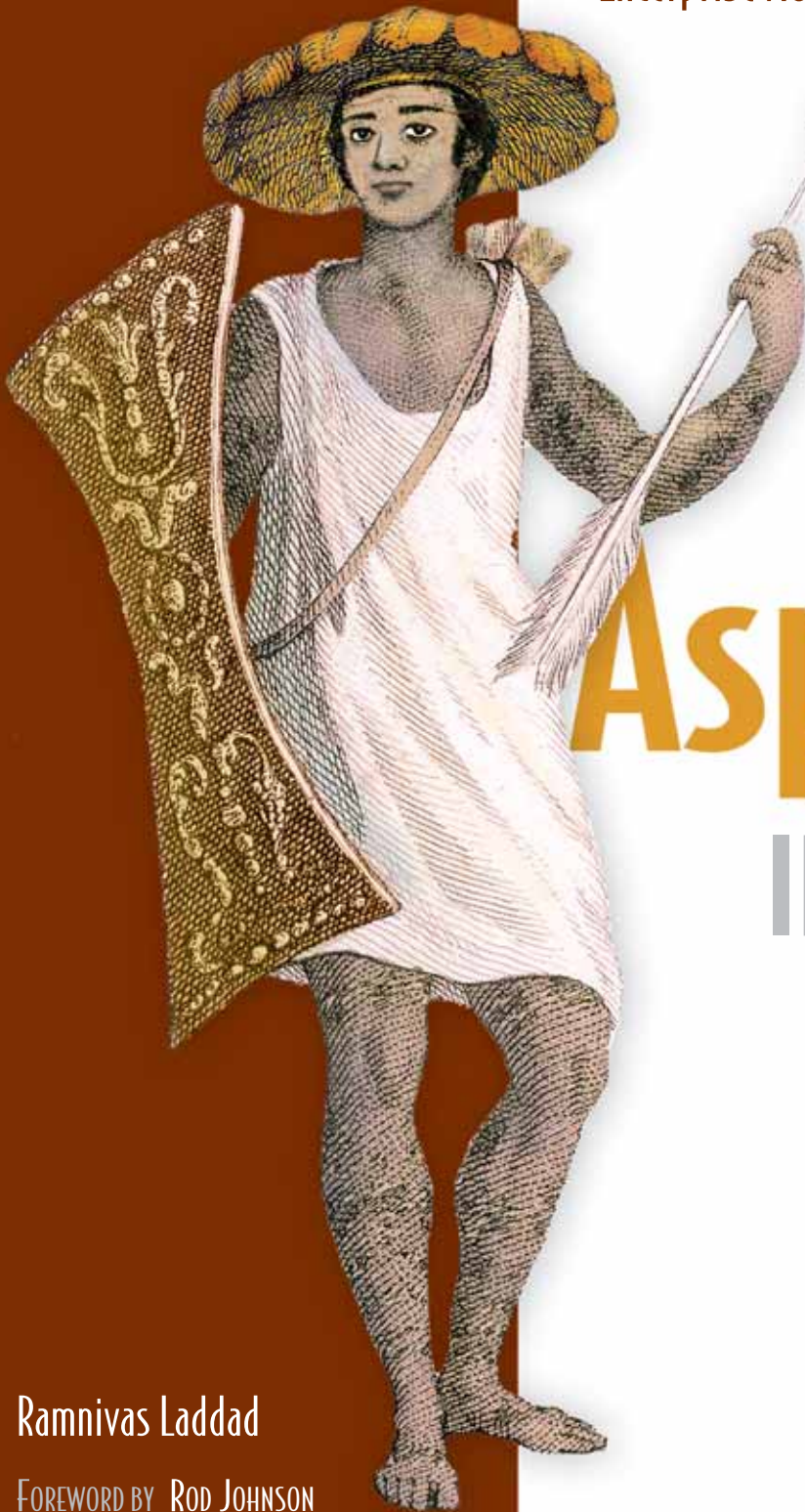


Enterprise AOP with Spring Applications



AspectJ

IN ACTION

SECOND EDITION

SAMPLE CHAPTER

Ramnivas Laddad

FOREWORD BY ROD JOHNSON

 MANNING



AspectJ in Action
Second Edition

by Ramnivas Laddad

Chapter 10

Copyright 2010 Manning Publications

brief contents

PART 1 UNDERSTANDING AOP AND ASPECTJ1

- 1 ■ Discovering AOP 3
- 2 ■ Introducing AspectJ 27
- 3 ■ Understanding the join point model 51
- 4 ■ Modifying behavior with dynamic crosscutting 87
- 5 ■ Modifying structure with static crosscutting 116
- 6 ■ Aspects: putting it all together 136
- 7 ■ Diving into the @AspectJ syntax 168
- 8 ■ AspectJ weaving models 199
- 9 ■ Integration with Spring 217

PART 2 APPLICATIONS OF ASPECTJ WITH SPRING249

- 10 ■ Monitoring techniques 251
- 11 ■ Policy enforcement: keeping your design intact 288
- 12 ■ Learning design patterns 319
- 13 ■ Implementing concurrency control 344
- 14 ■ Managing transactions 373

- 15 ■ Securing applications 404
- 16 ■ Improving domain logic 431
- 17 ■ Taking the next step 459
- appendix A ■ Setting up the example 469
- appendix B ■ Using Ant with AspectJ 486
- appendix C ■ Using Maven with AspectJ 491

Part 2

Applications of AspectJ with Spring

Part 2 puts the knowledge you gained in the first part to practical use by showing how AspectJ and the Spring Framework simplify enterprise applications. Although Spring provides dependency injection and enterprise service abstraction, AspectJ completes the picture by modularizing crosscutting concerns. We'll explore the two ways Spring integrates with AspectJ: through dynamic proxies and through byte-code weaving. You should be able to use most of the example code in your applications without much modification. Even if you aren't using Spring, you'll find that you can adopt these examples to suit your applications. We include a few examples from outside of Spring—specifically Swing and EJB—to show that AOP concepts are applicable in any kind of applications.

We begin by examining a classic application of AOP: monitoring and tracing. Then, we'll modularize the policy-enforcement concerns to create a safety net that ensures you won't get into trouble by violating programming policies. The examples in chapters 10 and 11 demonstrate how you can use AOP to improve your personal productivity during the development phase. You can take out these aspects when you deploy your system without affecting the correctness of the core system. Of course, as we explain, you can continue using these aspects in the deployed system and gain even more benefits.

We'll then implement complex functionality through AOP. First, chapter 12 introduces a few AOP design patterns that are used in the remaining chapters. Chapters 13 through 16 deal with functionality such as concurrency control, security, transaction management, and domain logic. In each case, you'll put

Spring dependency injection and enterprise abstraction to good use. Chapter 17 concludes this book with a discussion of how you can incorporate AOP into your organization.

Monitoring techniques

This chapter covers

- Noninvasive tracing
- Performance monitoring
- Spring AOP for monitoring

Monitoring encompasses a variety of techniques such as tracing important events in an application, profiling performance-critical portions of the app, and observing a system's vital parameters. You need to monitor enterprise applications in all phases of their lifecycle: development, deployment, and production. During the development phase, monitoring helps you understand the interactions between components in the system and spot any deviations from what is expected. During the deployment phase, it lets you profile the application under load to plan the hardware needed for a successful deployment. During production, monitoring helps you verify that the system is working within the expected range of operating parameters; it alerts you about any impending problems and lets you extract useful diagnostic information when things go wrong.

All monitoring techniques share some common problems when implemented using conventional techniques. First, their implementations cut across multiple modules, causing code scattering. Second, the code for monitoring intertwines with business logic, causing code tangling. The sheer amount of code needed is often a reason

not to implement monitoring in many applications. Furthermore, even if a determined team is ready to add the needed code, implementing monitoring functionality consistently is a tall order that is seldom achieved with the needed precision.

Recent advances in AspectJ, such as load-time weaving (LTW), make deploying monitoring aspects a much simpler task while providing control over the monitored points without needing recompilation. Spring's AspectJ integration also provides simpler opportunities to implement monitoring functionality for the key components in a typical enterprise application without any changes to build or deployment environments.

In this chapter, we'll examine ways to introduce monitoring in a systematic and noninvasive manner through use of aspects. We'll start with the perennial favorite of AOP: tracing! By introducing simple aspects, you can consistently trace important events in the system, such as execution of and calls to selected methods and throwing of exceptions. For each of these techniques, we'll examine common patterns in implementing tracing the AOP way. We'll also discuss how AOP can enhance conventional logging. Then, we'll extend these techniques to performance monitoring. You should be able to use the information provided in this chapter immediately during development. The experience gained in this process will help you decide about further use of AOP.

10.1 *Tracing in action*

Tracing is one of the most common monitoring techniques used to understand a system's behavior. In its simplest form, tracing logs messages that describe the occurrence of interesting events during the execution of a system. For example, in a banking system, you would log each account transaction with information such as the nature of the transaction, the account number, and the transaction amount. You could also log exceptions that occurred during execution of the system along with the context under which they occurred.

During the development cycle, tracing plays a role similar to a debugger. It's also usually the only reasonable choice for debugging distributed systems or concurrency-related problems. By examining the log, you can spot unexpected system behavior and correct it. A log also helps you see the interactions between different parts of a system to detect exactly where the problem might be. Likewise, in fully deployed systems, tracing acts as a diagnostic assistant for finding the root cause of the problem.

A poster child of AOP

Tracing has always been a poster child of AOP applications (as well as a popular target of the myth that all AOP can do is tracing—something I'll debunk throughout this book by providing a vast range of applications other than tracing). Tracing and other monitoring techniques are good examples of crosscutting functionality at its extreme. AOP makes implementing monitoring techniques a breeze. It not only saves a ton of code but also establishes centralized control, consistency, and efficiency.

Let's examine an AOP implementation of tracing in action through a simple console application based on code provided in appendix A. In listing 10.1, you add a product to an order.

Listing 10.1 The Main class: a simple test driver

```
package ajia.main;

import ...

public class Main {
    public static void main(String[] args) {
        ApplicationContext context
            = new ClassPathXmlApplicationContext("META-INF/spring/*.xml");

        OrderService orderService
            = (OrderService) context.getBean("orderService");
        ProductService productService
            = (ProductService) context.getBean("productService");
        Order order = new Order();
        Product product = productService.findProduct(1001L);
        orderService.addProduct(order, product, 1);
    }
}
```

Now, let's look at tracing implemented using AspectJ; later, we'll compare it with conventional techniques.

