

A study of modular reasoning in AspectJ  
AspectScope によるアスペクト指向プログラミングの  
支援

by

堀江 倫大

Michihiro Horie

06M37243

January 2008

A Master's Thesis Submitted to  
Department of Mathematical and Computing Sciences  
Graduate School of Information Science and Engineering  
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements  
for the Degree of Master of Science.

Supervisor: Shigeru Chiba

Copyright © 2008 by Michihiro Horie. All Rights Reserved.

## Abstract

This thesis proposes *AspectScope*, which is our programming tool for AspectJ. It automatically performs a whole-program analysis and visualizes the result, and shows how aspects affect module interfaces in the program. When an aspect extends a method behavior, it will also extend the specification which includes its behavior and its signature. In addition, the other methods in the call graph of the advised method are also extended these specifications. It is ideal to write a program only looking at a method specification. One of the typical examples is programming with API (Application Program Interface) in OOP. Developers need not investigate a method implementation because the specification of the method is unchanged and therefore reliable. In AOP, however, the existing specification after deploying aspects are no longer reliable.

Based on this notion, AspectScope can have developers look at the extended method specification on its view to let them understand which method is extended by an aspect. A developer who writes an aspect should consider the influence not only to the target method but also to the other methods that are indirectly affected by it. Therefore, AspectScope lets them append several javadoc comments for the affected methods on each advice. Besides, developers can decide which comment should be seen from the specified method. This feature to control the view is named *comment aspect*. Because of its control, developers can do modular reasoning in a method. To reinforce the function of comment aspect, AspectScope also provides the original advice that name is `comment`. This advice can weave a javadoc comment to a method in case the method has no comment. The woven result can be seen through the AspectScope editor.

## Acknowledgments

I would like to express my profound gratitude to my supervisor, Shigeru Chiba. He gave me closely teaching with his assiduous guidance and support. Our numerous discussions and his constructive comments have greatly improved my work. I also thank Kenichi Kourai, who has been a research associate of Tokyo Institute of Technology. He gave me various important ideas and advice. Yoshisato Yanagisawa has supported me since I was a bachelor student. He also taught me how to write a paper and to make a presentation. Muga Nishizawa gave me various important advice for my study. Finally, I greatly thank my colleagues of the Chiba Shigeru Group in Tokyo Institute of Technology; Yuji Takizawa, Yohsuke Kurita, Ryunosuke Imayoshi, Hidekazu Tadokoro, Shunpei Akai, Takashi Azumi, Satoshi Morita.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivating problem . . . . .	1
1.2	Solution by this thesis . . . . .	2
1.3	The structure of this thesis . . . . .	3
<b>2</b>	<b>Proposals for modular reasoning</b>	<b>5</b>
2.1	Aspect-Oriented Programming . . . . .	5
2.1.1	Obliviousness . . . . .	7
2.2	Aspect-Aware Interface . . . . .	8
2.3	Open Modules for AspectJ . . . . .	9
2.4	XPI (crosscutting program interface) . . . . .	13
2.5	AJDT . . . . .	14
2.6	Active model . . . . .	17
2.7	<i>Assistants</i> and <i>Spectators</i> . . . . .	18
2.8	Pointcut Interfaces . . . . .	24
2.9	Join Point Encapsulation . . . . .	24
2.10	Summary . . . . .	25
<b>3</b>	<b>AspectScope</b>	<b>27</b>
3.1	AspectScope editor . . . . .	27
3.2	comment aspect . . . . .	29
3.3	comment advice . . . . .	34
3.4	Outline Viewer . . . . .	36
3.4.1	The execution and call pointcuts . . . . .	36
3.4.2	The within and cflow pointcuts . . . . .	37
3.4.3	Other pointcuts and Inter-type declarations . . . . .	38
3.5	Summary . . . . .	39

<b>4</b>	<b>Implementation Issues</b>	<b>40</b>
4.1	Extended AJDT and JDT Parser . . . . .	40
4.1.1	Org.eclipse.ajdt.core . . . . .	40
4.1.2	Org.eclipse.ajdt.ui . . . . .	41
4.1.3	Org.aspectj.ajde . . . . .	43
<b>5</b>	<b>Evaluation</b>	<b>47</b>
5.1	A case study with an web-based information system . . . . .	47
5.1.1	Design Patterns . . . . .	47
5.1.2	Logging . . . . .	49
5.1.3	Exception Handling . . . . .	50
5.2	Experiment . . . . .	51
5.3	Summary . . . . .	53
<b>6</b>	<b>Future work</b>	<b>54</b>
6.1	Modularization of comments . . . . .	54
6.2	comment advice control . . . . .	56
<b>7</b>	<b>Concluding Remarks</b>	<b>57</b>

# List of Figures

2.1	A simple drawing editor written by AspectJ . . . . .	10
2.2	Interfaces in the AOP code . . . . .	11
2.3	The representation of <code>setX</code> method in the AJDT editor . . . .	15
2.4	Arrows in <code>moveBy</code> method, in case of <code>execution</code> pointcut . . .	15
2.5	Arrows in <code>moveBy</code> method . . . . .	16
2.6	A diagram focusing on <code>Billing</code> aspect . . . . .	18
2.7	<code>telecom</code> element after applying the abstraction operation . . .	19
3.1	The display by <code>AspectScope</code> . . . . .	28
3.2	The call graph on <code>setX</code> method . . . . .	29
3.3	The description of javadoc comments with <code>@comment</code> annotation . . . . .	30
3.4	The extended comment by <code>DisplayUpdate</code> . . . . .	34
3.5	The outline view presents the effect of the <code>execution</code> pointcut. .	36
3.6	The outline view presents the effect of the <code>call</code> pointcut. . . .	36
3.7	A conditional extension by the <code>within</code> pointcut (the red underline was drawn by the authors) . . . . .	38
3.8	There is a <code>before</code> advice associated with the <code>get</code> pointcut. . .	38
3.9	An intertype declaration of the <code>distance</code> method . . . . .	39
3.10	Two advices extend the <code>setX</code> method. . . . .	39
4.1	Overview of <code>AspectScope</code> . . . . .	42
5.1	Calculations of cross-cutting structures . . . . .	52
5.2	Comprising data of calcuations in <code>AspectScope</code> . . . . .	53

# List of Tables

2.1	Part of pointcuts in AspectJ . . . . .	6
2.2	The order of precedence in the expansion operation . . . . .	17
2.3	Example aspects and their categories . . . . .	22
2.4	Example aspects from Kiselev's book . . . . .	23
4.1	The necessary files for build a jdt parser . . . . .	46
5.1	The numbers of extended comments that developers will have to add for upper methods in the call graph . . . . .	49
5.2	The numbers of extended comments that developers will have to add for lower methods in the call graph . . . . .	49
5.3	The number of extended comments that developer have to add for the logging aspect . . . . .	50
5.4	The numbers of extended comments that exception handling aspects extend for upper methods in the call graph . . . . .	51
5.5	The number of extended comments that exception handling aspect extend for lower methods in the call graph . . . . .	51

# Chapter 1

## Introduction

### 1.1 Motivating problem

The pointcut and advice mechanism of Aspect-Oriented Programming (AOP) languages such as AspectJ [14] helps developers decompose software into modules and compose them into software. This mechanism allows us to (de)compose software of several modules, including crosscutting ones, without editing the client source code that the modules cut across. The client code does not have to explicitly invoke the code of the crosscutting modules.

This *obliviousness* [6] property of AOP languages is significant but it has caused much debate. Some say that obliviousness is an essential property [6] but others say that it is desirable but not mandatory [26]. Because of the obliviousness property, when one module is executed in an AOP language, other modules might be implicitly invoked from that module. This means that developers cannot understand the behavior of a module as long as they are looking at only the source code of that module. The behavior might be changed by the deployment of other modules (*i.e.* aspects). Therefore, AOP languages require a whole-program analysis for understanding a program. This fact has let AOP detractors claim that the obliviousness of AOP makes modular reasoning difficult although AOP was invented for better modularity [28].

To address this issue, several programming tools for AOP have been developed. One of the most popular tools is AJDT, AspectJ Development Tools of Eclipse IDE [29]. It automatically performs a whole-program analysis and visualizes the crosscutting structures in the program according to the result of the analysis. The developers do not have to manually perform a whole-program analysis any more. However, AJDT does not seem to sat-

isfy AOP detractors. One (but ignorable) reason is that some developers still prefer simple text editors such as *vi* and they do not want to write programs with rich programming environments such as Eclipse. Another reason worthy to consider is that developers want to see *static* module interfaces for understanding their programs. Here, the module interfaces include the specifications of the behavior of the modules. Although AJDT automatically performs a whole-program analysis while a developer is editing a program, the visualization by AJDT does not much help the developer see the module interfaces. It does nothing except simply showing the join points where modules are connected to aspects. Even worse, module interfaces in AOP languages are never static or stable. It changes according to the deployment of aspects. In this sense, the module interfaces in AOP are essentially different from traditional ones.

Although making module interfaces in AOP be static might be nonsense (because being not static might be essential), it should be possible to improve the visualization by a programming tool so that developers can easily see the module interfaces under the current deployment of aspects. This would hopefully give better impression of AOP to the developers who do not think AOP really helps modular programming due to its obliviousness property.

## 1.2 Solution by this thesis

This thesis proposes *AspectScope*, which is our programming tool for AspectJ. We have developed it for realizing our idea. Like AJDT, it automatically performs a whole-program analysis and visualizes the result. However, it shows how aspects affect module interfaces in the program. It interprets an aspect as an extension to other classes and it displays the extended module interfaces of the classes under the deployment of the aspects. It thereby helps developers understand crosscutting structures in the program.

When an aspect extends a method behavior, it will also extend the specification which includes its behavior and its signature. In addition, the other methods in the call graph of the advised method are also extended these specifications. It is ideal to write a program only looking at a method specification. One of the typical examples is programming with API (Application Program Interface) in OOP. Developers need not investigate a method implementation because the specification of the method is unchanged and therefore reliable. In AOP, however, the existing specification after deploying aspects are no longer reliable.

Based on this notion, *AspectScope* can have developers look at the

extended method specification on its view to let them understand which method is extended by an aspect. A developer who writes an aspect should consider the influence not only to the target method but also to the other methods that are indirectly affected by it. Therefore, AspectScope lets them append several javadoc comments for the affected methods on each advice. Besides, developers can decide which comment should be seen from the specified method. This feature to control the view is named *comment aspect*. Because of its control, developers can do modular reasoning in a method. To reinforce the function of comment aspect, AspectScope also provides the original advice that name is `comment`. This advice can weave a javadoc comment to a method in case the method has no comment. The woven result can be seen through the AspectScope editor.

### 1.3 The structure of this thesis

From the next chapter, we presented background, our tool AspectScope, and implementation issues of AspectScope. The structure of the rest of this thesis is as follows.

#### **Chapter 2: Proposals for modular reasoning**

This chapter first explains the feature of aspect-oriented programming and the nature of it, obliviousness. Then we discuss the several existing proposals to overcome obliviousness for modular reasoning.

#### **Chapter 3: AspectScope**

To address the problems described in the previous chapter, we propose our tool AspectScope. To resolve the obliviousness problem in AspectJ, the functional capabilities of this tool are presented.

#### **Chapter 4: Implementation Issues**

To develop AspectScope, we based on the implementation of AJDT. We extended AJDT model and its view. Besides, We modified jdt parser to accomplish `comment` advice.

#### **Chapter 5: Evaluation**

To evaluate the usefulness of AspectScope, we measured it using a web-based information system, which contains the cross-cutting concerns such

as transaction, authentication, exception handling, and so on. Besides, we did performance test by comparison with AJDT.

**Chapter 6: Future Works**

This chapter discuss the future work of AspectScope. The functional feature of it should be more sophisticated. We address the current problems and describes the solutions against them.

**Chapter 7: Concluding Remarks**

Finally, we conclude this thesis in this chapter. We present contributions.

