Develop aspect-oriented Java applications with Eclipse and AJDT

Updated tools make AOP easier for beginners and veterans alike

Matt Chapman
Helen Hawkins

September 21, 2004

AspectJ is an aspect-oriented extension of the Java™ language that enables a modular implementation of crosscutting concerns. This crosscutting behavior, which can be static or dynamic, presents an extra challenge to tools that support AspectJ. The AJDT project aims to provide Eclipse platform-based tools for AspectJ, consistent with the Eclipse Java Development Tools (JDT), with additional capabilities for visualizing and understanding the crosscutting nature of aspect-oriented applications. In this article, AJDT contributors and IBM aspect-oriented software development team members Matt Chapman and Helen Hawkins introduce you to AJDT. You'll learn how to install the tools; how to create, run, and debug AspectJ applications; and how to visualize and navigate the crosscutting structures inherent to aspect-oriented programming.

The AspectJ Development Tools for Eclipse (AJDT) is an open source Eclipse Technology Project that provides the tooling required to develop and run AspectJ applications. We believe good tools have a key role to play in realizing the full benefits of aspect-oriented programming, and particularly in helping newcomers understand the concepts involved.

After reading this article, we hope you will have a good feel for the support that AJDT provides for using AspectJ. We'll start by showing you how to get up and running with AJDT. Then we'll dive straight into things, creating an AspectJ application from scratch. You'll see how to create aspects and Java classes, and how to run the application. Our first example is very simple to keep the focus on the tools and introduce the different visual and navigational features that AJDT provides to help with your development. Even so, this simple application illustrates one real-world use of aspects.

We then move onto a slightly more complex example so that we can explore the tools in more detail, and show you how to work with existing projects. This example has several aspects that are integral to the application, as well as a debug aspect that we'll only want to include if we're actually debugging the application. We show how AJDT enables you to selectively apply and remove this debug aspect. After that, we visit the powerful Visualization perspective, to gain a higher-level
understanding of how the many different aspects affect the rest of the application. Finally, we look at a couple of more advanced features of the tools, including those for debugging AspectJ applications and generating documentation.

Throughout this article, we are assuming some basic familiarity with AspectJ and aspect-oriented programming (AOP). For introductory material on AOP and programming in AspectJ, please see the Related topics; alternatively, you can install AJDT and follow the Help links to AspectJ, where you will find the AspectJ documentation.

**Installing AJDT**

To follow along with this article, you'll need to be running AJDT version 1.1.11 or later, which requires Eclipse 3.0 (final). There is an earlier version of AJDT available that works with Eclipse 2.1; however, many of the features we discuss in this article were added in version 1.1.11.

To install the latest AJDT for Eclipse 3.0, follow these steps:

1. Start Eclipse, then go to Help>Software Updates>Find and Install....
2. Search for new features to install and add a new remote site called AJDT Update Site with the URL http://download.eclipse.org/technology/ajdt/30/update. If you need a proxy server to access the Internet, set your proxy preferences via Window>Preferences>Install/Update.
3. Expand the AJDT Update Site node and select AspectJ. Choose to install the Eclipse AspectJ Development Tools, then read and accept the copyright statement.
4. Click Finish.

As of version 1.1.11, AJDT is fully integrated into the Eclipse 3.0 framework, so there are several ways to make sure that it is properly installed. The most useful of these is the AJDT documentation's integration with both the Eclipse help and the Eclipse Platform welcome pages. To navigate to both the AJDT help and the documentation for AspectJ 1.2 select Help>Help Contents. To see the AJDT additions, select Help>Welcome and choose any one of the Overview, Tutorials, Samples, or What's New links. In particular, following the Tutorials>Build a simple AspectJ application link activates the Simple AspectJ Application cheat sheet. Cheat sheets are a new feature in Eclipse 3.0, and the AspectJ cheat sheet extends the Java "Hello World" application detailed in the Java cheat sheet. (You can also access the cheat sheet by going to Help>Cheat Sheets and selecting the one you want.)

