# Test-based Pointcuts: A Robust Pointcut Mechanism Based on Unit Test Cases for Software Evolution

Kouhei Sakurai
Graduate School of Arts and Sciences,
University of Tokyo
sakurai@graco.c.u-tokyo.ac.jp

Hidehiko Masuhara
Graduate School of Arts and Sciences,
University of Tokyo
masuhara@acm.org

## ABSTRACT

This paper proposes *test-based pointcuts*, a new aspect-oriented programming language construct that uses unit test cases as interface of crosscutting concerns. A test-based pointcut primarily specifies a set of test cases associated to a program. At execution time, it matches the join points that have the same execution history to the one of the specified test cases. The test-based approach improves pointcut definitions in two respects. First, test-based pointcuts are less fragile with respect to program changes because rather than directly relying on type and operation names in a program, they indirectly specify join points through unit test cases, which are easier to be kept up-to-date. Second, test-based pointcuts can discriminate execution histories without requiring to specify detailed execution steps, as they use test cases as abstractions of execution histories. With the abstractions, the second respect contributes to the first respect. We designed and implemented the test-based pointcuts as an extension to AspectJ, and confirmed, through an case study, test-based pointcuts are more robust against evolution when used for a practical application program.

## Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.3.3 [**Programming Languages**]: Language Constructs and Features

## General Terms

Languages, Design

## Keywords

Test-based Pointcuts, Fragile Pointcut Problem, Aspect-oriented programming language, Unit Test Cases

## 1. INTRODUCTION

Current aspect-oriented programming (AOP) languages, such as AspectJ[7], modularize crosscutting concerns using the pointcut-and-advice mechanism. Pointcuts match dynamic join points, which are well-defined points in the execution of a target program[1] (e.g., method calls). The advise consists of statements that run when the join points match an associated pointcut.

This paper proposes a novel pointcut mechanism called *test-based pointcuts*, which can make aspects more robust and more abstract by addressing two problems associated with pointcuts in current AOP languages: the *fragile pointcut problem*[6, 10], and the *execution history problem.* The fragile pointcut problem is that, when a developer changes a program with aspects, the join points actually matching a pointcut become different from the ones that the developer originally intended. The execution history problem describe the difficulty of defining pointcuts that match join points depending on the computation history so far. Such definitions cause the fragile pointcut problem by manual history management, even if a developer can define those pointcuts.

Instead of relying on type, method, and field names in the target program, test-based pointcuts specify variable names in unit test cases associated to the target program. At runtime, test-based pointcuts match join points through selected unit test cases *indirectly*; they match join points that have the same execution history as one of selected cases. For example, we want to advice a login method for the authentication failure case. With test-based pointcuts, we can define a pointcut that specifies the variable in test cases that contains an invalid password. Then the pointcut matches (the end of) calls to the login method that have the same execution histories as the test cases that uses the invalid password. The pointcut is faithful even when the target program is changed as long as the test cases are properly maintained along with the changes. It is also an easier way to distinguish execution histories as it uses test cases as abstractions of execution histories.

We designed test-based pointcuts as an extension to the AspectJ language, and implemented a prototype system on top of the Aspect Bench Compiler[4]. As a framework to describe unit test cases, we chose JUnit[2] with additional restrictions.

The rest of the paper is organized as follows: Section 2 presents an example application program that has the fragile pointcut and execution history problems. Section 3 introduces test-based pointcuts. Section 4 presents evaluation of test-based pointcuts. Section 5 discusses related work.

[1]A target program is the program whose execution can be affected by aspects. Note that in most AOP languages target programs can include aspects as well.

```
class FtpConnection {
 public boolean login(String n, String p) {
   send("USER "+n); if(!checkUser()) return false;
   send("PASS "+p); if(!checkPass()) return false;
   return true;
 }
 private void send(String data) {...}
 private boolean checkUser() {...}
 private boolean checkPass() {...}
 ... //other methods such as download follow here
}
```

**Figure 1: `FtpConnection` class**

```
 1  aspect ConnectionUpdate percflow(login()) {
 2    boolean passSent = false;
 3    pointcut login():  call(* login(..));
 4    before():  call(boolean checkPass()) &&
 5               cflow(login()) {passSent = true;}
 6    after() returning(boolean r):  login() {
 7     if     (!r && !passSent) l.loginFailUser();
 8     else if (!r && passSent) l.loginFailPass();
 9     else                     l.loginOk();
10    }
11    ...//methods for listener (l) management and
12       //advice for other notifications follow here
13  }
```

**Figure 2: `ConnectionUpdate` apsect**

Section 6 concludes the paper.

