An Aspect-Aware Outline Viewer

Michihiro Horie Shigeru Chiba
Tokyo Institute of Technology

1 Introduction

An aspect-oriented programming (AOP) is for modularising a crosscutting concern so that it can be easily attached and detached to/from software. Because of this functionality, AOP is one of key technologies for enabling evolvable software. However, critics have been mentioning that AOP makes modular reasoning difficult since join points where an aspect and an object are connected to each other tend to spread over a whole program. Developers often have a problem finding join points specified by pointcut definitions in an aspect. To help developers, a tool such as AJDT has been developed.

For better modular reasoning in AOP, this paper presents a new interpretation of AOP, in which an aspect is an extension to an existing module but the extension may be effective only when the module is accessed from specific accessor modules. This interpretation should let developers consider an aspect is just an extension in the same sense that a subclass extends a super class and override some methods. Thus developers would be able to think that each module has an external interface and the internal implementation of the module is never directly accessed by other modules including an aspect.

To support AOP according to this interpretation, we have developed an Eclipse plugin. It is a programming tool for AspectJ and it shows an outline view of a class woven with an aspect. It presents how each method is extended by showing javadoc comments taken from the definitions of the class and the aspect. This tool gives developers a totally different illustration of AOP programs from AJDT, which is a standard programming tool for AspectJ. AJDT mainly shows the locations of join points (or join point shadows) selected by pointcuts. In other words, it only illustrates where an aspect and an object is connected to each other.

2 An event-based interpretation

A famous paper by Filman and Friedman [2] explained that AOP is quantification and obliviousness. According to their interpretation, program execution is modeled as a sequence of events, such as method calls and field accesses. An advice is an reaction to an event, *i.e.* a join point, selected by a pointcut. Thus,

```
class Line {
  Point p1, p2;
  void move(int x, int y) {
        :
    p1.setX(newX);
        :
  }
     :
}
```

Figure 1: The move method in Line calls the setX method in Point.

to understand an AOP program, developers must know which events (*i.e.* join points) are selected for connecting an object and an aspect.

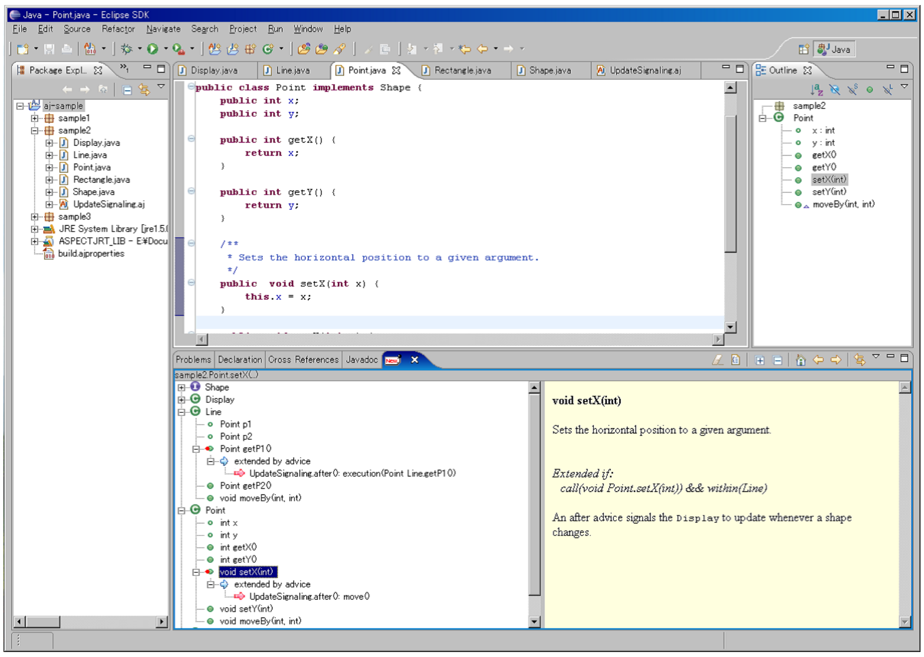
This event-based interpretation makes modular reasoning difficult since most of selected join points are part of the internal implementation of a module. For example, if a pointcut selects a join point representing a call to a setX method within a move method in a Line class (Figure 1), then that method call is part of the implementation of the move method and it should not be exposed to the outside of the Line class. Note that, here, the move method is not a *callee* method but a *caller* method. However, to understand the behavior of an aspect, developers must know the body of the move method contains the call to the setX method and it causes the execution of an advice body. The readers would think that the encapsulation principle is broken.

3 An extension-based interpretation

Although the encapsulation principle might seem broken in AOP, it is not really broken. To illustrate this fact, we present a different interpretation of AOP.

According to our interpretation, an aspect is an extension to a class although it might be effective only under some conditions. This is obviously acceptable if an aspect includes an advice associated with an execution pointcut, which selects the execution of a method body as a join point. Since the advice is executed together with that method body, the aspect can be regarded as an extension to the method body. Note that the extension does not break the encapsulation of the extended method body as an extension by inheritance does not. The extended method body is reused *as is* or the whole body is overridden.

