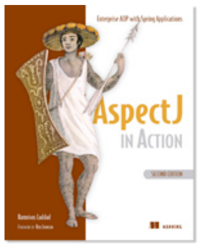

AOP by Analogy

By Ramnivas Laddad, author of *AspectJ in Action, Second Edition*

For core concerns, object-oriented programming (OOP), the dominant methodology employed today, does a good job. You can immediately see a class such as `LoanManagementService` *implementing business logic and* `AccountRepository` *implementing data access. But what about crosscutting concerns? Wouldn't it be nice if you could implement a module that you identify as* `Security`*,* `Auditing`*, or* `Performance-Monitor`*? You can't do that with OOP alone. Instead, OOP forces you to fuse the implementation of these functionalities in many modules. This is where aspect-oriented programming (AOP) helps. In this article, based on chapter 1 of AspectJ in Action, Second Edition, the author helps you to understand AOP by comparing it with other technologies you may be more familiar with.*

You may also be interested in…

Aspect-oriented programming (AOP) is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization—an aspect. Each aspect focuses on a specific crosscutting functionality. The core classes are no longer burdened with crosscutting concerns. An aspect weaver composes the final system by combining the core classes and crosscutting aspects through a process called weaving. Thus, AOP helps to create applications that are easier to design, implement, and maintain.

When you're learning a new technology, it sometimes helps to compare it with existing technologies. In this section, we'll attempt to help you understand AOP by comparing it with Cascading Style Sheets (CSS), database programming, and event-oriented systems. The purpose of this article is to help those familiar with at least one of these technologies to understand AOP by analogy.

Cascading Style Sheets (CSS)

CSS is a widely supported mechanism to separate content from presentation in HTML pages. Without CSS, formatting information is fused with content (causing tangling), and similar content elements have presentational information spread into multiple places (causing scattering). CSS helps the situation by letting the main document focus on content by separating the formatting information into a document called a stylesheet.

A core concept in CSS is a *selector* that selects document elements matching a certain specification. For example, the `body p` selector can select paragraphs inside the body element. You can then associate presentational information with a selector and, for example, set the background color of such elements to blue using the `body p {background: blue;}` element.

AOP acts on classes in the same way that CSS acts on documents. AOP lets you separate crosscutting logic from the main-line logic. AOP's pointcuts have the same selection role as CSS selectors. Whereas CSS selectors select structural elements in a document, pointcuts select program elements. Similarly, the blocks describing the formatting information are analogous to AOP advice in functionality.

Often, the selection mechanism requires more information than merely using the inherent characteristics of a structure such as `body p`. It's common practice to supplement content elements with additional metadata through the `class` attribute. For example, you can mark an HTML paragraph element as `menu` by using the tag `<p class="menu">`. Then, in the stylesheet, you can select such an element by using the `p.menu` selector and apply appropriate presentation characteristics. In AOP, practitioners face the same problem—selection through a pointcut often requires information beyond merely relying on inherent characteristics of the program elements such as class and method names. The use of Java annotations plays a role similar to the class attribute in HTML documents. You can, for example, mark a method as `@Transactional` and utilize it in a pointcut expression.

There are similarities from the adoption perspective as well. Through WYSIWYG HTML editors, it's easy to create a good-looking HTML. That apparent simplicity led to many initial web documents embedded with formatting information. But when we realized that it's difficult to create a consistent look when every element's formatting is specified independently, developers started to look favorably at CSS. AOP has encountered a similar trend. There is a level of comfort in embedding the implementation of crosscutting functionality inside classes; you can see exactly what's going to happen. But you soon realize that creating a consistent implementation is nearly impossible when similar code is scattered in many places. In addition, using CSS requires a level of expertise and understanding of the semantics associated with the elements in a document. Using AOP requires similar understanding of the semantic separation between core and crosscutting elements.

CSS works at the structure level, and database triggers offer similar separation at the programming level. Let's see how that technique compares with AOP.

Database systems

Database systems offer "dynamic crosscutting" targeted toward data access, whereas AOP offers a similar mechanism toward general programming. It offers two good analogies to AOP concepts: SQL with pointcuts and triggers with advice.

SQL and pointcuts

A join point is like a row in a database, whereas a pointcut is like an SQL query. An SQL query selects rows according to a specified criterion such as "rows in accounts table, where the balance is greater than 50." It provides access to the content of the selected rows. Similarly, a pointcut is a query over program execution that selects join points according to a specified criterion such as "method execution in the `Account` class, where the method name starts with 'set'". It also provides access to the join point context (objects available at the join point, such as method arguments).

Triggers and advice

Database programming often uses triggers to respond to changes made in data. For example, you can use a trigger to audit changes in certain tables. The following snippet calls the `logInventoryIncrease()` procedure when inventory increases:

```
CREATE OR REPLACE TRIGGER inventory_increase_trigger
    AFTER UPDATE OF count ON inventory
    FOR EACH ROW
        WHEN (new.count > old.count)
        CALL logInventoryIncrease(:new.itemID, :old.count, :new.count);
```

The static condition, such as the name of the table and the modified column, as well as the dynamic condition, such as the difference in the column value, are analogous to AOP 's pointcut concept. Both describe a selection criterion to "trigger" certain actions. The stored procedure specified in the trigger is analogous to AOP 's advice.

Database triggers and AOP 's advice both modify the normal program execution to carry additional or alternative actions. But there are some obvious differences. Database triggers are useful only for database operations. AOP has a more general approach that can be used for many other purposes. But note that AOP doesn't necessarily obviate the need for database triggers, for reasons such as performance and bringing uniformity to multiple applications accessing the same tables.

Similar to database triggers, event-oriented programming includes the notion of responding to events.

Event-oriented programming

Event-oriented programming is essentially the observer design pattern. Each interested code site notifies the observers by firing events, and the observers respond by taking appropriate action, which may be crosscutting in nature.

In AOP, the program is woven with logic to fire virtual events and to respond to the events with an action that corresponds to the crosscutting concern it's implementing.

But note this important difference: Unlike in event-based programming, there is no explicit code for the creation and firing of events in the subject classes. Executing part of the program constitutes the virtual-event generation. Also, event systems tend to be more coarse-grained than an AOP solution implements.

Note that you can effectively combine event-oriented programming with AOP. Essentially, you can modularize the crosscutting concern of firing events into an aspect. With such an implementation, you avoid tangling the core code with the event-firing logic.

Summary

With AOP, crosscutting concerns are modularized by encapsulating them in a new unit called an aspect. Core concerns no longer embed the crosscutting concern's logic, and all the associated complexity of the crosscutting concerns is isolated into the aspects. AOP marks the beginning of a new way of dealing with a software system by viewing it as a composition of mutually independent concerns. By building on top of existing programming methodologies, AOP preserves the investment in knowledge gained over the last few decades.