## 2. MOTIVATING EXAMPLE

This section, demonstrates the fragile pointcut problem and the execution history problem of an FTP client program written in AspectJ. The problems arise in the class that establishes network connection with an FTP server.

### 2.1 An FTP Client Written in Java

`FtpConnection` is a Java class that represents connections established by the FTP protocol. The `login` method shown in Figure 1 realizes a login operation to an FTP server. The `login` method takes a user name and a password as its parameters, and returns the result of the login operation as a boolean value. It calls three private methods, namely `send`, `checkUser` and `checkPass`. It first sends a user name to a connected server by calling the `send` method. It then confirms if the user name is accepted by the server by calling the `checkUser` method. If so, it sends a password. Otherwise it immediately returns `false` to indicate failure of the login operation. Finally, it checks acceptance of the password by calling the `checkPass` method. If the password is accepted, it returns `true` to indicate success of the login operation. Otherwise it returns `false`.

### 2.2 Notification Aspect in Pure AspectJ

We consider an aspect that implements a notification concern of the FTP client, whose implementation in AspectJ is shown in Figure 2. The concern notifies the user of the results of network operations such as login and download.

The main advice (lines 6-10) in the `ConnectionUpdate` aspect calls one of the three notification methods at the end of `login` method-call join point. It determines a method to call by the return value from the `login` method and a `passSent` flag that indicates whether a password is sent to the server. An auxiliary advice declaration at lines 4-5 runs before a call to the `checkPass` method from the `login` method and sets the flag. Because the `login` method calls the `checkPass` method only when the server accepts the given user name, the `passSent` flag denotes how far the `login` method proceeded inside.

### 2.3 Problems in Pure AspectJ Aspect

There are two problems associated with pointcuts in current AOP languages: the *fragile pointcut problem*[6, 10], and the *execution history problem*.

The fragile pointcut problem is the situation that, when a developer changes a target program without knowing the aspects, the aspects accidentally match the join points that are different from the ones initially intended. For example, consider that the name of the `login` method in the `FtpConnection` class is changed to something like `doLogin`. Then the pointcut `call(* login(..))` at line 3 in Figure 2 no longer matches intended join points.

The execution history problem is the difficulty of defining an advice declaration that runs at a join point depending on the execution history of the join point. The `ConnectionUpdate` aspect in the above example distinguishes execution histories by explicitly managing the flag `passSent` with the help of auxiliary advice. Such a complicated aspect is not easy to define, understand and maintain.

Moreover, the execution history problem often causes the fragile pointcut problem because the means of recognizing execution history has to depend on the target program implementation. In the `ConnectionUpdate` example, the pointcut of the auxiliary advice depends on the names of internal methods.

## 3. TEST-BASED POINTCUTS

To address the above mentioned problems, we propose a new mechanism called *test-based pointcuts* as an extension to AspectJ.

### 3.1 Overview

Figure 3 illustrates how test-based pointcuts use unit test cases as interfaces of crosscutting concerns.

There are three key language elements that enable the test-based pointcuts:

**Test methods.** Test methods, showing on the left-hand side in the figure, verify the target program meets the specifications by actually running each method in the target program with typical parameter sets, and by comparing return values with expected values. We use an extended JUnit framework for describing test methods. For example, `testLoginFailPass` runs the `login` method with a valid user name and invalid password, and confirms that the login operation is actually failed.

**Fixtures.** A fixture class, shown below the test methods in the figure, holds a set of variables that store test parameters and expected results. We require all the test methods to be written in such a way that they always access test parameters and expected results through

