On Identifying Bug Patterns in Aspect-Oriented Programs

Sai Zhang
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
saizhang@sjtu.edu.cn

Jianjun Zhao
Department of Computer Science
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
zhao-jj@cs.sjtu.edu.cn

Abstract

Bug patterns are erroneous code idioms or bad coding practices that have been proved fail time and time again. They mainly arise from the misunderstanding of language features, the use of erroneous design patterns or simple mistakes sharing the common behaviors. Aspect-oriented programming (AOP) is a new technique to separate the crosscutting concerns for improving modularity in software design and implementation. However, there is no effective debugging technique for aspect-oriented programs until now and none of the prior researches focused on the identification of bug patterns in aspect-oriented programs. In this paper, we present six bug patterns in AspectJ programming language and show the corresponding example for each bug pattern to help to illustrate the symptoms of these patterns. We take this as the first step to provide an underlying basis on testing and debugging of AspectJ programs.

1 Introduction

In modern software development, testing and debugging software is done as part of an integrated method of software development. An appropriate method of bug finding can easily help developers locate and remove bugs. A software bug is regarded as the abnormal program behaviors which deviates from its specification [4], including poor performance when a threshold level of performance is included as part of specification. Bug patterns, which is related to anti-patterns, are recurring relationships between potential bugs and explicit errors in a program; they are common coding practices which shares the similar symptoms and have been proven to fail time and time again. Those bug patterns are raised from the misunderstanding of language features, the misuse of positive design patterns or simple mistakes having the common behaviors. Such bug patterns are an essential complement to the traditional design pattern [9], just as a good programmer needs to know design patterns which can be applied in various context and improve the software quality, also to be a good software developer or problem solver the knowledge of common causes of faults is a need in order to know how to fix the software bugs.

In the previous research of bug patterns, most of the work has been focused on the Java programming language. Allen [4] summarized more than 14 categories of bug patterns in Java and later more pattern classifications was identified by the FindBugs research group at the University of Maryland. Farchi [8] also shows us some concurrent bug patterns and how to test them.

Aspect-oriented programming (AOP) [17] is a new technique to separate the crosscutting concerns. It can clearly modularize the crosscutting structure of concerns such as exception handling, synchronization, performance optimizations and resource sharing, which are usually difficult to express clearly in source code using existing programming techniques. However, the current research so far in AOP is focused on problem analysis, implementation techniques and testing approaches. Even though the importance of program debugging is know, it has received little attention in aspect-oriented paradigm. The new factor of complexity introduced in AOP such as behavior modification by the injected advice makes it difficult to find the defects or bugs in the source code. Until now there is no effective approach to find or locate bugs in aspect-oriented programs and none of the previous work was focused on the bug pattern identification in AOP language. The debugging issues still remain a big problem for aspect-oriented programs.

We may not know what types of bugs are unique or common happened to aspect-oriented programs without a proper bug pattern classification, and this poses several restrictions on the research and development of programs in the language:

- Developers do not know what kind of bugs are most likely to happen in a program, and therefore do not know how to prevent them. In other words, programmer would be lack of a fundamental knowledge on how to write bug-free code.

- Testers do not have suffice knowledge of how to write adequacy test cases that can effectively cover most of the common potential errors. Only when having an idea of how the common bugs happened in programs, can tester set up criteria for better addressing the specific bugs.

- Software maintenance staffs do not know which features of the language are more likely to result in the faulty code; so they cannot have a clear view on the current system when doing the maintenance tasks.

The bug patterns may help to solve these problems. To identify such patterns in AspectJ explicitly, we can leverage the experience of many programmer to improve their productivity in bug finding and eliminate the cost of software maintenance. The bug pattern identification can also help language designers or tool developers to develop the corresponding bug finding techniques or bug detectors, which for example could be applied to automatically locate syntactic bugs in the source code by program analysis.

Furthermore, the bug patterns provides a basis for further research on debugging AspectJ programs. It not only gives insight into possible consequences of different types of bugs but summarize the common behaviors among the similar ones. It can be used to recognize faults that have been already existed and prevent the potential bugs. The bug patterns taxonomy for AspectJ language could even be seen as a starting point for creating the general bug patterns for aspect-oriented languages.

