A Unit Testing Framework for Aspects without Weaving

Yudai Yamazaki
Shibaura-Institute of Technology
307 Fukasaku, Minuma-ku,
Saitama-shi, Saitama-ken, Japan
+81 48-683-2020
l01104@sic.shibaura-it.ac.jp

Kouhei Sakurai
Shibaura-Institute of Technology
307 Fukasaku, Minuma-ku,
Saitama-shi, Saitama-ken, Japan
+81 48-683-2020
sakurai@komiya.ise.shibaura-it.ac.jp

Saeko Matsuura
Shibaura-Institute of Technology
307 Fukasaku, Minuma-ku,
Saitama-shi, Saitama-ken, Japan
+81 48-683-2020
matsuura@se.shibaura-it.ac.jp

Hidehiko Masuhara
University of Tokyo
3-8-1 Komaba, Meguro-ku,
Tokyo, Japan 153-8902.
+81 3-5454-6679
masuhara@acm.org

Hiroaki Hashiura
Shibaura-Institute of Technology
307 Fukasaku, Minuma-ku,
Saitama-shi, Saitama-ken, Japan
+81 48-683-2020
hashiura@komiya.ise.shibaura-it.ac.jp

Seiichi Komiya
Shibaura-Institute of Technology
307 Fukasaku, Minuma-ku,
Saitama-shi, Saitama-ken, Japan
+81 48-683-2020
skomiya@sic.shibaura-it.ac.jp

ABSTRACT

Unit testing of aspects can verify aspects implementations of aspects against their specification. Current technique for unit testing of aspects requires to weave the aspect definition into a target program, which thus makes it difficult to write comprehensive test cases and to avoid interference from other aspects. In this paper, we propose a framework for unit testing aspects without weaving. Our framework generates testing methods from an aspect definition so that test cases can directly verify properties of aspects such as the advice behavior and pointcut matching.

Keywords

Aspect, Unit Testing, Framework, AOP, AspectJ

1. INTRODUCTION

Aspect-Oriented Programming(AOP) [1] allows modular implementation of crosscutting concerns in software development subjects. A typical software system comprises several core concern and the other crosscutting concerns. An aspect offers the unit of modular definition of crosscutting concerns. Comparing with the other programming paradigms such as object-oriented programming, the separation of crosscutting concerns improves modularity of the program. On the other hand, it becomes difficult to test whether the woven program operates correctly. In the unit testing in AOP, there is a work like [2]. But the method of unit testing without weaving an aspect with associated classes is not clear. Also in the aspectj-users mailing list [3], it argues about this problem actively, and Adrian Colyer pointed out as follows.

Current unit-testing approaches for aspects are lacking in the following ways:
** you cannot easily unit test an individual aspect in isolation from the rest of the program*
** you cannot easily test whether the pointcut expression associated with a piece of advice matches the join points you expect*
** you cannot easily test whether the pointcut expression associated with a piece of advice matches unwanted join points*
** you cannot easily test the body of advice in isolation from the rest of the program*

We propose a method of unit testing without weaving an aspect by describing test cases from the same viewpoint as describing the aspect for the program. For this purpose, the framework for describing the test cases with the method generated from aspect description is offered. This aims at solving the second, third and fourth problems that Adrian Colyer pointed out.

The rest of the paper is organized as follows. Section 2 presents problems of unit testing in AOP. Section 3 explains our unit testing framework and describes how to use the framework. Section 4 describes how to implement the framework in AspectJ. Section 5 discusses unit testing for aspect modules in AOP. Section 6 concludes the paper.

2. PROBLEM OF UNIT TESTING IN AOP

This section presents an example to clarify a problem of unit testing in AOP. Section 2.1 presents an example program with an aspect that tracks movement of figures. Section 2.2 shows a process to test the program after weaving aspects. Section 2.3 analyzes some problems that lie in the process.

2.1 Aspect Example

Graphical applications often re-draw figures such as points and lines. A typical requirement to such applications is to re-drawonly when a figure actually moves; thus it is necessary to observe movement of all figures. Tracking movement of figures is a crosscutting concern for the applications as shown in Figure 1.

