BugPatterns: Static Bug Detection Tool

Gunpreet Ahuja
Concordia University
Montreal Quebec Canada
gunpreetahuja0528@gmail.com

Manpreet Kaur
Concordia University
Montreal Quebec Canada
man.kaur3710@gmail.com

Sachleen Kaur
Concordia University
Montreal Quebec Canada
sachleen19@gmail.com

Gurpreet Lotey
Concordia University
Montreal Quebec Canada
gurpreet.rajarh@gmail.com

Nandhini Anbalagan
Concordia University
Montreal Quebec Canada
nandhininandhu94@gmail.com

Sanghavi Kartigayan
Concordia University
Montreal Quebec Canada
sanghavikartigayan@gmail.com

ABSTRACT

Static analysis tools help to detect bugs in the source code without its execution, and capture the defects in the code early, this eventually avoids the later expensive fixations. In order to find bugs, these tools require specifications, which are sometimes called bug patterns. We have implemented ten pre-defined bug patterns using Java language. In this study, we provide analysis of design that we followed for implementing a static analysis tool that help detecting pre-determined bug patterns in Java source code. This study also provides the description of result retrieved after running this tool on test suite that we developed for testing validity of this tool. Also, to verify our results we run this tool on existing open source Java projects and compared our results with that of Findbugs [1] which is static analysis tool widely used within industry for Java projects. We also manually inspected the bugs that are reported by our tool for the data set that we have selected to test our tool.

Keywords: Abstract Syntax tree, JDT, Bug Patterns.

INTRODUCTION

Static code analysis is a software verification technique used to analyze source code without the execution of that program, accomplished by various static analysis tools such as FindBugs, SpotBugs etc.

The use of such tools has been controversial [2], some developers believe that static analysis tools are glorified compilers, while the truth is believed to be that static analysis tools perform their task more accurately as apparently they can find more complex issues than compilers cannot. It can be difficult to find errors, violations and also to apprehend complex source codes that involve unfamiliar third-party or legacy code, this is where these tools prove to be valuable as they assist the experienced developers to figure out and resolve the issues that were challenging for them. Hence ,Static analysis improves the effectiveness of developers, whereas others think that it's for junior developers. Static analysis is believed to be capable of catching real issues disguised as false-positives, it's important to realize that for static analysis tool, developers have emphasized on reducing the false-negative results. They don't want their tool to pass issues in the code that could be potentially important. Some tools also provide permission to mark areas of code as 'ignored' or 'accepted' so that a developer can apply direct control over certain, acceptable false-positives. On the other hand there are some developers who believe that Static analysis is noisy and is responsible for the generation of too many false-positives. In case of a good quality code, these tools will be able detect and deal with code problems faster and will provide feedback that will help developers write better code, and be a better software developer.

In the effort to make such a tool , we have implemented a static analysis tool that follows the desired specifications i.e. the bug patterns that will work on hadoop and cloud-stack operating systems. The study describes in detail the design of static tool that we implemented, results achieved after running this tool using Junit test suite, created in order to test the whole code. We have also verified our results by comparing the output generated by using existing Java projects with that of FindBugs.

The report is divided into various sections 1. Detection Approach, 2. Summary of test suite, 3. Detection Results and discussion.The detection approach section, gives detailed explanation about the task that abstract syntax tree does , how the nodes are processed, how the information is

obtained from the Abstract syntax tree nodes and the steps to install and run our tool for the Java source code files. The Section 2 provides, Summary of Test Suite and Test Cases, gives detailed description about JUnit test cases that were created for each and every bug pattern.the explanation includes the objective of the test plan i.e the purpose of the test case that was designed, the outcome of the bug pattern execution and the results of the test cases and if they actually passed and got verified or not.The section 3, detection results and discussion gives description about the comparison of results of first four patterns and the static analysis tools i.e. Spotbugs which is extended version of FindBugs for 100 projects of Hadoop 3.0.0 [3] and similarly 134 projects of CloudStack 4.9 [4], also we mention why this comparison was performed .