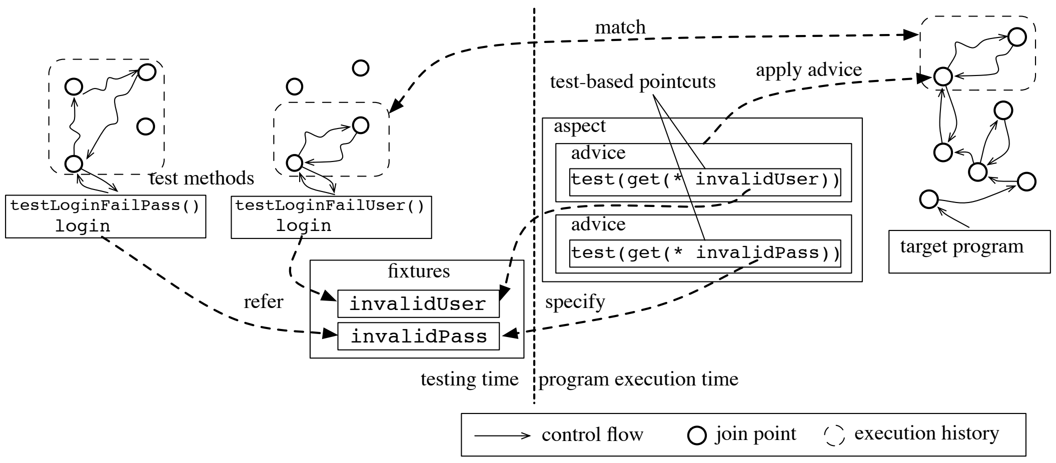

**Figure 3: An overview of test-based pointcuts**

fixture variables. For example, `invalidPass` is a variable for storing an invalid password.

**Test-based pointcuts.** Test-based pointcuts, which appear in the aspect in the figure, can be used as a pointcut primitive in AspectJ. A test-based pointcut is written like `test(`*p*`)` where *p* is a sub-pointcut description that specifies the test methods. For example, `test(get(* Fixtures.invalidPass))` specifies `testLoginFailPass`.

Execution of a program with test-based pointcuts is divided into the following two stages:

1. At *testing time* (the left-half of the figure), all the test methods are executed. The system records an execution history (shown as a dashed rounded box in the figure), which is a sequence of dynamic join points, and a set of referenced fixture variables for each test method.

2. At *program execution time* (the right-half of the figure), when the target program reaches a join point, the history of the current execution is compared against the recorded histories of test methods. When there is a test method that has the same execution history and it references all fixture variables specified in a test-based pointcut, the join point matches the test-based pointcut and hence the advice body runs.

## 3.2 Notification Aspect with Test-Based Pointcuts

Figure 4 shows the redefinition of `ConnectionUpdate` aspect with test-based pointcuts. Compared to Figure 2, it is more straightforward because each advice body merely calls one notification method thanks to each pointcut matching only one use case of the login operation. Note that the `loginOk` pointcut combines two `test` pointcuts in order to match login operations that provide valid user name and password.

Figure 5 shows a test case class for `FtpConnection` in the JUnit framework. It consists of test methods that correspond to unit test cases for the `login` method. We introduce the `Phase` class so that the system can distinguish the join points of actual test execution from the ones for setting up

```
aspect ConnectionUpdate {
 pointcut loginFailUser():
             test(get(* Fixtures.invalidUser));
 pointcut loginFailPass():
             test(get(* Fixtures.invalidPass));
 pointcut loginOk():
             test(get(* Fixtures.validUser)) &&
             test(get(* Fixtures.validPass));
 after(): loginFailUser() { l.loginFailUser(); }
 after(): loginFailPass() { l.loginFailPass(); }
 after(): loginOk()       { l.loginOk(); }
    ...//methods for listener (l) management and
       //advice for other notifications follow here
}
```

**Figure 4: The test-based version of `ConnectionUpdate` aspect**

parameters and for asserting results. To do so, we require every test method to call `Phase.test()` and `Phase.cond()` before and after actual test execution respectively. The `Fixtures` class is defined as a set of fixture variables that are commonly used to store the values for testing in test cases.

## 3.3 Advantages

We claim that the aspects defined with test-based pointcuts are less fragile and easier to handle the execution history problems due to the fact that test cases have the following properties. Generally, unit test cases are:

- **Up-to-date**: when a developer changes a target program, he or she (or another developer) also changes all the test cases associated to the program so as to pass. Therefore, test-based pointcuts automatically reflect the changes to the target program as long as the associated tests are maintained.

