XFindBugs: eXtended FindBugs for AspectJ

Haihao Shen, Sai Zhang, Jianjun Zhao, Jianhong Fang, Shiyuan Yao
School of Software, Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{haihaoshen, saizhang, zhao-jj, fjh1226, ysyadam}@sjtu.edu.cn

ABSTRACT

Aspect-oriented software development (AOSD) is gaining popularity with the wider adoption of languages such as AspectJ. However, though the state-of-the-art aspect-oriented programming environment (such as AJDT in the Eclipse IDE) provides powerful capabilities to check the syntactic or grammar errors in AspectJ programs, it fails to detect potential semantic defects in aspect-oriented software systems. In this paper, we present XFindBugs, an eXtended FindBugs for AspectJ, to help programmers find potential bugs in AspectJ applications through static analysis. XFindBugs supports 17 bug patterns to cover common error-prone features in an aspect-oriented system, and integrates the corresponding bug detectors into the FindBugs framework. We evaluate XFindBugs on a number of large-scale open source AspectJ projects (306,800 LOC in total). In our evaluation, XFindBugs confirms 7 reported bugs and finds 257 previously unknown defects. Our experiment also indicates that the bug patterns supported in XFindBugs exist in real-world software systems, even for mature applications by experienced programmers.

1. INTRODUCTION

Static analysis for software defect detection is a promising technique to improve software quality. Because of the sheer complexity of modern programming languages, the potential for misuse of language features, API rules or simply bad programming practice may be enormous. Static analysis techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective. Static analysis tools, such as [8,10,11,19,22], serve an important role in raising the awareness of developers about subtle correctness issues. In addition to finding existing bugs, these tools can also help programmers to prevent future defects. FindBugs [8], one of the most popular static analysis tools, is becoming widely used in Java community. FindBugs implements a set of bug *detectors* for a variety of common *bug patterns* (code idioms that are likely to be errors [25]), and uses them to find a significant number of bugs in real-world applications and libraries [14, 15].

Aspect-Oriented Programming (AOP) [27] has been proposed as a technique for improving separation of concerns in software design and implementation. It is gaining popularity with the wider adoption of languages such as AspectJ. AspectJ is a seamless extension of Java. An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. However, though the state-of-the-art aspect-oriented programming environments (such as AJDT [5] in the eclipse [7] IDE) provide powerful capability to check the syntactic or grammar errors in AspectJ programs, they fail to detect potential semantic defects in software systems. For example, the type checker used in AJDT only checks the syntactic correctness of the program, but fails to identify or even to generate a warning about the *type conflicts* introduced by an aspect. In such cases, once the class containing an introduced field with a conflicting type is instantiated, the whole program will be terminated abruptly.

Although the executable code of an AspectJ program is pure Java bytecode, the existing bug patterns and defect detection tools for Java bytecode might not be applied directly. In addition to the bytecode that corresponds to the source code (e.g., to bodies of advices), the compiled bytecode of an AspectJ program contains extra code inserted by the compiler during the weaving process. After weaving, the source code level *aspect*, *advice* or *intertype declaration* information has been translated into pure Java bytecode instructions, and therefore is no longer preserved. In fact, in our experimental study, none of the bugs found by XFindBugs presented in Section 5, can be detected directly by FindBugs.

In this paper, we present XFindBugs, an eXtended FindBugs for AspectJ. XFindBugs defines a catalog of 17 bug patterns for aspect-oriented features, and implements a set of bug detectors on top of the FindBugs analysis framework. Bug patterns abstract common misunderstandings of language features, API rules and bad programming practice. They help programmers to get a better understanding of how to write bug-free code. We also perform an empirical evaluation of XFindBugs on several AspectJ benchmarks and third-party large-scale applications (like GlassBox [9], AJHotDraw [1], and AJHSQLDB [2]). XFindBugs confirms 7 reported bugs and finds 257 previously unknown defects in these subjects, some of which may even result in a software crash. The experiment also indicates that the bug patterns XFindBugs supports exist in real-world software systems, even in mature AspectJ applications by experienced programmers.

In summary, the main contributions of this paper are: **(1)** a systematic catalog of bug patterns for AspectJ programs, **(2)** design and implementation of XFindBugs, a static defect detection tool for AspectJ software, and **(3)** an empirical evaluation of XFindBugs on over 300KLOC, which evidences the practical issues.

The rest of the paper is organized as follows. We start to briefly

PASTE'08, November 9-10, Atlanta, Georgia USA.
Copyright 2008 ACM 978-1-60558-382-2/08/11 ...$5.00.

introduce the background of FindBugs and discuss the error-prone features of AspectJ programs in Section 2. Section 3 presents the a catalog of bug patterns for AspectJ in detail. Section 4 summarizes the implementation issues on XFindBugs. Section 5 reports an empirical evaluation on XFindBugs. Related work and concluding remarks are given in Section 6 and Section 7, respectively.