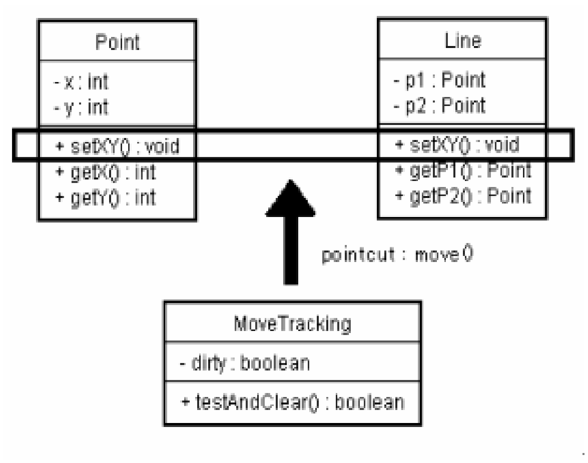

Figure 1. Tracking Movement of Points and Lines.

There are the following specifications in the tracking movement of figures concern:

- A minimum composition unit of a figure is a point that has coordinates. All figures in the program are defined by this premise. The figure in this example is defined by the **Point** class or the **Line** class shown in Figure 1.
- A figure moves when it is called the method **setXY(int,int)**, which changes the coordinates of the figure.
- The **testAndClear()** method checks movement of figures.
- When once any figure moves to anywhere by the time the program calls the **testAndClear()** method, it returns **true** and the state is cleared.
- When any figure does not move to anywhere by the time the program calls the **testAndClear()** method, it returns **false**.

Figure 2 is an implementation of the tracking movement in AspectJ under the abovementioned specification.

```
aspect MoveTracking{
   private static boolean dirty = false;
   public static boolean testAndClear(){
      boolean result = dirty;    dirty = false;    return result;
   }
   pointcut move() : call(void *.setXY(int, int));
   after() returning : move(){
      dirty = true;
   }
}
```

Figure 2. An Implementation of Tracking Movement in AspectJ.[1]

We will test whether this implementation code satisfies its specification by unit testing for associated modules.

Unit testing is a testing method for one program module. Since it does not contain any testing element of the other modules, it has following merits in software testing [5].

- It is not necessary to test by two or more modules combining.
- The place where some errors arose becomes clear.
- It can run in parallel.

2.2 Unit Testing after Weaving

To verify that **MoveTracking** aspect meets the specification mentioned in Section 2.1, we define the following test cases.

- When the **testAndClear()** method is called twice consecutively, the second call returns **false**.
- Immediately after the construction of a **Point** object, or after a call to **getX()** or **getY()** method of a **Point** object, the **testAndClear()** method returns **false**.
- After the **setXY(int,int)** method of a **Point** object is called, the **testAndClear()** method returns **true**.

We also define the test cases for all methods of the **Line** class in a similar manner to the last two cases.

Figure 3 shows an implementation of these test cases in JUnit [6].

```
1.   class MoveTrackingTest extends TestCase{
2.      Point p;
3.      void setUp(){
4.         p = new Point();
5.      }
6.      void testTestAndClear(){
7.         MoveTracking.testAndClear();
8.         assertFalse(MoveTracking.testAndClear());
9.      }
10.     void testPointNew(){
11.        assertFalse(MoveTracking.testAndClear());
12.     }
13.     void testPointGetX(){
14.        p.getX();
15.        assertFalse(MoveTracking.testAndClear());
16.     }
17.     void testPointGetY(){
18.        p.getY();
19.        assertFalse(MoveTracking.testAndClear());
20.     }
21.     void testPointSetXY(){
22.        p.setXY(1, -2);
23.        assertTrue(MoveTracking.testAndClear());
24.     }
25.     // Test cases for the Line class are omitted.
26.  }
```

Figure 3. Test Cases for the Woven Program.

2.3 Problems

- Problem of object creation and method calls

In order to verify the behavior of the advice bodies, the test cases have to include "glue code" that creates objects and calls methods. In other words, the testing method requires the definitions in of the target program even for verifying the sole behavior of aspects. This is because AspectJ offers no direct means of running bodies of the advice.

