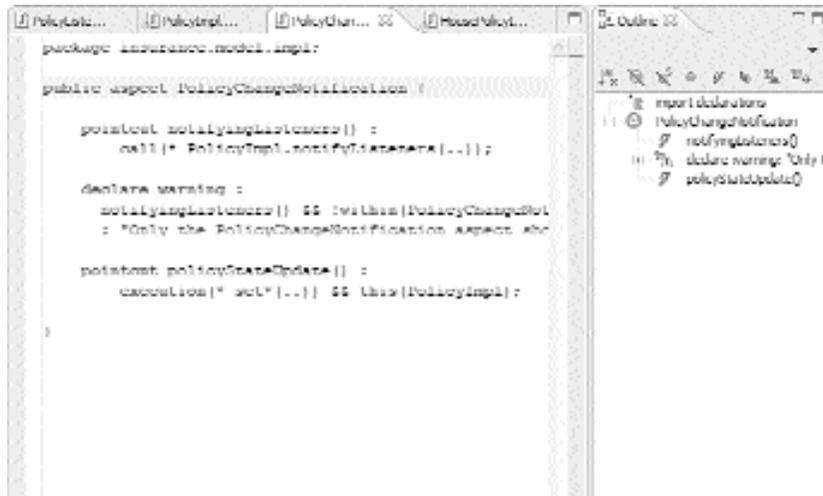
The more astute readers will have noticed a slight anomaly in the descriptions we just gave in this section. Join points are events that occur during the runtime execution of a program, so how can a pointcut match any join points at compile time when the program isn't running? The answer is that it doesn't; but what the compiler can do, is look at a line of code containing a call to the `notifyListeners` method and say "when this program executes, that line of code is going to give rise to a join point that will match this pointcut." It is the results of this static analysis that display as warnings. Chapter 6 covers some kinds of pointcut expressions that cannot be fully evaluated at compile time. (They require a runtime test to confirm the match.) These kinds of pointcut expressions cannot be used with the `declare warning` statement (which obviously needs to know at compile time whether or not there is a match).

## 2.6 Implementing the Notification Policy

Section 2.5 showed you how to declare how you would like the world to be. This is the section where we get to make it that way. Remember that when we started on this journey, we had the simple requirement that whenever the state of a policy is updated, we should notify all of its listeners. It seems that a useful next step would be to write a pointcut that captures all the join points where the state of a policy is updated. We could call it `policyStateUpdate`:

```
pointcut policyStateUpdate() :
    execution(* set*(..)) && this(PolicyImpl);
```

This pointcut defines a `policyStateUpdate` to be the execution of any method whose name begins with “set,” returning any value and taking any arguments. In addition, the object executing the method must be an instance of `PolicyImpl`. Because we have been following the JavaBeans naming convention in our domain model, this pointcut matches the set of state-updating methods in the policy class hierarchy very well. If we type this pointcut declaration into the editor buffer and save it, the editor should now look like Figure 2.28.



**Figure 2.28** Adding the `policyStateUpdate` pointcut.

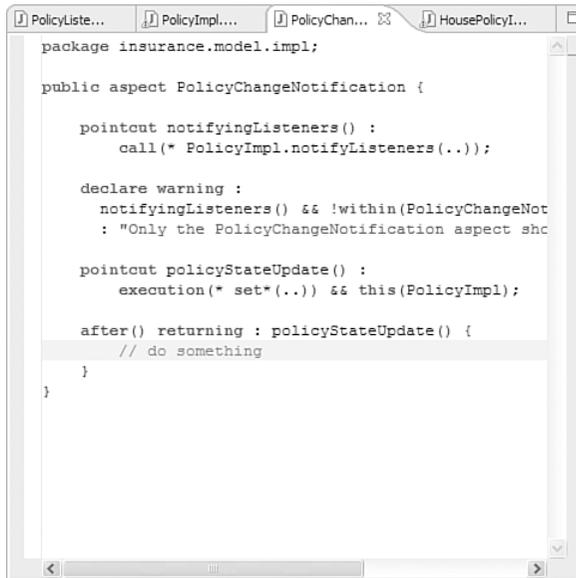
We have a way of matching all the join points where the state of a policy is updated. Now all we need to do is find a way to specify some action to take at those join points (notifying listeners). What we need is *advice*.