2. BACKGROUND

We next briefly introduce the background of FindBugs and error-prone features in aspect-oriented programs.

2.1 FindBugs

FindBugs [8], an open-source static analysis tool, has been widely used for detecting programming defects in Java community. It provides an extensible plugin architecture in which bug detectors can be easily defined, each of which may correspond to several different bug patterns. Until now, there are more than 300 bug patterns supported by FindBugs. The detectors implemented in Findbugs use a variety of techniques. Many simple detectors use a visitor pattern over the classfile, often using a state machine to reason about values stored on the stack or in local variables. Some detectors also incorporate control flow and data flow information into analysis, for finding sophisticated defects like `Null Pointer Bugs` [26]. Findbugs has been evaluated on a number of commercial and open source projects. For example, Findbugs analyzed all 89 publicly available builds of JDK (from builds b12 to b105) [14] and generated over 370 warnings with high/medium priority. Google has also incorporated static analysis into its software development process [14, 30]. The developers in Google run FindBugs on Google's Java code base, manually evaluated warnings, and filled bug reports as deemed appropriate. As a result, there are totally 1127 warnings reported by FindBugs in Google with medium/high priority. In both experiences, the help of Findbugs facilitates programmers to find the potential defects and then modify them quickly.

2.2 Error-prone Features in AspectJ Programs

AspectJ [3], a seamless aspect-oriented extension to the Java programming language, encapsulates crosscutting concerns for better modularity using constructs like *pointcut, advice* and *intertype declarations*. These new aspectual features ease separation of concerns in software design and implementation, but also introduce new error-prone features to the traditional Java programs. For example, the join point model in AspectJ is defined in a lexical-level, and the selection relies on naming conventions. It is easy for programmers to pick up incorrect join points, particularly when using wildcards in pointcut designators (such as the bug patterns of the `Pointcut` category in Table 1). Also, more than one advice can be activated at the same join point, in which the advice invocation sequence would affect the program execution and one advice's behavior may be altered by another advice (such as the bug patterns of the `Advice` category in Table 1). Furthermore, the intertype declaration mechanism (also called introduction or structural superimpositions [18]) offered by AspectJ can easily add new class members to override an existing one, or even alter the original class hierarchy dramatically (such as the bug patterns of the `Introduction` category in Table 1).

The error-prone features of AspectJ may easily lead to potential defects in an aspect-oriented system, even for experienced programmers or in mature applications (Section 5). However, the current state-of-the-art programming environment for AspectJ (such as AJDT [5]) fails to report such defects or even fails to generate any warnings. Therefore, in order to handle the unique aspectual features, there is a need to define new bug patterns and develop corresponding tool supports for AspectJ programs.

3. BUG PATTERNS IN ASPECTJ

As previously mentioned, a key part of XFindBugs is the definition of bug patterns for AspectJ programs. Table 1 lists a set of 17 bug patterns supported by XFindBugs in the current implementation. The bug patterns listed in Table 1 include the 5 categories[1] from [37], plus 12 new bug patterns (the bottom 12 rows of the table). We collect and abstract these newly added bug patterns from various sources, such as AspectJ Bugzilla reports [4], both beginner and professional programmers' feedback and our own experience in AspectJ programming.

Table 1: A catalog of bug patterns in AspectJ programs

Pattern ID	Short Description	Category	Priority
TMAI	**T**he **M**ultiple **A**dvice **I**nvocation	Advice	Low
TIL	**T**he **I**nfinite **L**oop	Pointcut	Medium
TSOA	**T**he **S**cope **O**f **A**dvice	Advice	Low
TOTC	**T**he **O**bject **T**ype **C**hange	Advice	Medium
MOG	**M**isuse **O**f **G**etTarget	Advice	Medium
TROP	**T**he **R**eturn **O**f **P**roceed	Advice	Medium
AFBI	**A**ccess **F**ield **B**efore Object **I**nitialization	Advice	Medium
AFOS	**A**ssign **F**ield **O**f **S**uperClass	Advice	Medium
TNP	**T**he **N**egated **P**ointcut	Pointcut	Low
IGMI	**I**nvoke **G**enericType **M**ethod **I**ndirectly	Introduction	Medium
ULIF	**U**se**l**ess **I**ntroduced **F**ield	Introduction	Medium
TDAM	**T**he **D**ifferent **A**ccess **M**odifier	Introduction	Medium
SA	**S**ingleton **A**spect	Advice	High
URT	**U**nchecked **R**eturn **T**ype	Introduction	High
UID	**U**nchecked **I**ntertype **D**eclarations	Introduction	Medium
TMCD	**T**he **M**ultiple **C**lass **D**eclaration	Introduction	Medium
MOAR	**M**ismatching **O**f **A**fter **R**eturning	Advice	Medium

As shown in Table 1, we classify 17 bug patterns into three categories (`Advice`, `Pointcut`, and `Introduction`) according to their root causes. We also define a priority (Low, Medium, or High) for each bug pattern to indicate the severity. The instances of bug patterns with low severity are reported to warn programmers about the bad programming practices, while the reported warnings with medium or high severity may lead to program crash. Since most bug patterns have a number of representation variants and alternatives, we choose the one that appears to be the most generally applicable. We next select six typical bug patterns in Table 1 to explain the symptom, cause, and cure. Explanations of other bug patterns can be found in an extended technical report [32] and [37].