DETECTION APPROACH

In this study, we tried to detect pre-determined bug patterns in Java projects. For this, we implemented a static analysis tool in Java that will be able to detect 10 known bug patterns in Java source code. We implemented this tool as a Maven project that takes as input Java source code files and reports all the bugs detected from the 10 bug patterns defined within our tool.

For detection of bug patterns, we tried analysing the source code by creating an AST for the source code and parsed the AST using two tools namely, Java Parser and JDT which is an inbuilt Eclipse tool for analysing and manipulating Java source code.

We developed a test suite that contains code containing bugs, the patterns for which are implemented within our tool. Using these tools, Java source file is entirely represented as tree of AST nodes and these nodes represent the elements of Java programming language. Figure 1 represents AST tree representation of Java source code illustrating the types of nodes created for a program. Each node contains the structural value of element it represents and value of these elements can be obtained using certain generic and structural methods. By visiting these nodes within our tool, we tried to detect bug patterns within our test suite and analysed the results obtained.

In order to further verify the validity of our tool, we run our project on buggy files of two existing Java open source projects and analysed the result obtained. Also, we performed the manual inspection for all bugs reported by our static analysis tool to determine if the bugs reported by our tool are valid and if that bug should be reported by our tool for that line number. All these results are further discussed within our report.

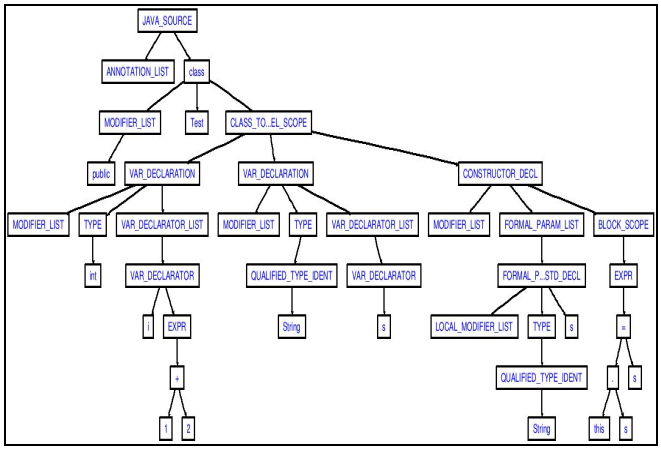

Figure:1 Parse Tree

We also made comparison of the result we obtained by running our tool with that of Findbugs. Basically, we first ran FindBugs static analysis tool on all projects within Hadoop and Cloud Stack open source projects and collected the errors being reported by FindBugs. Then, we filter the bugs that match first four bug patterns implemented within our tool. The files which are reported as Buggy by Findbugs containing those bugs are then isolated and retrieved from Github repository of that particular project and we run our tool on those files and made comparison to check whether same bugs are reported by our tool also.

Below are the steps required to install and run our tool for any Java files:

- Download or clone the project from github[1].
- Then this project need to be compiled and build using pom.xml as it is a Maven project.
- All the files that you need to test for any of the bug patterns are to be saved under "Bug Patterns/src/resources"
- These files will then be parsed by parser.java and all the bugs present within these files will be displayed in the console along with line numbers.

After following above steps, this static analysis tool will be able to depict all the bug patterns defined within our tool for target Java files in console.

SUMMARY OF TEST SUITE AND TEST CASES

A test suite was created with a class name "PatternTestSuite", which holds two different test classes named, "CatchStmTest" and "AppTest".

[1] https://github.com/GunpreetAhuja/BugPatterns

The first test class “CatchStmTest” was designed to address the pattern number 5. The second test class “AppTest” was designed to address some of the patterns numbered 1, 2, 3, 4, 6, 7, 8, 9, 10.