## 2.6.1 Introducing Advice

Pointcuts match join points, but advice is the means by which we specify what to do at those join points. AspectJ supports different kinds of advice—*before* advice enables you to specify actions to take before a matched join point, *after* advice enables you to specify actions to take after a matched join point, and *around* advice gives you complete control over the execution of the join point. In our case we want to notify listeners after returning from a `policyStateUpdate`:

```
after() returning : policyStateUpdate() {  
    // do something  
}
```

Figure 2.29 shows what happens when we type this into the editor buffer and save it. You can see that `after` and `returning` are AspectJ keywords. Also notice the similarities between the advice block and a method—both can take parameters (although we have none here yet), and both specify a block of code to execute when they are called. A key difference though is that methods are called explicitly, whereas the advice is implicitly invoked by AspectJ whenever a join point matching its associated pointcut expression occurs. Chapter 7 contains a full discussion of advice in AspectJ.



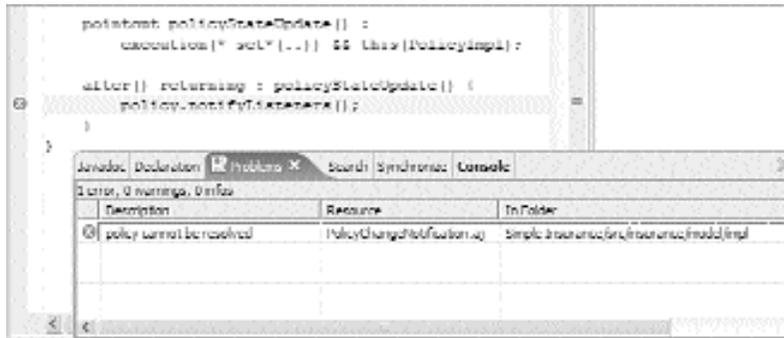
**Figure 2.29** Adding advice to the aspect.

### 2.6.2 Calling the Notify Method

Finally we get to implement the advice body and put in the call to `notifyListeners`. All we need to do is put in a call to `policy.notifyListeners()` in the body of the advice:

```
after() returning : policyStateUpdate() {
    policy.notifyListeners();
}
```

If we enter this into the editor buffer, and save, the compiler tells us that there is a small problem with our implementation as it stands (see Figure 2.30). “policy” cannot be resolved, the variable is not defined. How can the advice get ahold of the policy object whose state has just been updated?

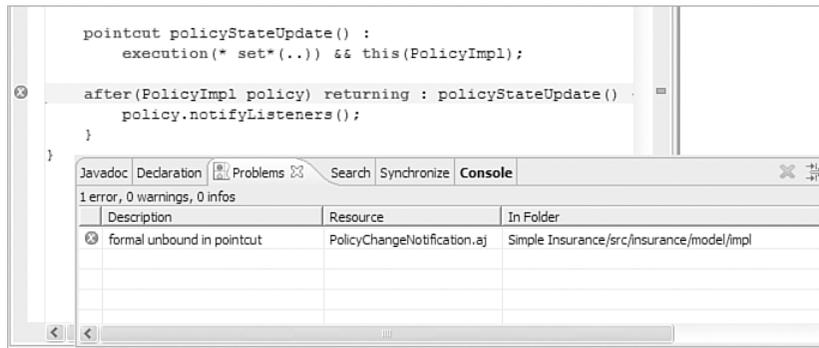


**Figure 2.30** “policy cannot be resolved.”

We need to pass the policy object into the advice as a parameter, which is done in the same way as specifying parameters for methods:

```
after(PolicyImpl policy) returning : policyStateUpdate() {
    policy.notifyListeners();
}
```

Let’s try that out in the editor. Figure 2.31 shows the result: a “formal unbound in pointcut” error.



**Figure 2.31** Formal unbound in pointcut.

What could that mean? Recall that unlike a method, there are no explicit calls to advice. So if you do not call the advice explicitly, passing in the parameters it needs, from where does the advice get its parameter values? The answer is that the advice parameter values have to be provided by the pointcut: When the pointcut matches a join point, it needs to extract some information from that join point (in our case, the policy object that has just been updated), and pass it into the advice. The error message is telling us that the “formal” (advice parameter) is not “bound” in the pointcut—or to put it another way, the pointcut is not giving the advice the parameter it needs yet.

Take another look at the definition of the `policyStateUpdate` pointcut:

```

pointcut policyStateUpdate() :
    execution(* set*(..) && this(PolicyImpl);

```

This matches any join point that is the execution of a method whose name begins with “set,” where the currently executing object (the object bound to “this” within the method body) is an instance of `PolicyImpl`. What we need is for the pointcut to tell us not just whether the currently executing object is an instance of `PolicyImpl`, but which instance it is. The set of values provided by a pointcut when it matches a join point is specified in its parameter list:

```

pointcut policyStateUpdate(PolicyImpl aPolicy) : ...

