What is aspect-oriented programming?

As stated earlier, aspects help to modularize cross-cutting concerns. In short, a cross-cutting concern can be described as any functionality that affects multiple points of an application.

ADVICE

When a meter reader shows up at your house, his purpose is to report the number of kilowatt hours back to the electric company. Sure, he has a list of houses that he must visit, and the information he reports is important. But the actual act of recording electricity usage is the meter reader’s main job.

Likewise, aspects have a purpose – a job they’re meant to do. In AOP terms, the job of an aspect is called advice.

Advice defines both the *what* and *when* of an aspect. In addition to describing the job that an aspect will perform, advice *addresses the question of when to perform* the job.

Spring aspects can work with five kinds of advice:

* Before 🡪 The advice functionality takes place before the advised method is invoked.
* After 🡪The advicce functionality takes place after the advised method completes, regardless

of the outcome.

* After-returnning 🡪 The advice functionality takes place after the advised method successfully completes.
* After-throwing 🡪 The advice functionality takes place after the advised method throws

an exception.

* Around 🡪 The advice wraps the the advised method, providing some functionality before and after the advised method is invoked.

Join Points

An electric company services several houses, perhaps even an entire city. Each house has an electric meter that needs to be read, so each house is a potential target for the meter reader. The meter reader could potentially read all kinds of devices, but to do her job, she needs to target electric meters that are attached to houses.

In the same way, your application may have thousands of opportunities for advice to be applied. These opportunities are known as join points. A *join point* is a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modifed. These are the points where your aspect’s code can be inserted into the normal flow of your application to add new behavior.

POINTCUTS

It’s not possible for any one meter reader to visit all houses serviced by the electric company. Instead, each one is assigned a subset of all the houses to visit. Likewise, an aspect doesn’t neccessarily advise all join points in an application. Pointcuts help narrow down the join points advised by an aspect.

If advice defines the *what* and *when* of aspects, then pointcuts define the *where*. A pointcut definition matches one or more join points at which advice should be woven. Often you specify these pointcuts using explicit class and method names or through regular expressions that define matching class and method name patterns. Some AOP frameworks allow you to create dynamic pointcuts that determine whether to apply advice based on runtime decisions, such as the value of method patterns.

ASPECTS

When a meter reader starts his day, he knows both what he’s supposed to do (report electricity) and which houses to collect that information from. Thus he knows everything he needs to know to get his job done.

An aspect is the merger of advice and pointcuts. Taken together, advice and point-cuts define everything there is to know about an aspect – what it does and where and when it does it.

INTRODUCTIONS

An *introduction* allows you to add new methods or attributes to existing classes. For example, you could create an *Auditable* advice class that keeps the state of when an object was last modified. This could be as simple as having one method, *setLastModified(Date),* and instance variable can then be introduced to existing classes without having to change them, giving them new behavior and state.

WEAVING

*Weaving* is the process of applying aspects to a target object to create a new proxied object. The aspects are woven into the target object at the specified join points. The weaving can take place at several points in the target object’s lifetime:

* Compile Time – Aspects are woven in when the target class is compiled. This requires a special compiler. AspectJ’s weaving compiler weaves aspects this way.
* Class Load Time—Aspects are woven in when the target class is loaded into the JVM. This requires a special ClassLoader that enhances the target class’s byte-code before the class is introduced into the application. AspectJ 5’s *load-time weaving(LTW)* support weaves aspects this way.
* Runtime—Aspects are woven in sometime during the execution of the application. Typically, an AOP container dynamically generates a proxy object that delegates to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

Spring’s AOP support

There’s a lot of synergy between the Spring and AspectJ projects, and the AOP support in Spring borrows a lot of from AspectJ project.

Spring’s support for AOP comes in four styles:

* Classic Spring proxy-based AOP
* Pure-POJO aspects
* @AspectJ annotation-driven aspects
* Injected AspectJ aspects (available in all versions of Spring)

The first three styles are all variations on Spring’s own AOP implementation. Spring AOP is built around dynamic proxies. Consequently, Spring’s AOP support is limited to method interceptions.

SPRING ADVICE IS WRITTEN IN JAVA

All the advice you create in Spring is written in a standart Java class. That way, you get the benefit of developing your aspects in the same integrated development environment(IDE) you’d use for normal Java development. The pointcuts that define where advice should be applied may be specified with annotations or configured in a Spring XML configuration, but either will be familiar to Java developers.

