Static Code Analysis Tools: Find Bugs and PMD

Aishwarya Rana and Alexander Innes

Texas State University

ABSTRACT

Static code analysis is a methodology to debug the program without executing it. Static analysis is only first step in a comprehensive software quality. Today in the fields of Software Development, Quality Assurance takes a massive amount of time to properly find and analysis for bugs as well as ensure constancy with coding methodologies across various programs. To help with that, various Static Code Analysis Tools have been developed to help developers with analyzing the source code and even ensure code constancy across programs and projects. In this report two Static Code Analysis Tools, FindBugs and PMD, are analyzed, and then are compared. FindBugs and PMD are open source static code analysis tool. To be specific, these tools are used for debugging the Java Code. First these tools were observed to categorize the error type and its pattern by testing on some source code obtained from GitHub. Then, these tools were evaluated on manually written program. Manually errors were injected according to error type and those error were evaluated with FindBugs and PMD. The experimental results showed that FindBugs and PMD are valuable tool as they can find real problems.

1. Introduction

This report is over the implementation and review of the Static Analysis Tools FindBugs and PMD. The reason for using these Static Analysis Tools is so that we can check the code without having to compile it. Thus, saving developer's time that might have be taken up by having to trace through and find bugs or managing code consistency.

Both tools come with a default set of rules which can be used to detect mistakes such as having empty try-catch blocks, variables that are never used, objects that are unnecessary, etc.

While FindBugs doesn't allow for customization of rule set, PMD on the other hand does allow users to customize and create new rules set. Another customization feature that is available to both tools, is the ability to weight each issue differently. The user can customize the weight of each issues in both tools, meaning that a user can define what they consider more or less sever with regards to each issue.

Finally, both tools also come with a relatively easy user command line interface and at the same time, can also be integrated with popular development environments such as Eclipse.

2. Overview of Tools

2.1 FindBugs

Inspects the Java Byte code, which is saved in the form of complied class files, to detect occurrences of bug patterns. It finds all possible bugs. Each finding is reported as warning. Bug patterns like potential bugs that are found during compiling, improper use of .equals() and hashCode(), unsafe, etc., these issues and more are the types that are generally found in the Byte Code and thus are found by FindBugs.

FindBugs customizability varies, one thing about FindBugs is that you can't create news rules set, in the case that the user wishes to use FindBugs to detect other issues or complications in the code is not applicable, meaning that the test sets that FindBugs releases or has are the ones that users have to work with, but FindBugs does allow users to customize the severity of the issues or bugs that are discovered by FindBugs. This customization allows user to judge which issues and cases that they feel may or may not severely impact the code that they are working on.

2.2 PMD

A tool that inspects the Java Source Code based on the evaluation rules that have been enabled during a given execution. The evaluation rules detect such errors like dead code, duplicated code, lack of curly braces, etc.

Customizability of the PMD is more flexible than that of FindBugs, in that while both FindBugs and PMD have a default set of rules, you can add and change the rule for PMD. Just as you can change the severity of bugs in FindBugs, you can also customize the severity of issues in PMD. This customization of the severity applies to both default rules set, and any rules sets that a user creates.

3. Overview of Analyzed Problems

3.1 FindBugs

FindBugs scans all possible bugs in Java Code. Each finding is reported as warning. All warnings are classified in four ranks: (1) scariest, (2) scary, (3) troubling and (4) of concern. This bug rank is a number between 1 and 20 and grouped in four values:

- Scariest (1-4),
- Scary (5-9),
- Troubling (10-14)
- Of Concern (rank 15-20)

The current version reports warning in ten categories:

Category	Description	Example
Bad Practice	Practices that violate recommended coding practices	Using "==" to compare string objects.
Dodgy	Code that is confusing, anomalous, and error-prone.	Dereferencing null without a prior null check.
Performance	Inefficient memory usage/buffer allocation, usage of non-static classes.	Creating a new String (String) constructor.
Correctness	Apparent coding mistakes.	Unsafe cast
Malicious code Vulnerability	Variables or fields exposed to classes that should not be using them.	Returning a reference to mutable object may expose internal representation
Internationalization	Use of non-localized methods	Use of non-localized String.toUpperCase
Multithreaded Correctness	Thread synchronization issues.	
Security	Similar to malicious code vulnerability	Passing a dynamically created string to a SQL statement.
Style	Bad coding practices	Assigns a value to a local variable, but the value is not read or used in any subsequent instruction

Table 1 Categories of FindBugs

Most of them were analyzed according to its category. Following are the description for few analyzed problem and its detail observation is explained in section 4.3.