```

(So that’s what those parentheses after the pointcut name are for!) Now it just remains to specify where the value of the policy parameter comes from. This is done via name binding in the pointcut expression:

```
pointcut policyStateUpdate(PolicyImpl aPolicy) :
    execution(* set*(..)) && this(aPolicy);
```

This revised pointcut expression matches the same join points as the previous version (the execution of any method whose name begins with “set,” and where the object executing the method is an instance of `PolicyImpl`), but also makes available the actual `PolicyImpl` object at each join point it matches.

We are nearly there now; we just need a way to say that the policy object provided by the pointcut when it matches a join point should be matched to the `policy` parameter we specified in the advice definition. This is done by name binding, too:

```
after(PolicyImpl policy) returning :
    policyStateUpdate(policy)
{
    policy.notifyListeners();
}
```

If we make these changes in the editor buffer, the aspect compiles successfully. Figure 2.32 shows the completed aspect in the editor.



**Figure 2.32** The completed *PolicyChangeNotification* aspect.

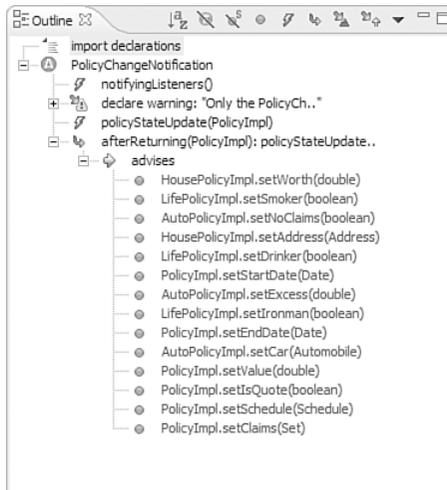
The next section shows you how you can use the tools to understand the effect of the advice that we just wrote.

## 2.7 Showing Advice In AJDT

AJDT contains a lot of features designed to help you understand and be aware of the effects of advice in your program. The primary means are the editor and Outline view, but there is also a visualization view and the capability to generate documentation.

### 2.7.1 The Outline View

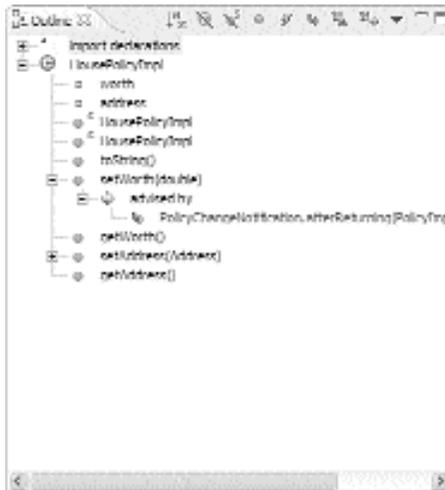
Let's look at the Outline view for the `PolicyChangeNotification` aspect. This is shown in Figure 2.33, with the node for the `after` returning advice fully expanded. Notice how the Outline view shows us all the places that the advice advises. To be slightly more precise, the Outline view is showing us all the places in the program source code that will give rise to a join point at runtime that will be matched by the pointcut expression associated with the advice. Just like the `declare warning` matches in the Outline view, these matches are links, too. If you click one, it will take you to the corresponding source location.



**Figure 2.33** Showing the effect of advice in the Outline view.

If we click the first match in the list, `HousePolicyImpl.setWorth(double)`, the editor opens on the source file containing the definition of the `HousePolicyImpl` class, at the `setWorth` method. The Outline view updates to display the outline for the `HousePolicyImpl` class, as shown in Figure 2.34.

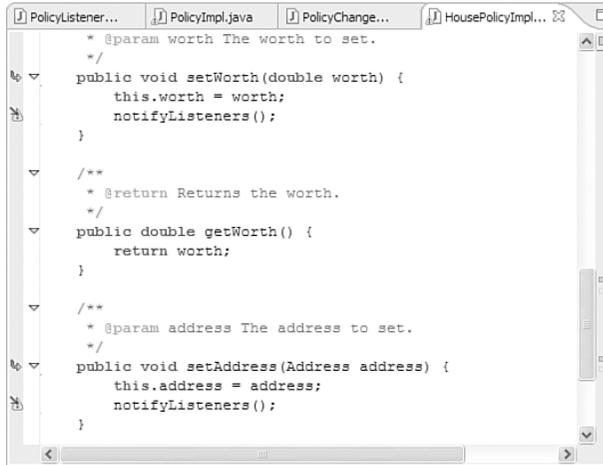
The node for the `setWorth` method has been expanded, and we can see that it has been “advised by” the `afterReturning` advice in the `PolicyChangeNotification` aspect. The `setAddress` method has also been advised, the plus sign (+) next to its node in the tree view gives us a cue, and if we expanded it we would see the “advised by” relationship as we do for `setWorth`. Once again the entries shown under “advised by” are hyperlinks that can be used to navigate to the definition of the advice affecting the method, simply by clicking.