3.1 Access Field Before Object Initialization

Accessing an object before its initialization will result in a `NullPointerException` at runtime. The AspectJ language provides a mechanism (the `execution(ClassName.new())` pointcut) to intercept constructor calls. The advice can also take over the control flow of object initialization in the program, and leave the object uninitialized. For example, Programmers sometimes incorrectly define an advice like in Figure 1. In this example, accessing caller object through `this(a)` will cause a Null Pointer deference. The detector in XFindBugs looks for instructions in a classfile, and generates a warning where an object might be accessed before its initialization.

An example (found by XFindBugs) of this bug pattern is shown in Figure 1. The before advice accesses object `a` before its initialization, and the dereference of `a` in statement `if(!a.s.equals(" some value")` will lead to a `NullPointerException` at runtime. As later described in Section 5, we are surprised to find that a similar bug instance could find its way into a mature, well-tested application.

We recommend programmers avoid accessing any uninitialized field in the `before` advice, or use defensive programming (as shown

[1] The Multiple Advice Invocation, The Infinite Loop, The Scope of Advice, The Object Type Change, and Misuse of GetTarget

in Figure 2) to check the `nullness` when using fields, especially when the advice intercepts constructor calls.

```
public class A {
   public String s = "Initialize s!";
} public aspect B {
   pointcut beforeInitialize(A a):
      execution( A.new()) && this(a);
   before(A a): beforeInitialize(a) {
      if(!a.s.equals("some value")) {...}
   }
}
```

Figure 1: Example of Access Field Before Object Initialization

```
   before(A a): beforeInitialize(a) {
      if(a.s != null) {...}
   }
```

Figure 2: Corrected Example of AFBI

3.2 Mismatching Of After Returning

There are two special cases of after advice in AspectJ, *after returning* and *after throwing*, corresponding to the two ways a subcomputation can return through a join point. Since the AspectJ compiler is unaware of the return type exposed by the *after returning* advice, when an *after returning* advice matches a `void` method, the return type will be treated as `NULL`.

A bug example found by XFindBugs is shown in Figure 3. The advice `after(): returning(Object o): withoutReturntype()` matches `public void print()` method in class `A`. Therefore, the object `o` used in this advice body will be dereferenced as `NULL` (a `NullPointerException` will be thrown).

```
public class A {
   public void print() {
      System.out.println("The return type is void!");
   }
}
public aspect B {
   pointcut withoutReturntype():
      execution(public void print());
   after () returning(Object o): withoutReturntype() {
      System.out.println(o.toString());
   }
}
```

Figure 3: Example of Mismatching Of After Returning

3.3 Singleton Aspect

In AspectJ, the *default* behavior of a non-abstract aspect is to have a single instance [6], and advices run in the context of this instance. The aspect declaration also accepts a modifier, called `"of"` that provides other kinds of aspect instance behavior. This aspect instance initialization mechanism poses extra complexity and error-proneness to the programs. In case of the careless pointcut definition, when an aspect has an `initialization` pointcut which accidently matches the self-initialization join point, the program will terminate with a runtime exception. This bug pattern, though it seems tricky or even ironic, is also found in our experiment.

We take a code snippet found by XFindBugs as an example (Figure 4). The pointcut `initPoint()` unintendedly matches the initialization of aspect `B`, and causes the program to terminate when the `after():initpoint()` advice body is executed. XFindBugs detects such anomaly in the AspectJ programs, and generates a warning, informing programmers should add a **`perthis`** keyword in the aspect declaration (Figure 5).