10.1.1 Tracing the aspect-oriented way

Let's use AspectJ to introduce the tracing functionality into each method in all the classes in the example. All you need to do is add the aspect in listing 10.2 to your system and compile it with the classes using the AspectJ compiler (note that you could have written the aspect using `@AspectJ` syntax without losing any functionality). That's it! You'll have tons of output to impress your colleagues.

Listing 10.2 Tracing method entries

```
package ajia.tracing;

import ...

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(TraceAspect.class);

    pointcut traced()
        : execution(* *.*(..)) && !within(TraceAspect);

    before() : traced() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.log(Level.INFO,
            "Entering [" + sig.toShortString() + "]");
    }
}
```

1 Selects methods to trace

2 Advises traced methods

- ① The `traced()` pointcut selects methods to trace—in this case, it selects all the methods in the system. You also follow a common idiom in AspectJ to exclude join points in the aspect itself to avoid a potential problem of infinite recursion. The `!within(TraceAspect)` part helps you avoid recursion caused by tracing method calls in the aspect itself. Although such exclusion won't matter in the current version of the aspect because there are no methods in the aspect itself, it's never too early to start using good practices.
- ② You use `thisJoinPointStaticPart` to get information about the method selected by the pointcut. The method `getSignature()` on `thisJoinPointStaticPart` returns the signature of the selected method. You use the `toShortString()` method to obtain the shorter version description. If needed, you could obtain the longer version using the `toLongString()` method or obtain individual parts of the signature and assemble the output yourself. You could also obtain the line number by using `thisJoinPointStaticPart.getSourceLocation().getLine()`. Note that the weaver creates static information about the join point at weave time, and neither compiler optimization nor the presence of the Just in Time (JIT) and hotspot virtual machine alters this information. See chapter 4 for detailed information about using reflection in an advice body.

When you compile this aspect together with the classes and run the test program, you get output similar to this:

```
Entering [Main.main(..)]
Entering [ProductServiceImpl.findProduct(..)]
Entering [JpaGenericRepository.find(..)]
Entering [OrderServiceImpl.addProduct(..)]
Entering [InventoryServiceImpl.removeProduct(..)]
Entering [InventoryServiceImpl.isProductAvailable(..)]
Entering [JpaInventoryItemRepository.findByProduct(..)]
Entering [InventoryItem.getQuantityOnHand()]
Entering [JpaInventoryItemRepository.findByProduct(..)]
Entering [InventoryItem.deplete(..)]
Entering [JpaGenericRepository.update(..)]
Entering [Order.addProduct(..)]
Entering [Order.isPlaced()]
Entering [Order.getItemFor(..)]
Entering [Product.getPrice()]
Entering [LineItem.setQuantity(..)]
Entering [OrderServiceImpl.updateOrder(..)]
Entering [JpaGenericRepository.update(..)]
```

With the tracing aspect, you get good insight into the system's execution just by writing a simple tracing aspect. Later in this chapter, you'll see improvements that will help you gain an even deeper understanding of the system's inner workings.

To truly appreciate the AOP implementation, let's examine how you could implement the same functionality without it.

Standard logging toolkit vs. log4j vs. commons logging

Throughout this chapter, we use log4j as the primary logging API. Although the use of Jakarta Commons Logging or slf4j would offer a more abstract solution, a few examples use facilities available in log4j without the associated API in Commons Logging and slf4j. Specifically, Commons Logging doesn't support Nested Diagnostic Context (NDC) and Mapped Diagnostic Context (MDC); slf4j supports MDC but not NDC. Because the focus of this chapter is combining AspectJ with a logging toolkit, discussing multiple logging toolkits would distract us from that goal. The good news is that with AspectJ, switching from one logging toolkit to another is a simple task.

10.1.2 Tracing the conventional way

If you were to implement tracing without using AspectJ, you'd end up with code such as in listing 10.3. To illustrate the additional work needed on your part, let's look at the modifications to the `Order` class required using the log4j API.

You instrument each method of the class to log the entry into it. You log each method at the `Level.INFO` level because you're writing informational entries. Listing 10.3 shows the changed `Order` class.

Listing 10.3 The `Order` class with conventional tracing

```
package ajia.domain;
import ...
...
public class Order extends DomainEntity {
    ...
    private Logger logger = Logger.getLogger("trace");
    public void addItem(Product product, int quantity) {
        logger.log(Level.INFO,
            "Entering [Order.addItem]");
        ...
    }
    public void removeItem(Product product, int quantity) {
        logger.log(Level.INFO,
            "Entering [Order.removeItem]");
        ...
    }
    public boolean isPlaced() {
        logger.log(Level.INFO,
            "Entering [Order.isPlaced]");
        ...
    }
    public void place() {
        logger.log(Level.INFO,
            "Entering [Order.place]");
```

```

    ...
}
    ...
}

```

Every class in the system would have similar duplication. That would be quite a task, right? Granted, the job would be mostly mechanical—you’d probably copy and paste code. After pasting, you’d have to be sure you changed each argument to the `log()` method correctly; if you didn’t, you’d end up with a log message that was inconsistent with the operation being performed. Furthermore, if you wanted entry *and* exit traced rather than just a log message for entry, it would be even worse.

NOTE The code in listings 10.2 and 10.3 isn’t exactly equivalent; the former uses an aspect-specific logger, whereas the latter uses a class-specific logger. We’ll examine a way to eliminate the difference in section 10.4.5.

Now, consider how long it would take to introduce tracing in a real system with hundreds of classes. How sure could you be that the methods would log the right information? This is a typical scenario with any crosscutting concerns: too much effort to implement them, and too many opportunities for bugs.

Logging toolkit and automatic extraction of caller information

When you’re using `log4j`, avoid using it with `%C`, `%F`, `%L`, `%M`, or a combined `%l` layout pattern. These patterns examine the call stack to extract the caller information (such as the class name, method name, and line number) at the location of the `log()` method invocation. The performance hit from using the call stack to deduce a caller is significant, because it involves obtaining the call stack and parsing its contents—not a trivial job. The `log4j` documentation explicitly warns against using these patterns (“WARNING Generating caller location information is extremely slow. Its use should be avoided unless execution speed isn’t an issue.”). Further, in the presence of an optimizing compiler and hotspot/JIT-enabled virtual machine, the deduced caller may be incorrect. When you don’t use these formats, you need to supply the caller information yourself the way you did in listing 10.3. But note that this isn’t a `log4j`-specific issue; the same issue exists in any logging kit that offers to deduce the caller from the call stack.

Alas, the need for caller information is too common; without such information, the log output is much less useful. But adding location information to each log invocation adds additional code to something that is already complex. AOP tracing, as you saw in listing 10.2, provides an efficient and accurate means to obtain local information. In section 10.7.1, we’ll examine a way to obtain the caller information in the same manner even when you use conventional logging.

Now that you’ve seen tracing using conventional and AspectJ-based techniques, let’s compare the two approaches.

10.2 Conventional vs. AOP tracing

The base implementation technique used in conventional tracing involves a logging toolkit and calls to its APIs. The logging toolkit simplifies the job of categorizing and formatting the log message. It also provides runtime control over the messages logged. For example, with `log4j`, you can have a logger associated with a category (typically the class name). You can then specify a `log4j.xml` file to control the level for each log category. Figure 10.1 illustrates this schematic of conventional tracing solutions.

As you can see, the tracing calls are all over the core modules. When a new module is added to the system, all of its methods that need tracing must be instrumented. Such instrumentation is *invasive*, causing tangling of the core concerns with the tracing concern.

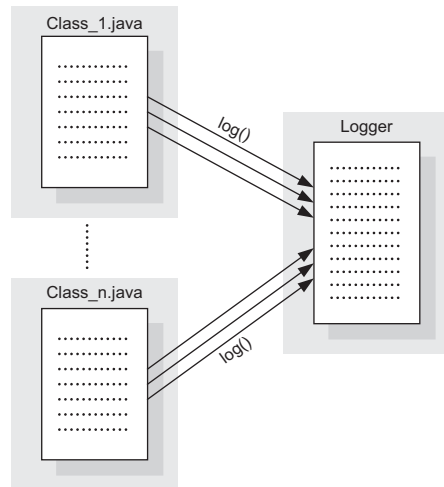


Figure 10.1 Conventional tracing, where all log points issue calls to the logger explicitly

Logging vs. tracing

There is some confusion about the use of terms *logging* and *tracing*. Most developers (and this book) refer to logging as an act of producing messages specific to the logic carried by a piece of code. Tracing is commonly considered as the act of producing messages for lower-level events: method entry and exits, object construction, exception handling, state modification, and so on. Both techniques often use a logging toolkit to simplify their implementation.

Consistency is the single most important requirement of tracing. If the tracing specification requires that a certain kind of operations be logged, then the implementation must log *every* invocation of those operations. When things go wrong in a system, doubting the tracing consistency is probably the last thing you want to do. Missed tracing calls can make output hard to understand and sometimes useless. For example, if you were expecting a certain method to have been invoked, and you didn't see a log output for that method, you couldn't be sure if the method wasn't called or wasn't logged due to inconsistent implementation.

Achieving consistency using conventional tracing is a lofty goal; and although systems can attain it initially, it requires continuous vigilance to maintain. For example, if you add new classes to the system or new methods in existing classes, you must ensure that they implement tracing that matches the current tracing strategy.

AOP fixes problems with conventional tracing by combining the logging toolkit with dynamic crosscutting. Essentially, AOP modularizes the invocation of the logging

toolkit's APIs. The beauty of this approach is that you don't need to instrument any log points; writing an aspect does a functional equivalent automatically. Further, because there is a central place to control tracing operations, you achieve consistency easily. Figure 10.2 shows an overview of AspectJ-based tracing.

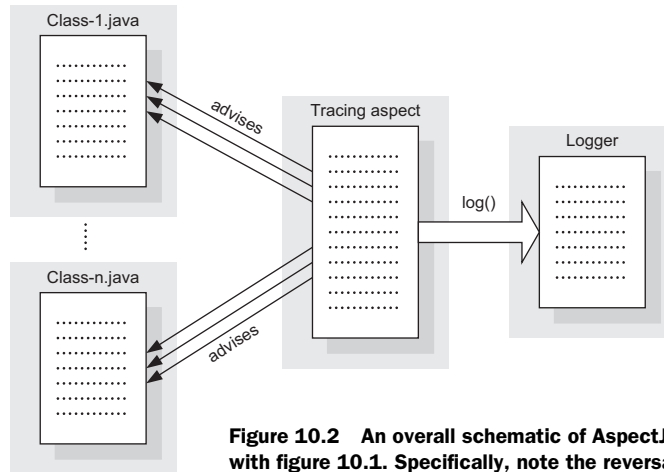


Figure 10.2 An overall schematic of AspectJ-based tracing. Compare this with figure 10.1. Specifically, note the reversal of the arrows to the classes.