**First steps**

Now that you've installed AJDT, let's explore how it supports the creation and running of AspectJ applications. To this end, we'll create a simple AspectJ application consisting of just one aspect and one class, and in so doing examine the AJDT features that we'll need to achieve this. The application is slightly contrived, but it serves our purpose. In it, we simply calculate the square of two numbers. At first, every time the program is given a number, it calculates the square from scratch -- and we've added a Thread.sleep() call within this method to ensure that this takes some time. By adding a cache aspect, however, we can check to see whether the result has already been calculated for a given input. If so, the cached value is returned; if not, then the
square is calculated as normal and the result is added to the cache. Thus, the cache aspect will improve our application's performance. (For a discussion on using aspects for caching, please follow the Aspects Blog link in the Related topics section.)

If you've used the JDT Eclipse tools for Java development, then using AJDT for AspectJ development will be a familiar experience. To create a new AspectJ project, follow these steps:

1. Select Window>Open Perspective>Java to open the Java perspective.
3. Call the project Caching Example, click Next, and add a source folder named src.
4. Click Finish to create the new project in your workspace.

Note: If this is the first AspectJ project you've created in your workspace, the AJDT Preferences Configuration wizard will appear after step 4. This wizard configures some Eclipse settings that will make your life easier. Accept the wizard's defaults and click Finish.

Expand the project node in the Package Explorer and you'll see that the AspectJ runtime library aspectjrt.jar has been added in the same way that the JRE system library was. Now, select your project and go to File>New>Package to create a package called caching, and go to File>New>Class to create a class called Application in your caching package. This is just the same as if the project were a Java project.

Now edit your class so that it looks like Listing 1.

**Listing 1. Main class for our application**

```java
package caching;

public class Application {
    public static void main(String[] args) {
        System.out.println("The square of 45 is " + calculateSquare(45));
        System.out.println("The square of 64 is " + calculateSquare(64));
        System.out.println("The square of 45 is " + calculateSquare(45));
        System.out.println("The square of 64 is " + calculateSquare(64));
    }

    private static int calculateSquare(int number) {
        try {Thread.sleep(7000);} catch (InterruptedException ie) {} return number * number;
    }
}
```

From the code in Listing 1, you can see that in our class we have a `main()` method that makes four calls to the `calculateSquare()` method, the second pair of calls being a repeat of the first pair. The `calculateSquare()` method does nothing for a bit and then returns the square of the input argument.

Before adding our cache aspect, let's run the application. As with a Java application in a Java project, to run the application you need to select in the Package Explorer the Java class that contains the `main()` method. In our case, this is our only class: Application.java. Once you've
selected this class, right-click it to open the contextual menu and select Run>Java Application. You'll notice that it takes a while to calculate the squares.

Now let's add an aspect to our application. The process of creating an aspect has the same look and feel as the process of creating a Java class. To create an aspect called cache, go to File>New>Aspect to open the New Aspect wizard. As with the New Java Class wizard, you'll need to choose the package in which to create your aspect, give it a name, and then click Finish. In the Package Explorer, you should see that a file called Cache.aj has been created. By default, aspects are created in files with the .aj extension, although you can have aspects in .java files, too.

Edit Cache.aj so that it contains the code in Listing 2.

Listing 2. An aspect to add a cache to the our application

```java
package caching;
import java.util.Hashtable;

public aspect Cache {
  private Hashtable valueCache;

  pointcut calculate(int i) : args(i) && (execution(int Application.calculateSquare(int)));

  int around(int i) : calculate(i) {
    if (valueCache.containsKey(new Integer(i))) {
      return ((Integer) valueCache.get(new Integer(i))).intValue();
    }
    int square = proceed(i);
    valueCache.put(new Integer(i), new Integer(square));
    return square;
  }

  public Cache() {
    valueCache = new Hashtable();
  }
}
```

From the code in Listing 2, you can see that in our cache aspect, we first create a hashtable to record the numbers and their squares. We then specify a pointcut called calculate. This matches the execution of any method named Application.calculateSquare() that accepts and returns an integer value. Because we only want to execute the Application.calculateSquare() method if the square for the given number hasn't already been calculated, we'll use around advice. The advice takes an integer argument and returns an integer value. In the body of our advice, we first check to see if the input value is in the hashtable. If it is, we return the corresponding result; if it isn't, then we proceed with the execution of the calculateSquare() method. Before returning from the around advice, we update the hashtable with these new values.