3.4 The Return Of Proceed

The around advice in AspectJ has the special usage of selectively preempting the normal computation at the join point. Around advice runs in place of the join point it operates over, rather than before or after it, while the `proceed` form takes as arguments the context exposed by the around's pointcut, and returns whatever the around is declared to return. However, within the body of an around advice, calling `proceed()` will invoke the next most specific piece of around advice (if there is any), which may lead to unexpectedly problems.

```
public class A {
   public void printString() {...}
   public static void main(String[] args) {
      new A().printString();
   }
}
public aspect B {
   pointcut point(): execution(* A.printString());
   before(): point() {...}
   pointcut initpoint(): initialization(B.new());
   after(): initpoint() {...}
}
```

Figure 4: Example of Singleton Aspect

```
public aspect B perthis (point()) {...}
```

Figure 5: Corrected Example of SA

An example of *The Return of Proceed* found by the detector is shown in Figure 6. The program initializes the field `Integer i` in the interface `I` with value **`new`** `Integer(3)`, after executing statement `Object result =` **`proceed`**`(val)`. However, after the `Integer` **`around`**`(`**`int`** `val)` advice, field `i` has been assigned twice (with the same value **`new`** `Integer(3)`). This violates the Java language specification[2], and a `ClassFormatError` will be thrown at runtime.

XFindBugs detects such defect and generates a warning for programmers, informing the potential conflicts (the field `i` in the interface `I`, and the **`proceed`**`(val)` statement in this example).

```
public interface I {
   public Integer i = new Integer(3);
}
public aspect B {
   Integer around(int val):
      call(Integer.new(int))&& args(val) {
      Object result = proceed(val);
      return (Integer)result;
   }
}
public class A implements I {
   public static void main(String[] args) {...}
}
```

Figure 6: Example of The Return Of Proceed

3.5 The Negated Pointcut

As introduced in Section 2.2, the join point model in AspectJ is defined in a lexical-level, and the selection relies on naming conventions. So, when using wildcards specified in a pointcut to match join points, it is easy for programmers to pick up an incorrect join point. The *negated pointcut* here refers to those pointcuts like `negatepoint(): !`**`execution`**`(* A.subString())`, which use `"!"` to exclude certain join points during execution. The use of *negated* pointcut may ease to express certain complex pointcut expressions, but it also has serious side-effects which may even result in a program crash.

Consider the negated pointcut example in Figure 7, which will throw an `ExceptionIntitializeError` at runtime. The `negatepointcut()` is likely to intercept a number of unexpected join points, such as object construction or the main method. These unexpected intercepted join points will lead to an *Object Access Be-*

[2] Public fields in a Java interface all have "public static final" modifiers after being compiled, and should not be assigned twice.

fore Initialization or *Infinite Loop* bug pattern.

```
public class A {
   public void printString() {...}
   public static void main(String[] args) {
      new A().printString();
   }
}
public aspect B {
   pointcut negatepoint(): !execution(* A.subString());
   before(): negatepoint() {...}
}
```

Figure 7: Example of The Negated Pointcut

3.6 Unchecked Intertype Declarations

The intertype declaration mechanism of AspectJ can introduce a new class member, to change the program structure statically. However, the type checker of the AspectJ compiler does not verify the type correctness of intertype declarations, which may result in program ambiguity [21], introduction conflicts or even runtime errors. The bug pattern *Unchecked Intertype Declarations* refers to the large extent problems caused by the type checking of ambiguous intertype declarations.

An example of this bug pattern is shown in Figure 8. This piece of code, which introduces an illegal intertype declaration **`public int`** `A.x = "hello"`, can surprisingly pass the AspectJ ajc v1.5 compiler. However, the program will terminate with a `VerifyError` as soon as it is executed.

Our XFindBugs implements the bug detector to check the type consistence of each intertype declaration, and generates corresponding warnings if any type rules in the Java language specification are violated.

```
public class A {
   public static void main(String[] args) {
      new A();
   }
}
public aspect B {
   public int A.x = "hello";
}
```

Figure 8: Example of Unchecked Intertype Declarations

4. IMPLEMENTATION ISSUES

To investigate the applicability of our proposed bug patterns, we implemented XFindBugs, an eXtended FindBugs for AspectJ programs. XFindBugs is built on top of the FindBugs analysis framework and supports the AspectJ compiler ajc [3] version 1.5 in the current implementation.

FindBugs provides an extensible plugin architecture. For each bug pattern presented in Section 3, we implemented a corresponding bug detector to detect the bug pattern instances. Like the existing bug detectors in FindBugs, our newly added detectors perform bytecode level analysis to detect potential defects. The detectors use a visitor pattern over the classfiles and/or the method bytecodes, getting information about types, constant values, and special flags (e.g., this value is the result of calling hashCode) about values stored on the stack or in local variables. For example, one of the most complex detectors we implemented is for bug pattern *Singleton Aspect* (Figure 4). The detector manipulates the bytecode instruction in the classfile to find the *signature* (Figure 9) of this bug pattern. The detector first verifies whether `ajc$perSingletonInstance` is in the CONSTANT POOL and then searches whether the method `aspectOf` is inside the `<init>` method declaration. If both signatures are found, the detector reports a warning.

```
private static synthetic void ajc$postClinit()
0:  new <B> (2)
4:  invokespecial B.<init> ()V (71)
7:  putstatic B.ajc$perSingletonInstance LB; (62)

public void <init>()
8:  invokestatic  B.aspectOf ()LB; (80)
11: invokevirtual B.ajc$after$B$2$4ccac9f1 ()V (82)

public static B aspectOf()
0:  getstatic B.ajc$perSingletonInstance LB; (62)
3:  ifnonnull     #19
6:  new <org.aspectj.lang.NoAspectBoundException> (64)
```

Figure 9: Bytecode Signature of Singleton Aspect Bug Pattern

5. EMPIRICAL EVALUATION

This section presents the empirical results produced by applying XFindBugs on selected subjects. We describe the objective, the subjects, and the presentation of empirical results.