With AspectJ-based tracing, the tracing aspect separates the core modules and the logger object. Instead of the core modules' embedding the `log()` method invocations in their source code, the aspect weaves the logging invocations into the core modules when they're needed. AOP-based tracing reverses the dependency between the core modules and the logger; the aspect encodes how the operations in the core modules are logged instead of each core module deciding for itself. This kind of arrangement is known as the *Dependency Inversion Principle* (see <http://aosd.net/2007/program/industry/I6-AspectDesignPrinciples.pdf> for more details). Furthermore, as we'll show later in this chapter, extending tracing to other events in the system, such as throwing an exception and modifying or accessing important object state, can be implemented easily owing to the powerful pointcut language.

As you can distill from the discussion so far, monitoring through AOP boils down to two parts:

- Selecting join points of interest
- Advising them with monitoring logic

The first part requires some considerations that we'll discuss next. We'll discuss the second part in sections that follow.

10.3 **Selecting join points of interest**

Typically, you'll want to select a list of monitored methods to make output more understandable and avoid degrading performance. You can achieve this goal by implementing the pointcut to select join points based on their static and dynamic characteristics. Let's look at several design options that will be useful for many pervasive aspects, like tracing.

10.3.1 Selection based on static structure

A simple way to select a set of monitored join points is based on their static structure. You can use characteristics such as the package hierarchy of the types, inheritance structure, method name patterns, and method return type. Leveraging annotations associated with various programming elements is also a powerful way to select the trace points. Typically, you'll use a combination of all these techniques to define your tracing pointcuts. Let's examine each one separately.

LEVERAGING PACKAGE STRUCTURE

A well-defined package structure is central to any good software system. It's common to have the package hierarchy reflect high-level design elements. For example, a software system following the Model-View-Controller (MVC) pattern has a package structure that dedicates a package element to each of the three concepts, leading to packages such as `ajia.domain` and `ajia.web`. Choosing monitoring points using packages therefore makes it easy to select elements based on the function they're playing. With such a selection, you can capture coarse-grained monitoring requirements such as "trace all domain model interactions." Listing 10.4 illustrates a selection based on static structure.

Listing 10.4 Trace aspect utilizing pointcuts based on package structure

```
package ajia.tracing;

import ...

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(TraceAspect.class);

    pointcut domainOp()
        : execution(* ajia.domain..*.*(..))
        || execution(ajia.domain..*.new(..));

    pointcut controllerOp()
        : execution(* ajia.web..*.*(..))
        || execution(ajia.web..*.new(..));

    pointcut repositoryOp()
        : execution(* ajia.repository..*.*(..))
        || execution(ajia.repository..*.new(..));

    pointcut traced()
        : domainOp() || controllerOp() || repositoryOp();

    ... advice to traced() ...
}
```

Note how you construct the final pointcut using simple pointcuts, each selecting a specific set of types. This style—a best practice in AOP—simplifies understanding and maintenance because you can focus on a simple concept at each level. It also makes the pointcut useful.

An alternative style is to use package scopes to define pointcuts, as shown in listing 10.5.

Listing 10.5 Trace aspect utilizing package scopes

```

package ajia.tracing;

import ...

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(TraceAspect.class);

    pointcut inDomainType()
        : within(ajia.domain..*);

    pointcut inControllerType()
        : within(ajia.web..*);

    pointcut inRepositoryType()
        : within(ajia.repository..*);

    pointcut inTracedType()
        : inDomainType() || inControllerType() || inRepositoryType();

    pointcut traced()
        : (execution(* *(..)) || execution(new(..)))
        && inTracedType();

    ... advice to traced() ...
}

```

Here, the first three pointcuts select all the join points within a certain package structure and combine them with pointcuts selecting all methods and constructors. Of course, nothing prevents you from further refining the `traced()` pointcut to, say, separate the selection of method executions from the selection of object creations.

Refactoring in action

Notice how you're progressively modifying the pointcuts to make them simpler and reusable. This is a typical refactoring process applied to aspects. Start by focusing on the problem at hand, and improve it as you go along. With experience, you'll develop your own style whereby you'll start with an implementation following the best practices and thus skip a few initial stages of refactoring.

With either style, typically, you promote pointcuts corresponding to high-level functionality and reusable concepts to a library aspect. For example, you can include the aspect shown in listing 10.6. Such an aspect is usable not only from a monitoring aspect but also from other aspects, as you'll see in the next chapter, where you use it to enforce system-level policies.

Listing 10.6 Reusable aspect to define the system architecture

```

package ajia.architecture;

import ...

public aspect SystemArchitecture {

```

```

public pointcut inDomainType ()
    : within(ajia.domain..*);

public pointcut inControllerType()
    : within(ajia.web..*);

public pointcut inRepositoryType()
    : within(ajia.repository..*);
}

```

Then, the tracing aspect can use these pointcuts to compose its own pointcuts, as shown in listing 10.7.

Listing 10.7 Trace aspect based on reusable aspect defining the system architecture

```

package ajia.tracing;

import ...

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(TraceAspect.class);

    pointcut inTracedType()
        : SystemArchitecture.inDomainType()
          || SystemArchitecture.inControllerType()
          || SystemArchitecture.inRepositoryType();

    pointcut traced()
        : (execution(* *(..)) || execution(new(..)))
          && inTracedType();

    ... advice to traced() ...
}

```

Although the package structure provides a coarse-grained selection of join points, you need to use other information to select join points to match tracing needs more precisely. Let's examine one such approach: using type-related information.

LEVERAGING TYPE STRUCTURE

Type-structure information, such as the naming pattern and inheritance hierarchy, provides useful techniques to select trace join points. If you use a consistent naming pattern for your types, you can leverage type names to define patterns using wildcards. For example, if all your types representing a controller are suffixed with `Controller`, you can use `*Controller` as the pattern to select them.

But because you're relying on a programmer's due diligence in adhering to a naming convention, it's best to find other ways before resorting to using a name pattern. One such way is to use the type hierarchy. For example, if you're using the traditional Spring MVC (that doesn't use the `@Controller` annotation), you can use `org.springframework.web.servlet.mvc.Controller+` as the pattern to select all controllers in your system. You can also use your own type hierarchy. For example, if all your domain entities extend the `DomainEntity` class, you can use `DomainEntity+` as the type pattern.

LEVERAGING ANNOTATIONS

You can leverage annotations to select types. For example, if you choose to use the Java Persistence API (JPA), you can use an annotation such as `@Entity` to select only

persistent objects; or if you use JAX-WS Web Services, you can use an annotation such as `@WebService` to choose only web services, as shown in the following pointcut:

```
pointcut entityOp() : execution(* (@Entity *).*(..));
pointcut webServiceOp() : execution(* (@WebService *).*(..));
```

Similarly, if you were to use Spring's annotation-based MVC, you could use the following pointcut:

```
pointcut controllerOp() : execution(* (@Controller *).*(..));
```

It's a best practice to implement pointcuts that utilize technology-specific type hierarchies and annotations in a separate library aspect (similar to listing 10.6). This lets you reuse them in multiple aspects or even in multiple projects using the same set of technologies.

You can even create your own custom annotations and use them for selection. For example, you can use the `@Audit` annotation to implement the tracing functionality required for a regulatory compliance. Such annotations may even specify the compliance code so the audit record can include it:

```
pointcut auditedAuditCode(Audit audit) : execution(@Audit * *(..)
                                                && @annotation(audit));
```

Similar to types, you can use annotations marked not only for the method but also for the return type, parameters, and declared exceptions. For example, the following pointcut selects all methods that return a type that is marked with the `@HIPPAData` annotation:

```
execution((@HIPPAData *) *(..))
```

When using custom annotations, you should avoid the use of tracing-specific annotations such as `@TraceMe`. Although such annotations provide a better degree of separation than a direct use of a logging API, it goes against the core idea of noninvasiveness behind AOP.

LEVERAGING METHOD SIGNATURE

Methods represent even finer granularity. Like type patterns, using name patterns is a possibility, and the same caveat of an unstable naming convention applies. But certain naming patterns—such as all setter methods starting with `set`—prove to be general, especially because other technologies such as dependency injection (DI) and persistence frameworks utilize those conventions.

When you're defining methods using a name pattern, exercise some care to specify the exact criteria needed. For example, although `execution(* set*(..))` certainly selects all setters, it may also end up selecting setters that take multiple arguments. A better pointcut for selecting setters would consider characteristics such as public access, void return type, and a single argument: `execution(public void set*(*))`. You still risk selecting a method such as `settle`, should it have public access, a void return type, and a single argument. If you want to be even more defensive, you can further tighten the definition to ensure that the character that follows `set` is an uppercase letter. You

can then settle the score with the `settle` method! Here is a pointcut that does the job (marginal improvements are possible to minimize typing, but you still need all 26 lines—one per each letter of the alphabet in English):

```
public pointcut setter()
    : execution(public void setA*(*))
      || execution(public void setB*(*))
      ...
```

You can similarly write a pointcut for getter methods that uses `!void` as the return type pattern for methods starting with `get` and `boolean` return type for methods starting with `is`. You must also ensure that the method doesn't take any arguments. You definitely want such pointcuts nicely tucked away in a reusable aspect so you can reap the benefits of the many lines of code you diligently typed!

Other method signature constituents, such as return type, parameter types, and thrown exceptions, can often help you select the right kind of join points. For example, the following pointcut selects all Remote Method Invocation (RMI) operations:

```
execution(public !static * Remote+.*(..) throws RemoteException+)
```

This pointcut leverages the type pattern `Remote+` along with the exception declaration in method to select all public non-static methods defined in the `Remote` type or its subtypes. This also illustrates how to use just the essential characteristics of join points to specify a precise selection criterion. This kind of selection is particularly useful when you're working with standard technologies such as Swing, SWT, Spring, and EJB. You'll see examples of such use throughout this book.

Through a judicious combination of package, type, and method patterns, you can select join points needed for most tracing functionalities mirroring their conventional equivalent. It's important to strike a balance between too general and too specific: the latter runs into the *fragile pointcut problem* because names change and it's easy to forget to change pointcuts.

AspectJ can offer additional possibilities by leveraging dynamic context that would be practically unattractive to implement conventionally.

