SOEN7481 Fall 2018: Assignment

Building a Bug Pattern Detector Tool

https://github.com/AdrienPoupa/soen7481-assignment

Yann Kerichard
Concordia University
Montreal, Quebec, Canada

Yousef Saatchi
Concordia University
Montreal, Quebec, Canada

Gagandeep Kaur
Concordia University
Montreal, Quebec, Canada

Gagandeep Singh
Concordia University
Montreal, Quebec, Canada

Karanvir Singh Sidhu
Concordia University
Montreal, Quebec, Canada

Adrien Poupa
Concordia University
Montreal, Quebec, Canada

ABSTRACT

Bug detection is crucial is modern software, as it helps developers to find their mistakes and correct them. Bugs can be detected statically or dynamically. Static bug detection is done without running the code; it happens in Integrated Development Environment such as Eclipse or IntelliJ for example. On the other hand, dynamic bug detection is performed at runtime.

In this study, we have created a Bug Pattern Detector, inspired by FindBugs [2] and SpotBugs [6], that can detect and classify bug patterns in ten different categories. Our parser is based on the JavaParser [4] open source library, that can parse Java files into abstract syntax trees. We have tested our solution by creating unit tests. We have run our tool on two open-source softwares: Hadoop 3.0.0 and CloudStack 4.9. We compared the results with FindBugs.

KEYWORDS

Bug Pattern, Java, Java Parser, jUnit, FindBugs

1 INTRODUCTION

In a world that heavily relies on software, reliability is crucial. To ensure it, several approaches have been used: bug detection, whether static or dynamic, testing (unit testing, integration testing, etc), reviewing, and so on.

To improve software reliability, we have created a static analysis tool that detects bug patterns in Java classes. We took inspiration from the open source software FindBugs [2] and its successor SpotBugs [6]. We used the book "JavaParser: Visited" [8] to understand how the JavaParser [4] library that we used to parse Java files. Our software covers ten different bug patterns that consist of the following:

(1) *Class defines* `equals` *but not* `hashCode`: if a class overrides the `equals` function but not the `hashCode` function, causing the `HashMap` and `HashSet` functions to fail;
(2) *Comparison of* `String` *objects using == or !=*: Java uses the `equals` function to test the equality between two string, and not a double equal sign unlike other languages;
(3) *Method may fail to close stream on exception*: when a stream is open, it should be closed in the finally block. Failing to do so may result in a file descriptor leak.
(4) *Condition has no effect*: finds useless conditions that are always true or false, resulting in redundant code;
(5) *Inadequate logging information in catch blocks*: logging the same information in different catch blocks can be confusing for the developer, can make debugging difficult and should not happen;
(6) *Unneeded computation in loops*: some variables created inside a loop may not be used later, creating an unnecessary overhead;
(7) *Unused methods*: functions that are not used create unneeded additional complexity and should be removed:
(8) *Empty exception*: if there is no information displayed when an exception occurs, difficulties can arise when debugging the program;
(9) *Unfinished exception handling code*: if there is a comment stating `TODO` or `FIXME` in the catch block of an exception, this means that the exception handling should be finished;
(10) *Over-catching an exception with system-termination*: when developers are catching high levels exceptions such as Exception or `RuntimeException` and there is a call to `System.exit(0)` in the catch block of an exception, the programs halts and this behavior is not desired.

We have created unit tests using the jUnit framework to ensure that those bugs were detected by our tool. Then, we tried our tool on two open-source softwares, Hadoop 3.0.0 and CloudStack 4.9, and we compared the results with FindBugs. We found that in some cases, our tool detected false positives that FindBugs did not and for some other cases, our tool detected bugs undetected by FindBugs.

2 DETECTION APPROACH

Our bug pattern detector relies on Java Parser, an open source library used to parse Java code.

2.1 Project Architecture

At the root of our project, the `filesToParse` folder contains, as its name states, the files that are to be parsed by our tool. Then, following Maven's standard layout [5], the java files are located in the `src` folder. Within this folder are the main and test directories, containing the project's sourcecode and the unit tests, respectively. The project can be run from the `Main` class, that acts as the driver. Alongside the `Main` class, the `Util` class contains the common functions used throughout the code (`getFunctionName` to get the name of a parent function from a `Node`, `getLineNumber` to get the line number of a given `Node`). The `FileUtil` class is reponsible for

