--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.