## Chapter 2

# Proposals for modular reasoning

In this chapter, we start with the base of Aspect-Oriented Programming, which is a mechanism to separate a cross-cutting structure as another module named *aspect*. The typical cross-cutting concerns are logging, synchronization, error handling, and so on. With a simple example, we explain the notion of *join point*, *pointcut*, and *advice*. We also mention *obliviousness* property of AOP, which is the basic problem in this thesis. Obliviousness prevents developers from doing modular reasoning in a method. The rest of this chapter presents the various related proposals against obliviousness, and describes these own problems.

### 2.1 Aspect-Oriented Programming

The pointcut and advice mechanism of Aspect-oriented Programming (AOP) languages such as AspectJ [14] allows developers to combine a module to a special module, called an *aspect*, without explicit method calls. This is useful to implement certain crosscutting concerns as a separate module. An aspect can define *pointcuts* and *advices*. An advice is implicitly invoked when a thread of control reaches some execution points in the other module. Those execution points are selected from the predefined set of points by the language. These points are called *join points*. Examples of join points are method call, method execution, field get, and field set. A pointcut is a set of join points. A Pointcut can also consist of the execution context of these join points. Several such examples are *this*, *target*, and *args*. These pointcuts supported in AspectJ is shown in Table 2.1.

pointcuts	join points
<code>call(<i>method pattern</i>)</code>	<i>method pattern</i> is called
<code>execution(<i>method pattern</i>)</code>	an individual method is invoked
<code>get(<i>field pattern</i>)</code>	a field of an object, class or interface is read
<code>set(<i>field pattern</i>)</code>	a field of an object or class is set
<code>within(<i>class pattern</i>)</code>	any joinpoints where the associated code is defined in the <i>class pattern</i>
<code>this(<i>object</i>)</code>	any join point where the currently executing object is an instance of <i>object</i>
<code>target(<i>object</i>)</code>	any join point where the target object is an instance of <i>object</i>
<code>cflow(<i>pcd</i>)</code>	any join point that is within the the dynamic extent of the join points matched by <i>pcd</i>

Table 2.1: Part of pointcuts in AspectJ

Pointcuts can be expressed logically with wild characters, `"*"`, `"+"`, and `".."` (The details are given in Section 3.2 of Chapter 3). As an example, following pointcut:

```
call(Point.new(*,*)) && !within(Figure+)
```

intercepts a call to a constructor of `Point` class only if the caller class is not a subclass of `Figure` class. An advice is a language construct similar to a function. An advice is invoked whenever a join point specified by a pointcut occurs in a base program. In AspectJ, the advice construct are of three types, `before`, `after`, and `around`. `Before` advice runs just before a specified join point arises, `after` advice runs just after a specified join point arises, and `around` advice gives control over the actual execution of a specified join point. An example of aspects is following `TimeLogger` aspect. A call pointcut with `target` pointcut takes a dynamic context of the caller join point. In the `before` and `after` advice bodies, `startTime` and `stopTime` field are referred. Although these field variables have private visibility in `Timer` class, aspect can refer them. When an aspect is declared with `privileged`, the aspect has access to all members beyond the principles of encapsulation. An aspect can have methods as `print` method in `TimeLogger` aspect.

```
class Timer {
    private long startTime, stopTime;
```

```

    public long start() {
        startTime = System.currentTimeMillis();
        return startTime;
    }
    public long stop() {
        stopTime = System.currentTimeMillis();
        return stopTime;
    }
}

public privileged aspect TimeLogger {
    before(Timer t): call(* Timer.start()) && target(t) {
        this.print("Started : " + t.startTime);
    }
    after(Timer t): call(* Timer.stop()) && target(t) {
        this.print("Stopped: " + t.stopTime);
        t.reset();
    }
    public void Timer.reset() {
        this.start = 0;
        this.stop = 0;
    }
    private void print(Object obj) {
        System.out.println(obj);
    }
}

```

An aspect can introduce new members in a class with inter-type declarations. The `reset` inter-type method in `TimeLogger` is an example. This method is not a normal method in Java, and the syntax is `"Timer.reset"`. Note that this in the inter-type method refers the target object of `Timer` class not `TimeLogger` aspect. The introduced `reset` method is invoked in `after` advice.

### 2.1.1 Obliviousness

Because of the property of obliviousness, when a method is executed, advices are implicitly invoked from aspects. This mechanism of AOP has let AOP detractors claim that the obliviousness property makes modular reasoning difficult. For example, when `start` method is called from a client class, the advice body is implicitly executed to print the a log message and its time.

As long as developers look at the implementation of `start` method, they do not understand the accurate behavior of it.

In AOP, a whole analysis of all aspects is required to figure out the cross-cutting structures of a program. In the above example, developers only have to investigate `TimeLogger` aspect and the advice body. Consider another aspect extends the same join point of a call to `start` method as the following `TimerContract` aspect.

```
aspect TimerContract {
    before(Timer t) : call(* Timer.start()) && target(t) {
        if (t.start != 0)
            throw new ContractBrokenException();
    }
}

aspect PrecedenceOrder {
    declare precedence : TimerContract, TimeLogger;
}
```

`TimerContract` aspect also extends the call to `start` method, and adds a contract that `start` method must satisfy before its execution. Besides, `PrecedenceOrder` aspect defines the precedence order of the aspects so that `TimerContract` aspect will be executed first. *declare precedence* decides the precedence order starting from the left side. If there is no precedence, it is not decidable which aspect is executed first. Therefore, to understand the exact behavior of `start` method, the analysis about three aspects is required.

## 2.2 Aspect-Aware Interface

`AspectScope` dynamically generates module interfaces according to current deployment of aspects. The generated interfaces are not statically determined ones. `AspectScope` shares this basic idea with aspect-aware interfaces [15]. We can say that `AspectScope` is a programming tool that realizes the basic idea of aspect-aware interfaces in `AspectJ`. However, interpreting aspects as both the callers and the callee sides extension is a unique idea of `AspectScope`. In the original article of aspect-aware interfaces, the interpretations of the `call`, `get`, and `set` pointcuts are open questions (in Section 4.2 of [15]). They even suggest interpreting an aspect including those pointcuts as a caller-side extension.

To illustrate the key property of aspect-aware interface, we below show a refactoring process of a figure editor as an example. A figure editor is a simple tool for editing drawings that are composed of points and lines. Figure 2.1 shows the AspectJ program of this figure editor. The concern of restricting the display size of editing pane is implemented in an aspect. Since a pane of the tool has the predetermined display size, the developer will write a constraint condition by defining **Contract** aspect. **Contract** restricts the horizontal size from 0 to 100, and the vertical size from 0 to 50. When figures in the editing pane are moved out of these range, **IllegalArgumentException** will be thrown. In this case, like figure 2.2, an aspect-aware interface should be presented after the whole program analysis of the figure editor. Once the aspect-aware interface has been presented, developers are able to understand through this interface both **setX** and **setY** are extended by **Contract** aspect. Note that **before** advices in **Contract** aspect designate execution pointcuts. Even when call pointcuts are designated instead of execution pointcuts (as described below), **setX** and **setY** behave similarly.

```
before(int x) : call(void Point.setX(int)) && args(x) { ... }
before(int y) : call(void Point.setY(int)) && args(y) { ... }
```

However, the resolution of call pointcut is not mentioned definitely whether the extension is written on the callee method **setX** or not. They mentions that an initial answer might be to list the extension on the caller method **moveBy** in **Line** as well.

AspectScope resolves these open issues. For example, when call pointcut is declared in an aspect, the outline viewer of AspectScope lists the extension on both caller method **moveBy** in **Line** class and the callee **setX**. AspectScope can also show the extension of an advice on the caller methods of **moveBy** method because the behavior of these caller methods are indirectly extended.

## 2.3 Open Modules for AspectJ

Open module for AspectJ is a mechanism to hide class implementations from advice. Since AspectJ is one of the powerfull languages, developers can select any joinpoints beyond information hiding rule. The concept of open modules is proposed by Aldrich, and open module for AspectJ is the one construct based on it. Onglingco *et.al* fully extended the notion of open modules to AspectJ. For example, first, Aldrich specifies only call pointcuts in his language. They also extend the notion of module compositions not

<pre> public interface Figure {     /**      * Moves the figure by dx along the x axis      * and dy along the y axis      */     void moveBy(int dx, int dy); }  class Line implements Figure {     private Point p1, p2;      /** Sets the starting point of this line */     public void setP1(Point np1) {p1 = np1;}      /** Sets the end point of this line */     public void setP2(Point np2) {p2 = np2;}      /** Returns the starting point of this line */     public Point getP1() {return p1;}      /** Returns the end point of this line */     public Point getP2() {return p2;}      /**      * Moves this line by dx along the x axis      * and dy along the y axis      */     public void moveBy(int dx, int dy) {         p1.setX(p1.getX() + dx);         p1.setY(p1.getY() + dy);         p2.setX(p2.getX() + dx);         p2.setY(p2.getY() + dy);     } } </pre>	<pre> class Point implements Figure {     private int x, y;      /** Sets the horizontal position      * to a given argument      */     public void setX(int nx) {x = nx;}      /** Sets the vertical position      * to a given argument      */     public void setY(int ny) {y = ny;}      /** Returns the horizontal position */     public int getX() {return x;}      /** Returns the vertical position */     public int getY() {return y;}      /**      * Moves this point by dx along the      * x axis and dy along the y axis      */     public void moveBy(int dx, int dy) {         x += dx; y += dy;     } }  aspect Contract {     /**      * the set value &lt;code&gt;x&lt;/code&gt; should be      * no fewer than 0, nor more than 100      * @throws IllegalArgumentException      */     before(int x) : execution(void Point.setX(int))         &amp;&amp; args(x) {         if (x &lt; 0    100 &lt; x)             throw new IllegalArgumentException();     }      /**      * the set value &lt;code&gt;y&lt;/code&gt; should be      * no fewer than 0, nor more than 50      * @throws IllegalArgumentException      */     before(int y) : execution(void Point.setY(int))         &amp;&amp; args(y) {         if (y &lt; 0    50 &lt; y)             throw new IllegalArgumentException();     } } </pre>
--	--

Figure 2.1: A simple drawing editor written by AspectJ

```

Figure
    void moveBy(int, int)

Point implements Figure
    int x;
    int y;
    int getX();
    int getY();
    void setX(int)    : Contract - before Contract.
                        execution(void Point.setX(int));
    void setY(int)    : Contract - before Contract.
                        execution(void Point.setY(int));
    void mvoveBy(int, int);

Line implements Figure
    Point p1;
    Point p2;
    Point getP1();
    Point getP2();
    void setP1(Point);
    void setP2(Point);
    void moveBy(int, int);

Contract
    before : Point.setX(int), Point.setY(int)

```

Figure 2.2: Interfaces in the AOP code

only to restrict the accesses but also to open some aspects by exposing almost all joinpoints.

