## Introduction

In this lab the objective was to learn more about static code analysis and get a feel for the type of analysis that can be done without ever actually executing the code by using analysis tools which help with identifying code issues.

# Aims/Objectives

* Install the required plugins used for analysing code from the Eclipse marketplace.
* Create a project and import the code for the Billing System.
* Check the code using each of the analysis tools previously installed.
* Choose two bugs in which to analysis for the purposes of this report.
* Determine whether the bugs are an actual fault or a false positive and if necessary, fix the code.
* Understand how static code analysis can be implemented into my own code to help me as a developer.

## Method

* Installed the PMD and SpotBugs analysis tools from the eclipse marketplace.
* Created a new project in Eclipse and downloaded the zip file containing the Billing System code from Blackboard and then extracted the files into the project.
* Checked the code for bugs using PMD and SpotBugs, analysed it and became familiar with the different levels of severity.
* In PMD, there are 5 different levels of severity shown as different coloured triangles as shown in Figure 1.

Red = Blocker – High, Aqua = Critical – Medium High, Green = Urgent – Medium, Pink = Important – Medium Low, Blue = Warning – Low.

* The issues are outlined in the program itself, in a violations overview window and in the package explorer as shown in Figure *2*.
* In SpotBugs, there are 4 different levels of severity. These include: Red Bug = Scariest, Orange Bug = Scary, Yellow Bug = Troubling and a Green Bug = Of Concern.
* In the case of the Billing System program there are only 3 levels present as shown in Figure 3 below. The bugs are also filtered again depending on the level of confidence give. This is determining how likely it is that the bug found is an actual fault within the system.
* After gaining some understanding of how the analysis tools work, I chose 2 bugs to analyse in further detail.
* The first bug I decided to analyse is shown below in Figure 4. This bug warns that the ‘==’ and ‘!=’ operators only checks for reference equality when dealing with String objects.
* The second bug I decided to analyse is shown below in Figure 5. This bug warns that the addCashier() method may fail to clean up the java.sql.Statement.
* I examined each of the chosen bugs and decided that both were actual faults in the program in which I had to implement changes to rectify.

## Results and Testing

* The analysis tools were easily found and installed from the Eclipse marketplace. There were no issues with creating the project in Eclipse and extracted the files to that location.
* The Billing System source code was easily accessible. I Used both tools to analyse the Billing System project.
* There were many potential issues, so some time was spent to get a feel for both analysis tools and to understand the different levels of severity. Two bugs were then chosen in which to analyse further:

*Bug 1:*

* In Figure 6 you can see the first bug chosen. It is marked as troubling in SpotBugs on line 67. Here the issue is that when comparing strings, the “==” operator checks whether each element references the same place in memory, but it does not actually check whether each of the strings contain the same string of characters.
* In the code, the objective of line 67 is to determine whether each of the fields contain an empty string and if so then handle that as some sort of error.
* However, in this case, the condition will always evaluate to false because the “==” operator is not comparing the actual strings for equality but rather for reference equality. So, knowing this, we can determine that this is an actual fault that will require a solution.
* In Figure 7, on line 68, the bug is handled. Using the equals method to compare two strings will achieve the required result as it checks the contents of each of the strings for equality.

*Bug 2:*

* In Figure 8 you can see the second bug chosen. The bug is highlighted on line 144 but from analysing the code and the bug further we can see that the problem does not lie with that line of code itself but that there is no statement.close() anywhere in order to close the stream of the database object.
* In Figure 9, this bug is handled by adding in the statement.close() line of code to the program which closes the stream.
* You could further protect the code by adding in a finally block to the try catch block where we could close both the statement and the conn.
* Adding that code here would ensure that these object stream would always be closed as the finally block is always run whether there is an error or not.

## Conclusion

I found the Eclipse marketplace easy to navigate and installing the plugins was very straight forward. There were no issues when checking for bugs with PMD but I did have some small issues when first trying to check the code for bugs with SpotBugs. It would start analysing the code and then would seem to be complete but there would be no analysis and also half of my classes would disappear from the Billing System project. There had been some network issues within the college, so I moved my project out of my one drive and from there I was able to use SpotBugs.

I chose two bugs which had actual faults because I wanted to implement the code changes to fix the faults and believe my solutions overcame the issues as when I re-checked the code with the changes implemented the bugs found previously were gone. However, by using these tools it has also given me some insight into areas in which I can research further and expand my knowledge.

I believe that the bugs found by the static code analysis are very helpful and should be highlighted by the system. In the case of the bugs chosen, I think both are important. For the first bug chosen, using the == operator on a String object, it is a common mistake in programming as in many cases such as comparing ints, it is sufficient in checking equality. However, for an object, it will only check for a reference to the same location in memory. This could cause quite an annoying issue within the program as you may believe that the original line of code is doing the job required. Having the static code analysis point out this possible flaw and, in this case, an actual bug with the system is very beneficial and I think it should indeed be found by the static code analysis. For the second bug chosen, a failure to clean up the java.sql.Statement object, I believe this is of importance as well as these stream object should always be closed and can sometimes be forgotten so I think that it too should always be found and highlighted in static code analysis.

I preferred using SpotBugs over using PMD as I just found it a little easier to navigate and get information from about the bugs. But overall I found the analysis tools very beneficial and easy enough to understand and think they will come in very useful in the future.

## **References & Bibliography**

Bhattacharjee, D. (2019) System.out.Println vs Loggers | Baeldung [online], available: https://www.baeldung.com/java-system-out-println-vs-loggers [accessed 23 Apr 2022].

Chess, B. and McGraw, G. (2004) ‘Static analysis for security, *IEEE Security Privacy*, 2(6), 76–79, available: https://doi.org/10.1109/MSP.2004.111.

Parahar, M. (2019) Differences between == and Equals() Method in Java [online], available: https://www.tutorialspoint.com/differences-between-and-equals-method-in-java [accessed 23 Apr 2022].

## **Appendices**

### Figure 1 – PMD severity levels.



### Figure 2 – PMD showing the different issues detected

Text

Description automatically generated

### Figure 3 - SpotBugs showing different issues detected

Graphical user interface, text, application

Description automatically generated

### Figure 4 – Bug 1

Graphical user interface, text, application

Description automatically generated

### Figure 5 – Bug 2

Graphical user interface, text, application, email

Description automatically generated

### Figure 6 – Bug on line 67 in the code

A screenshot of a computer

Description automatically generated with medium confidence

### Figure 7 – Code implemented to fix bug on line 68, original line 67 commented out.

Text

Description automatically generated

### Figure 8 – Bug highlighted on line 144 in the code

Text

Description automatically generated

.

### Figure 9 – Code implemented to handle bug on line 147.

A screenshot of a computer

Description automatically generated with medium confidence