Bug pattern detection

Behnam Ghavimi
ghavimi@iai.uni-bonn.de

Ugochukwu Chimbo Ejikeme
ejikeme@informatik.uni-bonn.de

May 21, 2014

Abstract

Static code analysis is the process of detecting errors and defects in the code. It can be viewed as an automated code review process. FindBugs and JTransformer are tools for doing static code analysis on Java programs. This paper tries to look at static analysis code and These tool briefly from technical aspect.

1 Introduction

Many aspects of development effects the quality of the software. Using the proper tools and technologies is as important as proper management and proper user acceptance testing. It is good to know that maintenance on software products can take more than half the resources of the entire product life cycle. [Kagan Erdil, December 16, 2003] Making the right decisions early can greatly reduce costs later.

FindBugs scans byte code for finding bug patterns and suspicious codes In the contrast of JTransformer which works on source code. Java has many features and APIs which are inclined to be used incorrectly. Finding Bugs tool can be also a motivation for improving the skills of development teams to a write better code. Even a code written by experts contains a surprising number of obvious bugs which will be discuss in the practical experience section. FindBugs chooses static analysis for finding bugs by using bug patterns. A bug pattern is a code idiom that is likely to be an error and it is found in places where code is not in correct practice in the use of a language feature or library API.Style checkers help to prevent certain violations of

some style guidelines but they are not particularly likely to be bugs, for example they always put constants on the left hand side of compressions. The difference between style checkers and bug checkers is that contraventions of style guidelines produce only problems for the developers but warnings which are found by bug checkers will cause problems for the users of the software. Another difference is that in contrast to style checkers, bug checkers may produce warnings that are unsound or complex to interpret. These fixes which are reported by the bug checker need judgments and should be fixed without making new bugs. [D. Hovemeyer, December 2004,]

2 Bug Classification

Each warning is classified in one of the following ways. Some detectors are very precise but for detecting a fix, a judgment call is required. For example, we can accurately tell there is a class which contains a static field in a code program and this static field can be altered by the other untrusted code. In this case human judgment is required to determine whether that class will ever run in this environment. FindBugs doesn't not try to judge but simply report the warnings.[D. Hovemeyer, December 2004,]

The other kind of warning is about reflecting violations of good programming practice but be unlikely to cause problems in practice. For example, many incorrect synchronization warnings correspond to data races that are real but highly unlikely to cause problems in practice so they are classified as harmless bugs. In contrast to the previous one, there are cases where the warning is accurate and in our judgment reflects a serious bug that warrants fixing. Such warnings are classified as serious.

In the field of software design and source code quality, good practices are called Design Patterns and bad practices are Bad Smell or Anti-Pattern. Design Patterns are known as good solutions for general design problems. Design patterns are usually defined as a relation between communicating objects of a software system or the way classes are structured. For example, the decorator pattern can dynamically add responsibilities to an object, rather than having the predefined responsibilities through inheritance so this can add flexibility to objects at runtime. It reduces coupling between components so they can be modified without affecting each other.

Opposed to Design Patterns are Anti-Patterns which are patterns applied in an inappropriate context; There are two types of Anti-pattern. The first one is used in the wrong context and the second one is a bad pattern which can be used anywhere. Another bad practice is Bad Smell which are

code corruption such as long methods and code duplication. Bad Smells is local code corruption within methods or classes. Anti-Patterns are usually related to structural problems, such as classes using an inappropriate hierarchy. Code Smells are usually implementation problems whereas Anti-Patterns are design problems.

In many cases, it is not clear whether a problem is a Bad Smell or an Anti-Pattern. Each bug pattern is identified by a short code and each detector falls into one or more of the following categories: 1. Single-threaded correctness issue , 2. Thread/synchronization correctness issue, 3. Performance issue and 4. Security and vulnerability to malicious untrusted code.[D. Hovemeyer, December 2004,]

3 Static Code Analysis

Static analysis is the analysis of software that is applied without executing. Finding all possible run-time errors in an arbitrary program is impossible. Huge projects spend a lot of time on reviewing of codes and finding and fixing of the defects generally. This effort and time can be extremely reduced by making the reviewing of code process in the automated way. The automation of static analysis ensures that program bugs are caught while testing. Static analysis made a mechanism for automated code reviews for finding defects early in the build phase. Static analysis supports early bug detection and fixing by comparing code with predefined patterns, improving development productivity and end-product reliability. It also helps making maintainability easier. Benefits from static analysis include a huge improvement in the reliability and quality of the software. The biggest benefit of all is the ability to identify defects at the coding stage. The automation tools catch defects in categories such as coding standards, potential performance issues and design defects.