In this paper, we choose the most well supported and widely used AOP language AspectJ as our target language and identify the most common and typical bug patterns in it. We also show the corresponding example for each bug pattern to help to illustrate the symptoms of these patterns. To the best of our knowledge, the work described in this paper is the first attempt to identify and summarize the bug patterns existing in AspectJ programs systematically.

The rest of the paper is organized as follows. Section 2 briefly introduces the background knowledge of aspect-oriented programming in AspectJ and the new error-prone features introduced by aspect-oriented programs. Section 3 describes the identified bug patterns in AspectJ in detail. Related works is discussed in section 4 and concluding remarks are given in Section 5.

2 Background

We next briefly introduce background information on programming in AspectJ and the error-prone features in the AspectJ programs.

2.1 AspectJ

AspectJ [17], an seamless extension of Java with aspects, is one of the most popular and best supported programming languages in AOP community. AspectJ provides three main programming constructs to modularize crosscutting concerns: pointcuts, advice and inter-type declarations. Pointcuts define the interception point among the normal execution flow at the specified joint points, and the advice code can be executed either before, after or around (instead of) the intercepted join points. The intertype declaration mechanism offered by AspectJ is used to introduce new methods or fields to the existing classes or make them implement interfaces or extend specialized superclasses previously unrelated to them. Whereas pointcuts and advice alter the dynamic behavior of the program execution, intertype declarations operate statically on the class member and structure by modifying the source code.

Figure 1 shows an AspectJ program that uses the features of pointcut, advice and intertype declaration. This sample code actually includes some implicit common bug patterns such as the "*scope of advice*" and the "*infinite loop*" which we will explain later. The program contains one base class `BankCustomer` and one aspect `CheckedCustomer`. The base class has only one field named `credit`, which represents the customer's credit in a bank. The aspect declares that the interface `Serializable` is implemented by class `BankCustomer`, which is achieved by means of the AspectJ construct `declare parents`. Moreover, the aspect define two pointcuts `creditResetting` and `creditLogging` to intercept the join points of the `resetCredit(int credit)` and `printCredit()` method calls respectively. The advice to be executed before the pointcut `resetCredit` is declared by means of the AspectJ keyword `before`. Its formal parameters match those exposed in the corresponding pointcut, the keyword `target` is used to expose the object on which the invocation is performed (in this case, the current `BankCustomer`) and the `args` keyword is used to pass parameters between the base code and the aspect code. The exposed context is then available within the advices associated with the pointcut. The code in this advice checks the setting credit in the sense that if the credit is less than 300, it will create a new `BankCustomer` object instead of the former one with 300 credit to adjust customer's credit. Another advice declared in the `CheckedCustomer` aspect is executed before `printCredit()` method, it is used to log the current credit of the target `BankCustomer` object.

2.2 Error-Prone Features in AspectJ programs

The basic idea of AOP is to encapsulate the crosscutting concerns which scatter previously on many modules

```
public class BankCustomer {
  private int credit;

  public void resetCredit(int credit) {
    this.credit = credit;
  }

  public int printCredit() {
    return this.credit;
  }
}

aspect CheckedCustomer {
  declare parents:
    BankCustomer implements Serializable;

  pointcut creditResetting
    (BankCustomer cust,  int credit):
      call(BankCustomer.resetCredit(..))
        && target(cust) && args(credit);

  pointcut creditLogging(BankCustomer cust):
    call(BankCustomer.printCredit()) && target(cust);

  before(BankCustomer cust, int credit):
    creditResetting(cust, credit) {
    if (credit < 300 ) {
      cust = new BankCustomer(300);
    }
  }

  before(BankCustomer cust):
    creditLogging(cust) {
    log("Credit: "+cust.printCredit());
  }
}
```

Figure 1. A sample AspectJ program

of a specific software system in a new kind of module called aspect. Although this separation facilitates to handle situations like synchronization policies, resource sharing and performance optimizations, it also introduces new complexities or more error-prone features to the traditional paradigm:

- The join point model in AspectJ is defined in a lexical-level, and the selection relies on naming conventions. When using wildcards specified in a pointcut to match join points, it is easy for programmers to pick up an incorrect join point. Also during the software evolution, a pointcut definition may become obsolete resulting from the renaming, moving or deleting classes, methods and fields of base code changes.
- More than one advice can be activated at the same join point, which causes the sequence of those advices would affect the program execution result. Moreover, the aspect's behavior may be altered by another aspect as long as it is accidentally executed inside another advice execution.
- The inter-type declaration mechanism of AspectJ can introduce a new class member to override an existing one or alter the original class hierarchy, which changes the behaviors of dynamic dispatch implicitly or causes a conflict.

In this paper, we focus on the new features of AspectJ particularly those error-prone language complexities to explain the bug patterns in AspectJ. As AspectJ is an extension of the Java programming language, all the existing bug patterns in Java are also applicable for AspectJ programs. The bug patterns proposed below are the patterns for AOP constructs, which involve AOP program elements explicitly.

3 Bug Patterns in AspectJ

We next present six bug patterns in AspectJ as examples and explain them in detail. When introducing each bug pattern, we also show an example that contains this specific pattern. Since most bug patterns have a number of representation variant and alternatives, we choose the one that appears to be the most generally applicable. A catalog of all bug patterns we identified in AspectJ will be appended in the end of this section.

AspectJ is a seamless aspect-oriented extension to Java, which means that programming in AspectJ is effectively programming in Java plus aspects. Therefore, all the bug patterns identified in [2,4,11] are also applicable in AspectJ programs. By focusing on the language features of AspectJ, the bug patterns in this section can be classified into three categories as showed below:

- **Visibility of Join Point.** In AspectJ programs, the definition of an aspect contains the advice code and a reference to a so called pointcut. The pointcut represents the join points which this aspect will intercept during execution. Since the definition of the pointcut can explicitly name the methods which designate the join points, alternatively, the pointcut can be defined by lexical matching using wildcards. Those features enhance the difficult for a developer without proper tools to know exactly which join point the aspect is applied and also cause the visibility of the joint point become even more loose. This is the category we called **Visibility of Join Point**, which is the root of common bug patterns such as *"The Infinite Loop"*, *"The Unmatched Join Point"* etc.
- **Impact of Advice.** Since AspectJ make explicitly where and how a concern is addressed, in the form of join points and advice. In static program text, the specification for concern handling is concentrated on one place, the aspect. However, dynamic execution of advice happens around all join points in classes as defined by pointcuts. One aspect can be applied at several join points, and multiple advice can be invoked simultaneous at one join point definition. As a defined aspect can change the program behavior dynamically, the gap between static text and dynamic execution imposes a new potential bug.

- **Modification of Source Code.** Intertype declaration is an AspectJ language construct to add new members to existing classes or interfaces. Adding members to class can result in changes of dynamic lookup if the introduced method redefines a method of a superclass, called dynamic interference in [16]. Moreover, the program semantics such as class hierarchy may be changed without modifying any base class directly. These side-effects are hard to find for programmers and become one of the most error-prone language features.

3.1 The Infinite Loop

The bug pattern "*The Infinite Loop*" is due to the accidental matching of unexpected join point, because the selection of join points in the pointcut relies on lexical-level matching. This bug pattern will be classified into the category of **Visibility of Join Point**.

Programmers sometimes carelessly define incorrect pointcuts, which cause a piece of advice may be invoked during another advice execution. In such a case, an aspect may change the behavior of another aspect. When the incorrect pointcuts are unexpectedly defined and matched, if two advices are activated during an execution of each other advice accidentally, these cause the infinite loop [12].

In an AspectJ program, accidental advice execution is the root of this "*Infinite Loop*" bug pattern, a typical syntactic pattern indicating that an instance of the fault type could be presented. We take the code snippet from [10] as an example in Figure 2.

```
pointcut Location(DVD dvd) :
  call(public String DVD.getLocation())
            && target(dvd);

before(DVD dvd):Location(dvd) {
  System.out.println(dvd.getLocation());
}
```

Figure 2. The infinite loop

This program intends to log the location information of the `DVD` object before it is used, so it intercepts the method call of `dvd.getLocation()` using aspect. The careless programmer just forgets to exclude the join point inside the aspect itself, and the advice is executed again in this aspect code, which causes an infinite loop. Actually this program will never reach the `println()` call, since it will abort due to having no stack trace.