reading and writing the HTML report. Finally, the `DirExplorer` class handles the exploration of folders recursively.The `BugPattern` package contains the `Bug` interface that is implemented by the abstract `BugPattern` class. Then, each of the ten bug patterns extend the `BugPattern` class, overriding the `getIdentifier`, `getName` and `getDescription` methods used in the report.
The `Checker` package contains the `Checker` interface that is implemented by the bug checkers classes, one for each bug pattern. Each checker defines the `check` function, used to search for a specific bug pattern in a given class. Figure 1 shows a package diagram for our solution. In listing 1, the common base for all the checkers is shown. Each checker relies on the Visitor pattern; we override the appropriate visitor function provided by the Java Parser library to collect the required information for the bug pattern, and when the requirements are met, we add the bug pattern into the list that we return at the end of the function. `DirExplorer` ensures that this is run for every file contained in the parsing folder.

```
public List<BugPattern> check(File projectDir) {
    List<BugPattern> bugPatterns = new ArrayList<>();

    new DirExplorer((level, path, file) ->
         path.endsWith(".java"), (level, path, file) -> {
        try {
            new VoidVisitorAdapter<Object>() {
                @Override
                // The prototype of the visit function that
                     we override depends on the bug pattern
                     that we want to implement
                public void visit(Node n, Object arg) {
                    super.visit(n, arg);
                    // Here is the specific code
                    ...
                    // When the conditions are met, add the
                         bug pattern to the list
                    if (bug conditions are met) {
                        // Get the line number
                        int lineNumber =
                             Util.getLineNumber(n);

                        // Get the method name
                        String functionName =
                             Util.getFunctionName(n);

                        // Add the bug pattern to the list
                        bugPatterns.add(new
                             BugPatternName(lineNumber, file,
                             functionName));
                    }
                }
            }.visit(JavaParser.parse(file), null);
        } catch (IOException e) {
            new RuntimeException(e);
        }
    }).explore(projectDir);

    return bugPatterns;
}
```

Listing 1: Excerpt of the check function common to each checker

2.2 Generating the Report

Once all the checkers are executed and all the bug patterns are found, an HTML report is generated using a template. The report can be found in /results/report.html. An example of such a report is shown in figure 2. It gives a summary of the bug patterns found, and a detailed table that states where the bug was found (file name, line number) and which bug pattern it is.

2.3 Bug Checkers Explanation

(1) *Class defines* `equals()` *but not* `hashCode()` is using a visitor object inspecting each method of the provided file. Then, for each of those methods, our tool will try to identify the name and the return type of the method. If this name is *equals* and the return type is a Boolean, then a boolean value will be set to true, specifying that the equals method has been found. The tool also looks for a method named *hashCode*, having for return type a string, and not having any arguments. If this method is found, another boolean value is set to true. Finally, if one of both boolean values is false, the tool indicates an error.

(2) *Comparison of* `String` *objects using == or !=* uses 2 visitors objects. To implement this bug pattern detection, we decided to use an `HashMap` object to store all the variables of each class. First, thanks to the first visitor object, the tool will inspect each method of the file, in order to identify all String parameter's name. Then, a second visitor allows the tool to inspect each statement. In the case where the tool detects a line using an equal symbol with 2 string objects, we notify the user that a bug is detected.

(3) *Method may fail to close stream on exception* will be identified thanks to a visitor object, inspecting all method declarations. The tool will then be able to detect many kinds of stream declaration statements. Each stream is stored in an `HashMap` object. If the tool detect a line with a *close()* instruction, and if the object on which this instruction is executed is referenced in the stream's `HashMap`, the stream object is then removed. Finally, for each stream object still present in the `HashMap` at the end, an error message will be displayed.

(4) *Condition has no effect* inspects each single line of code. If the tool detects a condition statement where the condition is a literal expression (such as true for example), an error message is displayed. The tool also handles the case where the condition is comparing two objects with the same name (such as `a = a`). An error is then displayed.

(5) *Inadequate logging information in catch blocks* uses a visitor that only inspect each `Try` block statement. The error message is then stored in an ArrayList object. If the same error message appears twice, then, the tool will notify the user that there is an error.

(6) *Unneeded computation in loop* implements a visitor pattern that visits all the kinds of loops and check for any variable declaration which are then added to a list. Then we iterate through each line of code recursively and check if the variable in the list is present in the line other than the place of their declaration. If found, that variable is removed from the list and at last each variable still present in the list its the

line number and function name are fetched and added to the array list of `Bug Pattern`.

