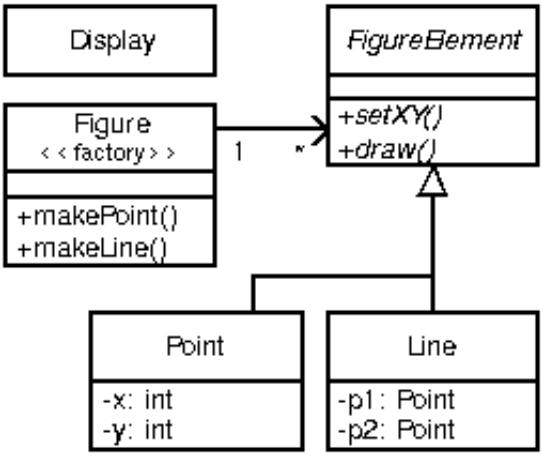
Introduction to AspectJ

Prev **Chapter 1. Getting Started with AspectJ** Next

Introduction to AspectJ

This section presents a brief introduction to the features of AspectJ used later in this chapter. These features are at the core of the language, but this is by no means a complete overview of AspectJ.

The features are presented using a simple figure editor system. A `Figure` consists of a number of `FigureElements`, which can be either `Point`s or `Line`s. The `Figure` class provides factory services. There is also a `Display`. Most example programs later in this chapter are based on this system as well.



UML for the `FigureEditor` example

The motivation for AspectJ (and likewise for aspect-oriented programming) is the realization that there are issues or concerns that are not well captured by traditional programming methodologies. Consider the problem of enforcing a security policy in some application. By its nature, security cuts across many of the natural units of modularity of the application. Moreover, the security policy must be uniformly applied to any additions as the application evolves. And the security policy that is being applied might itself evolve. Capturing concerns like a security policy in a disciplined way is difficult and error-prone in a traditional programming language.

Concerns like security cut across the natural units of modularity. For object-oriented programming languages, the natural unit of modularity is the class. But in object-oriented programming languages, crosscutting concerns are not easily turned into classes precisely because they cut across classes, and so these aren't reusable, they can't be refined or inherited, they are spread through out the program in an undisciplined way, in short, they are difficult to work with.

Aspect-oriented programming is a way of modularizing crosscutting concerns much like object-oriented programming is a way of modularizing common concerns. AspectJ is an implementation of aspect-oriented programming for Java.

AspectJ adds to Java just one new concept, a join point -- and that's really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type

declarations statically affects a program's class hierarchy, and aspects encapsulate these new constructs.

A *join point* is a well-defined point in the program flow. A *pointcut* picks out certain join points and values at those points. A piece of *advice* is code that is executed when a join point is reached. These are the dynamic parts of AspectJ.

AspectJ also has different kinds of *inter-type declarations* that allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.

AspectJ's *aspect* are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations.

In the sections immediately following, we are first going to look at join points and how they compose into pointcuts. Then we will look at advice, the code which is run when a pointcut is reached. We will see how to combine pointcuts and advice into aspects, AspectJ's reusable, inheritable unit of modularity. Lastly, we will look at how to use inter-type declarations to deal with crosscutting concerns of a program's class structure.

The Dynamic Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the dynamic structure of crosscutting concerns. This chapter describes AspectJ's dynamic join points, in which join points are certain well-defined points in the execution of the program.

AspectJ provides for many kinds of join points, but this chapter discusses only one of them: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including return (either normally or by throwing an exception).

Each method call at runtime is a different join point, even if it comes from the same call expression in the program. Many other join points may run while a method call join point is executing -- all the join points that happen while executing the method body, and in those methods called from the body. We say that these join points execute in the *dynamic context* of the original call join point.

Pointcuts

In AspectJ, *pointcuts* pick out certain join points in the program flow. For example, the pointcut

```
call(void Point.setX(int))
```

picks out each join point that is a call to a method that has the signature `void Point.setX(int)` — that is, `Point`'s void `setX` method with a single `int` parameter.

A pointcut can be built out of other pointcuts with and, or, and not (spelled `&&`, `||`, and `!`). For example:

```
call(void Point.setX(int)) ||
call(void Point.setY(int))
```

picks out each join point that is either a call to `setX` or a call to `setY`.

Pointcuts can identify join points from many different types — in other words, they can crosscut types. For example,

```
call(void FigureElement.setXY(int,int)) ||
call(void Point.setX(int))              ||
call(void Point.setY(int))              ||
call(void Line.setP1(Point))            ||
call(void Line.setP2(Point));
```

picks out each join point that is a call to one of five methods (the first of which is an interface method, by the way).

In our example system, this pointcut captures all the join points when a `FigureElement` moves. While this is a useful way to specify this crosscutting concern, it is a bit of a mouthful. So AspectJ allows programmers to define their own named pointcuts with the `pointcut` form. So the following declares a new, named pointcut:

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int))              ||
    call(void Line.setP1(Point))            ||
    call(void Line.setP2(Point));
```

and whenever this definition is visible, the programmer can simply use `move()` to capture this complicated pointcut.

The previous pointcuts are all based on explicit enumeration of a set of method signatures. We sometimes call this *name-based* crosscutting. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property-based* crosscutting. The simplest of these involve using wildcards in certain fields of the method signature. For example, the pointcut

```
call(void Figure.make*(..))
```

picks out each join point that's a call to a void method defined on `Figure` whose the name begins with "`make`" regardless of the method's parameters. In our system, this picks out calls to the factory methods `makePoint` and `makeLine`. The pointcut