**Figure 2.34** An advised method in the Outline view.

### 2.7.2 The Editor

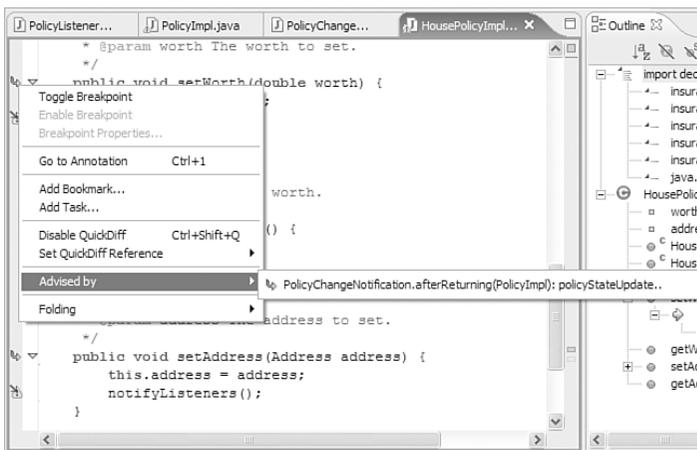
Now let’s look at the editor buffer we just opened on `HousePolicyImpl.java`. This is shown in Figure 2.35. There are two key points to note in this figure. First, in the left margin next to the `setWorth` and `setAddress` methods, we see an advice marker (the arrow pointing down and to the right). This tells us that there is after advice in place on these methods. In other words, after returning from the execution of the `setWorth` and `setAddress` methods, some advice will run.



**Figure 2.35** Advised methods in the editor.

In the right margin, we see the overview ruler for the whole source file. In this margin, there are check marks indicating the presence of advice (and also the warnings generated by the declare warning statement). You can click these check marks to navigate to the corresponding location in the source file.

Returning our attention to the advice markers in the left margin once more, they tell us that the method has after returning advice, but not which advice. As we have already seen, we can use the Outline view to find that out. Another option is to right-click the advice marker to bring up the context menu, as shown in Figure 2.36.

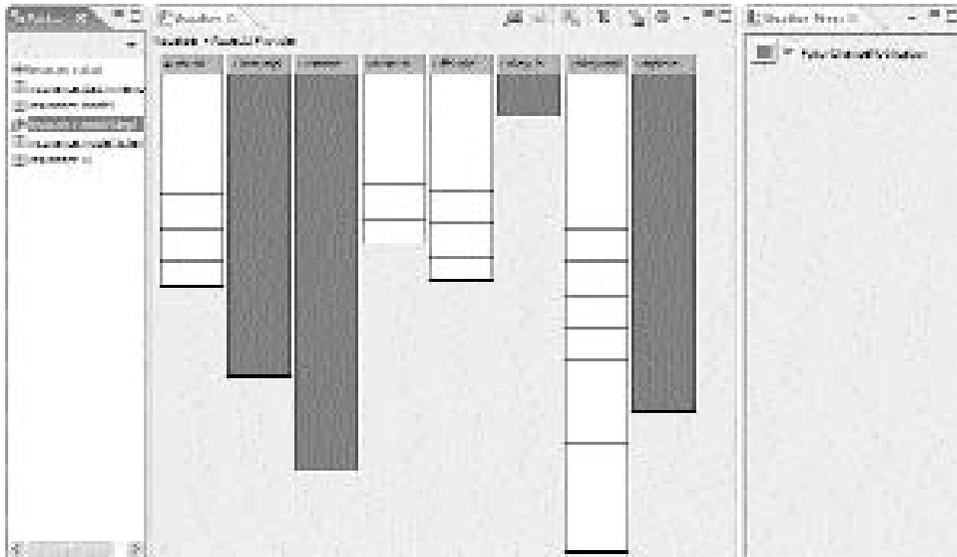


**Figure 2.36** Using the context menu to discover advice.

Selecting the **Advised by** option in the menu brings up a list of all the advice affecting the method. Selecting one of the items in the list (in Figure 2.36 there is only one) opens the editor on the advice declaration.

### 2.7.3 Documentation and Visualization

AJDT also provides a visualization view that gives you an overview of the source files in your application and where the advice defined in aspects matches lines of source code that will give rise to advised join points at runtime. Figure 2.37 shows an example of the kind of views it produces.