We below show an example of the drawing editor program (figure 2.1). The usage of open modules is mainly supposed the partitioned two groups of code: one can manipulate the code of aspects, and another can only manipulate the class codes. Unless a open module exposes particular joinpoints, aspect is not able to select any of them. Therefore, before an aspect writer adds the constraint condition to `setX` and `setY`, a class writer must declare an open module as described below.

```
module FigureModule {
    class Point;
    expose execution(void Point.setX(int))
        || execution(void Point.setY(int));
}
```

class in `FigureModule` is the declaration that means some joinpoints in `Point` class will expose to aspects. The details of the exposure is specified at `expose` statement. In the way, `Contract` aspect can be defined to select the execution of setter methods in `Point` class. There are other ways to define `FigureModule`. Since the developers who manipulate class codes understand `moveBy` method in `Line` class only calls the setter methods in `Point` class, `FigureModule` can be described as follows.

```
module FigureModule {
    class Line || Point;
    expose call(void Point.setX(int))
        || call(void Point.setY(int))
}
```

`call` pointcut intercepts the joinpoints in `Line` class that is the caller class. Therefore, the developer must specify `Line` class at the class. Open modules have several other rules to expose joinpoints. `friend` allows the aspect to intercept all joinpoints in the specified classes. This is useful to use logging aspects because logging inserts many places in codes. For example, the following definition allows `LoggingAspect` to hook all joinpoints in `Figure` class and `Point` class.

```
module FigureModule {
    class Figure || Point;
    friend LoggingAspect;
```

```

      :
    }

```

A module can be hierarchically constructed with other modules. Therefore, open modules can work well with larger systems. One of the basic composition of modules are shown below (in Section 3.6 of [25]). M2 gives M1 the further strict visibility of pointcuts.

```

module M1 {
  class C1, C2;
  friend A1, A2;
  expose : C1.pointcut1();
}
module M2 {
  class C3;
  friend A3;
  constrain M1;
  friend A4;
  expose : A4.pointcut2();
}

```

By using the keyword `constrain`, M1 is forced to be incorporated `expose` entity with one of M2. Note that, `friend` clause of M1 does not change. Instead of adding the `constrain` entity in M2, the replacement of the following `expose` entity in M1 behaves equally.

```

(C1.pointcut() && A4.pointcut2());
|| (C1.pointcut1() && thisAspect(A3 || A4));

```

## 2.4 XPI (crosscutting program interface)

XPI consists of an abstract aspect that declares several joinpoints. The Advice code can only designate these joinpoints declared in XPI to extend the class behaviors. Note that, there is no restriction rule in XPI. The developers are supposed to look first at XPI and then write advices which joinpoints are exposed in the XPI. In this way, the class writers are able to know only joinpoints exposed in XPI can be extended, and modular reasoning in methods are kept. William *et.al.* claim that developers need not know about specific aspects, such as logging, but they must decide which abstractions to expose as XPIs to facilitate aspect development and evolution.

An XPI consists of four elements: the name of the XPI, the visibility of pointcuts, the sets of abstract join points, and a partial implementations. For example, as described below, `XPointChange` XPI has two pointcuts whose visibility are public. Each pointcut designates execution and `args` pointcut. Then, `Contract` aspect can be defined. The advices in `Contract` aspect uses the public pointcuts in `XPointChange` XPI. Although the developers cannot know the details of advice behaviors, they can notice these setter methods are extended by some aspects. Therefore, the modular reasoning of these setter methods may be maintained.

```
public aspect XPointChange {
    public pointcut X(int x):
        execution(void Point.setX(int))
        && args(x);

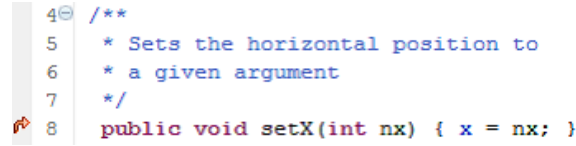
    public pointcut Y(int y):
        execution(void Point.setY(int))
        && args(y);
}
```

## 2.5 AJDT

The standard AspectJ support of Eclipse IDE, named AJDT [29], visualizes a crosscutting structure in an AspectJ program. This helps developers to reason about the program with a modular fashion despite the obliviousness property of AspectJ. However, the help by this visualization is still limited and thus developers sometime feel that AOP makes modular reasoning difficult. For example, AJDT only shows an arrow at which a target joinpoint (joinpoint shadow [20]) occurs in the ruler of the editor, and tells developers which event in the code are caught. Therefore, developers who use AJDT must investigate the implementation of a method and an advice to understand the program behavior. However, this investigation breaks modular reasoning. In the next section, we would detail its feature.

### execution pointcut

An execution pointcut selects joinpoints at which the specified method is executed. As shown in figure 2.3, in the ruler of the editor, AJDT puts a mark at `setX` method in `Point` class. The mark indicates that advices will extend the behavior of `setX` method when `setX` will be executed.

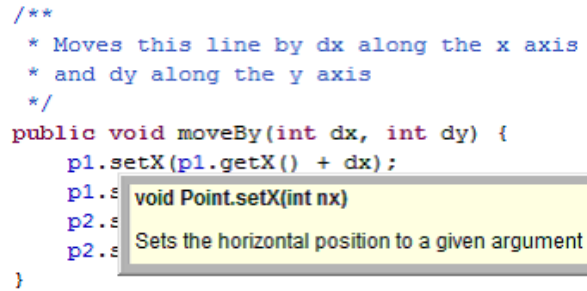


```

4  /**
5   * Sets the horizontal position to
6   * a given argument
7   */
8  public void setX(int nx) { x = nx; }

```

Figure 2.3: The representation of setX method in the AJDT editor



```

/**
 * Moves this line by dx along the x axis
 * and dy along the y axis
 */
public void moveBy(int dx, int dy) {
    p1.setX(p1.getX() + dx);
    p1.s
    p2.s
    p2.s
}

```

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

Figure 2.4: Arrows in moveBy method, in case of execution pointcut

However, this way of visualization does not let developers know how the specification of `setX` method is changed by an aspect. Developers must investigate the source code of `Contract` aspect to understand the detailed specification of `setX` method. In Figure 2.4, AJDT editor does not show any sign of extension in `moveBy` method in `Line` class. Note that `moveBy` is the caller method that calls `setX` method. This is because AJDT only shows joinpoints that advices capture. The pop-up display (the gray box in Figure 2.4) does not also indicate the advice extension of `setX` method. The content of the comment has not been changed after weaving. Developers can only notice the advice extension after looking at the `setX` method implementation.

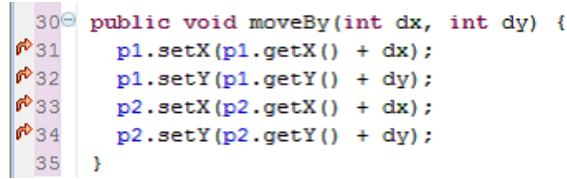
### call pointcut

A call pointcut selects joinpoints at which the specified method is called in client classes. In Figure 2.1, before advices in `Contract` aspect designate execution pointcuts. When these advices uses call pointcut insted of the execution pointcut, the behaviors of `setX` and `setY` method are not changed.

```

aspect Contract {
    /**

```



```

30 public void moveBy(int dx, int dy) {
31     p1.setX(p1.getX() + dx);
32     p1.setY(p1.getY() + dy);
33     p2.setX(p2.getX() + dx);
34     p2.setY(p2.getY() + dy);
35 }

```

Figure 2.5: Arrows in moveBy method

```

* ...
*/
before(int x) :
    call(void Point.setX(int))
    && args(x) {
    if (x < 0 || 100 < x)
        throw new IllegalArgumentException();
    }

/**
* ...
*/
before(int y) :
    call(void Point.setY(int))
    && args(y) {
    if (y < 0 || 50 < y)
        throw new IllegalArgumentException();
    }
}

```

AJDT puts arrows at moveBy method in Line class that is the caller class (Figure 2.5). As well as execution pointcuts, developers must investigate the implementation of Contract aspect to understand how setX method are extended. An arrow only shows that some aspects will capture the joinpoint when moveBy method calls setX method. AJDT displays arrows based on the event of joinpoints, and does not consider the modular programming.

In addition, there is no arrow indicating that setX method behavior is extended at the methods that call the moveBy method. As shown below, MultiLines class is one of the client classes that call moveBy method in Line class. This class represents a unicursal line that consists of random lines.

```
public class MultiLines implements Figure {
```

```

private List lines;
    :
public void moveBy(int dx, int dy) {
    for (Iterator it =
        lines.iterator(); it.hasNext();)
        ((Line) it.next()).
            moveBy(dx, dy);
}
}

```

There is no arrow indicating that the specification of `moveBy` method in `Line` class is extended. Therefore, only looking at `MultiLines` class does not tell developers that `setX` and `setY` methods are extended. They need to look at the `moveBy` method implementation in `Line` class. This breaks the encapsulation rule.

## 2.6 Active model

Active models [5] is another approach to represent a crosscutting structure better than AJDT. `ActiveAspect`, which is their tool based on the active models, presents a node-and-link diagram representing an interesting slice of the crosscutting structure of an AspectJ aspect. Although `ActiveAspect` and our `AspectScope` share the same goal, `ActiveAspect`'s approach is to visualize join points selected by aspects. On the other hand, our `AspectScope` visualizes module interfaces extended by aspects. It uses traditional tree-based representation.

An active model allows developers to focus on a piece of an aspect in the whole systems. Active model provide the three operations, *projection*, *expansion*, and *abstraction*. Projection operation shows joinpoint shadows

Order	Relationship
1	Method call from advice body
2	Reference from inter-type method
3	reference to inter-type method
4	reference from advice body to field in an other type
5	reference to inter-type field

Table 2.2: The order of precedence in the expansion operation

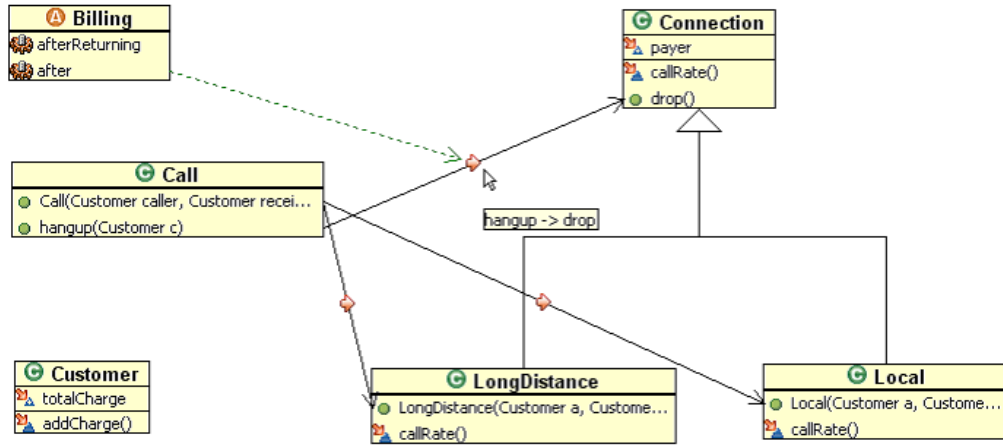


Figure 2.6: A diagram focusing on Billing aspect

in each models, and selects only essential entities in crosscutting structures to and from a particular models such as aspects and classes. For example, figure 2.6 (derived from Section 3.2 of [5]) shows the projection result generated from **Billing** aspect. Expansion operation is prepared for the further investigation of crosscutting structures. To present first the most important elements in all structures, the order of precedence is given as shown in table 2.2. Class diagrams tend to be complicating when many elements are displayed in one place. For better visualization and understanding, active models prepare the abstraction operation. The diagram in Figure 2.7 shows the **around** advice in **Profiling** aspect intercepts a number of modules in **telecom** package. The abstraction operation has been applied by aggregating classifiers, members, and these relationships.

Active models represents necessary information for developers who wants to know the influence of one aspect over other modules. However, advice models focus on showing aspect influence over other modules. Since developers would not find crosscutting structures that are originated from a class, obliviousness property of aspects has been only partly solved.