Each of these test scenarios and their corresponding outputs will be discussed in detail further:

1. Class defines equals() but not hashCode()

Test Plan Objective: The test case was designed to check the method name with its associated modifiers. If the method name matches equals or hashcode and the modifier matches @override, then the desired output is printed.

Test Cases: With equals() method; with equals() and hashCode() method

Bug Pattern Execution Outcome:

HE_EQUALS_NO_HASHCODE

Hash Code method not found: The class may violate the invariant that equal objects must have equal hashCode.

Test Result: Passed and Verified

2. Comparison of String objects using == or !=

Test Plan Objective: Test case was designed to check two equals strings. One among the two test cases compares two strings whereas, the other test case compares object with a string.

Test Case: With two strings; an object with a string.

Bug Pattern Execution Outcome:

ES_COMPARING_STRINGS_WITH_EQ

Line Number: 3 - Consider using the equals(Object) method instead

Test Result: Passed and Verified

3. Method may fail to close stream on exception

Test Plan Objective: Inside a method, file was created and accessed, but the file is not closed. So, the test case examines that the file is closed or not and prints the result accordingly.

Test Case: Creation of a file, Manipulations like read and write, Fail to close the file.

Bug Pattern Execution Outcome:

OS_OPEN_STREAM_EXCEPTION_PATH

Possibility of stream(s) left opened: Method may fail to close stream on exception.FileReader

Test Result: Passed and Verified

4. Condition has no effect

Test Plan Objective: To check whether the condition plays an active role in the manipulation and the outcome or it remains the same throughout the block.

Test Case: Placing a condition check, evaluation of the condition doesn’t determine any further operations.

Bug Pattern Execution Outcome:

UC_USELESS_CONDITION

Expression: true, Line Number: 3 - Possibility that this condition has no effect

Test Result: Passed and Verified

5. Inadequate logging information in catch blocks

Test Plan Objective:To check the duplicate logging statements in different catch blocks of the same try block.

Test Case: A file with different try catch and nested try catch is created and passed to test.

Bug Pattern Execution Outcome:

Inadequate logging information on line-39!

The description is System.out.println("Catch----4");

Inadequate logging information on line-31!

The description is System.out.println("file2");

Test Result: Passed and Verified

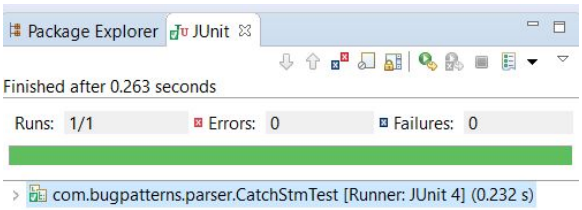

Figure: 2 (Test Case Result : Bug Pattern 5)

6. Unneeded computation in loops

Test Plan Objective: The test case is designed in such a way, that it checks whether the computation outcome (return type) produced inside the loop is used for further operations or not.

Test Case: Loop created with a return type, but it is never used anywhere in the program.

Bug Pattern Execution Outcome:

Declaration of 'k' at line 3

Usage of 'k' at line 3

Bug Pattern found:Unneeded computation in loops.

Test Result: Passed and Verified

7.Unused Method

Test Plan Objective: Test case was designed to detect the method which has declaration but has never been used or called.

Test Case: Method declaration given, but is not called anywhere.

Bug Pattern Execution Outcome:

Unused Methods

Line Number: 3 fun is not invoked anywhere within project

Test Result: Passed and Verified

8. Empty exception

Test Plan Objective: The Output stream is parsed to check the presence of Bug Indicator message for Empty Exception.
Test Case: Try block is designed to throw a ArrayIndexOutOfBoundsException, but the catch block doesn't print any error message.
Bug Pattern Execution Outcome: Line number: 23 - Empty Exception: There is no debug when an exception occurs
Test Result: Passed and Verified