10.3.2 Selection based on dynamic context

Consider a scenario in which you're utilizing repository classes through various services and you want to trace calls only if a web service originated those calls. The `cflow()` and `cflowbelow()` pointcuts can allow such a criterion. For example, the following pointcut selects all join points in the control-flow of any web service execution:

```
pointcut duringWebServiceOp() : cflow(webServiceOp());
```

You can then combine this pointcut with another that selects repository operations:

```
pointcut repositoryOp()
    : execution(* (@Repository *).*(..));

pointcut repositoryOpDuringWebServicesOp()
    : repositoryOp() && duringWebServiceOp();
```

This pointcut selects any repository method regardless of its lexical location and call depth if it's inside the control-flow of a web service operation.

The control-flow-based pointcuts are also useful in limiting tracing to only the top-level operations in a recursive call stack. For example, the following pointcut selects all top-level transaction-managed operations (assuming that all transactional methods are marked with a `@Transactional` annotation—we'll discuss other possibilities when we deal with transaction management aspect in chapter 14):

```
pointcut transactional()
    : execution(@Transactional * *(..));

pointcut topLevelTransactional()
    : transactional() && !cflowbelow(transactional());
```

By now, you should be convinced that you can select appropriate join points matching a specific monitoring requirement. It does take some experience to master writing good pointcuts, but be assured that it's easier than it may seem at first.

Let's examine the second part involved in each monitoring technique: the monitoring logic. We'll first complete our discussion of tracing. Then, we'll discuss exception monitoring, improving conventional logic, and performance monitoring.

10.4 *Tracing*

As you saw at the beginning of this chapter, the basic tracing implementation through AOP is easy. But AOP also offers some unique possibilities. In this section, we'll look at indenting trace calls, sharing aspect functionality, tracing intra-method activities, logging method parameters, and using type-specific loggers.

Now, get ready to do something that is practically impossible to do without aspects in any reasonable manner.

10.4.1 *Indenting trace calls*

You can make trace output tremendously more useful by exposing the caller-callee relationship in some fashion. Not only do you get to see the execution of join points, but you can also visualize the call graph—kind of like a sequence diagram. You may expose this relationship in a variety of forms. A simple way is to include the call-depth value in the log message. Alternatively, a visually more appealing way is to indent the log output to indicate the call-depth value.

Let's develop an aspect to illustrate how easy it is to accomplish this using AspectJ. The core idea here is to keep track of call depth on a per-thread basis and use it when logging a message. The implementation involves advising the traced method with a before advice to increment the depth as you enter the method and with an after advice to decrement it as you exit the method.

A simple way to keep the call depth is to use a thread-local wrapping an integer and update it from before and after advice. Then, when you log, compute the whitespace characters required, and prefix those whitespaces to the message.

Nested Diagnostic Context and Mapped Diagnostic Context

Log statements typically include information available in the local context, such as method names and parameters. To make log output more useful, certain logging toolkits let you set additional context. A log statement then may include that context in its output.

Nested Diagnostic Context (NDC) lets you arrange information in a hierarchical manner. Consider a layered enterprise application. Using NDC, you can push information identifying each layer as you enter it and pop that information when you exit. The log statement then may include the trail of layers that led up to that statement.

Mapped Diagnostic Context (MDC) lets you arrange information in a map. Consider a web application, where you need the log statements in the data access layer to include information such as the accessing user and the remote IP address available in the web layer. Using MDC, the web layer can add the needed information in a map. A log statement in the data layer may then include this information along with its local context.

When using log4j, you can leverage its capability to set up NDC for the same purpose to simplify the implementation. Here, each method pushes whitespaces in the nested context upon entry and pops them upon exit. The log statements then include all the accumulated whitespaces in the nested context. Listing 10.8 shows the aspect that implements the indentation functionality.

Listing 10.8 Indentation using log4j's nested diagnostic context

```
package ajia.tracing;

import ...

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(getClass());

    pointcut traced()
        : execution(* *.*(..)) && !within(TraceAspect);

    before() : traced() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.log(Level.INFO,
            "Entering [" + sig.toShortString() + "]");
        NDC.push("  ");
    }

    after() : traced() {
        NDC.pop();
    }
}
```

You need to make a small change to the output format configuration in the log4j.xml file to include the nested context, as shown in the following snippet. The only difference is the inclusion of the %x pattern, which stands for the nested context:

```
<layout class="org.apache.log4j.PatternLayout">
  <param name="ConversionPattern" value="%x%m%n"/>
</layout>
```

When you compile this aspect along with the classes and execute the system, you get output as follows:

```
Entering [Main.main(..)]
  Entering [ProductServiceImpl.findProduct(..)]
    Entering [JpaGenericRepository.find(..)]
  Entering [OrderServiceImpl.addProduct(..)]
    Entering [InventoryServiceImpl.removeProduct(..)]
      Entering [InventoryServiceImpl.isProductAvailable(..)]
        Entering [JpaInventoryItemRepository.findByProduct(..)]
          Entering [InventoryItem.getQuantityOnHand()]
        Entering [JpaInventoryItemRepository.findByProduct(..)]
      Entering [InventoryItem.deplete(..)]
    Entering [JpaGenericRepository.update(..)]
  Entering [Order.addProduct(..)]
    Entering [Order.isPlaced()]
    Entering [Order.getItemFor(..)]
    Entering [Product.getPrice()]
    Entering [LineItem.setQuantity(..)]
  Entering [OrderServiceImpl.updateOrder(..)]
    Entering [JpaGenericRepository.update(..)]
```

Take a moment to pause and consider how you might implement this functionality without using aspects. In every method, you might have to do something like:

```
try {
  NDC.push(" ");
  ... Log statements
  ... Business logic
} finally {
  NDC.pop();
}
```

This kind of code is an invitation to create bugs. First, you might forget to use a try/finally arrangement to perform NDC operations, thus missing `NDC.pop()` when an exception is thrown. Second, you might forget to include the NDC logic in every method, making the overall indentation scheme unreliable.

When you add functionality such as indentation logic, the trace aspects are no longer trivial, and it's desirable to share the tracing logic to avoid duplication and simplify maintenance. We'll examine such a possibility in the next section.

10.4.2 *Sharing tracing aspect functionality*

Tracing requirements come from multiple sources. Ideally, a separate aspect should implement each requirement to ease implementation and maintenance. But you'll still want to share all common tracing functionality. A good way to achieve both these goals is to create a reusable base aspect containing the core tracing functionality and one or more abstract pointcuts. You then create multiple subaspects targeting each

related set of requirements. Let's illustrate this idea by refactoring the aspects in listing 10.9 to create a reusable indentation-capable aspect.

Listing 10.9 Abstract aspect for indentation functionality

```
package ajia.tracing;

import ...

public abstract aspect IndentationTraceAspect {
    private Logger logger = Logger.getLogger(IndentationTraceAspect.class);

    public abstract pointcut traced();

    ... before and after advice remain unchanged since listing 10.8
}
```

Compared to listing 10.8, you make only two changes: you mark the aspect as abstract and mark `traced()` as a public abstract pointcut, and you remove the implementation body from the abstract pointcut. The advice remains unchanged. Each subsystem can now extend this aspect. For example, listing 10.10 shows an aspect that logs only the model classes.

Listing 10.10 Concrete aspect to trace domain classes

```
package ajia.tracing;

public aspect TraceAspect extends IndentationTraceAspect {
    public pointcut traced()
        : execution(* ajia.domain..*.*(..));
}
```

With reusable parts in place, you can create simple subspects and include them as required.

So far, we've limited ourselves to tracing at the entry and exit of a method. But you often need more granularity than that. For example, you may want to trace all remote calls occurring from within specified methods. Let's see how AspectJ can help with such requirements.

10.4.3 Tracing intra-method activities

AspectJ-based tracing helps to modularize logging invocations that are crosscutting in nature. A judicious combination of tracing options presented in this chapter will take care of a majority of the cases. For example, while processing an order, you may want to log specific steps such as securing payment, checking inventory, contacting the shipping division, and so on. If you implemented all these steps in one method, you would be able to use `call()` pointcuts perhaps in combination with `withincode()` to select individual calls. But some steps may encompass multiple calls, further reducing the possibility of selecting the required join points and hence tracing those steps. The first response in such situations should be to break each step into a separate method through Extract Method and similar refactoring (for details, see *Refactoring: Improving the Design of Existing Code* by Martin Fowler [Addison-Wesley, 1999]). You can then select

those methods for tracing purposes. This process yields an improved implementation, which is a good thing even without considering the benefits of simplifying tracing.

But real life isn't always considerate enough to offer such a choice. Sometimes you can't refactor a large method like this. Perhaps the requirements may call for logging specific states of the system instead of just the `this` object or the method arguments. In those cases, your only practical choice is to log the intra-method calls the conventional way, with in-line code. As you'll see in section 10.7, AspectJ can help even when you use conventional logging.

10.4.4 Logging the method parameters

Often, you not only want to log the method calls but also the invoked object and the method parameters. You can implement this requirement easily by using the `thisJoinPoint` reference. In each advice body, a special `thisJoinPoint` object is available that includes information about the advised join point and its associated context.

The aspect in listing 10.11 modifies the `before` advice in `TraceAspect` to log the method parameters.

Listing 10.11 `TraceAspect`: modified to log method parameters

```
package ajia.tracing;
import ...;

public aspect TraceAspect {
    private Logger logger = Logger.getLogger(getClass());

    pointcut traced()
        : execution(* *.*(..) && !within(TraceAspect));

    before() : traced() && !execution(* Object.*(..)) {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.log(Level.INFO,
            "Entering [" + sig.toShortString() + "]"
            + createParameterMessage(thisJoinPoint));
    }

    private String createParameterMessage(JoinPoint joinPoint) {
        StringBuffer paramBuffer = new StringBuffer("\n\t[This: ");
        paramBuffer.append(joinPoint.getThis());

        Object[] arguments = joinPoint.getArgs();
        paramBuffer.append("\n\t[Args: (");
        for (int length = arguments.length, i = 0; i < length; ++i) {
            Object argument = arguments[i];
            paramBuffer.append(argument);
            if (i != length-1) {
                paramBuffer.append(', ');
            }
        }
        paramBuffer.append(")]");
        return paramBuffer.toString();
    }
}
```

1 Augments pointcut to avoid infinite recursion

2 Formats log message

- ❶ You augment the pointcut with `!execution(* Object.*(..))` to avoid the recursive invocation that is caused by executing methods defined in `Object` such as `toString()` and `hashCode()`. Without this modification, the logger will prepare the parameter string in `createParameterMessage()` when it calls `toString()` for each object. But when `toString()` executes, it first attempts to log the operation, and the logger will prepare a parameter string for it again when it calls `toString()` on the same object, and so on, causing an infinite recursion. By avoiding the join points for `toString()` execution, you avoid infinite recursion. But you generalize it a bit further, to avoid tracing calls such as `equals()` and `hashCode()` as well by excluding all methods in the `Object` class. Note that the `!within(TraceAspect)` pointcut isn't sufficient here because it only selects the *calls* to `toString()` methods made within the aspect; the execution of the methods is still advised.
- ❷ The `createParameterMessage()` helper method returns a formatted string containing the object and arguments.