## 2.7 Assistatnts and Spectators

Clifton *et.al.* address the problem of modular reasoning via annotations that state which aspect may extend the module [4, 3]. They build their system on top of JML [18, 19]. The most distinctive feature of their proposal

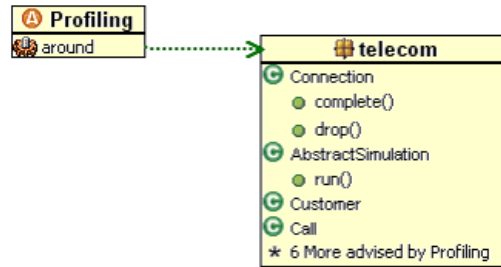


Figure 2.7: telecom element after applying the abstraction operation

is that they distinguish *assistants* and *spectators*. An assistant means an aspect that changes the semantics of the base programs, while a spectator means an aspect that merely observes class behaviors. Spectators are quite close to pure aspects [30]. For example, the following **Tracing** aspect is an spectator that does not change the effective specifications of any other classes. This spectator mutates its own state by appending `StringBuffer` and mutates global state of classes by printing an output. However, this advice does not change the effective pre- and postconditions of `Line`'s `moveBy` method. **Tracing** merely observes the arguments of the `moveBy` method and reports them. The arguments are passed on to the method unchanged and the method's results are unchanged. Since an spectator does not change the effective specification of the method they observe, the code outside an existing program can apply an spectator to any joinpoint in the original program without lack of modular reasoning.

```
aspect Tracing {
  private StringBuffer sbuf = new StringBuffer();
  before(Line l, int dx, int dy) :
    call(moveBy(int, int)) && target(l) && args(dx, dy) {
    String msg = l + " moves (" + dx + ", " + dy + ").";
    sbuf.append(msg);
    System.out.println(msg);
  }
}
```

An assistants can change the effective specification of a module. **Contract** aspect in figure 2.1 is an assistant. `before` advices can change the effective specification by throwing `IllegalArgumentException` in case that the argu-

ments dose not fulfills conditions. When assistants are present, developers cannot do modular reasoning only looking at a module. Curtis *et.al.* propose a class must explicitly name those assistants in itself. They say that a module accepts assistants when it inames the asssitants that are allowed to change its effective specification or the effective specification of modules that it uses. **Accept** clause can be declared in normal AspectJ program.

`accept TypeName`

where *TypeName* must be a fully qualified name of an assistant. For example, **Point** class accepts the **Contract** assistant by declaring: "accept Contract". Note that this acceptance is only allowed in **Point** class. The client classes that call **setX** and **setY** method in **Point** class would not have it applied to their calls. This is because the compiler doesn't know modularly where all accept culauses in a program might appear. We think that to keep modular reasoning in **Line** class as well, **Line** also must declare an acceptance as shown below. Even if exection pointcuts are declared in **Contract** aspect, a caller side **Line** class should also accept **Contract** assistant for better reasoning. Similary, when **Contract** uses **call** pointcuts, **Point** class that is a callee class should accept that assistant.

To reduce the burden of writting **accept** clause in the base classes, **accept** maps are introduced. An **accept** map allows developers to write specifications of acceptance in one place. This module can resolve not only the code tangling between class codes and **accept** clause, but also the scattering of **accept** clauses. An example aspect map is given as follows.

```
package figures;
Point {
    accept Contract;
}
Line {
    accept Contract;
}
* {
    accept FigureUpdate;
}
```

The **Line** pattern in the example says that **Line** class accepts **Contract** assistant. The next module starting with wildcard '\*' indicates that every classes accept **FigureUpdate** assistant. Therefore, **Line** class accepts both **Contract**

and `FigureUpdate` assistants. As with `accept` clauses in modules, the identifier in an `accept` maps is subject to Java's normal namespace rules for packages and imports. Aspect maps are similar to open modules. However aspect maps are far more restricted and they have no rule combining each module both hierarchically and even flat.

The change is to classify aspects in two categories, assistants and spectators. Curtis *et.al.* suggest no solution to categorize aspects automatically. Alternatively, they investigate some categorization data through several examples.

### A list from aspectj programing guide

They separate the example aspects into two categories based on how they would implement them with their restrictions. Table 2.3 (derived in Section 3.2.1 of [3]) lists the examples by category. They also describe the categories finely. In their categorization, most of the example aspects clearly meet their definition of spectator. To satisfy their restrictions these would only require the `spectator` annotation. Aspect in the examples that could be implements as assistants are divided into two kinds. *Client utilities* are used by client codes to change the effective behavior of the target objects. They premise that the changes in effective behavior of a target module do not affect the callee side class. Other example aspects can be categorized as *implementation utilities* in assistants. These assistants would be accepted by the module that it advises, for example, an aspect that uses an execution pointcut. There are some exceptions in their categorized aspects. `Coordinator` aspect is an abstract aspect and only refers to abstract pointcuts that are defined in `GameSynchronization` and `RegistrySynchronization`. In this case, they categorize `Coordinate` aspect as assistant because it does not change effective behavior of classes. `Debug` aspect is categorized as *combined*. This aspect would require a combination of assistants and spectators.

### A list from Kiselev's case study

Curtis *et.al.* also make a category list (derived from Section 3.2.2 of [3]) from the case study of Kiselev's book *Aspect-Oriented Programming with AspectJ* [?]. The examples derived from this book are related to the concerns of a web service. Examples are liberally divided into "development", "production", "runtime" and the others. `CodeSegregation` aspect are categorized *not defined* because this aspect declares `declare error` and `declare warning` in it. These constructs cannot be categorized in their current work. They mention

Examples	Category
telecom/TimerLog	spectator
tjp/GetInfo	spectator
tracing/lib/AbstractTrace	spectator
tracing/lib/TraceMyClasses	spectator
tracing/version1/TraceMyClasses	spectator
tracing/version2Trace	spectator
tracing/version2/TraceMyClasses	spectator
tracing/version3/Trace	spectator
tracing/ version3/TraceMyClasses	spectator
bean/BoundPoint	client utility
introduction/CloneablePoint	client utility
introduction/ComparablePoint	client utility
introduction/HashtablePoint	client utility
observer/SubjectObserverProtocol	client utility
observer/SubjectObserverProtocolImpl	client utility
spacewar/Display.DisplayAspect	client utility
spacewar/Display1.SpaceObjectPainting	client utility
spacewar/Display2.SpaceObjectPainting	client utility
telecom/Billing	client utility
telecom/Timing	client utility
spacewar/EnsureShipIsAlive	impl.utility
spacewar/GameSynchronization	impl.utility
spacewar/RegistrySynchronization	impl.utility
spacewar/Registry.RegistrationProtection	impl.utility
coordination Coordinator	assistant
spacewar Debug	combined
Coordinator refers only to abstract pointcuts.	
GameSynchronization and RegistrySynchronization	
extends sf Coordinator assistant.	

Table 2.3: Example aspects and their categories

Example	category
<i>Development Aspects</i>	
Logger	spectator
Tracer	spectator
Profiler	spectator
CodeSegregation	not defined
<i>Production Aspects</i>	
Authentication	client utility
Exceptions	client utility
NullChecker	spectator
<i>Runtime Aspects</i>	
OutputStreamBuffering	impl.utility
Pooling	impl.utility
ConnectionChecking	impl.utility
ReadCache	impl.utility
<i>not categorized</i>	
NewLogging	client utility
PaidStories	spectator
Profiler and NullChecker needs minor change to make this aspect a spectator. CodeSegregation introduces warnings and errors, which are outside the scope of the current work. Authentication includes some features (parent declaration) that are outside the scope of the current work.	

Table 2.4: Example aspects from Kiselev's book

that these constructs can be allowed in spectator aspects because they do not change the behavior of a program in any way. **Authentication** aspects declares **declare parents** that is also out of their current work. The aspects that declare **declare parents** can be allowed in either spectators or assistants in its own case.

By looking at the accept clause or the accept maps, developers can keep modular reasoning. Accept maps also ensure their visibility over other modules. At the beginning of a module, it lists all the possible locations where an aspect map naming that module might appear. From a module listed at the package clause, accept maps is visible. It is sure that, once a class accepts assistants by using accept maps or not, developer can do modular reasoning

in that class. However, class writers have to declare the acceptance clause each time of applying aspects. One of the benefit in AOP is applying aspects after defining classes with no modification of these base codes.

## 2.8 Pointcut Interfaces

In pointcut interfaces [10], named pointcuts provide a basis for a new kind of interface. There will be three possibilities for the scope of a named pointcut:

1. When the pointcut is semantically scoped within a class, then it can be placed in the class.
2. When the pointcut is semantically scoped within a single package, then it can be placed within a special "Pointcuts" class in this package.
3. When the pointcut is not semantically restricted to any particular package, a special "pointcuts" package may be introduced, with classes to hold these global pointcuts.

Note that the term *semantically scoped* means a scope of a subjective design feature. Pointcut interface enables class writers to do modular reasoning in a class. However, it is also weak against the future refactoring.

## 2.9 Join Point Encapsulation

*Restriction advice* [17] identifies which joinpoints are encapsulated against aspects. It specifies the join points that are not selectable for aspects. Although it prevent aspects modifying classes, class writers have to anticipate firstly which joinpoints are extensible for the future refactoring. This anticipation seems to be difficult. Basically, there is no restriction in aspects without a restriction advice. Therefore, it can be said that restriction advice is looser language mechanism than open modules are. Note that, open modules initially expose no joinpoints.

Restriction advice is alike as other advices. It uses the pointcut language to specify which joinpoints are restricted. Restriction advice can be declared in AspectJ program. For example, next `restrict` advice hides all private methods from aspects.

```
restrict() : call(private * *(..));
```

After joinpoints are affected by restriction advices, the weaver applies other aspects to classes. The procedure of the restriction is simple. It apply **restrict** entity to other normal pointcuts. For example, let's consider a piece of advice that attempts to modify methods in aspect:

```
before() : call(* Figure+.*(..)) {...}
```

Because there is restriction advice that matches above advice, the weaver does not allow this **before** advice to attach to the restricted joinpoints. The weaver processes this advice effectively as follows.

```
before() : call(* Figure+.*(..)) && !call(private * *(..)) {...}
```

## 2.10 Summary

To address obliviousness problem, several programming tools for AOP have been developed. One of the most popular tools is AJDT, AspectJ Development Tools of Eclipse IDE [29]. It automatically performs a whole-program analysis and visualizes the crosscutting structures in the program according to the result of the analysis. The developers do not have to manually perform a whole-program analysis any more. However, AJDT does not seem to satisfy developers. Their claim is that they want to see static module interfaces for understanding their programs. Here, the module interfaces include the specifications of the behavior of the modules. Although AJDT automatically performs a whole-program analysis while a developer is editing a program, the visualization by AJDT does not much help the developer see the module interfaces. It does nothing except simply showing the join points where modules are combined with aspects. Even worse, module interfaces in AOP languages are never static or stable. It changes according to the deployment of aspects. In this sense, the module interfaces in AOP are essentially different from traditional ones.

Another approach to address the drawbacks of the obliviousness property is to introduce language constructs into AOP languages. There have been several constructs proposed on this approach: for example, open modules [1, 25] and XPIs (crosscut programming interfaces) [9]. These approaches have no need to analyse the whole program including aspects. Instead, developers declare a module interface for pointcuts. They must explicitly specify selectable join points from external clients so that the fragile pointcut problem [16] can be avoided. The developers can take care of those selectable join points when they modify the implementation of the module. A disadvantage

of this approach is that developers must anticipate join points that will be selected by aspects deployed in future. Anticipating all necessary join points in advance is difficult. Otherwise, developers must manually update module interface whenever new join points must be selectable. The approach of AspectScope is to visualize currently selected join points and hence it complements the approach of open modules and XPI. After an aspect has been woven to a classe, focusing around the target method that are selected by the advice, other methods that call it or are called by it are indirectly affected by the advice. Therefore, the specification of these methods are also changed along the extension of the advice.

In object-oriented programming, developers can do modular reasoning of a method, only looking at the specifications of methods called by the method. One of the good case examples is API (Application Program Interface). Unlike AOP, the specification of a method does not change after compiling or executing a program in OOP. Therefore, developers can write codes with modular style.

## Chapter 3

# AspectScope

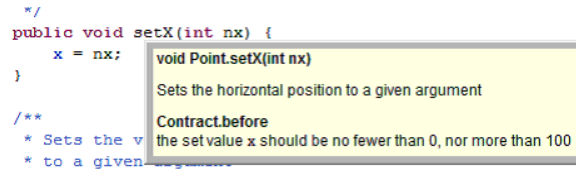
Although AJDT visualizes crosscutting structures in a program, it only indicates where a crosscutting structure joins other structures, that is, it only indicates join points in the source code. As we have seen in the previous section, this visualization is not sufficient to help developers understand crosscutting structures in their programs.