If you have Eclipse set to build projects automatically, all that's left is to run your AspectJ application! If this is not the case, select your AspectJ project in the Package Explorer and click the Build button, as shown in Figure 1. You're now ready to go. Since you've already run an earlier version of this application, you should have a run configuration all set up. Therefore, to run the AspectJ application, select Run button from the task bar (Figure 1), then select Application. You'll notice that it now takes much less time to calculate the squares the second time around!
Develop aspect-oriented Java applications with Eclipse and AJDT

The Outline view and editor markers

Now that we've written and run our simple AspectJ application, let's have a look at how AJDT helps you to understand and work with it. With the cache aspect open in the editor, take a look at the Eclipse Outline view, which should look like Figure 2.

Figure 2. Outline view of the Cache aspect

The Outline view shows the structure of the current document. In the case of our cache aspect, we have an import declaration, and then, instead of a class, we have an aspect called cache. You will notice that directly underneath cache there are four different nodes, each corresponding to a feature within the aspect. The first of these is the valueCache private field -- remember, aspects can contain methods and fields, just like classes. Next we have something a little different: a lightning bolt icon labelled calculate(int). This signifies a pointcut; click on it and the editor will jump to the appropriate line. Things get more interesting with the next node, representing the around advice. Expand this node, and you'll see an "advises" node, which itself expands to reveal a private static method called Application.calculateSquare(int). Here, the AspectJ-enhanced Outline view takes us beyond the basic structure of the current source file, showing us the relationship this construct (the around advice) has with another construct in the system -- a method that is being advised, in this case. Furthermore, the target of the relationship is a hyperlink -- simply click on the Application.calculateSquare(int) node, and the editor will switch to the containing source file (Application.java) and highlight the method that is being advised.

Now that we have Application.java open in the editor, the Outline view should look like Figure 3.
Initially, the Outline view for this source files looks pretty much as it would in the normal Java world, without AJDT. But notice that the node for the calculateSquare() method can be expanded to reveal an "advised by" node, and then a node showing the source of the advice -- in this case, the around advice in the Cache aspect (which is a link, as before).

Now, go back to the editor itself, which is showing the Application class. In the left edge of the window (known as the gutter), there is a marker next to the calculateSquare() method declaration, with the same icon as in the Outline view. There is also an orange block marking the same method declaration in the right-hand bar of the editor. Both show that some advice is taking effect here. Move your mouse arrow over these indicators; you'll get a message that tells you that they are marking a location advised by the around advice in the Cache aspect. Finally, you can right-click on the advice marker and open the “Advised by” menu to see the appropriate advice. Select this advice to navigate to it, as shown in Figure 4.

That's all it takes to create, run, and explore the crosscutting structure of a simple AspectJ project using AJDT!
The Spacewar example

Now that we have introduced AJDT with a simple example, it's time to explore the support it offers to AspectJ programmers working on slightly larger projects. We will revisit the Outline view, see how to selectively add and remove aspects from an application, and finally see how to get an overview of a project using the powerful Visualization perspective.

Adding aspects to existing Java projects

If you have an existing Java project to which you want to add aspects, you can easily convert it to an AspectJ project. Just select the project you wish to convert in the Package Explorer, right-click to display the contextual menu, and select Convert to AspectJ Project. Similarly, if for some reason you wish to convert it back to a Java project, select your AspectJ project in the Package Explorer and select Remove AspectJ Nature from the contextual menu.

Helpfully, AJDT includes the examples from the core AspectJ distribution, plus a wizard that makes adding them to your workspace very easy. The Spacewar example is one of the larger examples, and it serves our purpose well. (Again, for this portion of the article, you will need AJDT version 1.1.11 or later, as the Spacewar example wasn't included in earlier versions.) Let's get right to it:

1. In Eclipse, go to File>New>Other.
2. In the resulting dialog, expand the AspectJ folder, then the AspectJ Examples folder, and select Spacewar Example.
3. Click the Next button. You can enter a name for the project or accept the default.
4. Click Finish and the New Examples wizard will create an AspectJ project for you, add all of the required files for this example to it, and switch to the Java perspective, if you're not there already.

If you have Eclipse set to build automatically, the example will also be compiled -- if not, you'll need to trigger the build yourself, as we did above with our simple application.

Let's check to see that everything came out OK, and see what the example looks like when it runs. The main() method for the example is in Game.java in the spacewar package; to launch it, select the file, and go to Run>Run As>Java Application. Two game windows should then appear. You use the cursor keys and space bar to control the game. The rest you can figure out for yourself!