An interesting case is an advice associated with a call pointcut, which selects the execution of a method-call expression at a caller side. Suppose that a move method in a Line class calls a setX method in a Point class and a call pointcut selects a call to setX (Figure 1). We explain that the advice associated with that call pointcut extends the behavior of the setX method in the Point class. An advice always extends the behavior of a *callee-side* method even if a pointcut is call. It does not extend a caller-side method, for example, the move method



```
public class Point implements Shape {
    public int x;
    public int y;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    /**
     * Sets the horizontal position to a given argument.
     */
    public  void setX(int x) {
        this.x = x;
    }
```

```
sample2.Point.setX(..)
Shape
Display
Line
    Point p1
    Point p2
    Point getP1()
        extended by advice
            UpdateSignaling.after(): execution(Point Line.getP1())
    Point getP2()
    void moveBy(int, int)
Point
    int x
    int y
    int getX()
    int getY()
    void setX(int)
        extended by advice
            UpdateSignaling.after(): move()
    void setY(int)
    void moveBy(int, int)
```

void setX(int)

Sets the horizontal position to a given argument.

Extended if:
call(void Point.setX(int)) && within(Line)

An after advice signals the `Display` to update whenever a shape changes.

Figure 2: Our outline viewer for a class extended by aspects (lower panel surrounded by a blue rectangle). It shows javadoc comments on both a method and an advice.

in Line. Note that, under the event-based interpretation, that advice is often regarded as an extension to the caller-side method move.

A call pointcut can be combined with other pointcuts such as within and cflow. In this case, we explain that the behavior of a callee is extended by an aspect only when a caller satisfies the conditions specified by those other pointcuts such as within and cflow. For example, if a pointcut is the following:

```
call(void Point.setX(int)) && within(Line)
```

Then the advice associated with this pointcut extends the behavior of the setX method only when setX is called from a method declared in the Line class. This conditional extension cannot be implemented by subclassing; it needs AOP.

We similarly deal with get and set pointcuts as well. They extend the behavior of the fields that the pointcuts specify. For example, if a pointcut is get(int Point.xpos), then we consider that the advice associated with that get pointcut extends the behavior of the read access to the xpos field in Point. Without the extension, a read access to xpos simply returns the value of xpos. On the other hand, with the extension, a read access to xpos involves not only returning the value of xpos but also executing the associated advice.

void setX(int)
Sets the horizontal position to a given argument.

Extended if:
call(void Point.setX(int)) && within(Line)
An after advice signals the `Display` to update whenever a shape changes.

Figure 3: The Javadoc comments on the setX method

4 Tool support

Our extension-based interpretation encourages developers to treat modules only through external interfaces even if aspects are woven with a program. The effects by aspects can be described as part of external interfaces. To support this idea, we have developed a AspectJ programming tool on top of the Eclipse IDE (Integrated Development Environment). It is an outline viewer of a class (Figure 2); it lists all the methods and fields declared in a specified class. If some of those methods and fields are extended by aspects, then our outline viewer also shows that fact. Furthermore, the outline viewer shows javadoc comments taken from both a class and an aspect. If developers select a method or a field extended by an aspect, then the outline viewer shows the javadoc comments on a pointcut and an advice as well as that method or field. For example, in Figure 2, the setX method in the Point class is selected. Thus, the outline viewer shows comments (a larger image is presented in Figure 3) in the right pane.

5 Concluding remarks

This paper presents the extension-based interpretation of AOP, in which an aspect is an extension to a *callee* class. Each advice in an aspect extends the behavior of a target method or a target field; it never extends a method at a caller (or accessor) side. If a pointcut includes a pointcut designator such as within and cflow, the extension is effective only when the execution context satisfies such a pointcut designator.

Our outline viewer presented in this paper helps programming with this interpretation. It is different from existing AspectJ tools such as AJDT, which supports the event-based interpretation. The outline view shown by our tool is similar to the aspect-aware interface [3]. Although our work shares basic ideas with the aspect-aware interface, we have further pursued appropriate concrete representation of modules in the presence of crosscutting concerns. For example, the article about the aspect-aware interface [3] does not mention how call, get, and set pointcuts should be reflected on a module interface. It does also not mention javadoc comments. Our outline viewer considers that an extension by an aspect is conditional if a pointcut includes within *etc.* This conditional extension is similar to the idea of Classbox/J [1] although Classbox/J is not an AOP language.

References

[1] Bergel, A., S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 2005.

[2] Filman, R. E. and D. P. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," in *Aspect-Oriented Software Development* (R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, eds.), pp. 21–35, Addison-Wesley, 2005.

[3] Kiczales, G. and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," in *Proc. of the Int'l Conf. on Software Engineering (ICSE'05)*, pp. 49–58, ACM Press, 2005.