For better help, we have developed another programming tool for AspectJ. It is an Eclipse plugin named AspectScope. This tool visualizes crosscutting structures by showing how aspects affect the module interfaces in the program. Like AJDT, the tool performs a global analysis of the deployment configuration of aspects but it presents the result of the analysis from the viewpoint of how the module interfaces of classes are extended by aspects. Our tool performs projection to the methods that are indirectly affected by aspects. Developers have no need to look at these methods implementations to know the effects of aspects. This keeps the encapsulation rule of objects, and helps developers with modular programming. We have an assumption that AspectScope is effective to modify the programs developed by AspectJ.

### 3.1 AspectScope editor

AspectScope editor displays the specification affected by aspects on the method. For example, it displays the specification of `setX` method in `Point` class like Figure 3.1. It tells that `before` advice in `Contract` aspect extends the behavior of `setX` method with the javadoc comment of the `before` advice in the pop-up window and with the arrow in the ruler of the editor.

AspectScope also display the aspect extentions on the methods that are indirectly affected by aspects. Thus, AspectScope can reflect aspect effects



```

    */
    public void setX(int nx) {
        x = nx;
    }
    /**
     * Sets the v
     * to a given
  
```

void Point.setX(int nx)  
 Sets the horizontal position to a given argument  
**Contract.before**  
 the set value x should be no fewer than 0, nor more than 100

Figure 3.1: The display by AspectScope

not only on the target method that is selected by the a pointcut but also on the methods that are called by the target methods or that call the target method along the method call graph of the target method. For example, a call pointcut has an effect on both caller classes and callee classes [12]. An advice designates the call pointcut as follows.

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

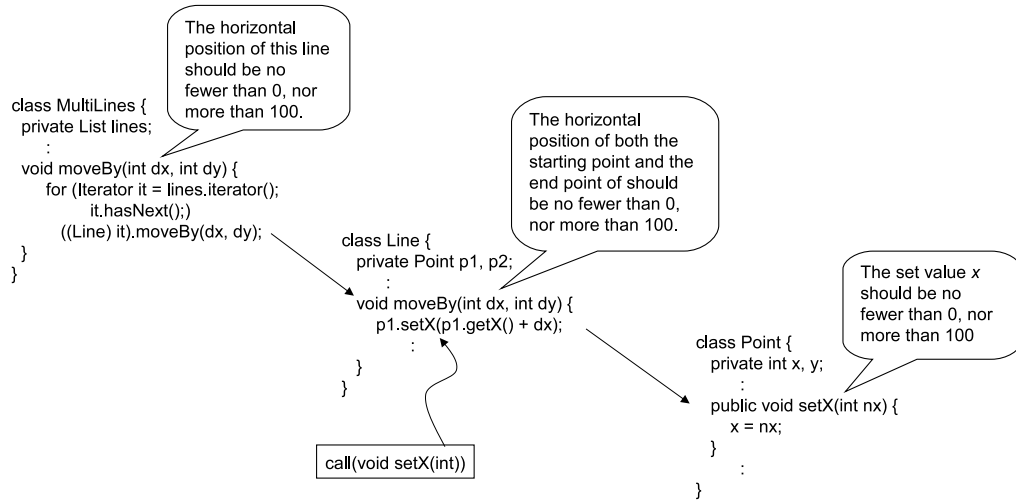
This pointcuts causes the extensions of `setX` method in `Point` class as well as `moveBy` method in `Line` class that calls the `setX` method. Besides, `moveBy` method in `MultiLines` class that calls the `moveBy` method in `Line` class is also extended because of this pointcut (Figure 3.2). The most important issue is that these extended specification must meet their each method abstraction (the upside comments in Figure 3.2). The specification on `setX` method meets its abstraction as shown bellow.

*The set value `x` should be no fewer than 0, nor more than 100, only if the caller is Line*

This specification can be actually seen in the the javadoc comment on `setX` method in Figure 3.1. Note that this properly specified specification is wrong with the caller method `moveBy` in `Line` class because `moveBy` method has no way to know the meaning of `x` in this comment. In addition, the content "only if the caller is Line" is clearly irrelevant to `moveBy` method that is the caller method.

A line is represented by the begging point and the end point, and `Line` class has the private field whose types are `Point`. Considering this feature of `Line` class, the following specification is appropriate to it.

*The horizontal position of both the starting point and the end point should be no fewer than 0, nor more than 100*

Figure 3.2: The call graph on `setX` method

This specification takes the phrase of the begging and end points instead of `x`, and keeps the abstraction of `moveBy` method. The `moveBy` specification of `MultiLines` class is as follows. The difference between caller `moveBy` and callee `moveBy` is the comment of saying "The horizontal positions of this line".

*The horizontal positions of this line should be no fewer than 0, nor more than 100*

These specifications suitable for the method abstractions should be written by aspect writers when they define aspects. When they implement an advice, they must also consider the actual behavior of it for consistency of the program.

### 3.2 comment aspect

AspectScope provides the feature to write the specification on a method abstraction. We call this feature *comment aspect*. Developers can append different javadoc comments on one method through comment aspect. The methods specified by the comment aspect can be seen the effects in the AspectScope editor. The display of arrows can be followed by the control

by the comment aspect.

```

aspect Contract {

    /** @comment
     *   The set value <code>x</code> should be
     *   no fewer than 0, nor more than 100
     *   @throws IllegalArgumentException
     *
     *   @comment (execution(void Line.moveBy(int, int)))
     *   The horizontal positions of both the starting point and
     *   the end point should be no fewer than more than 100
     *   @throws IllegalArgumentException
     *
     *   @comment (execution(void MultiLines.moveBy(int, int)))
     *   The horizontal positions of the lines should be
     *   no fewer than 0, no more than 100
     *   @throws IllegalArgumentException
     */
    before(int x) : execution(void Point.setX(int)) && args(x) {
        if (x < 0 || 100 < x)
            throw new IllegalArgumentException();
    }

}

```

Figure 3.3: The description of javadoc comments with **@comment** annotation

Note that the specifications on an advice cannot be always tracked back the top of the call graph. The extent of one advice influence depends on the each program architecture. Developers must estimate the applicable scope of an advice. For example, developers will write the **before** advice comment in **Contract** as shown in Figure 3.3. This comment has three **@comment** annotations. The top **@comment** annotation will be shown on the advised method that the pointcut designated. In this case, the top comment *"The set value ..."* is appended to **setX** method. This annotated comment needs no control statements. Under second annotated comments, they have the control statements that decide which methods should be appended them. Developers can write pointcut logical expressions in this control sentence

like AspectJ pointcuts. The second and third comments have execution statement.

`execution(method pattern)`

associates the javadoc comment under `@comment` with methods that fit in the `method pattern`. In this case, the second comment is appended to `moveBy` method in `Line` class, and the third one is associated with `moveBy` method in `MultiLines` class.

There are other control statements such as `within` and `caller`. `within` statement appends the comment on the methods that are in the target method call graph and are defined in the *patterns*.

`within(class pattern || method pattern)`

For example, `within(* csg.figures.*(..))` pattern associates methods that are contained in the call graph of the advised method and in `csg.figures` package. `caller` control statement takes an integer number as its argument as below. Developers will use this statement when they want to append the same comment on several methods at the same degree in the method call graph. It is useful for the methods that functions are very alike. These control statements can also be combined with each other like `"caller(2) && (within(Line) || within(Point))"`.

`caller(int)`

Besides, wild-cards can be written in the `method patterns` and the `class patterns` in these control statements. `"*"`, `"+"`, and `".."` are now available. `"*"` is the wild character. `"+"` can be used to express subclasses of the specified class name or interface name like `Figure+`. `".."` is the ellipsis pattern and available between a package name and a sub-package names, or a package name and a class name like `csg..figures.Figure` or `csg..Figure`. `".."` can omit no character, for example, `csg..figures.Figure` actually indicates `csg.figures.Figure`. This wild-card rules are same as ones available in AspectJ joinpoint patterns.

### The figure editor program

Figure 3.2 only shows the call hierarchy until `MultiLines` class. Figure classes including `MultiLines` are furthermore called in runtime through `moveBy` method

in Figure interface. The client class DrawApplication calls it as follows.

```
class DrawApplication implements
    MouseListener, MouseMotionListener {
    private int x0, y0;
    private Figure f;
    :
    public void mouseDragged(MouseEvent e) {
        if (f == null)
            return;
        f.moveBy(e.getX() - x0, e.getY() - y0);
        :
    }
    :
}
```

In `mouseDragged` method, `moveBy` method in Figure interface is called. Therefore, aspect writers must write specifications about an advice for `moveBy` method in Figure interface and `mouseDragged` method along these abstractions. First, for `moveBy` method in Figure interface, they will write a comment that the horizontal degree should be from 0 to 100. For `mouseDragged` method, they would write a comment that a mouse dragging are permitted only inside the window display of this editor. Besides, they do not need a mention about the exception handling because all exceptions that are possible to be thrown are caught in `mouseDragged` method.

```
aspect Contract {
/**
 *      :
 * @comment (execution(
 *          void Figure.moveBy(int, int)))
 * The horizontal positions of the edges
 * of this figure should be no fewer 0,
 * nor more than 100
 * @throws IllegalArgumentException
 *
 * @comment (execution(
 *          void DrawApplication.mouseDragged(
 *              MouseEvent)))
 * The released position of the mouse
```

```

    *   dragging should be inside the window.
    */
before() :
    execution(void Point.setX(int))
    && args(x) {
        :
    }
}

```

### A role as a debugging tool

Through AspectScope editor, a extended specification on a method can tell developers the program bugs. For example, consider `UpdateSignaling` aspect is defined as below. This aspect will update the display to redraw figures in the drawing editor whenever figures changed or moved.

```

aspect UpdateSignaling {
    /**
     * @comment (within(* figures.*(..))
     *   Signals the <code>Display</code>
     *   to update a shape changes.
     */
    after() :
        call(void Figure+.moveBy(int,int)); {
        Display.update();
    }
}

```

The classes defined in the figures package can be modified these specifications by `within` control statement. Then, some developer newly define `Arrow` class that represents an arrow. As shown below, `Arrow` class has a private field `tri` which type is `Triangle`. `MoveBy` method calls this `tri` field, and `tri` also calls `moveBy` method in `Triangle` class.

```

public class Arrow implements Figure {
    private Point p1, p2;
    private Triangle tri;
    :
    public void moveBy(int dx, int dy) {
        tri.moveBy(dx, dy);
        p1.setX(p1.getX() + dx);
        p1.setY(p1.getY() + dy);
    }
}

```

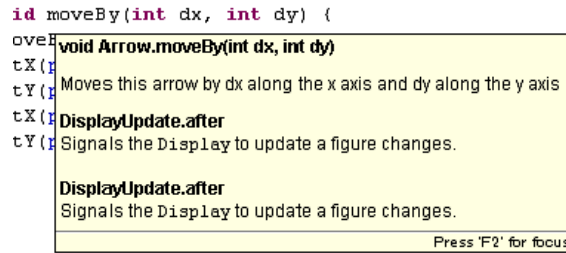


Figure 3.4: The extended comment by DisplayUpdate

```

    p2.setX(p1.getX() + dx);
    p2.setY(p2.getY() + dy);
}
}

```

Unfortunately, the program has got a bug. When an arrow drawn in the editor moves, the display flickers. In addition to `moveBy` method in `DisplayUpdate` class, another `moveBy` method in `Triangle` class is called, and the `after` advice executes the display update second time.

AspectScope tells this bug to show the `moveBy` method specification in `Arrow` class (Figure 3.4). The pop-up comment display twice the same comment of `after` advice in `UpdateSignaling`, and indicates that update method eventually executes twice.