- **Null pointer dereference** occurs when The return value from a method is dereferenced without a null check, and the return value of that method is one that should generally be checked for null. This may lead to a NullPointerException when the code is executed.
- **Unsafe Casts** occurs when code casts the result of an integral division (e.g., int or long division) operation to double or float. Doing division on integers truncates the result to the integer value closest to zero. The fact that the result was cast to double suggest that this precision should have been retained. What was probably meant was to cast one or both of the operands to double before performing the division
- **Array index out of Bound** occurs when Array operation is performed, but array index is greater than it is declared, which will result in ArrayIndexOutOfBoundsException at runtime.
- **Improper Use of equal() and hashCode()** occurs when class overrides equal(object), but not override the hashCode(), and inherits the implementation of hashCode() from java.langObject (which returns the identity hash code, an arbitrary value assigned to object by VM). Therefore, the class is very likely to violate the invariant that equal object must have equal hashCode.
- **Self-Assignment** occurs if the method contains a double assignment of a local variable or if the method contains a self-assignment of a local variable.
- **Infinite Loop** occurs when loop doesn't seem to have a way to terminate (other than by perhaps throwing an exception).
- **Ignoring Exceptions (Bad Practices)** occurs when a try-catch block that catches Exception objects, but Exception is not thrown within the try block, and Runtime Exception is not explicitly caught.

3.2 PMD

PMD scans the source code in order to generate reports on flaws that are found. This report classifies each flaw that is found in five separate categories, High, Medium High, Medium, Medium Low, Low, with distinct names given to each category. For High we refer to it as a Blocker, Medium High is a Critical, Medium is Urgent, Medium Low is Important, and Warning is Low.

- Blocker (High)
- Critical (Medium High)
- Urgent (Medium)
- Important (Medium Low)
- Warning (Low)

Rule Set	Description	Example
Basic	a collection of good practices which should be followed	Simplifying a for look to a while loop.
Braces	rules regarding the use and placement of braces	Lack of Curly Braces for if statements or loops
Design	rules that flag suboptimal code implementations. Alternate approaches are suggested	Using .equal() when you need to use ==, missing breaks or defaults in switch statements
Empty Code	find empty statements of any kind (empty method, empty block statement, empty try or catch block,...).	Empty Try-catch blocks, if statements, loops
Naming	rules regarding preferred usage of names and identifiers	Naming variable either too long or too short
Unused Code	unused or ineffective code	Variables or Objects that are created but never used.

Table 2 Rule Sets of PMD

PMD's issues were analyzed based on the Rule Sets. The following list are a few descriptions of Issues that have been observed and further detailed observation can be seen in 4.3

- **Dead Code** is variables that are unused in other parts of programs. This unnecessary code can mess with the consistency of the code and make reading code more difficult.
- **Using equals() instead of '=='**, some operators and conditions in JAVA either don't generate proper results or don't require the usage of .equals().
- **Overcomplicated Expressions** occurs when PMD determines if there are unnecessary if statements or for loops that could be while loops.
- **Violation of Naming Conventions**, is a rule that is set by PMD to determine the length that one names functions, classes and variables. This particular rule is flexible in that a user can determine if the naming of something is either too long or too short.
- **Lack of Curly Braces**, occurs when either conditions statements or loops don't have curly braces that show were they begin and were they end. This type of issues is more of a preferred style, meaning that some programmers don't like to use curly braces for single line condition statements/loops while others do. So this particular rule is one that a user can enforce.
- **Misplaced Null Check** is a condition that occurs when a user tries to check to see if an object is null, but is not required. The reason why this might not be required is that often times when this occurs a NullPointerException will be thrown. Due to the fact that checking for null will occur after the expression that is thrown.
- **Missing Break in Switch Statement** can cause issues in switch case statement. While this type of situation

could either be intentional, i.e. the users want to not have a break statement in switch cases, it still has the potential to be both a coding and a style error, meaning that it could potentially cause issues once the code is running and with regards to formatting of the switch statement is possible it was missed users.

4. Observations

4.1 Installing the tools

To install the FindBugs and PMD, Plugin for Eclipse, Eclipse 3.3 or later, and JRE/JDK 1.5 or later is required.

FindBugs

1. To install the FindBugs for Eclipse plugin:
2. In Eclipse, Help -> Install New Software...
   a. Click on **Add...**
   b. **Name**: FindBugs
   c. **URL**: http://Findbugs.cs.umd.edu/eclipse
3. Click OK**.**
4. You should see **FindBugs** in the list. Select checkbox next to it and click **next.**
5. You'll need to accept the license and confirm you want to install a plugin that is not digitally signed. Go ahead and install it anyway.
6. Restart eclipse.

PMD

1. To install the PMD for Eclipse plugin:
2. In Eclipse, click on Help -> Install New Software...
   a. Click on **Add...**
   b. **Name:** PMD
   c. **URL:** https://dl.bintray.com/pmd/pmd-eclipse-plugin/updates/
3. Click **OK**.
4. You should see **PMD for Eclipse 4**. Select the checkbox next to it and click **next.**
5. You'll need to accept the license and confirm you want to install a plugin that is not digitally signed. Go ahead and install it anyway.
6. Restart eclipse.

4.2 Learning the tools

FindBugs

To get started, right click on a Java project in Package Explorer, and select the option labeled "Find Bugs". FindBugs will run, and problem markers (displayed in source windows, and also in the Eclipse Problems view) will point to locations in your code which have been identified as potential instances of bug patterns.