We will now explore some of the Spacewar code to see more of the Outline view and editor markers in action. Let's take a look at the SpaceObjectPainting aspect, which is defined as an inner aspect in the Display1 class. Open Display1.java in the editor and expand the node for the SpaceObjectPainting aspect in the Outline view, as shown in Figure 5.
Figure 5. Outline view of SpaceObjectPainting aspect

Here we see several icons, different from the around advice icon we saw earlier. The first represents an abstract inter-type declaration -- the aspect is defining a `paint()` method on behalf of the `SpaceObject` class. You can see this by expanding the node. Next is another inter-type declaration, for a field this time, `ship.color`. Next comes some after returning advice declared on an anonymous pointcut, which applies when a method returns normally -- that is, without throwing an exception. Some of the other icons you might see in the Outline view, and as markers in the gutter of the editor, are:

- Before advice (▲)
- Declare parents (▼), where a class is defined to implement an interface, or extend a class
- Declare warning (△) or declare error (□), which can be used to trigger a compiler warning or error if a certain policy is not adhered to
- Declare soft (☆), where a checked exception has been softened to an unchecked exception

Build configurations

Some aspects form an integral part of a system, whereas others are used to provide particular features, such as debugging support, that are only required some of the time. Thus, you'll often want to selectively add and remove aspects from an application. AJDT provides this capability using the concept of build configurations, which define the source files within a project that are passed to the compiler. The tools make it easy to work with build configurations by enabling direct manipulation of source files from the Package Explorer within Eclipse. The Spacewar example has a debug aspect that is initially removed from the build configuration. If you look closely at the Package Explorer, as shown in Figure 6, you will see that the icon for Debug.java is hollowed out, indicating its exclusion. Notice also that the icon for the spacewar package is partially emptied out, to indicate that some, but not all, of the package has been excluded.
You can add the debug aspect to the project simply by right-clicking the Debug.java file and selecting the **Include in "demo" configuration** menu option. The source and package icons will now fill in, as shown in Figure 7, and the project will be built to include this file (unless Eclipse autobuilding is disabled).

If you now run the Spacewar program again, you'll see a new window, with debugging information.

To exclude an aspect, simply right-click the source file again and select the **Exclude** option. You can also hold down the Shift or Ctrl keys to select multiple source files, or even entire packages, to include or exclude. Changes made in this way affect the current, or **active**, build configuration, but sometimes it is useful to preserve a particular configuration for later use. This is done by storing build configurations in files with the .ajproperties extension. If you look in the Package Explorer for the Spacewar code, you will see two such files, called demo.ajproperties and debug.ajproperties, as shown in Figure 8. One of the icons has a triangle in it, which indicates that it is the active configuration. Any changes made to the configuration using the Include and Exclude contextual menu options are written to the currently active configuration file.
To switch to a different build configuration, simply select the appropriate properties file, right-click, and select **Activate this configuration** from the contextual menu. The project is then rebuilt with the new settings. Another way to switch between build configurations is to select **Project>Activate build configuration**, which displays a submenu with all of the build configurations that exist for the current project. This submenu also has a Save as... option that allows you to create new build configurations by writing out the current configuration to a new file.

Finally, you can also edit configuration files directly. Double-click on a build configuration file to open it in the associated build configuration editor. This action presents a tree view of the source files in the project, with checkboxes to add or remove resources from the configuration, as shown in Figure 9.

### Aspect Visualization perspective

If you're going to write aspect-oriented applications, you'll benefit from new ways of visualizing those applications. You need to be able to understand the relationships between your aspects and classes, and get an overall feeling for how your application behaves. For instance, how many classes do your aspects affect? As already described, AJDT provides advanced capabilities for navigating around your code, showing what is being advised, and so on. Building on this foundation, AJDT also provides its own Aspect Visualization perspective. This feature is specifically designed to help you view the crosscutting concerns in your application. To activate
this perspective, go to **Window>Open Perspective...** and select **Aspect Visualization**. You should get a display that looks like Figure 10.

**Figure 10. Visualization perspective**

![Visualization perspective](image)

After you’ve opened this perspective, you’ll notice that the focus is on visualization rather than coding. The main visualization tool integrated into the Aspect Visualization perspective is the Visualiser, located in the center of the perspective. If you’ve used earlier versions of AJDT, then the Visualiser is nothing new to you; however, in previous releases it used to be part of AJDT, whereas now it's a standalone plug-in. This means that it's possible to write your own provider to use the Visualiser to view your own data (for more information about the Visualiser select **Help>Help Contents>Visualiser**). In the Aspect Visualization perspective, though, we’re using the AspectJ provider, as you can see in Figure 10. This represents the classes and aspects within a project as bars and the places where aspects affect your code as stripes on those bars. The lengths of the bars are relative to file size -- the longer the bar, the more lines of code there are in the file that bar represents!