Tools should be able to scan Java source code or Java byte code in a structured way and should provide APIs to simplify the implementation of rules. Various techniques are commonly used for Java static analysis and these techniques can be categorized into two categories. The first type of analysis is performed on Java source and the second one is performed on byte code (class files).

3.1 Scanning Java Source

Java source scanner takes Java source as input and scans it entirely and then runs some predefined rules on that source code. A thorough understanding

of the language is a must for any tool to be able to identify bugs in the particular language program. Java parsers also simplify source code by transforming source code into tree structures and JavaCC and the JJTree parser are some example of that parser. The parsed source tool helps you in implementing rules that not only involves syntactic errors, but also errors requiring dataflow analysis of source code.

An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code. The syntax is "abstract" in not representing every detail appearing in the real syntax. For instance, an if-condition-then expression may be denoted by means of a single node with two branches. Abstract syntax trees are used in program analysis.Compared to the source code, an AST does not include certain elements, such as inessential punctuation and delimiters. An AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler. A simple example of the additional information present in an AST is the position of an element in the source code. This information is used in case of happening an error in the code, to show a user where is the location of the error. The AST is used intensively during semantic analysis, where the compiler checks for correct usage of the elements of the program and the language.The generated AST will be scanned by predefined patterns which are applied in rules.

JTransformer tool is a transformation engine for Java source code. This tool creates an Abstract Syntax Tree of the full source code of a Java project. Source code changes are propagated automatically and incrementally to the internal representation. Transformations of the internal model can be propagated to the source code upon explicit request.Analysis and transformations across multiple projects is as easy as in a single project. JTransformer can also handle an incomplete code. Figure 1 demonstrates a test case which is caught by JTransformer analysis. it is possible to fix options which are recommended by JTransformer. Figure 1 shows even the fix options which are recommended by JTransformer.Figure 2 shows Factbase inspector.The Factbase Inspectors let the user view and easily navigate through the fact representation of a Java program.

JTransformer searches a pattern in the code and when it faces this pattern it will show a warning. It is necessary to define a pattern for JTransformer and for this purpose we use prolog language.The following prolog code shows a pattern can reveal self comparison bug.

Example 3.1.
```
field_local_self_comparison(Cond,[Y,X]) :-
  field_self_comparison(Cond,[Y,X]);
```