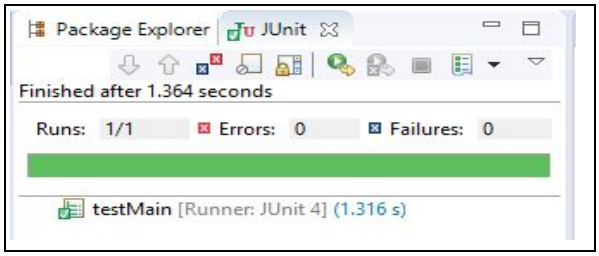

Figure: 3 (Test Case Result : Bug Pattern 8 and 9)

9. Unfinished exception handling code

Test Plan Objective: Parsing the output stream to check the presence of Bug Indicator message for Unfinished exception
Test Case: Try block is designed to throw a ArrayIndexOutOfBoundsException, whereas the catch block holds a comment FIXME.
Bug Pattern Execution Outcome: Line number: 23 - Unfinished exception handling code: There's a TODO or a FIXME in the catch block
Test Result: Passed and Verified

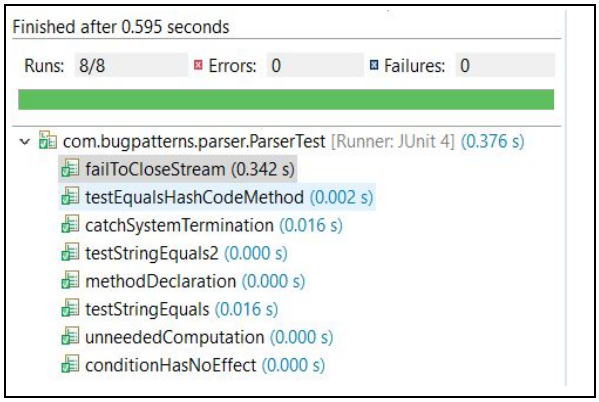

Figure: 4 (Test Case Result:Bug Pattern 1, 2, 3, 4, 6, 7, 10)

10. Over-catching an exception with system termination

Test Plan Objective: The test case is designed in a way that, it evaluates the Exception type, retrieve and typecast nodes of the statement into expressions, further converting the same into String and finally, detecting the count of System.exit. If it is present within the catch block, the message is printed as stated below.
Test Case: Try statement with a catch is created, system.exit is given within the catch block.
Bug Pattern Execution Outcome: Over-catching an exception with system-termination
Line Number: 3 Do not terminate system when catching very high level exceptions.
Test Result: Passed and Verified

DETECTION RESULTS & DISCUSSION

For the analysis of our project, we compared the result of first four patterns with that of FindBugs (SpotBugs). For this, two open source projects were used: Hadoop 3.0.0 [3] and CloudStack 4.9 [4]. The analysis was performed to measure the differences between FindBugs and our Static Bug Detection Tool (named BugPatterns).
Hadoop has multiple maven sub-projects that needed to be checked individually via FindBugs. We ran FindBugs on each sub-project and out of all the projects, we came up with four of the projects having 5 bugs that corresponded to three bug patterns:

- ES_COMPARING_STRINGS_WITH_EQ [5]
- UC_USELESS_CONDITION [6]
- HE_EQUALS_NO_HASHCODE [7]

The OS_OPEN_STREAM_EXCEPTION_PATH Bug Pattern was not present in the hadoop projects. The result and analysis for each of the bugs found in the hadoop projects is as follows:

1. **Project:** hadoop-aws
   Bug Pattern: Comparison of String objects using == or !=
   FindBugs Result:
   ES_COMPARING_STRINGS_WITH_EQ
   Our Tool (Figure 5):
   Bug Pattern found:
   ES_COMPARING_STRINGS_WITH_EQ
   Line Number: 773 - Consider using the equals(Object) method instead
   Limitation: Since the input of the resource files is taken as string, all the if condition operands are read as string. This makes the comparison between integers and objects as well the comparison between strings, thus, the result.
2. **Project:** hadoop-common
   Bug Pattern: Comparison of String objects using == or !=
   FindBugs Result:
   ES_COMPARING_STRINGS_WITH_EQ