The silence abort of the program is caused by multiple `StackOverflowExceptions`. First the infinite loop in the advice code body generates another call matched join point, which is turned out to be handled by itself. The infinite loop, which the current JVMs can not handle gracefully, leads to the completely silent abort. The overall problem is advice applying within its own body.

This kind of bug is insidious because it looks correct, and is hard to find by programmer. Detecting instances of this bug pattern simply involves examining the call graph of AspectJ program to check if there is any cycle existed. The simplest approach to cure and prevent this bug pattern from happening is to append the `!within(..)` to the pointcut declaration to exclude those join point inside the aspect code, such as showed in Figure 3.

```
pointcut Location(DVD dvd) :
  call(public String DVD.getLocation())
         && target(dvd) && !within(AspectName);
```

Figure 3. The modified pointcut of the infinite loop

3.2 The Scope of Advice

Advice is a method-like mechanism which is consisted of code that is executed `before`, `after` or `around` a pointcut. Like method parameters, advice parameters are local to the advice. In other words, if you reassign a parameter, it will have no effect outside the advice. You will not change the state of the parameters and do not have changes reflected outside the advice when misunderstanding the scope of advice. That causes the bug pattern of "*The Scope of Advice*", which should be fall into the category of **Impact of Advice**.

To show this bug pattern, consider the piece of code in Figure 4. There is an unscrupulous programmer attempts to reset the credit of bank customer:

```
before(BankCustomer cust, int credit):
  creditResetting(cust, credit) {
  if (credit < 300 ) {
    cust = new BankCustomer(300);
  }
}
```

Figure 4. The example of advice scope

The intention of this advice is to check the credit parameter of a customer when resetting his/her credit value. If the credit is less than 300, this advice adjusts it to 300. However, the effect of `cust=` lasts only until the end of the advice body, in other words, the advice does not affect the actual adjustment.

It is possible to do what this malicious advice attempts, but only with around advice. It is much easier to affect join point context by just changing the state of formal parameters. This approach adds an extra method like `adjustCredit` into the `BankCustomer` base class to set the `BankCustomer`'s state showed in Figure 5.

```
before(BankCustomer cust, int credit):
  creditResetting(cust, credit) {
  if (credit < 300 ) {
    cust.adjustCredit(300);
  }
}
```

Figure 5. Cures for advice scope bug Pattern

3.3 The Multiple Advice Invocation

AspectJ allows declaring more than one advice to intercept one join point, so multiple advices can be invoked at the same join point. The execution sequence of advices may affect the result of program. The missing of advice precedence declaration will cause both programmer and AspectJ compiler confused without any explicit error report, and that would be the root of a potential bug. We consider this kind of bug pattern "*The Multiple Advice Invocation*" into the category **Visibility of Join Point**.

Since AspectJ not only allows the programmer to declare multiple advices to be executed at one join point, also the order of advice execution can be controlled through advice precedence. The most common symptom of this pattern of bug is a program that has more than one advices invoked at one join point or define advice that forms a circular precedence relationship. To present how this can happen, consider the following code examples in Figure 6.

```
class BankCustomer {
  public void resetCredit(int x) { ... }
}

aspect LoggingAspect {
  before():
    call(void BankCustomer.resetCredit(..)){ ... }
}

aspect CreditValidationAspect {
  before(int x): call(void BankCustomer.resetCredit(..))
     && args(x) {
     if (x<0) {
       throw new BadCreditException(x);
     }
  }
}
```

Figure 6. Multiple advice without precedence definition

In Figure 6, the execution sequence of advices may affect the result of a program. When the `CreditValidationAspect` throws an exception, the output depends on whether or not the logging aspect is invoked before the credit validation. To prevent this problem, the precedence of each activated aspects must be defined explicitly clarify program logic. To avoid this kind of bug pattern instances, we can detect the multiple advice invocation information achieved by AspectJ compiler (such as the AJDT [1] in eclipse IDE) and add a proper precedence of aspects.

```
declare precedence: A, B;
declare precedence: B, C;
declare precedence: C, A;
```