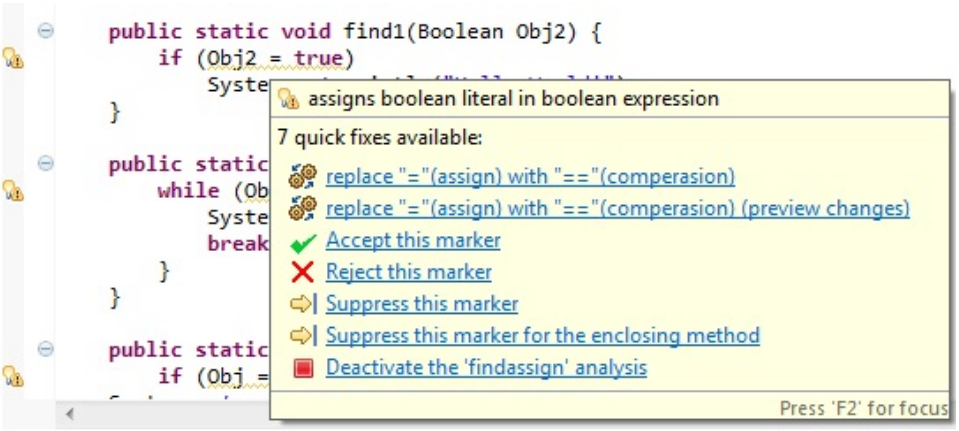

Figure 1: JTransformer Fix

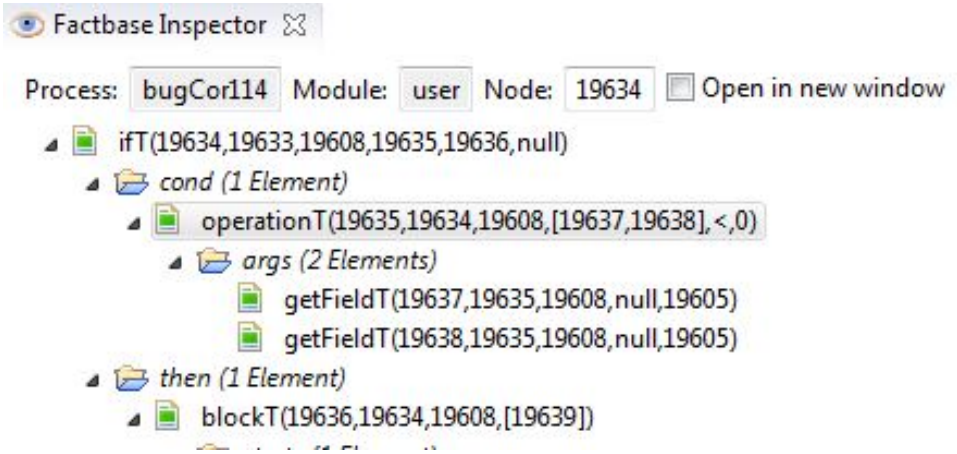

Figure 2: Factbase Inspector

```
     local_self_comparison(Cond,[Y,X]).
     field_self_comparison(Cond,[Y,X]) :-
      operation(Cond,[Y,X]),
     getFieldT(Y,Cond,_,_,FieldT),
     getFieldT(X,Cond,_,_,FieldT).
     local_self_comparison(Cond,[Y,X]) :-
    operation(Cond,[Y,X]),
        identT(Y,Cond,_,Ref),
        identT(X,Cond,_,Ref).
      operation(Id,[Y,X]):-
operationT(Id,_,_,[Y,X],<,_)
```

3.2 Scanning Java Bytecode

Java Runtime Environment contains JVM, class libraries, and other supporting files. JVM runs the code, and it uses the class libraries, and other supporting files provided in JRE. When a Java file is compiled the output is a .class file. This file consists of Java byte codes. Java Virtual Machine interprets the byte code into the machine code depending upon the underlying operating system and hardware combination.[Noll, Winter Semester 2012]

Java byte code libraries help a user in accessing byte code by providing interfaces for source level abstraction. users can use a class file without any knowledge of byte code. Understanding of Java byte code is necessary for developers as some patterns should be matched with bugs. Byte code verification is expensive and can exceed resources of small embedded systems and the implementation strategies of that can be in various ways. [D. Hovemeyer, December 2004,]

• First is the class structure and inheritance hierarchy. This type of detectors simply considers the structure of the classes without looking inside them. It is Easy to implement and very accurate for finding some kinds of bugs.

• Second is linear scanning of the code. This type of detectors makes a linear scan of the byte code of classes. These detectors do not make use of a complete control flow information. In the following code you can see part of a Java code and it's byte code.

Example 3.2. Java code :

```
public class Hello {
    public static void main(String[] args) {
```

```
    System.out.println("Hello, world");
  }
}
```

Byte code:

```
  getstatic #2;      //Field System.out
  ldc #3;           //String "Hello, world"
  invokevirtual #4; //Method PrintStream.println(String)
  return
```

A state machine will be formed to search for suspicious instruction sequences but the problem is that this technique can find bugs involving short code sequences(for example just in one class).

• The third one is control sensitive; this type of detectors uses an accurate control flow graph for analyzed methods.An example of a control flow and data flow diagram is shown in figure 3.

