SEON 7481 Assignment 1

Implementation of a static analysis tool

Amir Hossein Bavand
Department of Computer Science
and Software Engineering
Concordia University
Montreal
Amirhossein.bavand@gmail.com

Emad Fallahzadeh
Department of Computer Science
and Software Engineering
Concordia University
Montreal
emad.fallahzadeh@gmail.com

ABSTRACT

Software testing is one of the most important and costliest stages in the software development life cycle. a good approach to reduce its cost is using static analysis tools during code development or before sending the commit to be tested. FindBugs is a famous static analysis tool that is widely used by software developers. FindBugs can detect some bug patterns that are potential bugs in the system and report them to developers. it is a static tool, so it doesn't need to run the code. In this work, we have implemented 4 Bug patterns for static analysis and three of them have been extracted from FIndBugs patterns. We designed test cases and tested our code using Junit to prove its efficiency. We have also tested our tool on cloud stack, a large open-source project on Github, the results have been manually inspected by the authors and turned out that in 3 of the patterns the false-positive rate is low.

CCS CONCEPTS

• Software and its engineering • Software testing and debugging

KEYWORDS

Bug patterns, FindBugs, Software testing, Tool implementation,

1 INTRODUCTION

Software testing is one of the most important and costliest stages in the software development process. Prior research shows that about half of the cost of software development is related to software testing [1]. A good approach to reduce this cost is inspecting code before sending its commit to being tested. It means that we can develop static analysis tools to investigate code without running the project. A static analysis tool is a tool that can detect some patterns which are potential bugs. It means that by using it we can find bugs before testing or even during the programming process. For example, a common mistake between developers is not closing the IO stream after using it. any IO stream object should be closed at the end of using. A good general idea is to use a finally block to ensure that streams are closed [2]. A good example of a static analysis tool is FindBugs. FindBugs is an open source program which uses static analysis to look for bugs in Java code [2].

In this assignment, we have implemented a static analysis tool with having the ability to detect 4 bug patterns. Three of them have been taken from FindBugs bug patterns and one of them is a quite new pattern that does not exist in FindBugs. We implemented the following bug patterns.

1. **Class defines equals() but not hashCode (bug pattern number1)**

Some classes override equals method but do not override hashCode method. This class may violate the invariant that equal objects must have equal hashcodes. To implement it we should check that if every class that override equals method, also override hashCode method. We implemented it and we found 7 potential bug errors. By manually investigating them we found out that all of them haven't implemented hashCode. To be honest, we are not aware of details of the code but we are sure that at least 3 of them are false negative. Because they return false instead of true in some cases and it does not break the rule of equal object must have equal hashcodes.

2. **Method may fail to close stream on exception (bug pattern number2)**

Some methods create an IO stream object but does not assign it to any fields, pass it to other methods or return it, and does not appear to close it on all possible exception paths out of the method. This may result in a file-descriptor leak. It is generally a good idea to use a finally block to ensure that streams are closed [2]. We could implement it and obtain 118 potential bugs. We randomly picked 20 of them and inspected them manually. As we are not the developer of the cloudstack, like previous part we cannot report the false-positive rate exactly. But we could understand that 5 of 20 were related to the server and all of the server is shutdown, so we can consider them as a false-positive.

3. **Condition has no effect (bug pattern number3)**

Some conditions always produce the same result as the value of the involved variable was narrowed before. Namely, the condition or Boolean variable in if/while always returns either true or false. Probably something else was meant or

condition can be removed [2]. We implemented an algorithm using Abstract Syntax Tree to extract some parts of code that are suspicious to be a bug. Using it leads to obtain Potential bugs. We randomly selected 20 of them and investigated them manually and we found out that all of them were true-positive.

4. **Inadequate logging information in catch blocks (bug pattern number4)**

Developers usually rely on logs for error diagnostics when exceptions occur. However, sometimes, duplicate logging statements in different catch blocks of the same try block may cause debugging difficulties since the logs fail to tell which exception occurred [2]. We used to approach to detect this type of errors. Our first similar approach could find 373 potential bugs and after inspecting 20 samples we find out 95% of them false-positive. Our more complex approach could find 30 bugs and after inspecting 20 of them, we find 85% of them true positive.

2 DETECTION APPROACH

We implemented and designed algorithms to detect each of the above 4 bog patterns. We have Abstract Syntax Tree of Java code and all of the information mentioned below have been extracted from the parsing of AST.