Now, when you compile the classes with this aspect and execute the `Main` class, you get output similar to the following that includes the invoked object and the method parameters:

```

Entering [Main.main(..)]
    [This: null]
    [Args: ([Ljava.lang.String;@119298d)]
Entering [ProductServiceImpl.findProduct(..)]
    [This: ajia.service.impl.ProductServiceImpl@e0fcac]
    [Args: (1001)]
Entering [JpaGenericRepository.find(..)]
    [This: ajia.repository.impl.JpaProductRepository@ecb3f1]
    [Args: (1001)]
Entering [OrderServiceImpl.addProduct(..)]
    [This: ajia.service.impl.OrderServiceImpl@c135d6]
    [Args: (ajia.domain.Order@b5ad68,Product: ProductA,1)]
Entering [InventoryServiceImpl.removeProduct(..)]
    [This: ajia.service.impl.InventoryServiceImpl@18baf36]
    [Args: (Product: ProductA,1)]
Entering [InventoryServiceImpl.isProductAvailable(..)]
    [This: ajia.service.impl.InventoryServiceImpl@18baf36]
    [Args: (Product: ProductA,1)]
Entering [JpaInventoryItemRepository.findByProduct(..)]
    [This: ajia.repository.impl.JpaInventoryItemRepository@19c4091]
    [Args: (Product: ProductA)]
Entering [InventoryItem.getQuantityOnHand()]
    [This: ajia.domain.InventoryItem@4b12d9]
    [Args: ()]
Entering [JpaInventoryItemRepository.findByProduct(..)]
    [This: ajia.repository.impl.JpaInventoryItemRepository@19c4091]
    [Args: (Product: ProductA)]
Entering [InventoryItem.deplete(..)]
    [This: ajia.domain.InventoryItem@4b12d9]
    [Args: (1)]
Entering [JpaGenericRepository.update(..)]
    [This: ajia.repository.impl.JpaInventoryItemRepository@19c4091]

```

```

    [Args: (ajia.domain.InventoryItem@4b12d9)]
Entering [Order.addProduct(..)]
    [This: ajia.domain.Order@b5ad68]
    [Args: (Product: ProductA,1)]
Entering [Order.isPlaced()]
    [This: ajia.domain.Order@b5ad68]
    [Args: ()]
Entering [Order.getItemFor(..)]
    [This: ajia.domain.Order@b5ad68]
    [Args: (Product: ProductA)]
Entering [Product.getPrice()]
    [This: Product: ProductA]
    [Args: ()]
Entering [LineItem.setQuantity(..)]
    [This: ajia.domain.LineItem@11ed166]
    [Args: (1)]
Entering [OrderServiceImpl.updateOrder(..)]
    [This: ajia.service.impl.OrderServiceImpl@c135d6]
    [Args: (ajia.domain.Order@b5ad68)]
Entering [JpaGenericRepository.update(..)]
    [This: ajia.repository.impl.JpaOrderRepository@45aa2c]
    [Args: (ajia.domain.Order@b5ad68)]

```

So far, you've used a logger specific to the aspect. Let's see how you can use the logger specific to the types being traced.

10.4.5 *Choosing a type-specific logger*

A common logging idiom is to use a type-specific logger so that a configuration file can control the information being logged by specifying the log level for each type or a set of types. For example, the following snippet in a `log4j.xml` file declares that the minimum required log level for a message to appear from types in `ajia.web` package is `warn`:

```

<logger name="ajia.web">
    <level value="warn"/>
</logger>

```

The aspects so far in this chapter haven't had this kind of type-specific control through configuration. Instead, you have one logger per aspect. This arrangement provides aspect-specific control. If you have multiple tracing aspects, you can control them separately using configuration such as the following:

```

<logger name="ajia.tracing.JDBCTraceAspect">
    <level value="warn"/>
</logger>

<logger name="ajia.tracing.ExceptionMonitorAspect">
    <level value="error"/>
</logger>

```

But if you want class-level control, you can do so easily using the `pertypewithin()` aspect association. Recall from section 6.2.4 that `pertypewithin()` associates the aspect state with a type. If you include an aspect member for the logger, a separate

aspect instance and therefore a separate logger are associated with each advised type. Listing 10.12 shows a tracing aspect that uses type-specific loggers.

Listing 10.12 **TracingAspect: tracing using type-specific logger**

```
package ajia.tracing;

import ...

public aspect TraceAspect pertypewithin(*) {
    private Logger logger;

    after() returning: staticinitialization(*) {
        logger = Logger.getLogger(getWithinTypeName());
    }

    pointcut traced()
        : execution(* *.*(..) && !within(TraceAspect));

    before() : traced() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        logger.log(Level.INFO,
            "Entering [" + sig.toShortString() + "]);
    }
}
```

- ❶ The `pertypewithin()` aspect association selects all types to avoid filtering due to the type pattern specified, if any. This way, the pointcut alone determines the selected join points (otherwise, implicit limiting of join points comes into effect, as discussed in section 6.2.5).
- ❷ After loading any class being logged, you initialize the `logger` member to get the logger object corresponding to that class. Note that `getWithinTypeName()` returns the name of the type with which the aspect instance is being associated. Because a separate aspect instance is associated with each matching type, the logger initialized this way is type-specific.

So far, we've focused on build-time source-code weaving. But a popular approach is to use AspectJ LTW and Spring AOP. Let's discuss these deployment options before we resume discussing additional monitoring techniques.

10.5 A detour: deployment options for monitoring aspects

AspectJ load-time weaving (LTW) introduces a weaver into the runtime system without affecting the build system, thus simplifying its use. Similarly, if your application is based on Spring, you can use Spring's proxy-based AOP that also obviates any build-time changes and goes a step further by avoiding any deployment changes. Let's use tracing as a specific technique to see how AspectJ LTW and Spring AOP can be used with monitoring.

10.5.1 Utilizing load-time weaving

When you're just starting to play with AspectJ, a tracing or performance-monitoring aspect (which we'll discuss in section 10.8) is a good starting point. As we discussed in

chapter 8, utilizing LTW at this stage simplifies the deployment task quite a bit. LTW weaving along with XML definitions for the concrete aspects let you change part of the system being monitored without any recompilation. This choice requires that you have a base aspect that declares the monitored pointcut as an abstract pointcut. When you're defining concrete aspects in XML, you provide a definition for the pointcut. Refer to section 8.3.2 for details.

Let's apply the trace aspect you developed in listing 10.9 to your web application through LTW. First, you'll need to write an `aop.xml` file describing the aspects to weave in and target classes to be woven in. Listing 10.13 shows an example `aop.xml` that you'll use for the example.

Listing 10.13 `aop.xml` to weave in trace aspect

```
<aspectj>
  <aspects>
    <concrete-aspect name="ajia.tracing.SystemWideTraceAspect"
                    extends="ajia.tracing.IndentationTraceAspect">
      <pointcut name="traced" expression="execution(* *.*(..))"/>
    </concrete-aspect>
  </aspects>

  <weaver>
    <include within="ajia.*"/>
    <exclude within="*.*EnhancerByCGLIB*.*"/>
    <exclude within="*.*.*$$EnhancerByCGLIB$$*"/>
    <exclude within="*.*.*$$FastClassByCGLIB$$*"/>
  </weaver>
</aspectj>
```

- ❶ In the `aspects` section, you specify that a subspect of `IndentationTraceAspect` should be woven in. You use a `<concrete-aspect>` element along with a pointcut definition that selects all join points in the system. Note that as explained in section 8.3.2, the aspects declared using `<concrete-aspect>` elements such as `ajia.tracing.SystemWideTraceAspect` doesn't exist in the Java or AspectJ source code.
- ❷ You specify types to be woven in. You include all types in direct and indirect subpackages of the `ajia` package. Because you use Hibernate as the JPA implementation, it generates additional classes dynamically (with either `EnhancerByCGLIB`, `$$EnhancerByCGLIB$$`, or `$$FastClassByCGLIB$$` as part of their names). You exclude those classes through a series of `<exclude>` elements.

When you deploy the code using LTW (either through the `aspectjweaver.jar` discussed in chapter 8 or Spring-driven LTW discussed in chapter 9) and add a few items to the cart using the web interface, you get the output similar to the following:

```
Entering [OrderController.addToCart(..)]
  Entering [OrderController.getCurrentOrder(..)]
    Entering [OrderServiceImpl.updateOrder(..)]
      Entering [JpaGenericRepository.update(..)]
    Entering [ProductServiceImpl.findProduct(..)]
      Entering [JpaGenericRepository.find(..)]
```

```

Entering [OrderServiceImpl.addProduct(..)]
  Entering [InventoryServiceImpl.removeProduct(..)]
    Entering [InventoryServiceImpl.isProductAvailable(..)]
      Entering [JpaInventoryItemRepository.findByProduct(..)]
        Entering [InventoryItem.getQuantityOnHand()]
      Entering [JpaInventoryItemRepository.findByProduct(..)]
    Entering [InventoryItem.deplete(..)]
  Entering [JpaGenericRepository.update(..)]
Entering [Order.addProduct(..)]
  Entering [Order.isPlaced()]
  Entering [Order.getItemFor(..)]
  Entering [Product.getPrice()]
  Entering [LineItem.setQuantity(..)]
Entering [OrderServiceImpl.updateOrder(..)]
  Entering [JpaGenericRepository.update(..)]
Entering [ProductController.productSummary()]
  Entering [ProductServiceImpl.findProducts()]
    Entering [JpaGenericRepository.findAll()]
Entering [DomainEntity.getId()]
Entering [Product.getName()]
Entering [DomainEntity.getId()]
Entering [Product.getName()]
Entering [DomainEntity.getId()]
Entering [Product.getName()]
Entering [DomainEntity.getId()]
Entering [Product.getName()]

```

I strongly recommend that you play with this configuration and see the effects.