Figure 7. The circular precedence relationship among advices

Another variant of this bug pattern is the circular precedence relationship between advices in Figure 6, but the weaver will report these errors in this case. The same sort of circular error can arise within advice defined in a single aspect. Because AspectJ handle precedence on a per-joinpoint basis, if C does not apply to the same join points as both A and B, the circularity problem will never arises. A more complex example was mentioned in [15], the additional problem affecting the application of multiple aspects at one join point has been stated: granularity of aspect order showed in Figure 8. The only sound execution order is obviously: start- openDoor -goInside - goOutside - closeDoor - end, so neither A precede B nor vice versa is sound with necessary advice order. When the semantics is relevant with the aspect order, the detection of contradictory aspect is even difficult. Thus consider the understandability and clarity of the program, the circular relationship order between advices should be checked statically and then be broken off before runtime.

```
class Main {
  public static void main(String[] args) {
    start();
    end();
  }
  void start { }
  void end { }
}

aspect A {
  before: call(Main.start()) {
    openDoor();
  }
  after(): call(Main.end()) {
     closeDoor();
  }
  void openDoor() { }
  void closeDoor() { }
}

aspect B {
  before(): call(Main.start()) {
    goInside();
  }
  after(): call(Main.end()) {
    goOutside();
  }
  void goInside() { }
  void goOutside() { }
}
```

Figure 8. Contradict advice order example

As showed in Figure 6, the execution sequence of advices may affect the program result. Missing the precedence declaration when multiple advices would be invoked at one joint point will cause unexpectedly result. For the contra-

dict problem showed in Figure 8, a poor design or aspect modularization is the root cause, which is hard for AspectJ to solve.

3.4 The Unmatched Join Point

In a pointcut expression, when the wildcard (such as * or ..) is used for matching the intended join point's parameter, and the type declared in the `args()` primitive pointcut does not match with the intended one. That type unmatch will cause an implicit un-match problem. We call this kind of bug pattern "*The Unmatched Join Point*", and its classification category is **Visibility of Join Point**.

As we know, in AspectJ, the `args()` designator has two purposes: providing the arguments sent to the join point as parameters and limiting the matching on a specific call, execution or initialization designator. Even if the join point is matched, the program still can not work expectedly if the parameters passed to the `args` designator and the parameters declared in the pointcut are not identical. This causes *"The Unmatched Join Point"* bug pattern, which is difficult to detect. We take the related code snippet from the *BankCustomer* example to show the typical syntactic format of this bug pattern in Figure 9.

```
class BankCustomer {
  String name;
  public void setName(String name) { ... }
}

pointcut setName(int name):
  call(public void setname(..)) && args(name);
```

Figure 9. The implicit unmatched join point

Although the method name is matched by using the wildcards, the parameter appearing in the `args` primitive is not identical with the original parameter type. This implicit difference will cause this join point unmatched or become isolated.

For detecting and curing this kind of bug pattern, the type of each parameter declared in the pointcut should be checked in order to make sure that it is the same as the target matching one. On the other hand, in order to prevent this bug pattern happen, it is recommended to explicitly designate the parameter type inside the method signature instead of using wildcard.

3.5 Misuse of getTarget()

The bug pattern "*Misuse of getTarget()*" in category **Impact of Advice** is due to the AspectJ reflection mechanism. AspectJ provides special implicit variables to inspect and manipulate join point context. For example, we can access some information available through formal parameters using `getTarget()`, `getThis()` and `getArgs()`. However, for example, the `getTarget()` method of `thisJoinPoint` object will return `NULL` if using it inside a static method, this `NULL` return value involves a degree of obfuscation and may lead to a `NULLPointerException` or casting failure.