Developers who uses AJDT may have trouble in finding this bug. Although a mark is put on `moveBy` method in `Triangle` class, no mark is put on `moveBy` method in `Arrow` class. Therefore, developers cannot find `moveBy` method in `Arrow` class is extended by the same advice. To confirm this extension, they have to find classes that call `moveBy` method in `Arrow` class, and it is hard to do.

### 3.3 comment advice

We build an special advice named `comment` in AspectJ language. If a method has no comment, `comment` advice can weave an comment on it. Developers often put no comment on private methods or even other methods. Note that aspects in AspectJ can weave advices in even private methods. Therefore, if an advice extends the private method behavior, its influence can reach upper methods in the call graph. The syntax of `comment` advice is as follows.

```
/** the comment of a target method */
comment() : execution(method pattern) {}
```

A `comment` advice can designate only execution pointcut, and weaves a comment written on it. The comment on `comment` advice will be reserved for the target method. This advice weaves the comments on methods that matches *method pattern*. Note that these woven comments can be seen through the AspectScope editor. Like other normal advices, if `comment` advice is deleted, the comment is unwoven from the target method. Following is the example of how `comment` advice is used. The `getDistance` method are woven a comment "*Returns the distance between ...*" on it.

```
/**
 * @comment
 * Returns the distance between this line
 * and the point.
 */
comment() :
    execution(int Line.getDistance(Point)) {}
```

In case developers want to write the comment that is for the `comment` advice itself, following syntax will work. The upper comment is appended on the `comment` advice itself, and lower one is for the target method which the execution pointcut selects.

```
/**
 * This comment is for the advice itself.
 *
 * @comment
 * This comment is for the woven method.
 */
comment() : execution(method pattern) {}
```

Although a `comment` advice does not allow any codes in its body, we plan to enrich the syntax utility. For example, it lets developers write more constraint rules in it. In chapter 6, we will detail the use for future works.

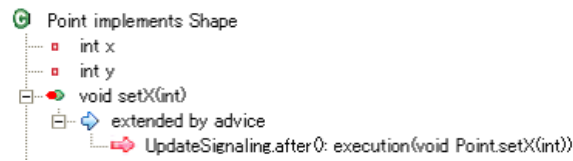


Figure 3.5: The outline view presents the effect of the execution pointcut.

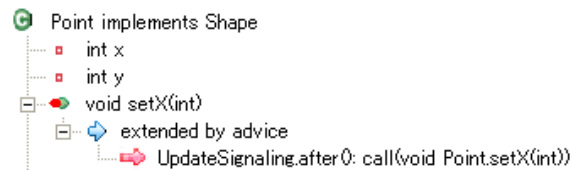


Figure 3.6: The outline view presents the effect of the call pointcut.

### 3.4 Outline Viewer

AspectScope has an outline viewer so that developers can easily find which method specifications are changed in programs. If developers attempt to find the changes of specifications, they do not need to look at the method implementations<sup>1</sup>. We'll show the current features in the outline viewer of AspectScope. As an example, `UpdateSignaling` aspect is defined as follows. `Display.update` method updates the editor display whenever a figure is moved.

```

aspect UpdateSignaling {
  after() : execution(void Point.setX(int)) {
    Display.update();
  }
}

```

#### 3.4.1 The execution and call pointcuts

If an `UpdateSignaling` aspect includes an `after` advice associated with a pointcut `execution(void Point.setX(int))`, then the outline view indicates that the `setX` method in the `Point` class is extended by the `after` advice in the `UpdateSignaling` aspect (Figure 3.5).

<sup>1</sup>At this time, AspectScope does not reflect an advice influence into all methods in their call graphs though we plan to add this feature

Note that even if the pointcut that the `after` advice is associated with is not `execution` but `call`, for example, `call(void Point.setX(int))`, then the outline view shown does not change except the description of the pointcut (Figure 3.6). AspectScope abstracts away from differences between `call` and `execution` because module interfaces affected by aspects are interesting concerns. AspectScope considers that the advice associated with either pointcut extends the behavior of the *callee-side* method. In AspectJ, both pointcuts select method calls. However, the join points (or join point shadow) selected by a `call` pointcut are method-call expressions at the *caller* side while the join points selected by an `execution` pointcut are the bodies of the specified methods at the *callee* (or *target*) side. Hence, for example, the advice associated with a `call` pointcut can obtain a reference to not only the target object but also the caller object. On the other hand, the advice associated with an `execution` pointcut cannot obtain such a reference.

Despite this difference, AspectScope uses the outline view of the *callee* side to indicate the extension by the `call` pointcut. Since the goal is to display the module interfaces affected by aspects, AspectScope must project the extension to a module interface, which is the outline view of the *callee* side in OOP. On the other hand, AJDT reflects this difference because it shows join points. If developers would like to look at the join points, AspectScope now has no difference in its view. Therefore, we plan to make the difference between the target method that a pointcut selects, and methods that have the indirect influence from an advice.

### 3.4.2 The within and cflow pointcuts

The `within`, `withincode`, `cflow`, and `cflowbelow` pointcuts select join points within a specified region. For example, the `within` pointcut selects only the join points included in the specified class. `call(void *.setX(int)) && within(Line)` selects method calls from the `Line` class to `setX` declared in any class. The selected join points are method-call expressions contained in the body of a method in the `Line` class. The `within` pointcut restricts the *caller* methods.

If the `call` pointcut is combined with the `within` pointcut, AspectScope interprets that the associated advice conditionally extends the behavior of the *callee* method. This is also true for the combination of `call` and `cflow`, `set` and `within`, and so forth. For example, if an `UpdateSignaling` aspect includes an `after` advice associated with a pointcut `call(void Point.setX(int)) && within(Line)`, then the outline view indicates that the `setX` method in the `Point` class is *conditionally* extended by the `after` advice (Figure 3.7).

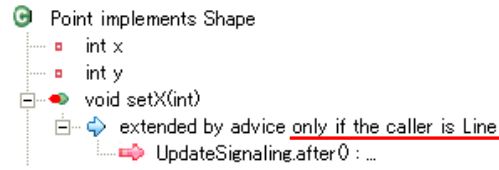


Figure 3.7: A conditional extension by the within pointcut (the red underline was drawn by the authors)

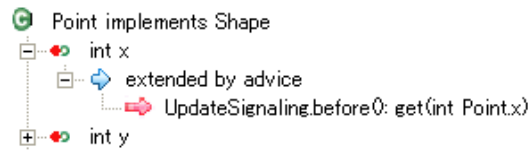


Figure 3.8: There is a before advice associated with the get pointcut.

Since the pointcut includes `within(Line)`, the outline view shows that the behavior of `setX` is conditionally “extended by advice only if the caller is Line”. The developers can see that the behavior of `setX` remains original if it is called from other classes than `Line`. If the combined pointcut is `cflow`, the outline view will show something like “extended if the thread is in the control flow of ...”

### 3.4.3 Other pointcuts and Inter-type declarations

The presentation of the `get` and `set` pointcuts in the outline view is similar to the `call` pointcut. In AspectJ, the join points selected by `get` and `set` pointcuts are field-access expressions at the accessor side (*i.e.* the caller side). Hence, AJDT shows an arrow icon at the line where the field is accessed. However, AspectScope interprets that an advice associated with a `get` or `set` pointcut extends the behavior of the target field. Figure 3.8 is an outline view presented by AspectScope. It illustrates the influence of an `UpdateSignaling` aspect that contains a `before` advice associated with a pointcut `get(int Point.x)`. Note that an arrow icon is shown below the `x` field in the `Point` class (*i.e.* at the target side) because the advice extends the behavior of the `x` field.

An aspect may include an intertype declaration. The methods and the fields appended by intertype declarations are also shown in the outline view.

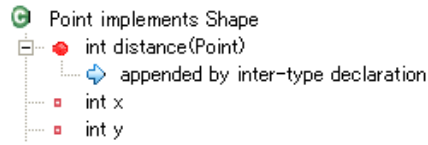


Figure 3.9: An intertype declaration of the distance method

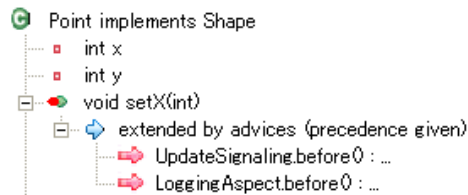


Figure 3.10: Two advices extend the setX method.

For example, Figure 3.9 indicates that an intertype declaration appends the distance method to the Point class.

If more than one advice extends a method or a field in an existing class, the outline view lists all the advices. If precedence rules are given by `declare precedence`, the multiple advice bodies extending the same method or field are listed in the execution order satisfying the given precedence rules (Figure 3.10).

### 3.5 Summary

In this section, we proposed a programming tool for AspectJ named AspectScope. AspectScope provides its editor for looking at modular interfaces extended by aspects, and developers can append an advice comment on a method for the feature of comment aspect. Besides, if a method has no comment, `comment` advice can be helpful. AspectScope is the only tool to do modular reasoning in AspectJ.

## Chapter 4

# Implementation Issues

### 4.1 Extended AJDT and JDT Parser

We extended AJDT implementation with AspectJ (Figure 4.1). AJDT mainly consists of three projects, that is, `org.aspectj.ajde`, `org.eclipse.ajdt.core`, and `org.eclipse.ajdt.ui`. The `org.aspectj.ajde` contains an aspectj compiler (`ajc`) and jdt parser for eclipse IDE. This project modifies AspectJ syntax to normal jdt syntax. For example, an advice and named pointcut are converted to a normal method in Java. Since an advice is anonymous, there is no difference between advices. Therefore, when there are more than two advices, they are renamed such as `"after()#2"`. The `org.eclipse.ajdt.core` receives the `ajc` weaving information from the `org.aspectj.ajde`, and then makes the core model of AJDT. This model has the information about all the crosscutting structures which advices extend methods and what methods are extended. The `org.eclipse.ajdt.ui` handles the view part. In AJDT editor, arrows are displayed which indicates the advice extension like Figure 2.3.

#### 4.1.1 `Org.eclipse.ajdt.core`

We took advantage of aspect-oriented programming, and extended the existing codes with aspects so that these codes will not be changed. The main purpose is to modify the original core model along with our purpose, that is, one-to-one relation between an advice and a method should be transformed one-to-many relations between an advice and methods that are in the a call graph. Therefore, `AspectScope` first obtains the one-to-one relation from the existing model. Then the tool searches for the call hierarchies of the target method and associates these methods with an advice for each. Af-

ter associating methods with an advice, AsepectScope next associates each comment of an advice with these methods. Finally, AspectScope puts back the modified model to the existing model.

The main aspect that intercept the existing org.eclipse.ajdt.core is AJ-ModelConverter aspect. This aspect gets the one-to-one relation from AJProjectModel class. With the basis of the target method information, AspectScope find its call hierarchies. To find them, this tool uses the search engine provided as the jdt libraries. The search engine is mainly used in the call hierarchy view of eclipse IDE. Concretely, the part of the implementations is as follows.

```
SearchEngine searchEngine = new SearchEngine();
IJavaSearchScope defaultSearchScope =
    CallHierarchy.getDefault().getSearchScope();
boolean isWorkspaceScope = SearchEngine.createWorkspaceScope()
    .equals(defaultSearchScope);
IJavaSearchScope searchScope = isWorkspaceScope ?
    getAccurateSearchScope(defaultSearchScope, member)
    : defaultSearchScope;
SearchPattern pattern = SearchPattern.createPattern(member,
    IJavaSearchConstants.REFERENCES,
    SearchUtils.GENERICS_AGNOSTIC_MATCH_RULE);
searchEngine.search(pattern, new SearchParticipant[] {
    SearchEngine.getDefaultSearchParticipant()},
    searchScope,
    searchRequestor, null);
return searchRequestor.getCallers();
```

After searching for the call graph, AspectScope append each method to the prepared comments of an advice. The comment parser parses all comments that are annotated @comment, and evaluates the entities of the constraint condition such as within(\*..figures.Figure+).