If you're still new to AspectJ but already using Spring as an architectural basis, you may want to start simple by utilizing Spring's proxy-based weaving.

10.5.2 Utilizing Spring AOP for tracing

Consider a situation where you need to trace the internal workings or monitor the performance of controllers and repositories in a web application. Although AspectJ-based weaving will work, you can get the functionality without needing an AspectJ weaver (build-time or load-time). This can significantly reduce resistance in getting started with writing your own aspect. If you're using the Spring Framework, you're probably already using aspects shipped with the framework; but many projects could gain a lot from custom aspects to meet their specific needs.

You can use Spring AOP to introduce tracing without any logistical overhead. The downside is that you can only trace public method execution on Spring beans, but this is often sufficient in many enterprise applications.

Let's implement a tracing aspect as shown in listing 10.14. This aspect differs from listing 10.8 only in its use of the `@AspectJ` syntax to make it work with Spring AOP.

Listing 10.14 Tracing aspect written in `@AspectJ`

```

package ajia.tracing;
import ...
@Aspect

```

```

public class TraceAspect {
    private Logger logger = Logger.getLogger(TraceAspect.class);

    @Pointcut("execution(* *.*(..))")
    public void traced() {}

    @Before("traced()")
    public void trace(JoinPoint jp) {
        Signature sig = jp.getSignature();
        logger.log(Level.INFO,
            "Entering [" + sig.toShortString() + "]");
        NDC.push(" ");
    }

    @After("traced()")
    public void exit() {
        NDC.pop();
    }
}

```

You can apply this aspect by adding the configuration file in listing 10.15, which is a part of the application context.

Listing 10.15 Configuration to use the trace aspect: monitoring-context.xml

```

<beans ...>
    <aop:aspectj-autoproxy/>
    <bean id="traceAspect" class="ajia.tracing.TraceAspect"/>
</beans>

```

When you execute the web application and exercise the functionality, you get output such as the following (the shown output is produced when adding a product to the cart):

```

Entering [TransactionAttributeSource.getTransactionAttribute(..)]
Entering [PlatformTransactionManager.getTransaction(..)]
    Entering [EntityManagerFactoryInfo.getNativeEntityManagerFactory()]
    Entering [DataSource.getConnection()]
Entering [ProductService.findProduct(..)]
    Entering [GenericRepository.find(..)]
Entering [PlatformTransactionManager.commit(..)]
Entering [TransactionAttributeSource.getTransactionAttribute(..)]
Entering [PlatformTransactionManager.getTransaction(..)]
    Entering [EntityManagerFactoryInfo.getNativeEntityManagerFactory()]
    Entering [DataSource.getConnection()]
Entering [OrderService.addProduct(..)]
    Entering [TransactionAttributeSource.getTransactionAttribute(..)]
    Entering [PlatformTransactionManager.getTransaction(..)]
    Entering [InventoryService.removeProduct(..)]
        Entering [InventoryItemRepository.findByProduct(..)]
        Entering [InventoryItemRepository.findByProduct(..)]
        Entering [GenericRepository.update(..)]
    Entering [PlatformTransactionManager.commit(..)]
    Entering [GenericRepository.update(..)]
Entering [PlatformTransactionManager.commit(..)]

```

Notice that you see output corresponding to the services, repositories, transaction manager, JPA manager, and datasource objects—all of those are Spring beans. But you don't see any calls for the domain objects such as `Product` and `LineItem`. This is due to the use of Spring AOP, which only applies to Spring beans.

Now you can have monitoring enabled in your application without needing the AspectJ weaver. When your tracing needs to expand beyond what can be handled by Spring's proxy-based weaving—say, to enable intra-method call tracing—you can consider using the full power of AspectJ weaving.

10.6 Exception monitoring

Because exception throwing is an important event in the system, tracing such occurrences is typically desirable. Exception monitoring is an extension of the method-tracing concept, except the focus is on exceptional conditions in a program rather than the execution of methods. The conventional way to trace exceptions involves surrounding the interesting parts of code with a `try/catch` block and instrumenting each `catch` block with a log statement. With AspectJ, it's possible to log exceptions thrown by a method without any modification to the original code.

First-failure data capture

First-failure data capture (FFDC) functionality requires logging all the data (`this`, arguments, method information) at the point of the first failure. By combining the aspect in this section with that in section 10.4.4, you can easily implement FFDC.

In this section, you'll develop an aspect that enables the logging of thrown exceptions in the system. The aspect in listing 10.16 logs any method in the system that throws an exception.

Listing 10.16 `ExceptionTraceAspect`: tracing exceptions using `log4j`

```
package ajia.tracing;

import ...

public aspect ExceptionTraceAspect {
    private Logger logger = Logger.getLogger("exceptions");

    private ThreadLocal<Throwable> lastLoggedException
        = new ThreadLocal<Throwable>();

    pointcut exceptionTraced()
        : execution(* *.*(..) && !within(ExceptionTraceAspect));

    after() throwing(Throwable ex) : exceptionTraced() {
        if (lastLoggedException.get() != ex) {
            lastLoggedException.set(ex);
            Signature sig = thisJoinPointStaticPart.getSignature();
            logger.log(Level.ERROR,
```

1 Tracks last handled exception


2 Selects traced method pointcut

3 Advises exceptions thrown

```

        "Exception trace aspect ["
        + sig.toShortString() + "]", ex);
    }
}

```



3 Advises exceptions thrown

- ❶ To avoid logging the same exception at each level in the call stack, you keep track of the last logged exception. If another thread throws the same exception object, you want it to be considered a new exception and logged. The use of thread-local ensures such behavior. Note that you could keep all exceptions thrown in a collection, instead of storing just the last exception. But the most common way to handle an exception is catching and rethrowing the same or a wrapped exception. Therefore, you don't need to worry about an old exception being thrown after throwing a new exception. In those situations, the aspect ends up logging the old exception multiple times, which may be a good idea anyway.
- ❷ The `exceptionTraced()` pointcut selects all the methods that need exception logging. Here, you're defining this as an execution of any method in the system. You can modify this pointcut to include a subset of methods, as described in section 10.3.1.
- ❸ The after throwing advice collects the thrown object as context. The advice uses `thisJoinPointStaticPart` to log the information about selected join points. You check and set the `lastLoggedException` member to avoid repeated logging of the same exception.

Let's write a simple program (listing 10.17) to exercise your aspect. You use a nested method that throws an exception to show the behavior of avoiding repeated logging of the same exception.

Listing 10.17 A driver that tests exception logging

```

package ajia.main;

public class Main {
    public static void main(String[] args) {
        try {
            perform();
        } catch (Throwable ex) {
            System.out.println("Error occurred during execution");
        }
    }

    public static void perform() {
        nestedPerform();
    }

    public static void nestedPerform() {
        nestedNestedPerform();
    }

    public static void nestedNestedPerform() {
        throw new IllegalStateException("Simulated exception");
    }
}

```

When you compile the Main class along with `ExceptionTraceAspect` and run the driver program, you get the following output:

```
Exception trace aspect [Main.nestedNestedPerform()]
java.lang.IllegalStateException: Simulated exception
    at ajia.main.Main.nestedNestedPerform(Main.java:24)
    at ajia.main.Main.nestedPerform(Main.java:20)
    at ajia.main.Main.perform(Main.java:14)
    at ajia.main.Main.main(Main.java:7)
Error occurred during execution
```

The output shows that `IllegalStateException`, which was thrown by the `nestedNestedPerform()` method, is logged once, but never logged again. All customizations discussed for method tracing in section 10.4 apply here equally well, including using a type-specific logger and creating reusable aspects.

As discussed in section 10.4.3, sometimes you have to resort to conventional logging. Even then, AspectJ won't quit helping you; it's such a good friend, as we'll discuss next.

10.7 Improving conventional logging

Unlike tracing, logging tends to be specific to the application. For example, you can log all the steps in processing an order. In many situations, common parts of logging in a class or a set of classes can be extracted into aspects. Further, because you can log calls and not just execution, you can target calls made from a few specific classes to modularize those calls. Essentially, you can try to reduce logging into somewhat specific tracing. In any case, these aspects tend to be specific to those classes.

With all that said, you're still left with situations where there is no apparent commonality that can be extracted. In those cases, you implement logging in a conventional manner and use AspectJ to enhance the logging functionality and simplify the implementation. Specifically, you can modularize setting up NDC and MDC.

10.7.1 Modularizing NDC with conventional logging

Consider a long method with several log statements. You'll like setting up NDC around that method so that all statements use that context (for example, to provide the indentation effect). If you perform this functionality using conventional coding alone, you must ensure that you call `NDC.push()` upon entering and `NDC.pop()` before exiting each method. This usually involves using a `finally` block such as the following:

```
public void processOrder(Order order) {
    try {
        NDC.push(" ");
        ... business logic and logging
    } finally {
        NDC.pop();
    }
}
```

If you don't need the indentation effect but rather wish to obtain nested caller information associated with each log statement, you use a string identifying the operation as the argument to the `NDC.push()` method. For example, you can use `NDC.push("processOrder")`. If done consistently, you get nested context in output such as “placeOrder processOrder”, if the `placeOrder()` method calls the `processOrder()` method.

This kind of arrangement is error prone. In particular, it's common to see a call to `NDC.pop()` without using the `finally` block (thus not clearing the context in case of an exception). AOP can help simplify such an implementation and provide consistency. As shown in listing 10.18, you can set up NDC to perform the indentation in a way similar to the indentation from listing 10.8.

Listing 10.18 Context indentation aspect

```
package ajia.logging;
import org.apache.log4j.NDC;

public abstract aspect ContextIndentationAspect {
    public abstract pointcut logContextOp();
    before(): logContextOp() {
        NDC.push(" ");
    }
    after(): logContextOp() {
        NDC.pop();
    }
}
```

The aspect declares an abstract pointcut and advises it to push and pop a few spaces. The derived aspects provide a definition for the `logContextOp()` pointcut to select the caller of the log method. If you need to establish nested context corresponding to the caller method, you can push `thisJoinPointStaticPart.getSignature().getName()` instead of spaces. You implement a subspect as shown in listing 10.19 to apply the context indentation to all service types.

Listing 10.19 System context indentation aspect

```
package ajia.logging;

public aspect SystemContextIndentationAspect
    extends ContextIndentationAspect {
    public pointcut logContextOp(): execution(* ajia.service..*(..));
}
```

To exercise this class, let's add the `ShippingServiceImpl` class (listing 10.20), which performs manual logging. It uses a few classes that we don't show, because all they do is log a message in each method.