**Figure 2.37** Visualization view.

Each vertical bar represents an individual source file. The highlighted bars contain lines of code that will give rise to join points matched by advice at runtime. The stripes on the highlighted bars indicate the places that advice applies. You can click these to navigate. In this example, the Visualiser is highlighting the classes in the policy hierarchy. Chapter 12 provides more detail on using the Visualiser.

A final way of understanding the effects of advice is through the documentation that AJDT produces via the `ajdoc` tool (`ajdoc` is AspectJ's version of the `javadoc` tool). We show you how to use AJDT to generate documentation in Chapter 4.

## 2.8 Evaluating the Implementation

We can see then from the tools that the advice we wrote seems to be calling `notifyListeners()` at all the right times. We're not finished yet though because we still have all the old calls to `notifyListeners()` scattered throughout the policy classes, and we haven't run the test suite.

We want to know that all the tests would still pass with the `PolicyChangeNotification` aspect in place, and all the old calls to `notifyListeners()` removed. Ideally we would like to know that now, before we actually go ahead and remove all those calls, because that will be a smaller step if we find we have to backtrack. We are going to show you an AspectJ technique you can use to do this. Remember we said that AspectJ supports several kinds of advice, including around advice. Around advice gives you complete control over the execution of a matched join point. One of the things you can do with around advice is decide whether and when the computation at the matched join point should proceed. A “no-op” around advice implementation looks like this:

```
Object around() : somePointcut() {
    return proceed();
}
```

It does nothing before proceeding with the computation at the join point, and returns immediately that computation has completed. As a transition stage in the development of the `PolicyChangeNotification` aspect, we can add the following around advice to the aspect:

```
void around() : notifyingListeners() &&
    !within(PolicyChangeNotification)
{
}
```

Because it contains no call to proceed, this advice has the effect of bypassing the computation at join points it matches. We previously defined the `notifyingListeners` pointcut to match all calls to `notifyListeners`, so this advice effectively removes all those calls that aren't made by our aspect from the runtime execution of the program—it lets us run the test cases and see what would happen if the calls weren't there. We should stress at this point that we are only using empty around advice as a transition stage in our refactoring. We do not advise you to create programs that use around advice to “stub out” unwanted calls as a permanent part of the design. See the next chapter for an example of the use of around advice as part of the program design.

With this temporary around advice in place, we can run the test suite. The results are shown in Figure 2.38. A test case has failed!

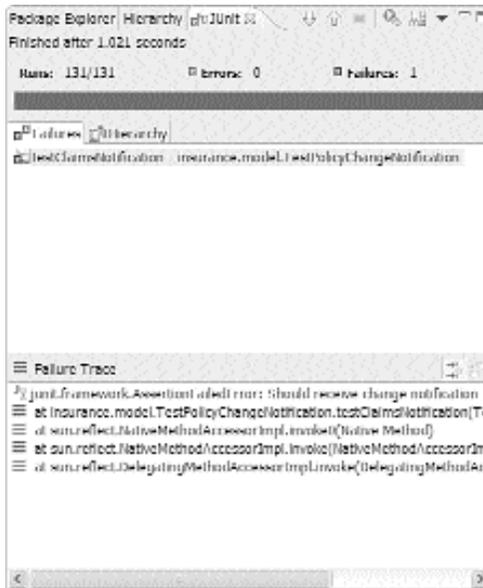


Figure 2.38 A failing test case.

What's going on? We can double-click in the JUnit view to go to the test case definition. The editor opens on the test case, with the failing assert highlighted (see Figure 2.39).