#### 4.1.2 Org.eclipse.ajdt.ui

The main purpose of the extension of this project is to show the pop-up display for the method specifications that are extended by the aspects. To show pop-up display, a newly defined class have to inherit org.eclipse.jdt.internal.ui.text.java.hover.JavadocHover class, and override getHoverInfo method. In AspectScope implementation, getHoverInfo method gets

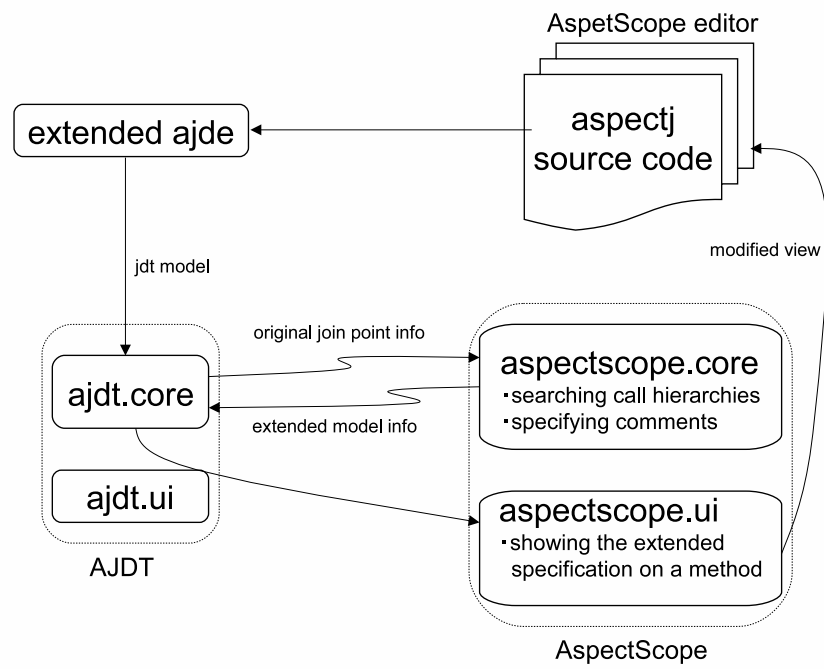


Figure 4.1: Overview of AspectScope

the crosscutting structure from the modified `org.eclipse.ajdt.core.AJProjectModel` class as shown below. The following `getRelations` method is the inter-typed method defined in `AJProjectModelConverter` aspect.

```
public String getHoverInfo(ITextViewer textViewer,
                           IRegion hoverRegion) {
    :
    Collection col = AJProjectModel.getRelations();
    :
}
```

### 4.1.3 Org.aspectj.ajde

We have to modify the JDT parser to introduce `comment` advice in the editor build in eclipse IDE. To implement jdt parser, we use the jikspg [13] that is the parser generator. Jikespg accepts as input an description for a language grammer and produces text files that is necessary for a parser such as jdt parser. We used the version 1.5 of jikspg. The grammer is LALR(1). A part of grammer file *java\_1.5.g*<sup>1</sup> is described as follows.

```
$Terminals
    comment
    :
    AJSimpleNameNoAround -> 'comment'
    :
    AspectBodyDeclaration -> BasicAdviceDeclaration
    :
    BasicAdviceDeclaration ::= BasicAdviceHeader MethodBody
    /.$putCase consumeBasicAdviceDeclaration(); $break ./
    /:$readableName AdviceDeclaration:/

    BasicAdviceHeader ::= BasicAdviceHeaderName
                        FormalParameterListopt
                        MethodHeaderRightParen
                        ExtraParamopt
                        MethodHeaderThrowsClauseopt
                        ':' PseudoTokens
    /.$putCase consumeBasicAdviceHeader(); $break ./
```

---

<sup>1</sup>The existing grammer file and other related files can be available from a cvs server of [aspectj.org](http://aspectj.org) [2]

```

/::$readableName AdviceHeader:/

BasicAdviceHeaderName ::= Modifiersopt 'comment' '('
/.$putCase consumeCommentHeaderName(); $break ./
/::$readableName CommentHeaderName:/

FormalParameterListopt ::= $empty
/.$putCase consumeFormalParameterListopt(); $break ./
FormalParameterListopt -> FormalParameterList
/::$readableName FormalParameterList:/

FormalParameterList -> FormalParameter
FormalParameterList ::= FormalParameterList ',' FormalParameter
/.$putCase consumeFormalParameterList(); $break ./
/::$readableName FormalParameterList:/

MethodHeaderRightParen ::= ')'
/.$putCase consumeMethodHeaderRightParen(); $break ./
/::$readableName ):/

ExtraParamopt ::= 'Identifier' '(' FormalParameter ')'
/.$putCase consumeExtraParameterWithFormal(); $break ./
/::$readableName ExtraParam:/

ExtraParamopt ::= 'Identifier' '(' ')'
/.$putCase consumeExtraParameterNoFormal(); $break ./
/::$readableName ExtraParam:/

MethodHeaderThrowsClauseopt ::= $empty
MethodHeaderThrowsClauseopt -> MethodHeaderThrowsClause
/::$readableName MethodHeaderThrowsClause:/

MethodHeaderThrowsClause ::= 'throws' ClassTypeList
/.$putCase consumeMethodHeaderThrowsClause(); $break ./
/::$readableName MethodHeaderThrowsClause:/

Modifiersopt ::= $empty
/.$putCase consumeDefaultModifiers(); $break ./
Modifiersopt ::= Modifiers
/.$putCase consumeModifiers(); $break ./

```

```

/:$readableName Modifiers:/

PseudoTokens ::= PseudoToken
PseudoTokens ::= ColonPseudoToken
/:$readableName type pattern or pointcut expression:/

PseudoTokens ::= PseudoTokens ColonPseudoToken
/.$putCase consumePseudoTokens(); $break ./

PseudoTokens ::= PseudoTokens PseudoToken
/.$putCase consumePseudoTokens(); $break ./

PseudoTokensNoColon ::= PseudoToken
PseudoTokensNoColon ::= PseudoTokensNoColon PseudoToken
/.$putCase consumePseudoTokens(); $break ./
/:$readableName allowable token in pointcut or type pattern:/

ColonPseudoToken ::= ':'
/.$putCase consumePseudoToken(":"); $break ./
/:$readableName any allowable token in pointcut or type pattern, except ':':/

PseudoToken ::= JavaIdentifier
/.$putCase consumePseudoTokenIdentifier(); $break ./
/:$readableName allowable token in pointcut or type pattern:/

PseudoToken ::= '('
/.$putCase consumePseudoToken("("); $break ./

PseudoToken ::= ')'
/.$putCase consumePseudoToken(")"); $break ./

PseudoToken ::= '.'
/.$putCase consumePseudoToken("."); $break ./

PseudoToken ::= '*'
/.$putCase consumePseudoToken("*"); $break ./

PseudoToken ::= '+'
/.$putCase consumePseudoToken("+"); $break ./

```

File	Explanation
javaAction.java	It contains <code>consumeRule</code> method in <code>Parser</code> class that manipulates the semantic action.
javadcl.java	It is used to generate resource files.
javasym.java	It contains field declarations that will be the part of <code>TerminalTokens</code> class.
javadef.java	It contains field declarations that can be <code>ParserBasicInformation</code> class.

Table 4.1: The necessary files for build a jdt parser

These `putCase`, `break`, and `readableName` are macros. For example, `putCase` is defined as following, and is extended as a part of the `switch`-statement.

```
$putCase
/.
case $rule_number : if (DEBUG) {
    System.out.println("$rule_text");  //$NON-NLS-1$
}
./
```

After compiling the grammar file, several files are generated as listed in Table 4.1. These files are used to replace these contents with the existing codes in `org.aspectj.org.eclipse.jdt.internal.compiler.parser` package. The `javaAction.java` defines the new `consumeRule` method that should be replaced the existing one in `Parser` class. The `javadcl.java` is used to generate the resource files that are the binary files and handles lexer function. `UpdateParserFiles`<sup>2</sup> class generates twenty four resource files which names are `parser<n>.rsc`, with `n` equals to 1 to 24. To generate resources files, following command line is run. The second program argument "`javahdr.java`" is acceptable even if it is an empty file.

```
> java UpdateParserFiles javadcl.java javahdr.java
```

These newly generated resource files should be moved to `org.aspectj.org.eclipse.jdt.internal.compiler.parser` package.

---

<sup>2</sup>is also available from the cvs of [2]

## Chapter 5

# Evaluation

### 5.1 A case study with an web-based information system

To evaluate the usefulness of AspectScope, we used it for browsing the source program of the health watcher program [8], which is web application server for customer complaints written by the third party. This web server is implemented in two aspect oriented language, CaesarJ [22] and AspectJ. We used the AspectJ program for our evaluation. The program is written in AspectJ and it consists of 692 classes (9,591 lines) and 25 aspects (1,989 lines). We can classify these aspects into six groups, design patterns, persistent systems, transactions, exception handlings, and loggings.

#### 5.1.1 Design Patterns

The health watcher program uses the design pattern [7] in aspect oriented programming [11]. Concretely, an *observer* pattern, a *command* pattern, a *factory* pattern, and *state* pattern are built in.

In observer pattern, The registration of observers and the notification of the observers are crosscutting structures, and are implemented as an aspect. For example, following advice extends seventeen method behaviors.

```
after(Subject subject):
    call(* Subject+.set*(..))
    && this(CommandServlet+)
    && target(subject) {
    Iterator iter =
        getObservers(subject).iterator();
```

```

while ( iter.hasNext() ) {
    updateObserver(subject,
                    ((Observer)iter.next()));
}
}

```

One of the extended methods is `setPassword` method in `UpdateEmployeeData`. When this method is called, the advice will be executed to notify observers. The `setPassword` is invoked in `executeCommand` method in the same `UpdateEmployeeData` class as showing below.

```

public class UpdateEmployeeData
    extends CommandServlet {
    public void executeCommand(CommandReceiver
                               receiver) {

        Employee employee = null;
        String newPassword = ...;
        :
        // the joinpoint caught by the advice
        employee.setPassword(newPassword);
        :
    }
}

```

The local variable `employee` is only called in `executeCommand` method. This `executeCommand` method is invoked at runtime through the call to `executeCommand` method in `Command` interface. Therefore, the hierarchical structure of the call graph is only one-tier.

As shown below, developers should append the advice javadoc comment reflected on `setPassword` method in `UpdateEmployeeData` class which is the caller class.

*Calls `<code>updateObserver</code>`* after setting the new password of the employee to update the Observer.

Besides, the following javadoc comment is essential for `executeCommnd` method. The `newPassword` is a local variable in `executeCommand` method, and this comment must not be said that a password will be modified.

*Updates the information of the employees by calling `<code>updateObserver</code>`*.

About the other design patterns, developers will have to add comments along the hierarchical structure of these own call graphs as in Table 5.1 and Table 5.2. As shown in Table 5.2, the observer aspects need callee class extensions because these advices designate `call` pointcuts. Note that a `call` pointcut intercepts a call to a target method. However, AspectScope can extend the callee class specifications using comment aspect. To keep modular reasoning, AspectScope shows the extended module interface without distinguishing a `call` pointcut and a `execution` pointcut.

In the command pattern, the hierarchical structure is one-tier in the same way of invoking the interface method in the observer pattern. In the state pattern, developers will must write comments for the nine-tier methods. The factory pattern in this program has no advice while it has inter-type declarations.

aspect patterns	method extensions	one	two	three	six	nine
Observer	17	17	0	0	0	0
Command	3	3	0	0	0	0
Factory	0	NA	NA	NA	NA	NA
State	15	14	13	10	5	2

Table 5.1: The numbers of extended comments that developers will have to add for upper methods in the call graph

aspect patterns	one
Observer	17
Command	0
Factory	NA
State	0

Table 5.2: The numbers of extended comments that developers will have to add for lower methods in the call graph

### 5.1.2 Logging

As we detail in Section 2.4 of chapter 2, the Logging code does not influence a program semantics (Clifton *et.al.* call them *spectators*), and developers may not always have to prepare comments for extended method by logging.