(7) *Unused methods* checker: Firstly. we check for `jUnit` importation in the class and if the method contains test annotation then it is not checked since test cases are never called directly from the source code. For rest of the methods they are added to the `hashMap` along with their class name. After that, we use visitor to visit each class and we iterate over `hashMap` to check if the current class contains the method in the `hashMap` then we set the flag for found variable to true and continue checking for other methods and if for some class, methods are not found in the `hashMap`, and also they are not a main method, then we set the flag to false and is added to to the array list of `Bug Pattern`.

(8) *Empty exception* checker: we used the catch clause visitor to parse the code containing the catch clause. After that we get the body of the catch block in `Block Statement` object and using this object we fetched all the statements of the block in the `NodeList` object. If there exists no statement, then the line number from where the catch block is starting and its function name are fetched and added to the array list of `Bug Pattern`.

(9) *Unfinished exception handling code* checker: For implementing the checker, we used the catch clause visitor pattern to parse the code containing the catch clause. After that we iterate through all the comments contained in the catch block and if the comment contains `fixme` or `TODO` as a text then the line number of the comment and its function name are fetched and added to the arraylist of `Bug Pattern`.

(10) *Over-catching an exception with system-termination* checker we used the catch clause visitor pattern to parse the code containing the catch clause and get its parameters. If the parameter contains any of the known exceptions then its body of catch block is fetched and from that we get the list of statements. After that we iterate through all the statements to get the values of `NameExpr` and `MethodCallExpr` and if they match with system and exit respectively we get the function name and its line number to add to the array list of `Bug Pattern`.

3 TEST CASES

Each bug pattern has two unit tests in the test folder that ensures the bug pattern is well detected when it is present in a dummy Java class and that it is not detected when a class does not exhibit the bug pattern.

In listing 2, a unit test class is shown, it is similar for other bug patterns. When the bug is found, the bug pattern list is checked for size (number of defects found), the file name is verified, the function name is verified as well, and we make sure that the line number is correct.

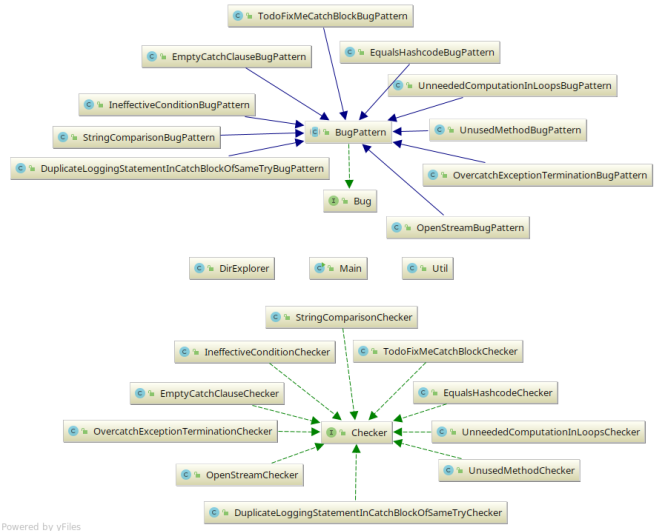

Figure 1: UML Package Diagram of our tool

Summary

Bug Pattern	Number
UF: Unfinished exception handling code	1
OS: Method may fail to close stream	3
OC: Over-catching an exception with system-termination	1
HE: Class defines equals() but not hashCode()	1
ES: Comparison of String objects using == or !=	1
EC: Empty Catch block found	2
Total	**9**

Bug Patterns

Click on a bug pattern row to see full context information.

Code	Bug Patterns
HE	filesToParse/EqualsHashcode/EqualsWithoutHashcode.java: Class defines equals() but r
	Bug type Equals Hashcode (click for details) In file filesToParse/EqualsHashcode/EqualsWithoutHashcode.java In method "equals" At line 2
OS	filesToParse/OpenStream/OpenStreamNotClosed.java: Method may fail to close stream

Figure 2: Excerpt of a report

```
public class TestEmptyCatch {
  @Test
   /**
   * Here we test for the empty catch clause bug
   * The EmptyCatch class contain the bug pattern,
   * so the bug checker returns one instance of the bug
   **/
  public void testEmptyCatchEmpty() {
    List<BugPattern> bugPatterns = new
         EmptyCatchClauseChecker().check(new
         File("filesToParse/.../EmptyCatch.java"));
    Assert.assertEquals(1, bugPatterns.size());
    Assert.assertEquals("EmptyCatch.java",
         bugPatterns.get(0).getFilename());
```

```
    Assert.assertEquals("testCatchBlock",
         bugPatterns.get(0).getFunctionName());
    Assert.assertEquals(9, bugPatterns.get(0).getLine());
  }
   /**
   * The bug pattern is not present in the NotEmptyCatch
       class,
   * so nothing should be returned
   **/
  @Test
  public void testEmptyCatchNotEmpty() {
    List<BugPattern> bugPatterns = new
         EmptyCatchClauseChecker().check(new
         File("filesToParse/.../NotEmptyCatch.java"));
    Assert.assertEquals(0, bugPatterns.size());
  }
}
```