```
call(public * Figure.* (..))
```

picks out each call to `Figure`'s public methods.

But wildcards aren't the only properties AspectJ supports. Another pointcut, `cflow`, identifies join points based on whether they occur in the dynamic context of other join points. So

```
cflow(move())
```

picks out each join point that occurs in the dynamic context of the join points picked out by

`move()`, our named pointcut defined above. So this picks out each join points that occurrs between when a move method is called and when it returns (either normally or by throwing an exception).

Advice

So pointcuts pick out join points. But they don't *do* anything apart from picking out join points. To actually implement crosscutting behavior, we use advice. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points).

AspectJ has several different kinds of advice. *Before advice* runs as a join point is reached, before the program proceeds with the join point. For example, before advice on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated.

```
before(): move() {
    System.out.println("about to move");
}
```

After advice on a particular join point runs after the program proceeds with that join point. For example, after advice on a method call join point runs after the method body has run, just before before control is returned to the caller. Because Java programs can leave a join point 'normally' or by throwing an exception, there are three kinds of after advice: `after returning`, `after throwing`, and plain `after` (which runs after returning *or* throwing, like Java's `finally`).

```
after() returning: move() {
    System.out.println("just successfully moved");
}
```

Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point. Around advice is not discussed in this section.

Exposing Context in Pointcuts

Pointcuts not only pick out join points, they can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations.

An advice declaration has a parameter list (like a method) that gives names to all the pieces of context that it uses. For example, the after advice

```
after(FigureElement fe, int x, int y) returning:
        ...SomePointcut... {
    ...SomeBody...
}
```

uses three pieces of exposed context, a `FigureElement` named fe, and two `int`s named x and y.

The body of the advice uses the names just like method parameters, so

```
after(FigureElement fe, int x, int y) returning:
        ...SomePointcut... {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

The advice's pointcut publishes the values for the advice's arguments. The three primitive pointcuts `this`, `target` and `args` are used to publish these values. So now we can write the complete piece of advice:

```
after(FigureElement fe, int x, int y) returning:
        call(void FigureElement.setXY(int, int))
        && target(fe)
        && args(x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ")");
}
```

The pointcut exposes three values from calls to `setXY`: the target `FigureElement` -- which it publishes as `fe`, so it becomes the first argument to the after advice -- and the two int arguments -- which it publishes as `x` and `y`, so they become the second and third argument to the after advice.

So the advice prints the figure element that was moved and its new `x` and `y` coordinates after each `setXY` method call.

A named pointcut may have parameters like a piece of advice. When the named pointcut is used (by advice, or in another named pointcut), it publishes its context by name just like the `this`, `target` and `args` pointcut. So another way to write the above advice is

```
pointcut setXY(FigureElement fe, int x, int y):
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);

after(FigureElement fe, int x, int y) returning: setXY(fe, x, y) {
    System.out.println(fe + " moved to (" + x + ", " + y + ").");
}
```

Inter-type declarations

Inter-type declarations in AspectJ are declarations that cut across classes and their hierarchies. They may declare members that cut across multiple classes, or change the inheritance relationship between classes. Unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time.

Consider the problem of expressing a capability shared by some existing classes that are already part of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to *each affected class* a method that implements this interface.

AspectJ can express the concern in one place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and

associates the methods and fields to the existing classes.

Suppose we want to have `Screen` objects observe changes to `Point` objects, where `Point` is an existing class. We can implement this by writing an aspect declaring that the class Point `Point` has an instance field, `observers`, that keeps track of the `Screen` objects that are observing `Point`s.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();
    ...
}
```

The `observers` field is private, so only `PointObserving` can see it. So observers are added or removed with the static methods `addObserver` and `removeObserver` on the aspect.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }
    ...
}
```

Along with this, we can define a pointcut `changes` that defines what we want to observe, and the after advice defines what we want to do when we observe a change.

```
aspect PointObserving {
    private Vector Point.observers = new Vector();

    public static void addObserver(Point p, Screen s) {
        p.observers.add(s);
    }
    public static void removeObserver(Point p, Screen s) {
        p.observers.remove(s);
    }

    pointcut changes(Point p): target(p) && call(void Point.set*(int));

    after(Point p): changes(p) {
        Iterator iter = p.observers.iterator();
        while ( iter.hasNext() ) {
            updateObserver(p, (Screen)iter.next());
        }
    }

    static void updateObserver(Point p, Screen s) {
        s.display(p);
    }
```

```
}
```

Note that neither `Screen`'s nor `Point`'s code has to be modified, and that all the changes needed to support this new capability are local to this aspect.

Aspects

Aspects wrap up pointcuts, advice, and inter-type declarations in a a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers in addition to the crosscutting members. Because only aspects may include these crosscutting members, the declaration of these effects is localized.

Like classes, aspects may be instantiated, but AspectJ controls how that instantiation happens -- so you can't use Java's `new` form to build new aspect instances. By default, each aspect is a singleton, so one aspect instance is created. This means that advice may use non-static fields of the aspect, if it needs to keep state around:

```
aspect Logging {
    OutputStream logStream = System.err;

    before(): move() {
        logStream.println("about to move");
    }
}
```

Aspects may also have more complicated rules for instantiation, but these will be described in a later chapter.

Prev | Home | Next
Chapter 1. Getting Started with AspectJ | Up | Development Aspects