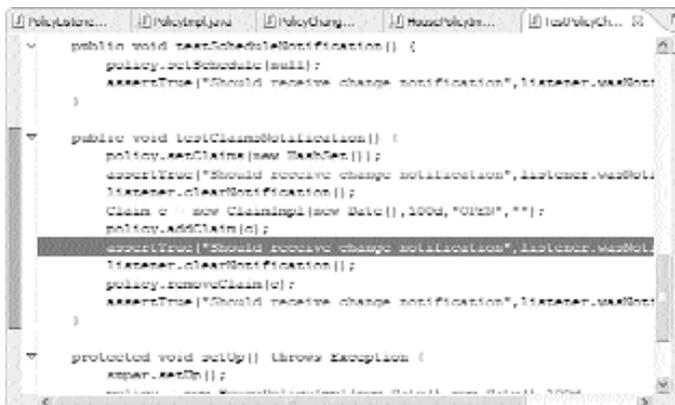
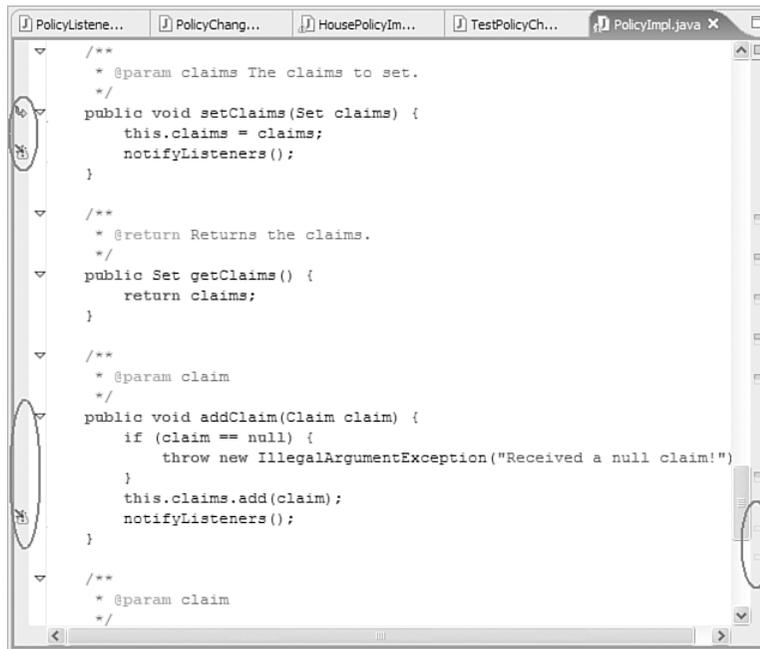


Figure 2.39 The failing test case definition.

You can see that the test case adds a new claim against a policy, and then checks to see that a notification of update was received. The method called to add the new claim against the policy is `addClaim`. We defined the `policyStateUpdate` pointcut to match the execution of methods beginning with “set” on policy objects. “addClaim” does not fit this pattern—our pointcut definition is not quite correct. If we open the editor on the `PolicyImpl` class, we can see what’s going on (see Figure 2.40).



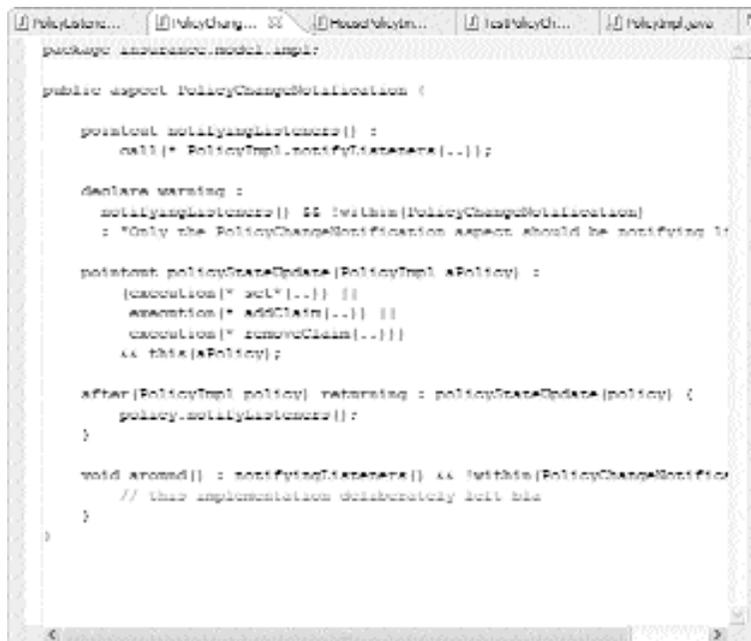
**Figure 2.40** Clues in the editor.

Notice that the `setClaims` method has an after advice marker next to it in the left margin. The `addClaim` method has no such advice marker—although it does have a warning marker because the old implementation contains a call to `notifyListeners`. Another clue is in the ruler in the right margin. You can see that the advice markers and warning markers are nicely paired throughout the file, apart from the last two (highlighted), which have warnings but no advice markers. These warnings are against the `addClaim` and `removeClaim` methods.

Aspects aren't a silver bullet—you can use them to improve the modularity of your programs, but they don't alleviate the need for test-driven development or any of the other best practices you have learned from working with Java.

### 2.8.1 Updating the Pointcut Declaration

Let's go back to the `PolicyChangeNotification` aspect and update the definition of the `policyStateChange` pointcut. We want to match join points that represent either the execution of a “set” method, or the “addClaim” method, or the “removeClaim” method. Figure 2.41 shows the updated pointcut definition.