5.1 Objective

The objective of the evaluation is to investigate the following questions:

- Do the bug patterns defined in this paper exist in real-world AspectJ applications?
- Can the tool XFindBugs find real potential defects?
- Can XFindBugs scale to large applications, or is there a real necessity for the usage of our tool?

5.2 Subject Programs

The subject programs used in this paper are collected from various sources. AJHotdraw [1], AJHSQLDB [2], and GlassBox [9] are three mature large-scale AspectJ applications. The ajc and abc benchmarks are obtained from the *ajc* and *abc* compiler distribution packages. The design patterns program suite, which implements 23 design patterns, is described in [20]. The subject programs used in this evaluation are also widely used by other researchers to evaluate their work [12, 31, 33, 34, 36]. We also run XFindBugs on the buggy code reported by real-world AspectJ programmers from AspectJ Bugzilla for evaluation. Table 2 shows the name of each subject program (Name), the number of lines of code in total (LOC), the number of advices (#Advice), the number of pointcuts (#Pointcut), and the number of intertype declarations (#Introduction). Several subject programs have multiple releases, we run XFindBugs on each available version.

Table 2: Subject Programs

Name	LOC	#Advice	#Pointcut	#Introduction
AJHotdraw	38846	48	33	54
AJHSQLDB	123661	30	38	0
GlassBox	39220	132	183	44
ajc Benchmarks	4656	44	30	27
abc Benchmarks	89596	54	54	87
Design Patterns	10821	15	24	43

5.3 Experiment Procedures

We conducted two different evaluations on XFindBugs to show XFindBugs can effectively detect potential defects in AspectJ programs.

- we extract the existing bug reports from AspectJ Bugzilla [4] and run XFindBugs on the reported buggy code, to see whether it can confirm reported bugs.
- we run XFindBugs on the subject programs listed in Table 2, to see whether XFindBugs can find any previously unknown defects.

XFindBugs performs fully automated analysis and generates warnings for each suspicious code snippet. For each warning produced by XFindBugs, we check the corresponding source code manually to confirm its validity.

5.4 Threats to Validity

Like any empirical evaluation, there are some threats to validity which must be taken into consideration in our experiment. Though XFindBugs has been evaluated on over 300KLOC Java and AspectJ code, the programs we investigated may be not representative enough. For example, the instances of six bug patterns (SA, URT, UID, TMCD, MOAR, and TOTC) are not found in these subject programs, but they do exist in our programming experience. Therefore, we need to investigate more AspectJ applications in our future work.

Since the bytecode generated after the advice weaving process is compiler-specific, different AspectJ compilers may produce different bytecode. For this reason, though the current implementation of XFindBugs fully supports the AspectJ compiler ajc v1.5, one of the most widely used and well supported compilers, we can not claim XFindBugs can find bug pattern instances in the woven bytecode produced by a different AspectJ compiler.

5.5 Result and Analysis

5.5.1 *Defects from AspectJ Bugzilla*

We evaluated XFindBugs to check whether it can confirm known defects described in the literature. We picked up six bug reports[3] from AspectJ Bugzilla, and ran XFindBugs on the reported buggy code snippets. As a result, XFindBugs reports seven bug instances which fail into six bug patterns[4], and pinpoints the faulty code correctly as described in the report.

5.5.2 *Defects in Subject Programs*

We used XFindBugs to detect potential bugs in a number of real-world AspectJ applications listed in Table 2. Table 3 shows the empirical results produced by XFindBugs. Column 1 shows the evaluated subject program's name, columns 2 - 12 show the number[5] of warnings on each bug pattern (Table 1) found by XFindBugs, column 13 (#All) summarizes the total warning number, and column 14 (%FP) shows the rate of false positives[6] that is determined by manually inspecting the source code. From the table, we observe that XFindBugs reports 287 warnings on the subject programs and AspectJ Bugzilla, in which 264 are confirmed as real defects (including 7 known bugs in AspectJ Bugzilla). The experimental result indicates that a large part of bug pattern instances exist in real-world programs and the other part come from our colleagues' coursework. In addition, XFindBugs can scale well to find them within an acceptable false positive rate.