Listing 10.20 Shipping service implementation

```
package ajia.service.impl;
import ...
```

```

public class ShippingServiceImpl implements ShippingService {
    private Logger logger = Logger.getLogger(ShippingServiceImpl.class);
    private CourierService courierService = new CourierServiceImpl();
    private EmailerService emailer = new EmailerServiceImpl();

    public void processOrder(Order order) {
        logger.log(Level.INFO,
            "[ShippingServiceImpl.processOrder] Processing order "
            + order.getId());

        Package packageToSend = createPackage(order);

        logger.log(Level.INFO,
            "[ShippingServiceImpl.processOrder] Notifying courier "
            + "service with " + packageToSend);
        Tracking tracking = courierService.send(packageToSend);

        logger.log(Level.INFO,
            "[ShippingServiceImpl.processOrder] Sending email with "
            + tracking);
        emailer.send(tracking);

        logger.log(Level.INFO,
            "[ShippingServiceImpl.processOrder] Finished processing");
    }

    private Package createPackage(Order order) {
        logger.log(Level.INFO,
            "[ShippingServiceImpl.createPackage] Creating package");
        return new Package("1234");
    }
}

```

When you call the `processOrder()` method from a driver class, you get the following output:

```

[ShippingServiceImpl.processOrder] Processing order 1000
  [ShippingServiceImpl.createPackage] Creating package
[ShippingServiceImpl.processOrder] Notifying courier service with
➤ Package number: 1234
  [CourierServiceImpl.send] Sending package
[ShippingServiceImpl.processOrder] Sending email with
➤ Tracking number: 12341234
[ShippingServiceImpl.processOrder] Finished processing

```

As you can see, the indentation makes output easier to comprehend by clarifying the caller-callee relationship. Note that following the warning in section 10.1 against using `%C`, `%F`, `%L`, `%M`, and `%l` layout patterns, you explicitly pass the class and method name to each log statement. Let's simplify that using AOP and MDC.

10.7.2 Modularizing MDC with conventional logging

You can also use MDC to enhance log messages. In the simplest form, you can register the caller method name and the defining type as the context. This is a much cheaper way to include the caller information without having to use a pattern that deduces the information by examining the call stack. Listing 10.21 shows an aspect that sets the caller name using `thisEnclosingJoinPointStaticPart`.

Listing 10.21 MDC establishment aspect

```

package ajia.logging;

import ...

public aspect MDCEstablishmentAspect {
    private static final String MDC_KEY = "caller";

    pointcut logCall() : call(* Logger.log(..));

    before() : logCall() {
        MDC.put(MDC_KEY,
            thisEnclosingJoinPointStaticPart.getSignature().toShortString());
    }

    after() : logCall() {
        MDC.remove(MDC_KEY);
    }
}

```

Next, because the aspect establishes the caller context, you don't need to add that information in each log statement. For example, the following log statement from the `ShippingServiceImpl` in listing 10.20 changes from

```

logger.log(Level.INFO,
    "[ShippingServiceImpl.processOrder] Processing order "
    + order.getId());

```

to a much simpler:

```

logger.log(Level.INFO,
    "Processing order " + order.getId());

```

Now you can use a pattern such as `[%X{caller}]` to log this context information. When you invoke the `processOrder()` method, you see output similar to the following:

```

[ShippingServiceImpl.processOrder(..)] Processing order 1000
[ShippingServiceImpl.createPackage(..)] Creating package
[ShippingServiceImpl.processOrder(..)] Notifying courier service with
➤ Package number: 1234
[CourierServiceImpl.send(..)] Sending package
[ShippingServiceImpl.processOrder(..)] Sending email with
➤ Tracking number: 12341234
[ShippingServiceImpl.processOrder(..)] Finished processing

```

You can combine MDC with NDC (discussed in the previous section). Furthermore, you can use any other diagnostic context to be automatically available for logging. For example, in the following snippet, we use Spring Security API to add context information related to the accessing user:

```

MDC.push("user",
    SecurityContextHolder.getContext().getAuthentication().getPrincipal());

```

The other context that you can establish using MDC includes the ongoing transaction's identifier, the business-specific context such as the shopping card identifier, and so on. Imagine the kind of tangling that would result if you were to provide diagnostic

context using conventional means alone. With AspectJ helping with crosscutting functionality, you can concentrate on only the bare minimum work.

That completes our discussion of tracing. Let's now move our focus to another kind of common technique—performance monitoring.

10.8 Performance monitoring

Performance monitoring involves measuring the time taken by interesting parts of the system as well as the number of times a particular method is invoked. Depending on how it's used, you can monitor activities at various levels. For example, in a web application, you can monitor requests by measuring time taken at the servlet and controller layers. You can also monitor the service layer to gain more fine-grained information. For development purposes, you can focus on parts suspected of slowing down the system. For example, you can monitor JDBC or ORM calls to monitor database access. You can also monitor the concurrency characteristics of the system to determine parameters such as thread pool size and throttling.

Performance monitoring of an application involves recording vital parameters of parts of the system: how long it takes to execute certain functionality, how many calls have been made for a particular method, and so on. You can extend the basic idea of tracing to implement performance monitoring. The idea is to compute the difference in timestamps before and after each monitored operation and record that difference against the currently advised join point. You can also count the number of invocations for each advised operation.

Let's develop a simple aspect to show this functionality. You'll also use `@AspectJ` syntax to show that most monitoring aspects can be implemented easily in either the traditional syntax or the `@AspectJ` syntax. The use of the `@AspectJ` syntax will also allow you to use the aspect through Spring's proxy-based weaving.

Prebuilt solutions for monitoring

The purpose of aspects in this section is to illustrate the core performance monitoring technique. For a more complete solution, you may want to consider prebuilt solutions such as Glassbox and the SpringSource Application Monitoring Suite (AMS) that use AspectJ internally as an implementation technology.

Listing 10.22 shows an abstract aspect that uses `System.nanoTime()` as the way to get timing information.

Listing 10.22 AbstractPerformanceMonitoringAspect: monitoring operations

```
package ajia.monitoring;
import ...

@Aspect
public abstract class AbstractPerformanceMonitoringAspect {
```

```

private Logger logger
    = Logger.getLogger(AbstractPerformanceMonitoringAspect.class);

@Pointcut
public abstract void monitoredOp();

@Around("monitoredOp()")
public Object monitor(ProceedingJoinPoint pjp) throws Throwable{
    long start = System.nanoTime();
    try {
        return pjp.proceed();
    } finally {
        long complete = System.nanoTime();
        logger.log(Level.INFO,
            "Operation " + pjp.getSignature().toShortString()
            + " took " + (complete-start) + " nanoseconds");
    }
}
}

```

You're using the timing API directly. But it's best to encapsulate such functionality into a separate interface to allow a pluggable implementation. For example, you can use *Jamon* (<http://jamonapi.sourceforge.net>) or *Simon* (<http://code.google.com/p/javasimon>) to simplify gathering data and computing statistics. It's also best to leverage a DI mechanism to make such pluggability a configuration option.

Let's enable monitoring of repository classes by creating a subaspect as shown in listing 10.23.

Listing 10.23 `SystemMonitoringAspect`: monitoring repository operations

```

package ajia.monitoring;

import ...

@Aspect
public class SystemMonitoringAspect extends
    AbstractPerformanceMonitoringAspect {
    @Pointcut("execution(* ajia.*.*(..)) && !within(ajia.tracing..*)")
    public void monitoredOp() {}
}

```

You can now enable this aspect through AspectJ LTW using an `<aspect>` element inside `aop.xml`. Alternatively, you can define a subaspect of `AbstractPerformanceMonitoringAspect` using an `<concrete-aspect>` element to the same effect. You may also use build-time weaving to weave the aspect prior to deployment. Either way, when a user adds an item to a cart, you get output similar to the following:

```

Operation InventoryItem.getQuantityOnHand() took 14760 nanoseconds
Operation InventoryServiceImpl.isProductAvailable(..) took
➤ 54309707 nanoseconds
Operation JpaInventoryItemRepository.findByProduct(..) took
➤ 762442 nanoseconds
Operation InventoryItem.deplete(..) took 10447 nanoseconds
Operation JpaGenericRepository.update(..) took 52935 nanoseconds
Operation InventoryServiceImpl.removeProduct(..) took 58732125 nanoseconds

```

```

Operation Order.isPlaced() took 8793 nanoseconds
Operation Order.getItemFor(..) took 26898 nanoseconds
Operation Product.getPrice() took 2481 nanoseconds
Operation LineItem.setQuantity(..) took 13210 nanoseconds
Operation Order.addProduct(..) took 610647 nanoseconds
Operation JpaGenericRepository.update(..) took 7343705 nanoseconds
Operation OrderServiceImpl.updateOrder(..) took 7466786 nanoseconds
Operation OrderServiceImpl.addProduct(..) took 67595893 nanoseconds
Operation OrderController.addToCart(..) took 102952612 nanoseconds
Operation JpaGenericRepository.findAll() took 675822 nanoseconds
Operation ProductServiceImpl.findProducts() took 853805 nanoseconds
Operation ProductController.productSummary() took 1350510 nanoseconds
...

```

You can play with the aspect to make the output more interesting and useful. For example, you may

- Modify the pointcuts to select join points of your interest.
- Modify the base aspect to log more information about the join points.
- Apply NDC context to examine how much time is spent in each constituent method.

The same aspect can be applied using Spring AOP by adding the code in listing 10.24 into a file that forms the application context.

Listing 10.24 Adding performance monitoring through Spring AOP

```

<beans ...>
  <context:component-scan base-package="ajia" >
    <context:exclude-filter expression="ajia.web.*" type="aspectj"/>
  </context:component-scan>

  <aop:aspectj-autoproxy/>

  <bean id="monitoringAspect"
        class="ajia.monitoring.SystemMonitoringAspect"/>
</beans>

```

Now, when you run the web application and add a product to the shopping cart, you get output similar to the following:

```

Operation GenericRepository.find(..) took 36026925 nanoseconds
Operation ProductService.findProduct(..) took 39170621 nanoseconds
Operation InventoryItemRepository.findByProduct(..) took
↳ 160806192 nanoseconds
Operation InventoryItemRepository.findByProduct(..) took
↳ 4458387 nanoseconds
Operation GenericRepository.update(..) took 86045 nanoseconds
Operation InventoryService.removeProduct(..) took 166793545 nanoseconds
Operation GenericRepository.update(..) took 16229158 nanoseconds
Operation OrderService.addProduct(..) took 184289344 nanoseconds

```

Notice that only Spring beans and not other classes such as domain entities are being monitored. This is due to the proxy-based AOP being applied only to the Spring beans.