A screenshot of an IDE window showing the source code for the `PolicyChangeNotification` aspect. The code is as follows:

```
package insurance.model.impl;

public aspect PolicyChangeNotification {

    pointcut notifyingListeners() :
        call(* PolicyImpl.notifyListeners(..));

    declare warning :
        notifyingListeners() && !within(PolicyChangeNotification)
        : "Only the PolicyChangeNotification aspect should be notifying it";

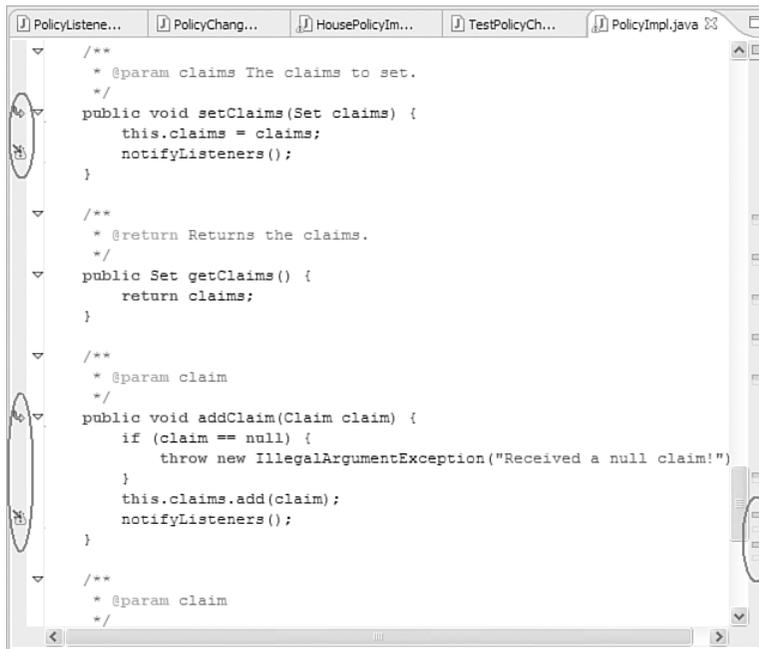
    pointcut policyStateUpdate(PolicyImpl &Policy) :
        execution(* set*(..)) ||
        execution(* addClaim(..)) ||
        execution(* removeClaim(..))
        && this(&Policy);

    after(PolicyImpl policy) returning : policyStateUpdate(policy) {
        policy.notifyingListeners();
    }

    void around() : notifyingListeners() && !within(PolicyChangeNotificatio
    // this implementation deliberately left blank
}
}
```

Figure 2.41 The updated pointcut definition.

We can use either the Outline view or the editor to check that we are now indeed matching the execution of the add and remove claim methods. Figure 2.42 shows how the `PolicyImpl.java` source file looks in the editor now.



**Figure 2.42** New matches as a result of the updated pointcut.

Notice that the advice markers are now appearing next to the `addClaim` and `removeClaim` methods. In the right margin you can also see that the advice markers and the warnings are now balanced: every time there is a call to `notifyListeners` coded in the `PolicyImpl` class, there is also advice in effect to achieve the same result. Now when we re-run the test cases, they all pass.

### 2.8.2 Removing the Old Calls to Notify

Now the time has come to remove all the calls to `notifyListeners` that are scattered throughout the classes in the policy hierarchy. We can use the warning tasks in the Problems view (see Figure 2.27) to navigate to all the offending places and remove the call. Now we can take out the around advice as well, and the job is done. Save all the files and re-run the test cases—they all pass.

Figure 2.43 shows the completed `PolicyChangeNotification` aspect.

A screenshot of an IDE window showing the source code for a Java aspect named PolicyChangeNotification. The code is as follows:

```
package insurance.model.impl;

public aspect PolicyChangeNotification {

    pointcut notifyListeners() :
        call(* PolicyImpl.notifyListeners(..));

    declare warning :
        notifyListeners() :: !within(PolicyChangeNotification)
        : "Only the PolicyChangeNotification aspect should be notifying.";

    pointcut policyStateUpdate(PolicyImpl pPolicy) :
        [execution(* act*(..)) ||
         execution(* addClaim(..)) ||
         execution(* removeClaim(..))]
        && this(pPolicy);

    after(PolicyImpl policy) returning : policyStateUpdate(policy) {
        policy.notifyListeners();
    }

}
```

Figure 2.43 The finished aspect.