Specifically, the test cases in Figure 3 have to create a **Point** object (ll.2-5), and call the **getX(), getY()** and **setXY(int,int)**

[1] This aspect is defined by modifying the sample code of [4].

method (ll.14, 18, and 22, respectively) in order to verify the behavior of advice bodies. The code for creating objects and calling methods would became more complicated for practical applications.

- Problem of interference from other aspects
Even if the aspect definition correctly implements the specification of the concern, the results of test cases may change due to interference from the other aspects. This is obviously because the testing method examines the woven program. For example, when the program also has the following aspect, which prevents calls to the **setXY()** method with a negative argument, the test case **testPointSetXY()** will fail even though the definition of the **MoveTracking** aspect itself is correct against the specification.

```
aspect NegativeArgumentPrevention{
   pointcut negativeCall(int x, int y) :
      call(void *.setXY(int, int)) && args(x, y)
            && if (x < 0 || y < 0);
   before(int x, int y) : negativeCall(x, y){
         throw new RuntimeException();
   }
}
```

In this case, a test may be interrupted when the **NegativeArgumentPrevention** aspect throws **RuntimeException**.

As it is required to enable it to test an aspect without weaving other aspects, it becomes hard to check other all aspects of which the target aspect for a test is influenced. Whenever we run tests, we have to select appropriate aspects and recompile all the program.

3. UNIT TESTING FRAMEWORK FOR UNWOVEN ASPECTS

We propose a method of unit testing without weaving an aspect by describing test cases from the same viewpoint as describing the aspect for the program. This paper assumes that aspects are written in AspectJ. The proposed unit testing framework, which is presented in this section, is integrated into the AspectJ language and its compiler.

Section 3.1 presents how to describe test cases for the example mentioned in Section 2. Section 3.2 shows our unit testing framework and how to implement these test cases using the framework.

3.1 Test Cases with Our Unit Testing Method

With our unit testing method, the following test cases verify the **MoveTracking** aspect against the specification in Section 2.1:

- When the **testAndClear()** method is called twice consecutively, the second call returns **false**.
- Whenever any join point matches the **move()** pointcut, its advice body runs. A subsequent call to the testAndClear**()** method returns **true.**
- Any method call to **Point.setXY(int,int)** matches the **move()** pointcut. So does to **Line.setXY(int,int)**.
- No method call to **Point.getX()** matches the **move()** pointcut. Similarly, there are test cases for the methods in Point and Line class except for setXY(int,int).

3.2 Unit Testing Framework

In order to directly describe such test cases, we design our framework as follows.

(A) Test cases can directly run advice bodies; i.e., without manipulating objects of target programs. This makes it possible to describe the second test case without requiring the target classes.

(B) Test cases can manually generate join points and test the pointcut expressions against those join points where join points are points at which advice can run. AspectJ employs a dynamic join point model [7], in which the join points are the points in execution. This makes it possible to describe the third and fourth test cases without the target classes.

The framework offers the following two kinds of methods in order to describe the test cases (B) :

1. TestJoinPointFactory.create(String) creates a join point object that matches to a given primitive pointcut expression.
2. For each pointcut expression in the aspect, a method that matches the pointcut and the join point object as created above. In the following example, MoveTrackingTester.testMove(TestJoinPoint) is for the move() pointcut in the MoveTracking aspect.

Figure 4 shows an implementation of these test cases in our unit testing framework.

```
1.   import framework.TestJoinPoint;
2.   import framework.TestJoinPointFactory;
3.   class MoveTrackingTester{
4.       static void afterReturningMove(){…}
5.       static boolean testMove(TestJoinPoint jp){…}
6.   }
7.   class MoveTrackingTest extends TestCase{
8.       void testTestAndClear(){
9.       MoveTracking.testAndClear();
10.          assertFalse(MoveTracking.testAndClear());    }
11.      void testAfterReturningMove(){
12.          MoveTrackingTester.afterReturningMove();
13.          assertTrue(MoveTracking.testAndClear());    }
14.      void testMove(){
15.          TestJoinPoint jp1 = TestJoinPointFactory.create(
16.                  "call(void Point.setXY(int, int))");
17.          TestJoinPoint jp2 = TestJoinPointFactory.create(
18.                  "call(int Point.getX())");
19.          assertTrue(MoveTrackingTester.testMove(jp1));
20.          assertFalse(MoveTrackingTester.testMove(jp2));    }
21.  }
```

Figure 4. An Implementation of Test Cases in our Framework.