AJHotDraw. AJHotDraw is an aspect-oriented refactoring of JHotDraw, which is a well-designed open source Java framework for technical and structured 2D graphics. However, XFindBugs still finds two defects on its latest version. These two defects fall into the *Misuse of GetTarget* bug pattern. We show the code sample taken from AJHotDraw in Figure 10 and 11.

The two advices shown in Figure 10 and 11 both advise static methods in AJHotdraw. In this example, the **thisJoinPoint**.getTarget() would return NULL, and a NullPointerException will be thrown when dereferencing the return object.

AJHSQLDB. Like AJHotDraw, AJHSQLDB is another aspect-oriented refactoring case study on HSQLDB. It has more than 120 KLOC in the investigated version. XFindBugs also finds several unknown defects. We show the sample code of bug pattern *Access Field Before Object Initialization* in Figure 12. In this piece of code, the advice in line 72 intercepts the exception handling block in line 35. That is, if an exception is throw from line 35, the advice in line 72 will take the control flow. However, the variable pw used in line 77 still remains uninitialized (it is intended to initialized in line 43), and a NullPointerException will be thrown.

```
84: void around(DrawingView drawingView):
      callCommandFigureSelectionChanged(drawingView) {
85:   AbstractCommand command =
     (AbstractCommand)thisJoinPoint.getTarget();
86:     command.hasSelectionChanged = true;
87:     proceed(drawingView);
88:   }
```

Figure 10: Misuse Of GetTarget in AJHotdraw: Line 84 – 90 in org.jhotdraw.ccconcerns.commands.UndoableCommand.aj

```
168: Undoable around(): callCommandGetUndoActivity() {
169:   AbstractCommand command =
         (AbstractCommand)thisJoinPoint.getTarget();
170:   return new
         UndoableAdapter(command.getDrawingEditor().view());
171: }
```

Figure 11: Misuse Of GetTarget in AJHotdraw: Line 168 – 171 in org.jhotdraw.ccconcerns.commands.UndoableCommand.aj

```
34: public ExceptionHandlingAbstractAspect() {
35:   try {...} catch (Exception e) {}
41:   if(LOG_CATCH_BLOCKS || LOG_THROW_EXCEPTION) {
42:   try {...
43:     pw = new PrintWriter(fw);
44:   }...
45: }
72: before (Exception e): exceptionCatchBlocks(e)
      && if(LOG_CATCH_BLOCKS) {
76: if(thisJoinPoint.getThis() != null) {
77:   pw.println(thisJoinPoint.getThis().getClass()
      .getName() + " => " + thisJoinPointStaticPart
      .getSignature().toShortString());
80:   }...
81: }
```

Figure 12: Access Field Before Object Initialization: Line 76 – 80 in org.hsqldb.aspects.ExceptionHandlingAbstractAspect.aj

In AJHSQLDB, another three *Misuse Of GetTarget* bug pattern instances come from line 2619 to 2627, line 2646 to 2664, and line 2676 to 2685 in the org.hsqldb.util.DatabaseManagerSwing.aj file. Due to the space limitation, we do not show the code here.

Glassbox. Glassbox is a widely-used troubleshooting agent for Java applications that automatically diagnoses common problems. It is well tested and the latest version available is 2.0GA. XFindBugs finds one defect, falling into *The Scope Of Advice* bug pattern category. We show the sample code in Figure 13.

In Figure 13, the advice parameter sql has been modified within advice before(Statement, String sql) :topLevelDynamicsSqlExcc(statement, sql). However, like method parameters, advice parameters are local to the advice. In other words, reassigning sql in the above example will have no effect outside the advice. Therefore, XFindBugs treats it as a potential defect and generates a warning with priority Low.

AspectJ Benchmark Suite and Design Patterns. In the Tetris, Dcm, and LawOfDemeter programs in the benchmark suite, XFindBugs detects 34 *The Multiple Advice Invocation* defects in total. For example, in Tetris, two data dependent advices Levels.before(): newgame() and Counter.before(): newgame() are declared without aspect precedence. These two advices will be invoked at the same join points during program execution. In such cases, the advice invocation sequence is not explicitly defined, or sometimes depends on the compiler. XFindBugs detects such symptom as a bad practice and generates a warning.

[3] Bugzilla number: 195794, 148644, 165810, 145391, 218023, and 72834.

[4] Assign Field Of SuperClass, The Return Of Proceed, The Negated Pointcut, Invoke GenericType Method Indirectly, Useless Introduced Field, and The Different Access Modifier

[5] The number in parentheses is the warnings reported by XFindBugs, while the number outside the parentheses is the real defects confirmed by us.

[6] A false positive is a warning reported by XFindBugs that is not a real defect.