SPRING ADVISES OBJECTS AT RUNTIME

In Spring, aspects are woven into Spring-managed beans at runtime by wrapping them with a proxy class. The proxy class poses as the target bean, intercepting advised method calls and forwarding those calls to the target bean. Between the time when the proxy intercepts the method call and the time when it invokes the target bean’s method, the proxy performs the aspect logic. Spring doesn’t create a proxied object until that proxied objects will be created when it loads all the beans from the BeanFactory. Because Spring creates proxies at runtime, you don’t need a special compiler to weave aspects in Spring ‘s AOP.

SPRING ONLY SUPPORTS METHOD JOIN POINTS

As mentioned earlier, multiple join-point models are available through various AOP implementations. Because it’s based on dynamic proxies, Spring only supports method join points. Because it’s based on dynamic proxies, Spring only supports method join points. This is in conteast to some other AOP frameworks, such as AspectJ and JBoss, which provide field and constructor join points in addition to method pointcuts. Spring’s lack of field pointcuts prevents you from creating very fine-grained advice, such as intercepting updates to an object’s field. And without constructor pointcuts, there’s no way to apply advice when a bean is instantiated.

But method interception should suit most, if not all, of your needs. If you find yourself in need of more than method interception, you’ll want to complement Spring AOP with AspectJ.

Selecting join points with pointcuts

As mentioned before, pointcuts are used to pinpoint where an aspect’s advice should be applied. Along with an aspect’s advice, pointcuts are among the most fundamental elements of an aspect. Therefore, it’s important to know how to write pointcuts. In Spring AOP, pointcuts are defined using AspectJ’s pointcut expression language.

The most important thing to know about AspectJ pointcuts as they pertain to Spring AOP is that Spring only supports a subset of the pointcut designators available in AspectJ.

-args() , @args(), execution(),this(),target(), @target(), within(), @within(),@annotation

As you browse through the supported designators, note that the execution designator is the only one that actually performs matches. The other designators are used to limit those matches. This means execution is the primary designator you’ll use in every pointcut definition you write. You’ll use the other designators to contain the pointcut’s reach.

*Writing pointcuts*

To demonstrate aspects in Spring, you need something to be the subject of the aspect’s pointcuts. For that purpose, let’s define a Performance interface:

Package concert;

public interface Performance{

public void perform();

}

Performance represents any kind of live performance, such as a stage play, a movie, or a concert. Let’s say that you want to write an aspect that triggers off Performance’s perform() method. The figure at bellow shows a poincut expression that can be used to apply advice whenever the perform() method is executed.

//------: execution(\* concert.Performance.perform(..))

You use the execution() designator to select Performance’s perform method. The method specification starts with an asterisk, which indicates that you don’t care what type the method returns. Then you specify the fully qualified class name and the double dot(..) indicating that the pointcut should select any perform() method, no matter what the argument list is.

Now let’s suppose that you want to confine the reach of that pointcut to only the concert package. In that case, you can limit the match by tacking on a within() designator, as a shown at bottom:

//------: execution(\* concert.Performance.perform(..)) && within(concert.\*)

Note that you use the && operator to combine the execution() and within designators in an “and” relationship (where both designators must match for the pointcut to match). Similary, you could use the || operator to indicate an “or” relationship. And the ! operator can be used to negate the effect of a designator.

Because ampersands have special meaning in XML-based configuration. Likewise, or and not can be used in place of || and ! , respectively.

Selecting beans in pointcuts

Spring adds a bean() designator that lets you identify beans by their ID in a pointcut expression. bean() takes a bean ID or names as an argument and limits the pointcut’s effect to that specific bean.

For example, consider the following pointcut:

//------execution(\* concert.Performance.perform()) and bean(‘woodstock’)

Here you’re saying that you want to apply aspect advice to the execution of Performance’s perform() method, but limited to the bean whose ID is woodstock. Narrowing a pointcut to a specific bean may be valuable in some cases, but you can also use negation to apply an aspect to all beans that don’t have a specific ID:

//------ execution(\* concert.Performance.perform()) and !bean(‘woodstock’)

In this case, the aspect’s advice will be woven into all beans whose ID isn’t ‘woodstock’.

Creating annotated aspects

A key feature introduced in AspectJ 5 is the ability to use annotations to create aspects. Prior to AspectJ 5, wiring AspectJ aspects involved learning a Java language extension. But AspectJ 5, wiring AspectJ’s annotation-oriented model makes it simple to turn any class into an aspect by sprinkling a few annotations around.

You’ve already defined the Performance interface as the subject of your aspect’s pointcuts.Now let’s use AspectJ annotations to create an aspect.