HWLoggin aspect logs the invocation of a constructor when a constructor of HealthaWatcherFacade class is called. One of the caller class of this constructor is getInstance method in HealthWatcherFacade class as described below.

```
public synchronized static
    HealthWatcherFacade getInstance() {
    if (singleton == null)
        singleton = new HealthWatcherFacade();
    return singleton;
}
```

When developers add an advice comment on getInstance method, following statements may be adequate.

*Configures the logging before returning the instance*

Table 5.3 represents the number of hierarchical structure about which HWLogging aspect extends. In this login, there is no advice using call pointcut, and developers do not need the callee side extension of a specification. "()" indicates that they have no necessity of writing advice specifications.

Aspect	method extension	one	two
HWLogging	2	(2)	(1)

Table 5.3: The number of extended comments that developer have to add for the logging aspect

### 5.1.3 Exception Handling

We found aspects which execute exception handlings. ExceptionHandlingPrecedence is the aspect which defines the precedence of other aspects. Developers will not need to reflect the advice specifications on target methods in the three aspects except HWPersistenceExceptionHandler. For example, around advice in HWDistributionExceptionHandler aspect designates the pointcut shown below.

```
execution(
    * HWServlet+.do*(HttpServletRequest, HttpServletResponse))
&& args(..., response)
```

An advice that uses this pointcut extends `doGet` method in `HttpServlet` class. The `doGet` method is invoked when HTTP GET request is sent, and developers will not need to write the advice specification for this `doGet` method.

aspects	me <sup>1</sup>	one	four	six	seven	nine
<code>HWDistributionExceptionHandler</code>	4	0	0	0	0	0
<code>HWPeRsistenceExceptionHandler</code>	9	5	5	3	2	1
<code>HWTransactionExceptionHandler</code>	4	0	0	0	0	0
<code>HWUpdateObserverExceptionHandler</code>	5	0	0	0	0	0
<code>ExceptioHandlingPrecedence</code>	0	NA	NA	NA	NA	NA

Table 5.4: The numbers of extended comments that exception handling aspects extend for upper methods in the call graph

aspects	one
<code>HWDistributionExceptionHandler</code>	0
<code>HWPeRsistenceExceptionHandler</code>	2
<code>HWTransactionExceptionHandler</code>	0
<code>HWUpdateObserverExceptionHandler</code>	0
<code>ExceptioHandlingPrecedence</code>	NA

Table 5.5: The number of extended comments that exception handling aspect extend for lower methods in the call graph

## 5.2 Experiment

This section reports the results of our experiment to measure the cost of obtaining and modifying cross-cutting structures in AspectScope. The machine we used for experiment is Core Duo L2500 processor(1.83GHz), 1.5GB memory, Windows Vista. The IDE is Eclipse 3.2, and our Modified projects is `org.aspectj.ajde` 1.5.3, `org.eclipse.ajdt.core` 1.4.1, and `org.eclipse.ajdt.ui` 1.4.1.

Figure 5.1 illustrates the time until a tool gets all the cross-cutting structures, that is, which advice extends methods and which method is extended by advices. This averaged time is obtained by executing 100 times for each. An before advice extends `moveBy` methods in figure classes, and the javadoc

comment on its advice is :

```
/**
 * @comment
 *   An after advice signals the < code > Display < /code >
 *   to update whenever a figure changes.
 * @comment (execution(void run(..)))
 *   ...
 */
```

Since the hierarchical structures of these call graphss are each one-tier, the comment consists of two annotated comments. The pointcut declaration is:

```
execution(void moveBy(int, int))
```

and `moveBy` method are defined in four classes, `Line`, `Point`, `Triangle`, and `Arrow`. Besides, `moveBy` methods in these classes are all called in a client class. In AJDT, it takes 34.48(ms) to calculate cross-cutting structures. On the other hand, in AspectScope, it takes 2864(ms). From the result of this experiment, AspectScope assumes to consume a lot of time to compute call graphs and to parse the constrained condition on annotation.

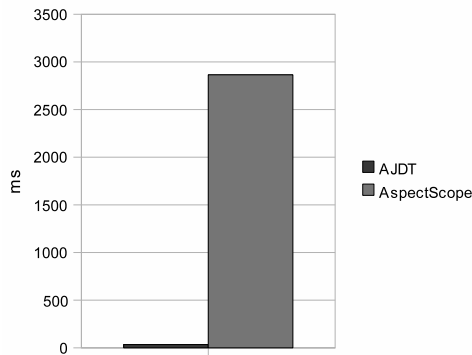


Figure 5.1: Calculations of cross-cutting structures

We then measured the actual time of computing call graphs and paring comments. Figure 5.2 illustrates the averaged time of these computation by executing 100 times. It takes 2707(ms) to obtain call graphs, and 31.46(ms) to parse comments. Getting call graphs information is intensive cost, and

we have to improve the implementation to calculate the call graphs.

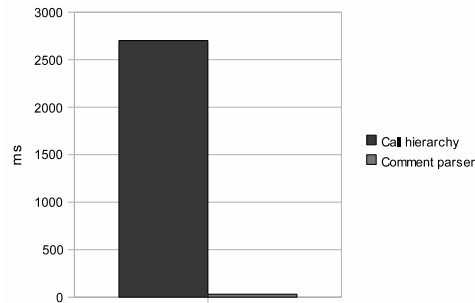


Figure 5.2: Comprising data of calculations in AspectScope

### 5.3 Summary

From this case study, we found that developers will need to write advice specification on the caller methods in most aspects. Advices in this program usually extend methods until three or four-tier in the hierarchical structure of the call graphs. This is in larger part due to that interface methods intervene the call graphs.

## Chapter 6

# Future work

### 6.1 Modularization of comments

AspectScope now let developers write several javadoc comment on one advice with simple enumeration. However, since these comment are not modularized, they are very fragile to a modification of a program. The better way to write comments may be modularising per each call graph. For example, following is call graphs spreading from `setX` method. When `setX` method is extended by an aspect, all specifications of methods in the call graphs will be also extended.

```
MultiLines.moveBy(int, int)      Arrow.moveBy(int, int)
      |                          |
      v                          v
Line.moveBy(int, int)      Triangle.moveBy(int, int)
      \                        /
      Point.setX(int)
```

With the feature of *comment aspect*, an advice can have several javadoc comments that are reflected on the methods as follows:

```
/**
 * @comment (execution(void Line.moveBy(int, int)))
 *   The advice comment reflected on Line.moveBy
 *   along its abstraction
 *
 *   ...
 */
after() : ...
```

This way of appending comments on an advice are fragile and inelegant. The modular style of comment will be written separately from an advice as described below. These modules should be generated automatically by the tool. Developers only write comments on each method.

```

/** The advice comment on Point.setX */
Module0: module(Point.setX(int)) {
}

/**
 * The advice comment reflected on
 * Line.moveBy along its abstraction
 */
Module1: module(Line.moveBy(int, int)) {
    Module0;
}

/** ... */
Module2: module(Triangle.moveBy(int, int)) {
    Module0;
}

/** ... */
Module3: module(MultiLines.moveBy(int, int)) {
    Module1;
}

/** ... */
Module4: module(Arrow.moveBy(int, int)) {
    Module2;
}

```

Each module has its own comment that represents an advice comment for this module, and contains other modules that is called in that module. In AspectJ, an advice such as `after` is anonymous. Therefore, the challenge is how these modules connect with a particular advice. We will have to extend the advice syntax in AspectJ. In addition, we will have to decide the syntax rule when aspects extend the same join point.

## 6.2 comment advice control

In chapter 3, we discuss that AspectScope can work as a debugging tool with an example of the Observer aspect pattern. AspectScope editor shows the same advice comment to make developers know the redundant updates. On the other hand, as shown below, `moveBy` method in `Line` class also shows developers being extended by `Contract` aspect though this extension is harmless.

*The horizontal positions of this line should be no fewer than 0, nor more than 100*

*The horizontal positions of this line should be no fewer than 0, nor more than 100*

In such a case, AspectScope will have to control the comment display. One possible way is that `comment` advice can have a control syntax as following. The display of `moveBy` method in `Line` class is aggregated by this `comment` advice. In this constraint statement, an advice have to designate a named pointcut to specify which extension display is redundant.

```
comment() : execution(void Line.moveBy(int, int)) {  
    aggregate : Contract.preCondition();  
}
```

## Chapter 7

# Concluding Remarks

This thesis has discussed a programming tool for AspectJ, named AspectScope. AspectScope performs a whole-program analysis of AspectJ programs and visualizes the result so that developers can understand their program behavior with local reasoning. It displays the module interfaces extended by aspects under current deployment. A unique idea of AspectScope is to reflect an aspect specification even on the methods that are indirectly influenced by an aspect. Developers can append several specifications on an advice by the feature of comment aspect, and can control the advice specifications to the extent necessary to each program. When a method has no comment, the newly defined `comment` advice allows developers to append a comment to it. The effect of weaving a `comment` advice can be seen through the AspectScope editor.

This notion of AspectScope enables expressing the effects of aspects through module interfaces. Developers thereby do AOP by using their OOP experiences of modular programming, in particular, modular extensions to classes by virtual classes [21], mixin-layers [27], nested inheritance [23, 24], and so on.

# Bibliography

- [1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *ECOOP 2005 - Object-Oriented Programming, LNCS 3586*, pages 144–168. Springer-Verlag, 2005.
- [2] AspectJ Organization. *AspectJ*. <http://www.eclipse.org/aspectj/>.
- [3] Curtis Clifton and Gary T. Leavens. Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning. Technical report, Iowa State University, 2002.
- [4] Curtis Clifton and Gary T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *FOAL 2002*, 2002.
- [5] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 158–168, New York, NY, USA, 2006. ACM.
- [6] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.
- [7] Erich Gamma, Richard Helm, Ralph Johnson, and John M Vlissides. *Design Patterns*. Addison-Wesley, 1994.
- [8] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Claudio Sant Anna, Sergio Soares, Paulo Borba, Uira Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *ECOOP 2007 - Object-Oriented Programming, LNCS 4609*, pages 176–200. Springer-Verlag, 2007.
- [9] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software

- Design With Crosscutting Interfaces. In *IEEE Software*, vol.23, pages 51–60, 2006.
- [10] S. Gudmundson and G. Kiczales. Addressing practical software development issues in aspectj with a pointcut interface, 2001.
- [11] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.
- [12] Michihiro Horie and Shigeru Chiba. An Outline Viewer for AspectJ Programs. TOOLS EUROPE 2007, 2007.
- [13] IBM. *Jikes Parser Generator*. <http://www.alphaworks.ibm.com/formula/JikesPG>.
- [14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, LNCS 2072*, pages 327–353. Springer-Verlag, 2001.
- [15] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.
- [16] Christian Koppen and Maximilian Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software (EIWAS'04)*, 2004.
- [17] David larochelle, Karl Scheidt, and Kevin Sullivan. Join Point Encapsulation. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies(SPLAT) 2003*, 2003.
- [18] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
- [19] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

- [20] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs, 2002.
- [21] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 52–67, New York, NY, USA, 2002. ACM.
- [22] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.
- [23] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 99–115, New York, NY, USA, 2004. ACM.
- [24] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.
- [25] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 39–50, New York, NY, USA, 2006. ACM.
- [26] Awais Rashid. Aspects and Evolution: The Case for Versioned Types and Meta-Aspect Protocols, 2006.
- [27] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
- [28] Friedrich Steimann. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.*, 41(10):481–497, 2006.
- [29] The Eclipse Foundation. *AspectJ Development Tools(AJDT)*. <http://www.eclipse.org/ajdt>.

- [30] Elcin Recebli Wolfson. Pure aspects. Master's thesis, Oxford University, 2005.