Example 3.3. Java code

```
public List expand(int k) {
   if (k != 0)
   this.tail = new List().expand(k - 1);
   return this;
}
```

Java bytecode

```
 0: iload_1
 1: ifeq 21
 4: aload_0
 5: new List // creates a List
 8: dup
 9: invokespecial <init>:()V // <init> is a constructor
 12: iload_1
 13: iconst_1
 14: isub
 15: invokevirtual expand:(I)LList; // LList; is a List
 18: putfield tail:LList;
 21: aload_0
 22: areturn
```

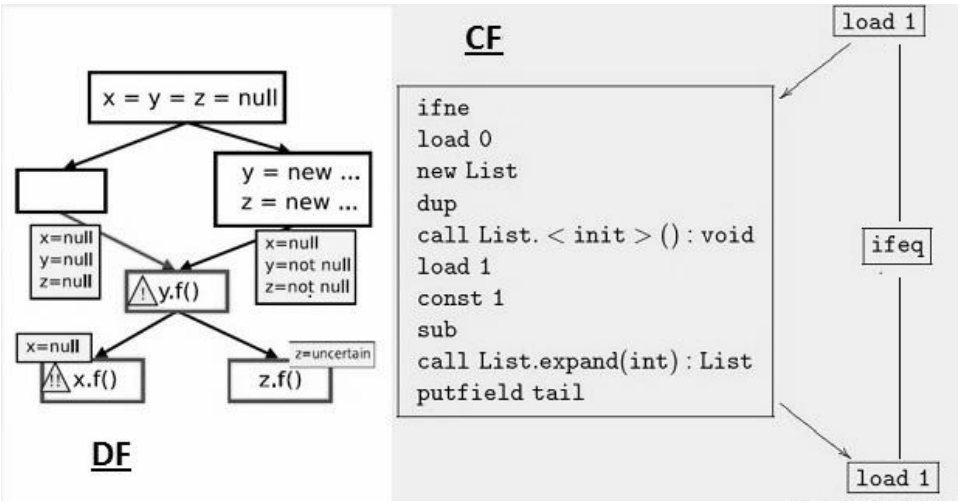

Figure 3: Data flow diagram and Control flow graph

[Spoto, 4th March 2008]

• The last one is data flow which is the most complicated detector. These detectors execute methods symbolically and try to keep track of values. They search for places where values are used in a suspicious way. An example is the null pointer dereference detector. The following code is an example of the null pointer dereference problem. Figure 3 shows the data flow diagram.

```
   x=y=z=null;
     If (cond){
   Y=new,,,;
   ........
   Z=new,,,;
 }
   y.f();

if(cond2)
 x.f();
else
 z.f();
```

3.3 Advantages and disadvantages of static code analysis

Static code analysis advantages:

- It can return the exact location of the bug in the code.
- It can be developed by developers who understand the code.
- It can recommend some fixes for each bug.
- It is fast.
- It can scan the entire code base.
- It permits weaknesses to be found earlier in the development life cycle, reducing the cost to fix.

Static code analysis limitations:

- It produces false positives or negatives.
- It can make a fake feeling in the developer that from aspect of security everything is ok.
- It is as good as the patterns which are using to scan with.
- It cannot find vulnerabilities in the runtime environment.

4 BUG PATTERN DETECTORS

Number of existed patterns in Findbug tool is 430 which are categorized in 131 types. For space reasons, just some of this type are named here.Cloneable Not Implemented Correctly (CN), Double Checked Locking (DC), Dropped Exception (DE), Equal Objects Must Have Equal Hashcodes(HE), Static Field Modifiable By Untrusted Code(MS), Null Pointer Dereference (NP), Non-Short-Circuit Boolean Operator (NS) and Uninitialized Read In Constructor (UR).