### 2.8.3 Comparing the Modular and Non-modular Implementations

While working through the tasks in the Problems view removing all the unwanted calls to `notifyListeners`, we noticed an interesting case in the `LifePolicyImpl` class. This is shown in Figure 2.44.



```
    }

    /**
     * @param drinker The drinker to set.
     */
    public void setDrinker(boolean drinker) {
        this.drinker = drinker;
        notifyListeners();
    }

    /**
     * @return Returns the drinker.
     */
    public boolean isDrinker() {
        return drinker;
    }

    /**
     * @param ironman The ironman to set.
     */
    public void setIronman(boolean ironman) {
        this.ironman = ironman;
    }

    /**
     * @return Returns the ironman.
     */
    public boolean isIronman() {
        return ironman;
    }
}
```

**Figure 2.44** Ironman.

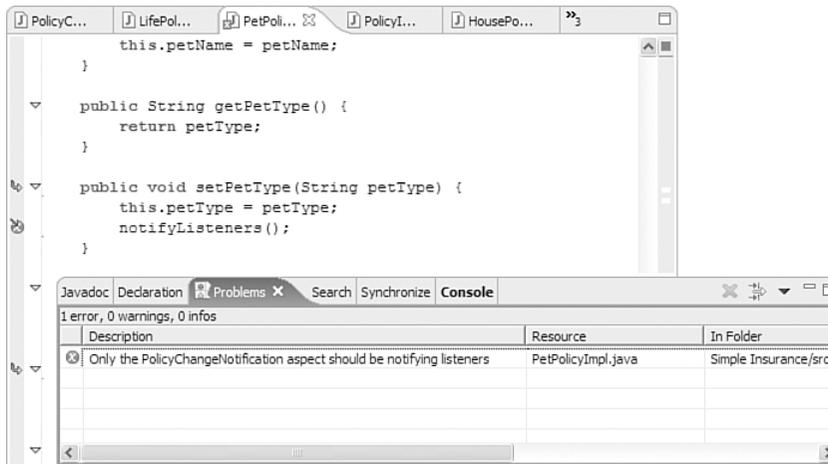
Notice that the `setDrinker` method contained a call to `notifyListeners`, but the call to `setIronman` a little farther down in the source file did not. This was a bug in the original implementation. How did this happen? Originally Simple Insurers Inc. just sold ordinary life policies pretty much like every other insurance company. Sometime after the original `LifePolicyImpl` class was written, the marketing department decided that anyone fit enough to compete in an Ironman competition ought to be a pretty good prospect for life insurance, and decided to go after the market niche with special discounts. The programmer that added the `ironman` field into the `LifePolicyImpl` class forgot to add in the call to `notifyListeners`.

The `PolicyChangeNotification` aspect *does* notify listeners after the `ironman` field has been updated. Because the pointcut specifies by property (all the `set` methods) rather than by exhaustive enumeration when a notification should be issued, it gets it right. In our experience, it is fairly common for an aspect-based implementation such as this to be more accurate than the scattered, hand-coded alternative.



Notice that all the methods that update the state of the `PetPolicyImpl` class are being advised by the `PolicyChangeNotification` aspect. The aspect continues to correctly notify changes even in code added to the system after the aspect was written. This is the power of capturing the design for change notification in the code (whenever the state of a policy changes), rather than coding by hand the implications of the design (putting calls to `notifyListeners` throughout the policy hierarchy).

If the `PetPolicyImpl` programmer somehow manages to miss the advice markers in the editor gutter and the “advised by” relationships in the Outline view, and starts to implement change notification the old way, the `PolicyChangeNotification` aspect soon tells him or her (see Figure 2.46).



**Figure 2.46** Catching a violation of the change notification design.

## 2.10 Summary

We’ve come a long way in this chapter. We identified a problem in the Simple Insurance application that came to light when we decided to add support for pet insurance to the application, whereby calls to `notifyListeners` were spread throughout the policy class hierarchy. We decided to refactor this implementation to get back to a one-to-one mapping from design requirement (notify whenever the state of a policy object changes) to implementation. We applied the eXtreme programming philosophy “make sure everything is expressed once and only once,” also known as the DRY principle (*don’t repeat yourself*). Using

AJDT and AspectJ, we were able to implement a modular solution to change notification using an aspect. The tools helped us during the refactoring process, both to explore the code and to incrementally test the changes we were making.

When we had finished, all the test cases were passing, we had replaced 15 calls to `notifyListeners` with one single call, found and removed a bug, and left behind a guard (the `declare error`) so that our chosen design modularity would remain in place in the code during evolution and maintenance. Even better, the solution we implemented in the aspect continued working even when the new `PetPolicyImpl` class was added—there was no additional effort spent implementing change notification for pet policies.

There is more that we could do to modularize change notification for policies—keeping track of listeners, and adding and removing them, also really belongs in the `PolicyChangeNotification` aspect. Chapter 8 shows you how to do that, too.