Our Tool (Figure 6):
Bug Pattern found:
ES_COMPARING_STRINGS_WITH_EQ
Line Number: 217 - Consider using the equals(Object) method instead

3. **Project:** hadoop-common
Bug Pattern: Condition has no effect
FindBugs Result: UC_USELESS_CONDITION
Our Tool:
Bug Pattern not found:
Limitation: All the conditions within program are using values whose initial value is not declared but retrieved and computed within different functions from other files. So, it can not be determined whether condition will always remain true or false as value of variable is not known within that file.
4. **Project:** hadoop-hdfs
Bug Pattern: Condition has no effect
FindBugs Result: UC_USELESS_CONDITION
Our Tool:
Bug Pattern found: UC_USELESS_CONDITION
Expression: i == RO_NODE_INDEX, Line Number: 103 - Possibility that this condition has no effect
(Since the value of i and RO_NODE_INDEX is 0)
5. **Project:** hadoop-yarn-server-resourcemanager
Bug Pattern: Condition has no effect
FindBugs Result:
HE_EQUALS_NO_HASHCODE
Our Tool:
Bug Pattern found:
HE_EQUALS_NO_HASHCODE
HashCode method not found: The class may violate the invariant that equal objects must have equal hashcodes

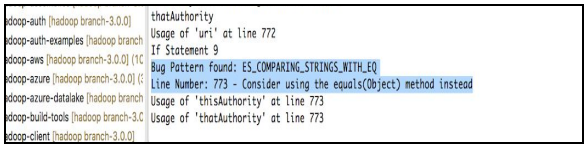

Figure: 5

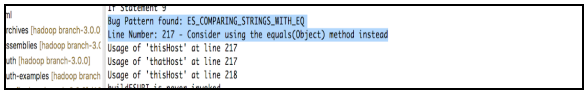

Figure: 6

Similarly, CloudStack has multiple , more than 100, maven sub-projects that needed to be checked individually via FindBugs. We ran FindBugs on each sub-project and out of all the projects, we came up with two of the projects having two bugs that corresponded to two bug patterns only:

- UC_USELESS_CONDITION
- ES_COMPARING_STRINGS_WITH_EQ

The other two were not present. The result and analysis for each of the bugs found in the cloudstack projects is as follows:

1. **Project:** cloud-plugin-network-vsp
Bug Pattern: Condition has no effect
FindBugs Result: UC_USELESS_CONDITION
Our Tool (Figure 7):
Bug Pattern found: UC_USELESS_CONDITION
Expression: !resourceConfigurationChanged, Line Number: 518 - Possibility that this condition has no effect

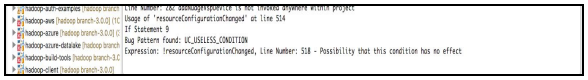

Figure: 7

2. **Project:** cloud-utils
Bug Pattern: Comparison of String objects using == or !=
FindBugs Result:
ES_COMPARING_STRINGS_WITH_EQ
Our Tool (Figure 8):
Bug Pattern found:
ES_COMPARING_STRINGS_WITH_EQ
Line Number: 62 - Consider using the equals(Object) method instead

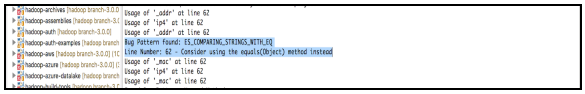

Figure: 8

```
try {
    session.connect(getSshConnectTimeout());
  } catch (JSchException e) {
    LOG.warn("Unable to connect to " + host
        + " as user " + args.user, e);
    return false;
  }
```

Figure: 9

Apart from these four bug patterns compared, we also analyzed the rest of the bug patterns within our tool. Some of them we are explaining here. Figure 9 shows the Empty

Exception pattern for which the code is shown in Figure 10. This code snippet from hadoop does not contain any logging for debug mode so the bug.