- **Thorough**: unit test cases cover most use-cases of most methods in the target program. Therefore, it is easy to identify a test case that corresponds to a specific concern, even if it depends on execution history.

```
public class TestFTPConnection extends TestCase {
 Fixtures f = new Fixtures();
 ...//setUp and tearDown are omitted.
 public void testLoginFailUser() {
  f.invalidUser = "unknown"; f.validPass = "mypass";
  Phase.test();
  boolean r = con.login(f.invalidUser, f.validPass);
  Phase.cond();
  assertFalse(r);
 }
 public void testLoginFailPass() {
  f.validUser = "john";      f.invalidPass = "?";
  Phase.test();
  boolean r = con.login(f.validUser, f.invalidPass);
  Phase.cond();
  assertFalse(r);
 }
...//other test methods follow here.
}
```

**Figure 5: The `TestFTPConnection` test case class revised for test-based pointcuts**

## 3.4 Implementation

We implemented a prototype compiler that supports test-based pointcuts by extending the Aspect-Bench Compiler (abc)[4]. The size of the extension is approximately 3800 lines of code.

Due to space limitation, we only report an outline of the compiler implementation, which consists of the following two stages:

1. The first stage runs each test method, and records its execution history.

2. The second stage weaves aspects into target program. For an advice declaration with `test` pointcuts, it inserts code fragments at shadows of the test execution histories, and inserts a guarded advice invocation into the test target method. The former code fragments set the flags when the control passes the shadow. The guard can then test if the execution has the same history as one of the test methods by looking into the flags.

## 4. CASE STUDY ON EVOLVABILITY

In order to evaluate how test-based pointcuts make aspects more robust against software evolution, we compared two AOP implementations of the j-ftp network browser[1] over three versions in the repository. The original implementations are open-source, and written in Java. In the evaluation, 1) for each version, we modularized a notification concern that reports update of remote directories into an aspect by using pure AspectJ and AspectJ with test-based pointcuts. 2) We applied aspects of a previous version to the classes in the later version and compared the methods to be advised by the aspects with the original implementation. We chose versions 1.07, 1.15 and 1.48.

Figure 6 shows how the notification concern scatters in the three versions.

Table 1 summarizes how the notification concern scatters in the original implementations (**OR**), and in the classes with aspects of the previous version (**AJ** and **TB**). **AJ** and **TB** are the aspects in pure AspectJ and the aspects using test-based pointcuts, respectively. The marks (X) denote that the method is advised. The hyphen (-) are misses; a method is not advised by the aspect in the previous version, while the method in the original implementation notifies. The marks with underlined ($\underline{\text{X}}$) are accidentally captured against the original implementation. For instance, the `chdir` method of the `FilesystemConnection` class in the version 1.15 the original implementation notifies, but the pure AspectJ aspect version 1.07 does not capture the method because the pointcut specifies the `chdir` method in only the `FtpConnection` class.

If we closely examined the definitions, the changes can be classified into the following types:

**Concern Change** : The developer decided to notify at a different join point from the previous version. In such a case, we had to modify the aspect definitions by all means. The most misses and accidental captures fall into this category in fact.

**Birth of a Sibling Class** : The developer defined a new sibling class of an existing class. `FilesystemConnection` in version 1.15 is the case. In such a case, oftenly, the same set of the methods should be advised as the methods advised in the existing class. The test-based pointcuts automatically captured those methods thanks to the properly defined test methods, while the pure AspectJ pointcuts could not work because they used specific class names in the earlier versions.

**Signature Change** : The developer changed a name or the number of arguments of a method. `FtpConnection .upload(f,n,in)` in version 1.48 is the new signature. In such a case, the advice that are applicable to the method should also be applied to the method with the new signature. The test-based pointcuts automatically captured the method with the new signature because the test methods were modified to reflect the changes. The pure AspectJ pointcuts could not work. Moreover, since the developer decided to leave the method with the original signature for some reasons, the compiler could not detect the unused pointcut in this case.

Interestingly, `FilesystemConnection.upload(file)` in the version 1.15 of the original implementation does not notify even though the sibling method does. The test-based pointcut captures the method because the test for the upload method uses the same fixture variable. We consider this is rather a concern change.

To summarize, we observed that the test-based pointcuts work well for evolution that can be classified as birth of sibling class and signature change.