4.1 Useless self operation(SA)

It tries to find methods which contains a double assignment or a self-assignment of a local variable or a field and It also tries to find methods which compare a field or local variable with itself or performs a nonsensical computation of that element with another reference to the same element. It is implemented easily in the linear scanning way.[D. Hovemeyer, December 2004,]

Example 4.1.

```
public class test {static int y=0;
public static void find2(int x) {
   y=x-x;
    }
 }
```

Name	NCSS (Lines)	Class Files	Time (min:sec.csec)				Warning Count			
			ESC/Java	FindBugs	JLint	PMD	ESC/Java	FindBugs	JLint	PMD
Azureus 2.0.7	35,549	1053	211:09.00	01:26.14	00:06.87	19:39.00	5474	360	1584	1371
Art of Illusion 1.7	55,249	676	361:56.00	02:55.02	00:06.32	20:03.00	12813	481	1637	1992
Tomcat 5.019	34,425	290	90:25.00	01:03.62	00:08.71	14:28.00	1241	245	3247	1236
JBoss 3.2.3	8,354	274	84:01.00	00:17.56	00:03.12	09:11.00	1539	79	317	153
Megamek 0.29	37,255	270	23:39.00	02:27.21	00:06.25	11:12.00	6402	223	4353	536

Figure 4: Compression of Findbug and another tool on some big programs[N. Rutar, 2004]

4.2 Questionable Boolean Assignment

We expect in a condition that two or more variables compare to each other and then a true/false answer will be returned. Typically, one would use a compression operation like ==. However, this can easily be mistyped as =, which is an assignment operator and returns always true. This likely error is not detected by the Java compiler.

Example 4.2.

```
public void find_for_Boolean_Parameter(Boolean Obj) {
for (;((Obj = true));){
break;
    }
}
```

Note that there is no need to detect assignments of other types inside conditions, because those are caught already by the Java compiler (e.g. a=5 is not a boolean expression).

4.3 Suspicious Equals Comparison (EC)

This detector uses data flow analysis to determine when two incomparable objects are compared and using the equals () method. This compression always return false. For instance in compression of an array and a reference that doesn't seem to be an array or different types being compared are guaranteed to be unequal. Even if they are both arrays, the equals method on arrays only determines of the two arrays are the same object. (java.util.Arrays.equals(Object[], Object[]).)[D. Hovemeyer, December 2004,]

5 Practical experience

Even in the codes which are written by expert can be found some bugs.We try to observe this claim by running FindBugs and JTransformer on some big projects such as ArgoUml,AWT,jakarta and jrefactory.Just in ArgoUml version 0.17 We observe that FindBugs can catch 156 bugs in different categories;Null pointer reference,Impossible downcast and Unwritten field are some of that categories.We tried JTransformer on these projects as well and we found couple of bugs.The great thing that we found was that there are some overlap between these two tool and there are some bugs as well which can be detected just by one of them.It shows that the quality of detections are related to bug patterns and it can make some false positive and negative as well. The figure 4 shows the result of applying FindBugs and some the other tools on number of big projects.

6 Conclusion

There are lot of benefits which make using of bug detectors more reasonable, but still there are lot of works that we should do to make these tools more efficient. For instance there are still a lot of cases which human should decide about fixing these bugs and it is obvious that in some of these cases interference of human is not evadable. We can judge about this tool just by their bug patterns.This means that they can only find bugs which we define for them. There are still some false alarms and it is ideal for developers to have a precise bug detector tool which can cover most of the existing bugs in the program with as much as less possible false alarms. We should consider that there are some meta tools which merges some of these bug detectors together. This is a good idea because they can improve each other's weaknesses but still the biggest problem is remained which is the false positive. These tool use some patterns which will be applied for finding a matched bug inside the code. Each of these patterns use different strategies based on their definition which leads to have different effort times. In each of these cases still we can use much optimal algorithms to make these times as short as possible.

References

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

[Kagan Erdil, December 16, 2003] Kagan Erdil, Emily Finn, Kevin Keating Jay Meattle Sunyoung Park Deborah Yoon. (December 16, 2003). Software maintenance as part of the software life cycle. *Department of Computer Science Tufts University.*

[N. Rutar, 2004] N. Rutar, C. B. Almazan, J. S. Foster. (2004). A comparison of bug finding tools for java. *Page 245 256. of: Proceedings of the 15th International Symposium on Software Reliability Engineering* . IEEE Computer Society Washington.

[Noll, Winter Semester 2012] Noll, Thomas. (Winter Semester 2012). Static program analysis. *RWTH University.*

[Spoto, 4th March 2008] Spoto, Fausto. (4th March 2008). Static analysis of java bytecode.