Table 3: Warnings generated by XFindBugs in subject programs

Name	#TMAI	#TIL	#TSOA	#MOG	#TROP	#AFBI	#AFOS	#TNP	#IGMI	#ULIF	#TDAM	#All	%FP
AJHotdraw	8 (8)	0 (0)	0 (6)	2 (2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	10 (16)	37.5%
AJHSQLDB	142 (142)	0 (0)	0 (0)	3 (3)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	147 (147)	0%
GlassBox	0 (1)	0 (0)	1 (6)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (7)	85.7%
AspectJ Benchmarks	34 (34)	0 (0)	0 (2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	34 (36)	5.5%
abc Benchmarks	60 (60)	3 (3)	0 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	63 (64)	1.56%
Design Patterns	0 (0)	0 (0)	2 (10)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (10)	80%
AspectJ Bugzilla	0 (0)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)	1 (1)	2 (2)	1 (1)	1 (1)	1 (1)	7 (7)	0%
#All	244 (245)	3 (3)	3 (25)	5 (5)	2 (2)	1 (1)	1 (1)	2 (2)	1 (1)	1 (1)	1 (1)	264 (287)	8.0%
%FP	0.4%	0%	88%	0%	0%	0%	0%	0%	0%	0%	0%	8.0%	N/A

```
182: before(Statement statement, String sql) :
       topLevelDynamicSqlExec(statement, sql) {
183:   if (sql==null) {
184:     sql = "NULL dynamic sqlstatement";
185:   }
186: ...
187: }
```

Figure 13: The Scope Of Advice: Line 183 – 185 in glassbox.monitor.resource.JdbcMonitor.aj

In `NullCheck`, XFindBugs finds six defects falling into the *The Infinite Loop* bug pattern. We show the sample code of one defect in Figure 14. The method call of `thisJointPoint.getSignature()` in line 78 is intercepted by the pointcut `methodsThatReturnObjects` in line 71. Therefore, an infinite calling loop (`thisJoinPoint.getSignature()` → `around(): methodsTahtReturnObjects()` → `thisJoinPoint.getSignature()`) is formed, which will result in a `StackOverflowError`. In this case, XFindBugs generates a warning and suggests programmers to add `!`**`within`**`(aspectname)` to the pointcut designator, to prevent the defect.

```
71: Object around(): methodsThatReturnObjects() {
78:   ...thisJoinPoint.getSignature().toShortString() ...
84: }
```

Figure 14: The Infinitive Loop in NullCheck

There are also several kinds of warnings reported by XFindBugs in the AspectJ benchmark suite and AspectJ implementation of design patterns. These warnings are summarized in Table 3.

5.6 False Positives

We manually check each warning reported by XFindBugs in the source code and count the possible false positives. As shown in Table 3 (column %FP), the overall false positive ratio is 8.0%. For warnings generated on one specific bug pattern, the false positive ratio ranges from 0% to 88%. We can observe that most warnings reported in `AJHotdraw`, `AJHSQLDB` and `AspectJ benchmark suite` are valid. While for the `GlassBox` application, the false positive ratio is very high (85.7%). This false positive ratio might be related to the maturity of the investigated subject. For example, `GlassBox` has experienced many releases and is widely used in commercial applications, so it has nearly become a bug-free software. On the other hand, we find warnings on bug patterns like *The Scope of Advice* have a high false positive rate, due to the complexity and variance of assignment instructions for different types inside the advice body.

5.7 Experiment Conclusion

We have demonstrated the effectiveness of XFindBugs on a number of non-trivial AspectJ applications. XFindBugs scales well to over 300KLOC, and not only confirms the reported bugs in the literature, but also reports 257 previously unknown defects. We believe the potential defects (including some severe bugs) found by XFindBugs are valuable for programmers to improve the software quality. Especially for large AspectJ applications, the help of tools like XFindBugs should be undeniable.

6. RELATED WORK

We next discuss some related work in the areas of bug pattern and static bug finding techniques.

Bug patterns [25, 37] are erroneous code idioms or bad coding practices that have been proved fail time and time again. However, most of the previous research is focused on Java programming language [17, 25, 26]. Allen [13] summarized more than 14 categories of bug patterns in Java and later more pattern classifications were identified by the FindBugs research group at the University of Maryland. Farchi [17] and David [24] *et al.* also showed a catalog of concurrent bug patterns and how to find them. In our earlier work [37], we identified six bug patterns in AspectJ programs as a preliminary study, but without tool supports and empirical evaluations on real-world programs. In this paper, we focus on the unique aspectual features and present a systematic list of bug patterns for AspectJ programs. Our defect detection prototype XFindBugs has also been successfully applied to several large-scale applications.