## 5. RELATED WORK

There have been studies on defining interface between aspects and target programs, including Open Modules[3], Aspect-Aware Interfaces[8], Annotation style development (as discussed in the paper by Kiczales and Mezini[9]) and XPI[5]. The techniques proposed in those studies mainly prohibit changes of target programs when the change causes pointcut mismatch. Therefore, when the programmer has to change the target program against the interface, it is the

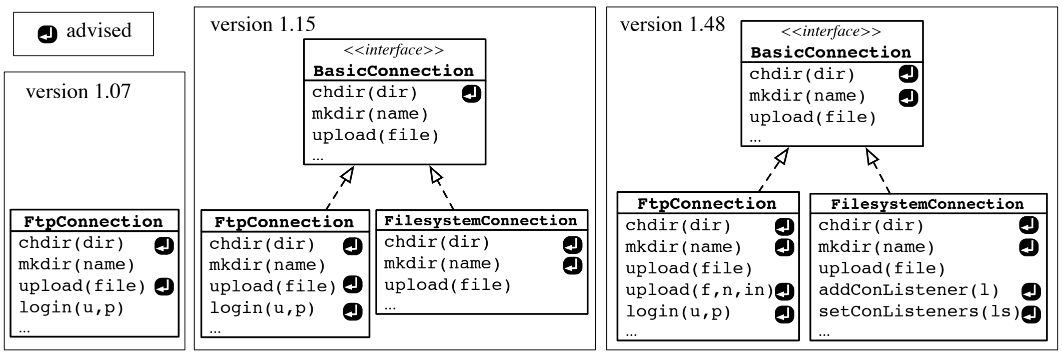

**Figure 6: Evolution of the connection classes and the notification concerns**

| | | 1.07 | 1.15 | | | 1.48 | | |
|---|---|---|---|---|---|---|---|---|
| class | method | OR | OR | AJ | TB | OR | AJ | TB |
| FtpConnection | chdir(dir) | X | X | X | X | X | X | X |
| FilesystemConnection | chdir(dir) | | X | - | X | X | X | X |
| FtpConnection | mkdir(name) | | | | | X | - | - |
| FilesystemConnection | mkdir(name) | | X | - | - | X | X | X |
| FtpConnection | upload(file) | X | X | X | X | | X | |
| FtpConnection | upload(f,n,in) | | | | | X | - | X |
| FilesystemConnection | upload(file) | | | | X | | | |
| FtpConnection | login(u,p) | | X | - | - | X | X | X |
| FilesystemConnection | addConListener(l) | | | | | X | - | - |
| FilesystemConnection | setConListeners(ls) | | | | | X | - | - |

**Table 1: The result of previous versions of aspect application to later versions**

programmer's responsibility to fix the interface and aspect definitions. Test-based pointcuts use test methods as interface between aspects and target programs.

Model-based pointcuts[6] propose more abstract pointcuts than the ones available in existing AOP languages to address the fragile pointcut problem. The view-based pointcuts, which are an instantiation of the model-based pointcuts, use *views* as a classification that reflects the developer's intention. From this perspective, the test-based pointcuts can be seen as another instantiation of the model-based pointcuts that use test methods instead of views.

## 6. CONCLUSION

We proposed test-based pointcuts, which use unit test cases as an interface of crosscutting concerns. A test-based pointcut matches join points in the execution of a target program that (potentially) have the same execution history as one of the unit test cases specified by the pointcut.

Test-based pointcuts address the fragile pointcut problem and execution history problem by indirectly matching join points through unit test cases. In other words, test-based pointcuts replace the fragile pointcut problem with maintainace of unit test cases whose cost should anyhow be paid with practical software development. Test-based pointcuts can match join points based on execution histories without relying on detailed execution steps by using unit test cases as abstractions of execution histories.

Although at a preliminary stage, we implemented a compiler, an AspectJ language extended with test-based pointcuts, and showed that test-based pointcuts solve the problems in a practical application when compared the necessary changes to pointcut definitions over three versions.

## 7. ACKNOWLEDGMENTS

This work is supported by the JSPS Research Fellowships for Young Scientists.

We would link to thank the anonymous reviewers of the workshop for their valuable comments. We also thank the members of the PPP research group and the TM-seminar at University of Tokyo for valuable comments.

## 8. REFERENCES

[1] j-ftp. `http://j-ftp.sourceforge.net/`.

[2] JUnit. `http://www.junit.org/`.

[3] J. Aldrich. Open Modules: Modular Reasoning about Advice. In *ECOOP '05*, pages 144–168, July 2005.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05*, pages 87–98, 2005.

[5] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[6] A. Kellens, K. Mems, J. Brichau, and K. Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *ECOOP '06*, pages 501–525, 2006.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01*, pages 327–353, 2001.

[8] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05*, pages 49–58, 2005.

[9] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *ECOOP '05*, pages 195–213, July 2005.

[10] M. Stoerzer and J. Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *ICSM '05*, pages 653–656, 2005.