2.1 Class defines equals but not hashCode

Some classes override the equals method but do not override the hashCode method. This class may violate the invariant that equal objects must have equal hashCodes. to detect these classes, first of all, we visit the abstract syntax tree of the class. Our focus is on the name of the methods of the class. If we see the method equal, we will set the flag **hasEqual** to true. If we see the method hashCode, we also make the flag hasHashCode true. At the end of visiting the class, we check if hasEqual is true and hasHashCode is false. If so, we will report a bug pattern. By using this approach, we do not miss any potential bugs. However, we could report false-positive. Because sometimes, the equal method can make a stronger condition for objects to be equal. In this case, the overriding equals method does not lead to any problem even if we do not override the hashCode method.

2.2 Method may fail to close stream on exception

Some methods create an IO stream object but do not close it or may fail on closing it. First of all, we detected IO objects that do not be closed. For each IO object when it is defined, we add it to a Map. **We also use binding to trace it during the code**. If it is closed, we will set a flag to true. For each IO stream object, we have a similar flag. In the end, for each object stream, we check that if its flag is true. If it is false, we report it as a bug. We could extract 91 patterns by this approach. Then we used a more complex approach. We added to our precious method the condition that the closing of the object should be in a finally block. If so, we ignore it. Otherwise, we will report it as a potential bug. We could obtain 118 potential bugs. The reason for adding this condition as that if the **close** function is run somewhere out of the finally block it could fail because of some exceptions and the method may fail to close the object stream.

2.3 Condition has no effect

Some conditions have no effect. To detect them, we extract all conditions that have just a **true** or **false** in their expression. For example, if a condition is **if(true)** or **while(true)**, we report it a potential bug. Also, we trace every Boolean variable using **binding**. For example, we have a condition like while(A) and A is a Boolean variable. If we do not have any assignment for A in the code, we will report it as a bug. The reason for this condition is that if we have an assignment for it, the code may use that assignment to change the condition. By using this approach, we may produce some false negative, but we prevent the happening of a lot of false-positive.

2.4 Inadequate logging information in catch blocks

As a first approach, we extracted all try blocks that have at least two catch clauses with the same string on their block. Using this approach, we could obtain 373 try blocks. It is obvious that we just see to the strings and do not consider the exact log that is produced during the block. For example, most of them contain exception e or ex in their log but this approach may detect them as potential bugs. So, we made our method more advance and consider exception e and ex as well. We check every simple name in a catch clause block. If it is e or ex, we do not report it as a potential bug. Otherwise, we check the block strings and if we have similar strings for a try block, we will report a potential bug. By this approach, we could reduce our extracted block to 30. We could miss some bugs that include e or ex, but they are not related to the exception. However, the probability of happening them is quite weak.

3 DESIGN OF THE TEST CASES

For each bug pattern, we designed some test cases to make sure of their correctness. We have used JUnit to test them. We will explain each of them bellow.

Bug pattern1

we designed 2 test cases to test the correctness of the code. in the first test case, we override both the equal and hashCode methods. In the second one, we just override the equal method. In this situation, a bug should be reported.

bug pattern2

We designed 3 test cases to test the correctness of the code. in the first test case, we open an Input stream and do not close it. In this situation, the tool should report a bug. In the second test case, we open a stream and close it, but not in a Finally block. like the previous test, the tool should report a bug. In the third test case, we check a valid situation. We open an Input stream and close it in the Finally block. The tool should not report a potential bug.

bug pattern3

We designed 3 test cases to test the correctness of the code. in the first test case, we just check the conditions with true or false in their condition. It means that we just check conditions like if(true) or while(true). In the second test case, we check conditions that check a Boolean variable. For example, Y is a variable. We design a test to check for conditions like if(Y) or while(!Y). In the third test case, we test a situation that should not report a bug. In this case, we just check a normal and usual situation.

bug pattern4
We designed 2 test cases to test the code. both of them include try catch with two catch blocks. In the first one, the catch blocks produce similar logs. In the second one, the contents of the catch blocks are the same, but they have e in their logs. It this situation the tools should not report a bug.

4 DETECTION RESULT

For each bug pattern, we run it on the cloud-stack, an open source repository on Github, and for each of them we have seen to the potential bugs.

4.1 Class defines equal but not hashCode

We obtained 7 bug reports. by manually investigating them we find out that at least three of them are false-positive. It means that they override the equal method to set some situations as false. It means that there is not any possibility to detect two objects equal while they are not. As we are not aware of the details of the code, we cannot get a certain opinion about the other 4 one, but it seems that they were true-positive.