The method that calls an advice body mentioned in (A) is the **afterReturningMove()** method in the fourth line of Figure 4. Using this method, the second test case mentioned above is described like 11-13 lines of Figure 4.

Using the **create()** and **testMove()** method, the third and the fourth test cases are described as follows.

1. Create join points that are expected to be included in the specified pointcut expression (as shown in the 15-16 lines of Figure 4).
2. Create join points that are expected not to be included in the specified pointcut expression (as shown in the 17-18 lines of Figure 4).
3. Check that join points defined in the step 1 are included in the specified pointcut expression (as shown in the 19 line of Figure 4).
4. Check that join points defined in the step 2 are not included in the specified pointcut expression (as shown in the 20 line of Figure 4).

The 1-2 lines of Figure 4 show the *import* declarations of some classes in the framework[2]. The method **create()** is defined in the TestJoinPointFactory class.

The 3-6 lines of Figure 4 is a definition of the MoveTrackingTester class which consists of the afterReturningMove**()** method and the **testMove()** method. This class definition is generated from the source code of the MoveTracking aspect. It is also offered by our framework. We call such a generated class *Tester Class*.

Using this framework, the test cases for the **MoveTracking** aspect is described as shown in the 7-21 lines of Figure 4. Using this framework, we can create any join point at the unit testing for any aspect. Therefore, it is not necessary to generate required objects or to call the methods.

4. IMPLEMENTATION

A unit testing framework for aspect modules is implemented using AspectJ compiler version 1.2. This tool is a unit testing support tool which offers three generator functions and some class to implement the functions of (A) and (B) mentioned in Section 3.2.

4.1 Tester Class Generator

Tester class consists of some methods to call each advice body defined in an aspect module and some methods to check each specified pointcut expression.

After we define any aspect module, our tool automatically generates its *Tester Class* from the source code.

Tester Class generator has the following three functions.

- A function of *Advice body method generation.*
- A function of *Pointcut expression checker generation.*
- A function of *Tester Class generation.*

4.1.1 Advice Body Method Generation

The AspectJ compiler translates an aspect module into a class file which runs on JavaVM. At this time, each advice body is changed into a public method of the class and a name of the method corresponding to the advice body is automatically generated according to a mechanical rule. It is an unreadable name for us like ajc$afterReturning$MoveTracking$1$c0539092.

This function gets these data from some classes in the AspectJ compiler and automatically generates a wrapper method which calls the unreadable named method. This wrapper method is named by its original keywords in the aspect description so that the name is more readable one for us.

For example, in the case of the advice in Figure 2, a method named **afterReturningMove** is automatically generated from such keywords as **after**, **returning**, and **move**. The function of *advice body method generation* generates the following method definition.

```
public static void afterReturningMove(){
  MoveTracking.aspectOf()
    .ajc$afterReturning$MoveTracking$1$c0539092();
}
```

In addition, a mechanism in which a name of the generated method is specified clearly is also prepared.

4.1.2 Pointcut Expression Checker Generation

This function gets all pointcut expressions from the advice declarations in the aspect, and generates a method to check each pointcut expression as follows.

```
public static boolean testMove(TestJoinPoint jp){
  return testPointcut(new TestPointcut(
    "call(void *.setXY(int,int))"), jp);
}
```

The **TestJoinPoint** class is a class for processing a *join point* as an object. The **TestPointcut** class is a class for processing a *pointcut expression* as an object. The **testPointcut()** method has both a pointcut expression object and a join point object as the parameters. If the former matches the latter, it returns true. Otherwise it returns false. These classes are offered by our framework.

4.1.3 Tester Class Generation

Using the above two functions, this function generates the class definition of Tester Class for the target aspect module. This class definition consists of the wrapper methods, the checking methods for each advice and the methods for initializing a Tester Class.

4.2 Other Components

Some classes are defined in order for the form shown in Section 3.2 using the method that checks a pointcut expression to validate it. Each test case on checking pointcut expressions is defined as follows.