There are several ways of this bug pattern, a typical syntactic form is: when using arbitrary wildcards to specify all join points, the execution of static method is accidentally matched. If the reflective access primitive `getTarget()` is used to get the context of join point, the value `NULL` will return. Figure 10 shows a piece of code containing this potential bug.

```
class BaseClass {
  public static void doSomething() { ... }
  ...
}

aspect A {
  pointcut BaseLogging(): execution( public * *(..));
   ...
  before():BaseLogging() {
   BaseClass base =
     (BaseClass)thisJointPoint.getTarget();
 }
}
```

Figure 10. Misuse of getTarget()

The potential `NULL` pointer exception or casting exception will occur during the program execution if `getTarget()` is incorrectly used inside a static method.

To avoid this bug pattern, every pointcut declaration for static method and those pointcuts with wildcards should be checked to see if they have matched static method or not. For the matching one, the applied advices are also needed to be inspected to make sure that no `getTarget()` is used.

3.6 Introduction Interference

The inter-type declaration mechanism of AspectJ allows inserting new members (both methods and fields) into existing classes. Although the AspectJ compiler will report errors of name clashes, the changes in class behavior by introducing new members (such as overriding a class method) in a class hierarchy, called dynamic interference [16], can alter the class runtime behaviors unexpectedly. Particularly when an introduced method override the existing one, that will alter the runtime behavior of the client object. Those common bugs are classified into the bug pattern "*Introduction Interference*" with the category **The Modification of Source Code**.

A typical situation when this bug pattern occurs is: AspectJ introduces a new method into the class hierarchy and overrides the previous class behavior. This described problem is a more subtle case of the fragile base class problem [14] - subclasses change behavior because of changes in the superclass. Consider the example in Figure 11.

```
class A { void m() { ... } }
class B extends A { }
class C extends B {
  void n() { m() }
}

aspect introduceM2B {
    void B.m() { ... }
}
```

Figure 11. Example of Intertype declaration conflicts

Aspect `introduceM2B` introduces a method `m` to class `B`, which results any call from class `C` now results in a call `B.m()`, and not `A.m()`. Neither Java nor AspectJ guarantees this kind of method redefinition. For class `B`, it might work with this change as intended. However, effects of these inter-type declarations modify the behavior of its subclasses implicitly. The similar situation occurs if one external aspect changes the hierarchy of existing classes; the aspect moves `B` to inherit the other classes which happen to have the same method `m()`. Then, the clients functionality is broken.

Changes introduced with AspectJ are not visible directly in the source code of the base system. Aspects are new modularization units usually stored in separate files. The effect of this code can influence semantics of the whole system, and therefore making program comprehension more difficult.

The intertype declaration conflicts with the previous class behavior may cause the following problems as well as tracking down bugs introduced by changing a base class is difficult.

- The actual virtual call execution may be changed, which causes the corresponding behavior of client changed.
- If the hierarchy of classes is changed, any `instanceof` operation is facing the risk of casting failure. More generally, if the type of a class is changed, that will result in a `ClassCastException` for all its subclasses.

Detecting this kind of bug pattern is somehow more difficult because it not only needs to statically analyze the impacted source code to find the altered dynamic dispatch chain but also needs to monitor the dynamic behavior of program execution.

4 Related Work

We discuss related work in the areas of bug patterns in Java and fault models for AspectJ.

4.1 Bug Patterns in Java

The previous research on bug patterns are mainly focused on object-oriented programming language such as Java. Allen [4] summarized more than 14 bug patterns categories, for example, including "*The Null Pointers*", "*Dangling Composite*" and "*The Orphaned Thread*". Following Allen's work, Hovemeyer and Pugh [11] presented a novel syntactic pattern matching approach to detecting the bug patterns in Java and implemented a bug finding tool called FindBugs [2], which uses various type of bug detectors to find bug patterns in Java. Broadly speaking, each of these detectors falls into one or more of the four categories: (1) single-threaded correctness issue, (2) thread/synchronization correctness issue, (3) performance, and (4) security and vulnerability to malicious untrusted code. Including the bug patterns reported by FindBugs, there are over 50 bug patterns in Java that have been identified, which cover from bad practices, performance issue, correctness, and multi-threaded correctness.