Listing 2: Excerpt of the `TestEmptyCatch` class to unit test the `EmptyCatchClauseChecker`

4 DETECTION RESULTS

We run our tool and FindBugs on CloudStack 4.9 and Hadoop 3.0.0.

4.1 Running Our Tool

To run our tool, we extracted the zip files of CloudStack and Hadoop at the root of our tool, next to the filesToParse folder. Then, we modified the following line from the Main.java file, to change the default folder to CloudStack's and Hadoop's. This is shown in listing 3.

```
File projectDir = new File("filesToParse");
```

Listing 3: Excerpt of the `Main` class that defines the folder to parse

At the end of the analysis, an HTML result file is generated. We faced scalability issues: since we only tried our project with a few classes, scaling from a dozen of classes to thousands of them was not trivial. We had to improve our checkers so they are more efficient memory-wise, and we have to raise the Java heap using the -Xmx argument. Memory usage has been reduced using a method's hashcode instead of the `MethodDeclaration` instance. We had to raise it to 9GB of RAM to make it work for Hadoop. Table 1 shows the results for our tool on on CloudStack 4.9 and Hadoop 3.0.0.

4.2 Running FindBugs

Next, we run FindBugs on Hadoop 3.0.0 and CloudStack 4.9. Both projects are from the Apache foundation and follow the same guidelines. To analyze bugs, we had to build and compile the two projects. We faced some issues doing so, for example we had to install Protobuf 2.5.0 otherwise Hadoop would not compile with a later version [1]. Then, one can run the compilation and FindBugs report generation with the following command described in listing 4.

```
mvn compile findbugs:findbugs
```

Listing 4: Maven command to run FindBugs generation

Bug Pattern	Hadoop	CloudStack
UF: Unfinished exception handling code	34	102
IL: Duplicate logging statements in different catch blocks	5	2
OC: Over-catching an exception with system-termination	13	48
OS: Method may fail to close stream	51	17
UM: The method is defined but never used	23,974	21,899
IC: Condition is ineffective	3	6
HE: Class defines equals() but not hashCode()	8	4
EC: Empty Catch block found	1,645	83
UC: Uneeded computation is present in a loop	917	89
ES: Comparison of String objects using == or !=	2	2
Total number of bugs found	26,652	22,252

Table 1: Bug detection result for our tool on Hadoop and CloudStack

This command makes FindBugs generate a `findBugsXml.xml` file for each subproject. This means that we have to merge them together, and this can be done using an undocumented option, UnionBugs [3]. Then, the merged XML file can be opened in FindBugs or SpotBugs, and a HTML or TXT report can be generated. Then, we found that Hadoop only had one bug, which does not seem realistic given the size of the project. Later, we found that the development team had FindBugs ignore bugs using XML filter files. We replaced those files with an empty filter to get the full report, since our tool does not implement a filter feature. The same files were present in CloudStack but applying the same process did not change the result output. We found that for Hadoop and CloudStack, SpotBugs was not supported and only FindBugs was available. Table 2 shows the results of the FindBugs analysis.

Project	Hadoop	CloudStack
Lines of code	943,502	318,866
Number of classes	14,564	5,800
Number of packages	546	473
Number of bugs	2,945	127
HE	0	0
ES	2	0
OS	1	0
IC	0	0

Table 2: Bug detection result for FindBugs on Hadoop and CloudStack

Most bugs (91%) found in CloudStack were internationalization warnings, meaning that methods would "perform a byte to String conversion, and will assume that the default platform encoding is suitable", possibly causing unwanted behaviors depending on the platform.

4.3 Comparing the Results

The table 3 sums up the differences for bug patters 1 to 4 between our solution and FindBugs.

Tool	Our Tool		FindBugs	
Project	Hadoop	CloudStack	Hadoop	CloudStack
HE	8	4	0	0
ES	2	2	2	0
OS	51	17	1	0
IC	3	6	0	0

Table 3: Recapitulating the bug number difference

(1) HE - Class defines `equals` but not `hashCode`: our tool scoped to the current class only, so that if a class defines a `hashCode` function in the parent class and `equals` in the child class, it will report a bug. This is the case for the `RMDelegationTokenIdentifierForTest` class: it extends the `RMDelegationTokenIdentifier` class which extends `YARNDelegationTokenIdentifier`, which extends `AbstractDelegationTokenIdentifier` that finally implements `hashCode`.