1. The **create()** method of the **TestJoinPointFactory** class creates an instance of the **TestJoinPoint** class with a join point sentence as a parameter.
2. The checking method is called with the above TestJoinPoint instance as a parameter.
3. Its return value is checked.

```
void testMove(){
  TestJoinPoint jp =
    TestJoinPointFactory.create(
      "call(void Point.setXY(int, int))");
    assertTrue(
```

[2] The package name of "framework" is a temporary name for the explanation.

```
        MoveTrackingTester.testMove(jp));
}
```

A pointcut expression is changed into a syntax tree using the parser of AspectJ compiler, and is tested by being compared with the specified **TestJoinPoint** instance.

Each primitive join point that the join points are decided at the compile time such as *call* and *execution* can be set a parameter of the **create()** method.

5. DISCUSSION

This section discusses some topics related to the unit testing for aspect modules in AOP and the future problems.

5.1 Unit Testing in AOP Framework

In the framework based on JBoss AOP [8], AspectWerkz [9], and AOP alliance [10], AOP is realized as a framework, without extending the Object-Oriented Language itself. An aspect or an advice definition is defined by these frameworks as one class. Therefore, the advice definition has the specific name, and the advice body can be tested by calling in code of pure Java.

In these frameworks, a point cut is described in the form other than programs, such as XML and annotation, so that generally it is difficult to test a point cut expression directly. Our framework can describe the test cases for an aspect from the same viewpoint as description of an aspect. It is enabled to define an aspect module based on the specification, and to also define the test program.

5.2 Combination with Other Testing Frameworks

The testing framework JUnit and mock object [11] can be used with our tool at the unit testing for aspect modules. The typical usage of our tool is as follows. Test cases are described using the **TestCase** class offered by JUnit as shown in the example of this paper. The code to test an aspect module is defined by such a class as the **MoveTrackingTester** class, which is generated by our tool.

Moreover, in case the advice body definition is tested, we can use mock object within the advice definition. For example, we assume that the advice of **MoveTracking** uses the **Point** class. If the advice definition uses the mock object instead of actual **Point** object, the unit testing becomes independent of the implementation of the **Point** class.

Since our unit testing framework is defined by pure Java classes, it is possible to combine it with such existing frameworks.

6. CONCLUSION

Some of the problems that Adrian Colyer pointed out can be solved by our proposed framework, which tests an aspect isolated from the rest of the program. The framework enables to test pointcut expressions and body of advice declarations in aspects without weaving. This also guarantees non-interference from other aspects when a program has more than one aspects without requiring recompilation of the whole system.

In this paper, test subjects are limited to the advice body and pointcut expressions. Since an aspect in AspectJ has the following elements, it needs to extend the framework to support all of them:

- Kind of advice
- Body of advice
- Pointcut expression
- Inter-type declaration

It is also our future work to evaluate the effectiveness of our unit testing framework with practical application programs.

7. ACKNOWLEDGMENTS

We would like to thank all the members of Masuhara's and Tamai's research groups at University of Tokyo for their comments on this work. We also would like to thank the SIGSE members of Information Processing Society of Japan for their comments.

8. REFERENCES

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[2] J. Zhao, *Data-Flow-Based Unit Testing of Aspect-Oriented Programs*, Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC' 2003), Dallas, Texas, USA, November 3-6, 2003.

[3] aspectj-users: https://dev.eclipse.org/mailman/listinfo/aspectj-users

[4] Production Aspects: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/starting-production.html#d0e67

[5] Glenford J.Myers, *The Art of Software Testing*, John Wiley and Sons, March 1979, 177p, ISBN 0-471-04328-1.

[6] JUnit, Testing Resources for Extreme Programming: http://junit.org

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold, *An Overview of AspectJ*, Proceedings of the 15th European Conference on Object-Oriented Programming, p.327-353, June 18-22, 2001.

[8] JBoss AOP: http://www.jboss.org

[9] AspectWerkz: http://aspectwerkz.codehaus.org

[10] AOP Alliance: http://aopalliance.sourceforge.net

[11] Tim Mackinnon, Steve Freeman, Philip Craig, *Endo-Testing: Unit Testing with Mock Objects*, eXtreme Programming and Flexible Processes in Software Engineering - XP2000, Cagliari, Sardinia, Italy, June 21-23, 2000.