Our work is to extend the bug patterns research and identification to the aspect-oriented programming languages using the AspectJ language as an example. The bug patterns presented in this paper are different in nature from the existing bug patterns in Java in the sense that they involve explicitly the aspect-oriented programming language features such as crosscutting concerns, join point, and advice.

4.2 Fault Models for AspectJ

Prehaps the most similar work to ours is the research of program fault models. The term of fault model was first proposed by Binder [6] in order to describe a model that identifies relationships and components of a system under test that are most likely to have faults. Because the numerous possible inputs, paths or states of a system, the testing cost would be tremendous even for the simplest programs. The fault model is designed to find an instance of a particular fault existed or potentially existed in a program, and provides programmers with some guidelines on how to localize the described faults, or to increase the observability of the common bugs.

For aspect-oriented programs, several fault models have been proposed. Alexander et al [3] proposed a model with six faults types and Ceccato et al [7] added three more fault types to that model later. Van Deursen et al [18] also presented an aspect-oriented fault model using AspectJ-like languages; it covered several faults due to inter-type decelerations, faults in pointcuts and faults in advice. The problem of "fragile pointcut" is discussed by Koppen and Storzer [13] ,which can also be regarded as a typical fault type, because using pointcuts based on wildcards and naming conventions can easily lead to surprising matches of

join points during the evolution of software system. In [5], Bakken and Alexander present a fault model for AspectJ pointcut by describing *fault name*, *fault category*, *summary*, *syntactic form* and *semantic impact*. While the fault models are claimed to provide a foundation mainly for software testing, program inspections and evaluation of testing strategies for AspectJ programs, our work for bug patterns in AspectJ is aimed to provide a basis for the AOP debugging, that is, as soon as the programmers find a similar bug symptom as described here in an AspectJ program, they can use the bug patterns to help them quickly identify which language features cause that failure and how to remove them.

5 Concluding Remarks

In this paper, we presented six bug patterns in AspectJ, that is to provide both researchers and programmers a clear view of what kind of bugs may happen in AspectJ programs and how to detect them. The study of bug patterns mainly focus on the aspects of bug pattern symptoms, cause root and cures and preventions. However, the bug patterns we presented here are the first research result of our work, and do not use every possible aspect construct or cover all AspectJ characteristics. New research should cover other remaining aspect constructs, as well as the interactions between them. For future work, we plan to further develop our approach to investigating more bug patterns in AspectJ programs. We will also develop a bug detecting tool based on the identified bug patterns in this paper to support debugging and fault finding in AspectJ programs.

References

[1] Ajdt: AspectJ development tools. `http://www.eclipse.org/ajdt/`.

[2] FindBugs. `http://findbugs.sourceforge.net/`.

[3] R. Alexander, J. Bieman, and A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, March 2004.

[4] E. Allen. *Bug patterns in Java (2nd Edition)*. Apress, 2002.

[5] J. S. Bakken and R. T. Alexander. Towards a fault model for aspectj programs: step 1 – pointcut faults. In *Proc. 2nd workshop on Testing aspect-oriented programs (WTAOP 2006)*, pages 1–6, 2006.

[6] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 2000.

[7] M. Ceccato, P.Tonella, and F. Ricca. Is aop code easier to test than oop code? In *Proc. Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)*, March 2005.

[8] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 17th International Symposium on Parallel and Distributed Processing*, page 286.2, 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns - elements of resuable object-oriented software*. Addison-Wesley, 1994.

[10] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: aspect-oriented programming in Java*. Wiley, 2003.

[11] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[12] T. Ishio, S. Kusumoto, and K. Inoue. Debugging support for aspect-oriented program based on program slicing and call graph. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 178–187, 2004.

[13] C. Koppen and M. Storzer. Pcdiff: attacking the fragile pointcut problem. In *Proc. European Interactive Workshop on Aspects in Software (EIWAS'04)*, September 2004.

[14] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proc. 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998.

[15] K. Ostermann and G. Kniesel. Independent extensibility – an open challenge for AspectJ and Hyper/J, 2000.

[16] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proc. 16th European Conference on Object-Oriented Programming*, pages 562–584. Springer-Verlag, 2002.

[17] T. A. Team. The aspectJ programming guide. Online manual, 2003.

[18] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw, 2005.