To populate the Visualiser, just select an AspectJ project in the Projects view on the far left, or a package in the Package view to the right of that. Look at the Spacewar example, and you’ll see that there are some bars that are greyed out, some bars with only one colored stripe, and some bars with more than one colored stripe. (Sometimes there’s even more than one stripe on a single line!) The greyed-out bars represent classes or aspects that are not affected by any advice. By pressing the button shown in Figure 11, you can choose to filter out these unaffected bars.
Figure 11. The Visualiser controls

Limit visualization to affected bars

Look at the remaining affected bars and their stripes, and you'll see that the colors match those on the buttons in the Visualiser menu (located to the right of the Visualiser). Next to each of the colored buttons is a checkbox and a label. The label gives the name of the aspect (or class that contains the aspect, in the case of inner aspects) that the color represents, and the checkbox indicates whether or not this aspect has been included in the current visualization. Thus, if you have several aspects, you can filter out any aspects you're not interested in. For example, if you build the Spacewar example with the debug.ajproperties build configuration and then return to the Visualiser, you may decide that you just want to see what your debug aspect is affecting. Deselecting all but the debug aspect in the Visualiser menu shows you precisely this, as you can see in Figure 12. Another way to tell which color represents a particular aspect is by mousing over the stripe you're interested in. This gives the name of the aspect that stripe represents. If there's more than one color to a stripe, then there's more than one aspect affecting that line of code!

Figure 12. Visualizing the effects of the debug aspect

Another very important feature of the Visualiser is that it too provides a way of navigating around your code. Having looked at the visualization of the Spacewar example, we can see that there is a Display aspect. We haven't looked into the code yet, so you're probably curious as to what this aspect does. We can see that it affects four classes, but what are they? Mousing over the dark grey stripe at the top of the bars for each of these affected classes, we find out that they are spacewar.Display, spacewar.Game, spacewar.SWFrame, and spacewar.Timer. But what exactly is the aspect advising? Double-click on the stripe on the spacewar.Game bar and you'll see: You go straight to the point in Game.java that is affected by the Display aspect, and it appears in the Editor view at the bottom of the Aspect Visualization perspective.

Debugging

Debugging applications containing aspects is as easy as debugging applications without them, so if you're used to using the Java debugger within Eclipse, there is again nothing really new for
you here. Since the advent of AJDT 1.1.11, setting breakpoints in before and after advice is the same as setting breakpoints in methods. Setting breakpoints in around advice is slightly different; if you want the debugger to stop at such a breakpoint, you need to turn inlining off. To do this, go to Window>Preferences>AspectJ>Compiler and navigate to the Advanced tab. Once there, check the No inline option and click OK.

To show in more detail how similar debugging AspectJ applications is to debugging standard Java applications, let's set a breakpoint in the Spacewar example and step through the code. First, switch to the Debug perspective by going to Window>Open Perspective>Debug. Since we looked at the display class in the Aspect Visualization perspective, let's put a breakpoint into the after() returning (Player player): call(Player+.new(..)) advice. Open Display.java class and navigate to the after returning advice, as shown in Figure 13. Then, either double-click in the gutter next to the line  Enumeration elements = DISPLAYS.elements(); or right-click and select Toggle Breakpoint from the same place.

**Figure 13. Setting a breakpoint**

![Figure 13. Setting a breakpoint](image)

After setting the breakpoint, to launch the application in debug mode, follow the same steps as you would with a plain Java application. From the green bug’s drop-down menu (the green bug is located on the workbench toolbar) select your Spacewar run configuration, and you're off! If you don't have a Spacewar run configuration set up, then from the same drop-down menu, select Debug... Create a new Java application configuration that runs the main() method in the spacewar.Game class, then press Debug.