You can also extend AspectJ-based profiling functionality to implement modular dynamic service-level monitoring. For example, let's say that you're using some

third-party services such as credit-card approval processing over the internet. You may have an agreement that provides you with certain performance guarantees. You can collect the time before and after each invocation of the services. When the service gets near or below the agreed level, you can alert the provider as well as use the information to collect penalties, if the agreement so specifies. If you're on the provider side, you can use the profile information to create alerts when the level of service approaches the agreed level. Such alerts may help you fix the problem before it reaches a critical level.

Let's complete the discussion of AOP-based monitoring by examining how you can control aspects once they're deployed.

10.9 *Runtime control of monitoring aspects*

Monitoring aspects often need to be turned on or off in production. This reduces the overhead associated with monitoring and the amount of data reported, thus improving comprehension.

A simple way to control an aspect is by using an `if()` check evaluating a boolean field and exposing that field through JMX. If you need to control the applicability of the aspect at application startup, you can assign the boolean field a system property (or a property read from a property file). You can also combine these two techniques to control the default applicability of the aspect as well as allow runtime control, as shown in listing 10.25.

Listing 10.25 *Enabling and disabling monitoring at runtime*

```
package ajia.monitoring;

import ...

@Aspect
public abstract class AbstractPerformanceMonitoringAspect {
    private Logger logger
        = Logger.getLogger(AbstractPerformanceMonitoringAspect.class);
    private boolean enabled = Boolean.getBoolean("ajia.monitoring.enable");

    @Pointcut
    public abstract void monitoredOp();

    @Around("monitoredOp() && !within(AbstractPerformanceMonitoringAspect)")
    public Object monitor(ProceedingJoinPoint pjp) throws Throwable {
        if(!isEnabled()) {
            return pjp.proceed();
        }
        long start = System.nanoTime();
        try {
            return pjp.proceed();
        } finally {
            long complete = System.nanoTime();
            logger.log(Level.INFO,
                "Operation " + pjp.getSignature().toShortString()
                + " took " + (complete-start) + " nanoseconds");
        }
    }
}
```

```

public void setEnabled(boolean enabled) {
    this.enabled = enabled;
}

public boolean isEnabled() {
    return this.enabled;
}
}

```

The aspect includes a boolean field so that it can be exposed to a JMX console. You assign this field the value of the `ajia.monitoring.enable` system property so that unless you start the application with the `-Dajia.monitoring.enable=true` argument, monitoring is disabled. You use this field inside the advice to control its application. Note that you can use an `if()` pointcut, if the aspect is woven using the AspectJ weaver (which can then perform certain optimizations to avoid creation of the `pjp` object if the condition in the pointcut evaluates to `false`). But because Spring AOP doesn't support the `if()` pointcut, you embed the check inside the advice.

For runtime control, you need to expose the field through JMX. If you're using the Spring Framework, exposing the aspect through JMX is a trivial task, as shown in listing 10.26 (assumes AspectJ LTW deployment).

Listing 10.26 Exposing the monitoring aspect using Spring's JMX support

```

<beans ...>
  <bean id="monitorAspect"
    class="ajia.monitoring.SystemMonitoringAspect"
    factory-method="aspectOf"/>
  <bean class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <util:map>
        <entry key="ajia:name=monitorAspect"
          value-ref="monitorAspect"/>
      </util:map>
    </property>
  </bean>
</beans>

```

1 Aspect as Spring bean

Exposes aspect over JMX

2

- ❶ You create a Spring bean corresponding to the aspect. Note that if you use Spring's proxy-based AOP, you omit the `factory-method="aspectOf"` part.
- ❷ You use Spring support for exporting beans to JMX to export the aspect instance under the `ajia:name=monitorAspect` object name.

Now, when you connect to the application using a JMX console such as JConsole, as shown in figure 10.3, you see a bean named `monitorAspect` and can modify its `enabled` property. (You may have to specify `-Dcom.sun.management.jmxremote` to the VM, depending on your web server and the VM).

As you can see, implementing performance monitoring using AspectJ is just as easy as tracing. At this point, if you haven't already done so, I urge you to download the source code for the book and try it yourself before proceeding to the next chapter.

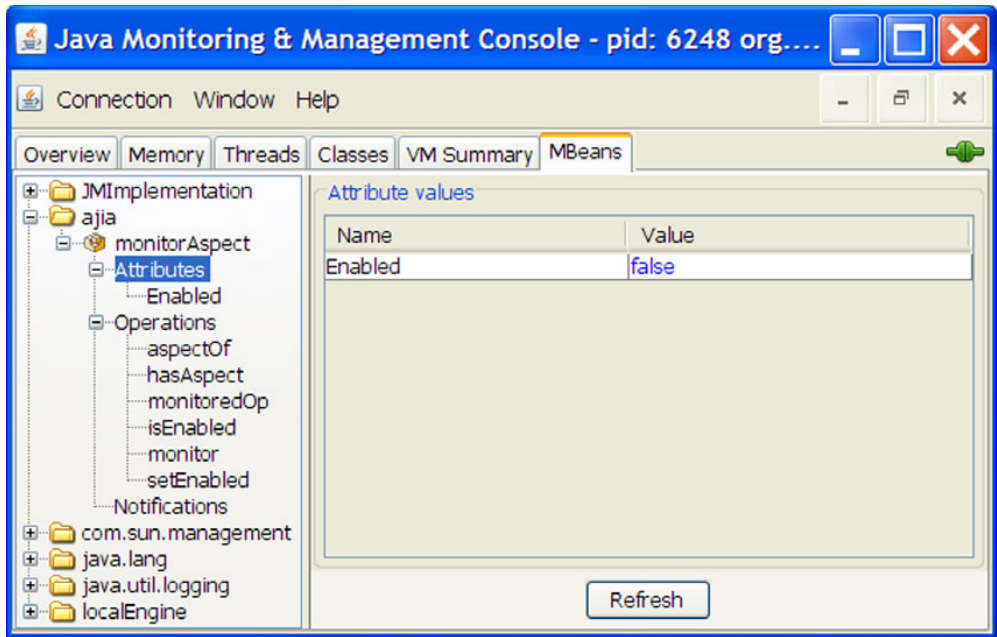


Figure 10.3 JConsole to monitor and control aspect exposed through JMX.

10.10 Summary

Software developers and management often look for a killer application: one that is so well suited that it makes adopting a new technology worthwhile, despite the risks. The reason behind this conservative approach is to balance the considerable investment associated with any new technology against the benefits it offers. A killer application supposedly provides enough benefits to outweigh the risks. But such applications are hard to find. Add to that the difficulty in measuring largely qualitative benefits, such as productivity improvements, cleaner design, ease of evolution, and improved quality. Such qualitative benefits make proving the advantages of a new approach to a skeptic challenging. The more practical approach is to find ways to reduce the investment involved with the new technology. If you can achieve such reduction, you no longer have to wait until you see a bonanza of benefits.

AspectJ-based tracing and monitoring techniques offer low-investment, low-risk ways to begin using AspectJ. The aspects and idioms presented in this chapter may be all that you need to start applying them in real world applications. These aspects also offer a unique plug-and-play nature. LTW offers an attractive way to weave aspects while leaving your build environment intact. The use of LTW further enhances the plug-and-play nature of the solution; a simple modification to the startup script is all you need to add aspects to or remove them from your system.

If this chapter has convinced you of the benefits of using AspectJ for monitoring, you may start out by using it for tracing and performance monitoring to understand

system interaction and performance bottlenecks. Later, you can demonstrate to your team the benefits you've experienced, which may lead them to adopt AspectJ as well. At any point, including during the final shipment, you can exclude the AspectJ and monitoring aspects. The overall effect is that you can start using AspectJ with minimal risk.

When you commit to AspectJ-based monitoring, you'll start seeing even more benefits. You can use AspectJ-based solutions for auditing and production performance monitoring. Implementation of all these concerns is now nicely modularized. This solution leads to increased flexibility, improved accuracy, and better consistency. It saves you from the laborious and boring task of writing nearly identical log statements in code all over your system. The use of AspectJ also makes the job of switching logging toolkits an easy task. You can start with any one that you're familiar with and feel comfortable that changing the choice later on will require modifying only a few statements.

Although it isn't a killer application, monitoring may be the perfect way to introduce yourself and your organization to AspectJ.

“This book teaches you how to think in aspects. It is essential reading for both beginners who know nothing about AOP and experts who think they know it all.”

— Andrew Eisenberg, AspectJ Development Tools Project Committer

AspectJ IN ACTION Ramnivas Laddad FOREWORD BY ROD JOHNSON

To allow the creation of truly modular software, OOP has evolved into aspect-oriented programming. AspectJ is a mature AOP implementation for Java, now integrated with Spring.

AspectJ in Action, Second Edition is a fully updated, major revision of Ramnivas Laddad’s best-selling first edition. It’s a hands-on guide for Java developers.

After introducing the core principles of AOP, it shows you how to create reusable solutions using AspectJ 6 and Spring 3. You’ll master key features including annotation-based syntax, load-time weaver, annotation-based crosscutting, and Spring-AspectJ integration. Building on familiar technologies such as JDBC, Hibernate, JPA, Spring Security, Spring MVC, and Swing, you’ll apply AOP to common problems encountered in enterprise applications.

This book requires no previous experience in AOP and AspectJ, but it assumes you’re familiar with OOP, Java, and the basics of Spring.

What’s Inside

- Totally revised Second Edition
- When and how to apply AOP
- Master patterns and best practices
- Code you can reuse in real-world applications

An expert in enterprise Java, **Ramnivas Laddad** is well known in the area of AOP and Spring. He is a committer on the Spring Framework project.

For online access to the author and a free ebook for owners of this book, go to manning.com/AspectJinActionSecondEdition



“Clear, concisely worded, well-organized ... a pleasure to read.”

— From the Foreword by Rod Johnson, Creator of the Spring Framework

“Ramnivas showcases how to get the best out of AspectJ and Spring.”

— Andy Clement
AspectJ Project Lead

“One of the best Java books in years.”

— Andrew Rhine
Software Engineer, eSecLending

“By far the best reference for Spring AOP and AspectJ.”

— Paul Benedict, Software Engineer,
Argus Health Systems

“Ramnivas expertly demystifies the awesome power of aspect-oriented programming.”

— Craig Walls
author of *Spring in Action*