4.2 Method may fail to close stream on exception

Finally, we could extract 118 bug reports in cloudstack. we manually and randomly inspected 20 of them. The code was very complex, as a consequence, we could not understand the code details, but we found out that 5 of them were related to a server as all of the servers are shutdown at the end of the process, so maybe they were false-positive. If so, the false-positive rate in this specific project is 25%. But the server process is not common and the false-positive rate is expected to be less than 10% in other projects. We also found that just 4 of them were related to finally blocks. In other cases, there is an IO stream object that is not closed at all.

4.3 Condition has no effect

We found out that all of the reports are true positive. This is because of our approach. We do not consider some situations to avoid a high rate of False-positive. So, all of the 20 manually inspected reports were true-positive.

4.4 Inadequate logging information in the catch blocks

By the first approach, we could extract 373. By choosing randomly 20 samples and inspecting them we found out that almost all of them (19 among 20) were false-positive. The reason is that most of them print exception e in their log and these logs for each e would be different, but our approach considered them as the same and reported a potential bug. By making the method more advance, we could reduce the extracted bugs to 30. We have manually inspected 20 them and found out at all of them and found out that most of them are true-positive. the rate of false-positive in the sample that we could find is 85%. we found out that the false-positive rate is because of our method. The method that we use does not consider nested blocks and try blocks. Sometimes there is a try block in another try block. In this situation, our approach may result in false-positive. There are also some blocks, about half of them, which do not produce any log. They just set a flag or return from the function. We also consider them as true-positive.

5 DISCUSSION

In this section, we summarize the lessons learned from this assignment

Abstract Syntax Tree. During this assignment, we got familiar with Java AST. We use the AST of the code to detect potential bugs during the system. We parsed the AST and looked for its nodes carefully. We also got familiar with binding a variable or any other object in java code. We also used AST view, a plugin of Eclipse, to visualize the AST of the code. We used it to obtain the name of each node and we have found it very helpful.

Junit. We learned how to work with JUnit. We designed test scenarios for each pattern and tested them using JUnit.

Bug Detections. We implemented 4 bug patterns during this assignment. For each one we started from a simple algorithm and tried to improve it using better algorithms and decisions. For example, for pattern 4, we developed two approaches to identify anomalies. However, we found out that sometimes the efficiency of a simple method is almost as good as the more complex one. Also, we found out that sometimes, we need to do a lot of work just to improve the efficiency of a method a little amount. For example, we know that our algorithm for bug pattern 2 has a problem and in some cases, it cannot detect the potential bugs. However, we have preferred not to change it. Because it needs a lot of developing time and also more computation time. So, we learned that by considering the time limitation of the development of the project it is better to use a simpler approach that is almost as good as the more complex approaches.

False-positive True-positive. During this assignment, we found out that there is a relation between true-positive and false-positive. A method with high true-positive comes with high false-positive. Likewise, a method with a low true-positive rate comes with a low false-positive rate. As a result, we should make a trade-off between them to get a better result. For each pattern we can also use different algorithms with different sensitivity to bugs and the developer could select each of them based on the result that expects.

Result evaluation. We run the tool on cloudstack project. We found out that there are lots of potential bugs in it that developers of the project didn't consider them. We understand that even this simple tool can prevent a lot of bugs before testing the projects in a real situation.

6 CONCLUSSION

As project and software development process become more complex, the testing process also becomes more expensive. Developers are always seeking for less costliest tools to test their code. They also need to be aware of the bugs of their code during the development process. In this assignment, we have implemented a static analysis tool to detect 4 patterns of potential bugs during the development of the code. We have designed test cases to prove the ability of our tool. We evaluate the performance of our tool by running it on Stackcloud (an open-source project on Github). We find out that in all of the patterns the false-positive rate is less than 50% and in 3 of them the false-positive rate is less than 30%.

REFERENCES

[1] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller. 2013. Model-Based Test Oracle Generation for Automated Unit Testing of Agent Systems. IEEE Transactions on Software Engineering 39, 9 (Sept 2013), 1230–1244. https://doi.org/10.1109/TSE.2013.10

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

Link to GitHub

https://github.com/emadfallahzadeh/BugDetector

link to Google Doc

https://docs.google.com/document/d/108hnFThJdecq1p4Tim-g7BzkHgoQ7aNlBzrGajH9XGU/edit