There is also a lot of work [19, 26, 29, 35] devoted to static bug finding techniques. Static techniques range in their complexity and their ability to identify or eliminate bugs. The bug patterns proposed in [25] are a very practical technique for finding bugs in real-word software. *Formal Proof* [16] is an effective static technique for eliminating bugs. Though the existence of a correctness proof is the best guarantee that the program does not contain bugs, the difficulty in constructing such a proof is still prohibitive for most programs. *Partial verification* [23] techniques have been proposed to prove that some desired properties of a program hold for all possible executions. Compared to these techniques, the bug pattern technique used in this paper is *unsound*. It can identify *probable* bugs, but may also produce false positives and false negatives.

7. CONCLUDING REMARKS

In this paper, we presented XFindBugs, an eXtended FindBugs for AspectJ, to detect potential defects in AspectJ software. In our current implementation, XFindBugs supports a catalog of 17 bug patterns, which cover common error-prone features of AspectJ programs. Our experience of using XFindBugs on several large-scale AspectJ applications highlights the practical issues of this tool. The experience also evidences most of the bug patterns presented in this paper do exist in real-world software systems.

In the future, we plan to identify more bug patterns in aspect-oriented software systems, and incorporate new bug detectors into XFindBugs. We would also like to investigate other sophisticated analyses (such as the verification approach [22, 28] and dynamic slicing [38]), to refine our existing detectors in XFindBugs to make them more accurate.

To encourage evaluation and further research in these and other directions, the source code of our bug detectors is available at

`http://cse.sjtu.edu.cn/~zhang/XFindBugs/`

Acknowledgements. This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Cheng Zhang, Qingzhou Luo, and Xin Huang for their valuable discussions on this work. We would also like to thank the anonymous PASTE 2008 reviewers for their insightful comments.

8. REFERENCES

[1] AJHotDraw. http://sourceforge.net/projects/ajhotdraw.

[2] AJHSQLDB. http://sourceforge.net/projects/ajhsqldb.

[3] AspectJ. http://eclipse.org/aspectj/.

[4] AspectJ Bugzilla. http://www.eclipse.org/aspectj/bugs.php.

[5] Aspectj development tools (ajdt). http://www.eclipse.org/ajdt/.

[6] AspectJ Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide.

[7] Eclipse. http://www.eclipse.org/.

[8] FindBugs. http://findbugs.sourceforge.net/.

[9] Glassbox. http://www.glassbox.com/.

[10] JLint. http://jlint.sourceforge.net/.

[11] PMD. http://pmd.sourceforge.net/.

[12] C. Allan, J. Tibble, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam. Adding trace matching with free variables to AspectJ. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 345–364, 2005.

[13] E. Allen. *Bug patterns in Java*. Apress, 2002.

[14] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

[15] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on production software. *OOPSLA 07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, 2007.

[16] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426, New York, NY, USA, 2006. ACM.

[17] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 7, 2003.

[18] R. Filman et al. *Aspect-oriented software development*. Addison-Wesley, Boston Toronto, 2005.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[20] J. Hannemann and G. Kiczales. Design pattern implementation in Java and aspectJ. *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, 2002.

[21] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95, 2007.

[22] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *In Tenth International Workshop on Model Checking of Software (SPIN), volume 2648 of Lecture Notes in Computer Science, pages 235–239. Springer-Verlag, 2003.*, 2003.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Not.*, 37(1):58–70, 2002.

[24] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. *2004 PODC Workshop on Concurrency and Synchronization in Java Programs*.

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

[26] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–14, 2007.

[27] G. Kiczales et al. Aspect-oriented programming. *ACM SIGSOFT Software Engineering Notes*, 26(5):313, 2001.

[28] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004.

[29] N. Rutar, C. Almazan, and J. Foster. A Comparison of Bug Finding Tools for Java. *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSREqŕ04)*, pages 245–256.

[30] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[31] H. Shen, S. Zhang, and J. Zhao. An empirical study of maintainability in aspect-oriented system evolution using coupling metrics. In *Proc. 2nd IEEE Theoretical Aspects of Software Engineering Conference (TASE 2008)*, June 2008.

[32] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. XFindBugs: eXtended FindBugs for AspectJ. Technical Report TR-SJTU-CSE-08-06, Center for Software Engineering, Shanghai Jiao Tong University. Download: http://cse.sjtu.edu.cn/XFindBugs, July 2008.

[33] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, 2006.

[34] G. Xu and A. Rountev. Regression Test Selection for AspectJ Software. *Proceedings of the 29 th International Conference on Software Engineering*, 20(26):65–74, 2007.

[35] M. Young and R. Taylor. Rethinking the taxonomy of fault detection techniques. *Proceedings of the 11th international conference on Software engineering*, pages 53–62, 1989.

[36] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. 24th IEEE International Conference on Software Maintenance*, Sep 2008.

[37] S. Zhang and J. Zhao. On Identifying Bug Patterns in Aspect-Oriented Programs. *Computer Software and Applications Conference, 2007. COMPSAC 2007-Vol. 1. 31st Annual International*, 1, 2007.

[38] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.