As soon as the breakpoint is reached, the debug mode pauses the Spacewar application, as shown in Figure 14. Information about all the current variables can be found in the Variables view in the top right-hand corner of the Debug perspective, and the stack trace of the current thread can be found in the Debug view in the top left of the Debug perspective. The point in the code where you are paused can be seen in the Editor. To step through the code, just press the Step Over button in the Debug view. It's as easy as that!
Generating documentation

You can use the javadoc tool to generate API documentation for Java projects, based on specially formatted comments in the source code. The AspectJ distribution, as of version 1.2, includes a tool called ajdoc. This is an extension to javadoc that adds details of the crosscutting nature of your aspects to the documentation generated by javadoc. The JDT tools in Eclipse include a wizard for invoking and configuring the javadoc tool, and with AJDT installed, a similar wizard is available for ajdoc.

To invoke the ajdoc wizard, select the Project>Generate ajdoc... menu option. In the dialog that appears, simply select the project for which you wish to generate the API documentation. In our case, this is the Spacewar example, as shown in Figure 15.
Many of the classes and methods in the Spacewar code are not public, so to make the generated documentation more useful, select the Private option in the visibility section of the dialog. The Destination field towards the bottom of the dialog specifies the output location, which defaults to a doc directory in the project.

Now click the Next button to go to the following page, which allows any additional options to be specified. Select the Open generated index file in browser option, so you can view the generated documentation as soon as it has been generated. After clicking Finish, you will be asked if you want to update the javadoc location -- if you say yes here, it simply remembers the output location for next time. The ajdoc tool is then launched, and you can switch to the Console view in Eclipse if you want to watch it running. Once it has finished, the index page should be shown in a Web browser. If you navigate to the spacewar.Registry aspect, you'll see that the regular javadoc output has been enhanced to show where the advice takes effect, as illustrated in Figure 16.
Further options

There are many more options and settings available to you through AJDT -- far too many to mention them all here! High points include advanced compiler options, where you can configure the compiler to either ignore or declare as warnings or errors various coding issues; additional project properties pages for specifying AspectJ InPath and Aspect Path classpath entries; and the option to send informational weaving messages to the Problems view (exploiting this new feature of AspectJ 1.2). There are also many AspectJ-specific code templates available (for a list of these, select Windows>Preferences to open the Preferences dialog, then go to Java>Editor>Templates). For further information regarding these features and more, we refer you to the Eclipse Help -- the What's new and AJDT Highlights pages in particular.

Conclusion

Whilst a number of more advanced topics were beyond the scope of this article, we hope you now have a good feel for the nature of the support AJDT offers for aspect-oriented programming with AspectJ. As you have seen, the crosscutting nature of your aspect-oriented applications is surfaced by the tools in several ways, from the Outline view and markers on the side of the editor, to the powerful Visualization perspective, which gives you a higher-level view of your application.

AJDT has improved in a number of ways fairly recently (see the new and noteworthy lists for the recent 1.1.11 and 1.1.12 releases for more details), but the extent of the challenge in this area means that there is still plenty of room for further enhancements. Experimental work has been underway on a project known as Lancaster, which has been investigating ways of achieving much closer integration with the JDT. This will enable more complete functionality in areas such as code completion, formatting, searching, browsing, refactoring, and surfacing the structure of aspects in the Package Explorer and other views. Recent progress on this has been good, so watch out for the first 1.2.0 beta release in the near future. The aim here is to put AJDT firmly on a par with the JDT. Beyond this the focus will be on developing new functionality to enable you to learn and exploit AOP even more, with capabilities such as a pointcut wizard and support for aspect-specific refactorings. There is much to look forward to!
Finally, AJDT is an open source project, and thus it relies on people getting involved in a variety of ways, from users giving feedback and raising bugs and feature requests to developers submitting patches. If you'd like to get involved, please go to the AJDT Web site (see Related topics) and look for the links to Bugzilla, the newsgroup, the developers mailing list, and the page of to-do tasks.
Related topics

- Find out more about AJDT and download the latest version from the AspectJ Development Tools home page.
- Learn more about the AspectJ language from the AspectJ home page.
- "Contract enforcement with AOP," Filippo Diotalevi (developerWorks, July 2004) shows how to design and define clear contracts between components while keeping your code clean and flexible.
- "Use AOP to maintain legacy Java applications," Abhijit Belapurkar (developerWorks, March 2004) explains how to use aspect-oriented programming to comprehend and maintain complex legacy systems.
- AspectJ in Action, Ramnivas Laddad (Manning Publications, 2003) is an excellent resource for learning aspect-oriented programming in AspectJ.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)