(2) ES - Comparison of `String` objects using == or !=: the two bugs detected by our tool in CloudStack should have been detected by FindBugs. Listings 5 and 6 show what has been detected by our tool:

```
public static final String ERR_TIMEOUT = "timeout";
```

Listing 5: Excerpt from the `Script` class, definition of the `ERR_TIMEOUT` attribute

```
String result = null;
...
if (result == Script.ERR_TIMEOUT) { // should not be
     done!
```

Listing 6: Excerpt from the `KVMHABase` class, method `runScriptRetry` in the CloudStack hypervisors KVM plugin

Here, the `result` variable, which is a `String`, is compared with the equality operator against the `Script.ERR_TIMEOUT` variable, also being a `String`.

(3) OS - Method may fail to close stream on exception: the difference is coming from the fact that our tool has a scope limited to the current method to check if the stream has been closed in the `finally` block. For example, the code in listing 7 is closing the stream without using the `close` function directly. Thus, our tool detects a bug because the stream is not explicitly closed within the function.

```
finally {
    IOUtils.cleanup(null, metaStream, dataStream,
         checksumStream);
}
```

Listing 7: Excerpt from the `DebugAdmin` class, method `run` in the Hadoop HDFS project

(4) IC - Condition has no effect: our tool detected real bug patterns in CloudStack, such as shown in listing 8 in the `DatabaseUpgradeChecker` class:

```
if (true) { // FIXME Needs to detect if management
     servers are running
```

Listing 8: Excerpt from the `DatabaseUpgradeChecker` class, method `upgrade` in the upgrade CloudStack package

We do not know why this was not detected by FindBugs after the removal of the FindBugs filter.

5 THREATS TO VALIDITY

Our study only focused on analyzing only two softwares: CloudStack and Hadoop. There is no certainty that those findings could be generalized to other software. We have no software to compare the results to for bugs 5 to 10.

6 RELATED WORK

The FindBugs software has been described by Nathaniel Ayewa et al. in 2008 [7]. In 2011, Antonio Vetro et al. have discussed about the validation of FindBugs issues [9].

7 FUTURE WORK

The findings of the paper could be extended by redoing the tests on SpotBugs. Additional software could be tested. Our solution could be improved to scale better for larger software.

8 CONCLUSION

In this paper, we created a software that is able to detect 10 different bugs patterns. We run this tool on two open source software, Hadoop 3.0.0 and CloudStack 4.9. We compared the results with the results obtained from running FindBugs on those two software for the common bug patterns. We found that the differences we observed were due to either a reduced scope in our software or a lack of detection in FindBugs.

ACKNOWLEDGMENTS

We would like to thank our supervisor, Dr. Tse-Hsun (Peter) Chen, for the patient guidance, encouragement and advice he has provided throughout the duration of the project.

REFERENCES

[1] 2018. Building hadoop by installing protoc 2.5.x. Retrieved October 19, 2018 from https://vrushaliscorner.wordpress.com/2016/02/19/installing-protoc-2-5-x-google-protocol-buffers-compiler-on-a-mac
[2] 2018. FindBugs - Find Bugs in Java Programs. Retrieved October 19, 2018 from http://findbugs.sourceforge.net/
[3] 2018. How to generate 1 report with findbugs-maven-plugin? Retrieved October 19, 2018 from https://stackoverflow.com/questions/23128275/how-to-generate-1-report-with-findbugs-maven-plugin
[4] 2018. JavaParser - For processing Java code. Retrieved October 19, 2018 from http://javaparser.org/
[5] 2018. Maven - Introduction to the Standard Directory Layout. Retrieved October 19, 2018 from https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html/
[6] 2018. SpotBugs. Retrieved October 19, 2018 from https://spotbugs.github.io/
[7] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, and William Pugh John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 10177397 (Aug. 2008), 22–29. https://doi.org/10.1109/MS.2008.130
[8] Federico Tomassetti Nicholas Smith, Danny van Bruggen. 2018. *JavaParser: Visited.*
[9] A. Vetro, M. Morisio, and M. Torchiano. 2011. An empirical validation of FindBugs issues related to defects. *15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE 2011)* (2011). https://doi.org/10.1049/ic.2011.0018