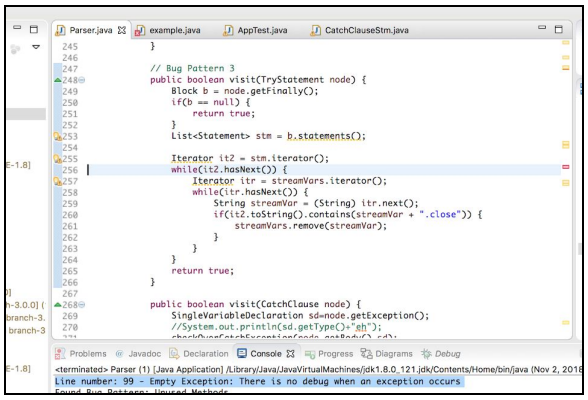

Figure: 10

Also, for one of the files in hadoop utils our code detected some errors, the explanation of which is provided below:

Bug Pattern (Figure 11): Unneeded Computations in loop

Result: Success

Explanation: The variable storing value returned from function call is used within if so there is possibility that value may never get used within loop. Refer to Figure 10 for the respective code snippet:

```
for (int i = 0; i < overlays.length; i++) {
  overlays[i] = new Configuration();
  if (i == RO_NODE_INDEX) {
    overlays[i].setEnum(SimulatedFSDataset.CONFIG_PROPERTY_STATE,
      i == RO_NODE_INDEX
        ? READ_ONLY_SHARED
        : NORMAL);
  }
```

Figure: 11

Bug Pattern: Unused Methods

Result: Success

Explanation: Our tool finds this bug pattern based on the methods that are not found in the list generated containing all the invoked methods.

Limitations: It also displays the constructor methods as unused because they are not called as such using the method invocation statements. They are just needed to create objects. It displays the result for each unused method repeatedly. This is due the fact that the methodNames for the classes are populated dynamically while the invokedMethods are checked from the methodNames list. This results in parsing of the whole methodNames list repeatedly which causes multiple output for the bug for one method.

For rest of the bug patterns, the tool gives results as expected.

CONCLUSION

This paper describes the implementation of the bug pattern tool that uses ten different specifications as it's key requirements. Initially, all the bug patterns were implemented and a proper test suite with its corresponding test cases were designed to verify whether the bug pattern which we have designed are able to detect their respective bugs. Then, we ran our bug detection tool over 100 projects of Hadoop 3.0.0 and 134 projects of Cloudstack 4.9. And finally, our results were compared with that of the Find Bugs tool outputs and the difference of the same were discussed. Also there are some limitations in our tool that are also specified within discussion e.g. It also displays the constructor methods as they are not called etc.

REFERENCES

[1] Findbugs.sourceforge.net. (2018). FindBugs™ -Find Bugs in Java Programs. Available at: http://findbugs.sourceforge.net

[2] Roguewave.com. (2018). [online] Available at: https://www.roguewave.com/sites/rw/files/documents/white-papers/klocwork-source-code-analysis-ebook.pdf

[3] GitHub. (2018). *apache/hadoop*. [online] Available at: https://github.com/apache/hadoop/tree/branch-3.0.0

[4] GitHub. (2018). *apache/cloudstack*. [online] Available at: https://github.com/apache/cloudstack/tree/4.9

[5] Anon, (2018). [online] Available at: http://findbugs.sourceforge.net/bugDescriptions.html#HE_EQUALS_NO_HASHCODE

[6] Findbugs.sourceforge.net. (2018). *FindBugs Bug Descriptions*. [online] Available at: http://findbugs.sourceforge.net/bugDescriptions.html#ES_COMPARING_STRINGS_WITH_EQ

[7] Anon, (2018). [online] Available at: http://findbugs.sourceforge.net/bugDescriptions.html#UC_USELESS_CONDITION