You can also run FindBugs on existing java archives (jar, ear, zip, war etc.). Simply create an empty Java project and attach archives to the project class path. Having that, you can now right click the archive node in Package Explorer and select the option labeled "Find Bugs". If you additionally configure the source code locations for the binaries, FindBugs will also link the generated warnings to the right source files.

You may customize how FindBugs runs by opening the Properties dialog for a Java project, and choosing the "Findbugs" property page. Options you may choose include:

- Enable or disable the "Run FindBugs Automatically" checkbox. When enabled, FindBugs will run every time you modify a Java class within the project.
- Choose minimum warning priority and enabled bug categories. These options will choose which warnings are shown. For example, if you select the "Medium" warning priority, only Medium and High priority warnings will be shown. Similarly, if you uncheck the "Style" checkbox, no warnings in the Style category will be displayed.
- Select detectors. The table allows you to select which detectors you want to enable for your project.
- Existing standard FindBugs detector packages can be configured via
  Window → Preferences → Java → FindBugs → Mi sc. Settings → Custom Detectors. Simply specify their locations of any additional plugin libraries.

PMD

To start PMD, right click on a Java project in Package Explorer, and select the option label “PMD”. PMD will then run, generate a marker on problem code and display a PMD explorer that will show the priority of the issue, the line where the issue was found, the timestamp, the rule that the issues is violating, and an error message.

Customization of PMD is done by opening the Properties dialog for a Java project, and choosing the “PMD”. In order to allow customization, you must check the checkbox Enable PMD, otherwise PMD will use the default ruleset that came with it. If you’d like PMD to check derived files you will hit the checkbox Include Derived Files.

Another customization feature is Handling High Priority Violations as Eclipse Errors or Full build enable, you can enable or disable those at your leisure.

Finally, you can select the working set you want to use or deselect the working set you don’t want, as well as configure the ruleset you want to use by either checking the checkbox of the desired ruleset or unchecking the checkbox of the ruleset you don’t want.

Now, PMD does have a more in-depth customization by allowing users to create and manage their own rules this can be done by going to Preferences and select the “PMD” tab.

This tab allows for all kinds of customization, but the main tab that handles most customization of new rules is the “Rule Configuration” tab. It is here where you can remove old rules or add new rules be either importing rules or creating your own via Rule Designer, see Figure 1.

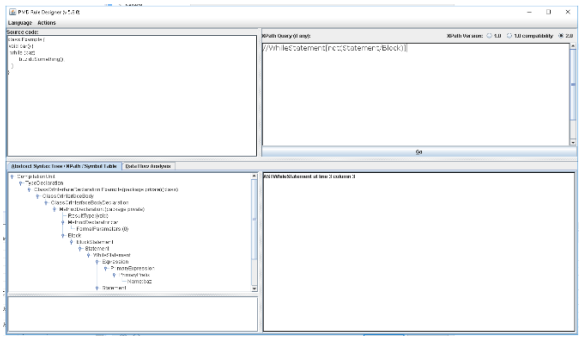

Figure 1 Rule Designer UI

4.3 Analyzing the Problems Using the Tools

FindBugs

To analyze the problem using FindBugs, Some GitHub examples were taken. 8 online source codes were evaluated. Out of then 4 had at least 200 LOC, one was specially coded to detect improper use of equals() and hashCode() and other 3 were small size programs. These codes were run in Windows 10, 16 GB RAM Intel processor i7 7th generation CPU 2.70 GHz 64-bit processor and Eclipse Version: Oxygen.1a Release (4.7.1a). FindBugs was successful to analyze the error according to its category. Various categories of warning were observed on those 8 programs extracted from GitHub. After that, to analyze specific problem, manually codes were written, and unnecessary error were injected to observe other potential issue that FindBugs can detect. Also, some different errors were detected which reflects one of the pattern type and named the category as "Experimental" but that is not yet included in the category of FindBugs. Still its usefulness of bug pattern is under analysis. A table in section 4.4 shows the error count according to its category which was experimented on those 8 projects. Also, potential error detected with respect to its rank, pattern and confidence were observed, which is further shown in section 4.4. Following are some detail analysis of the problem whose descriptions were already studied in section 3.1.

Null Pointer Dereference

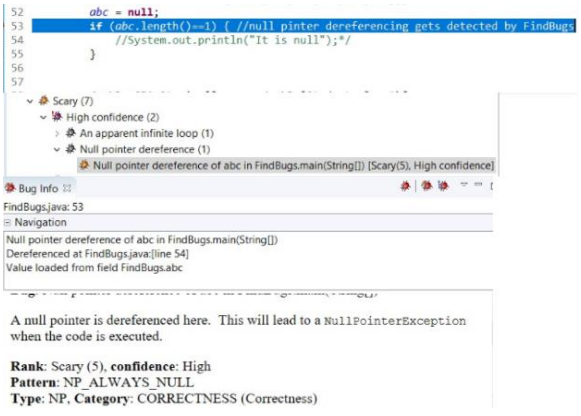

Figure 2

Unsafe Casts

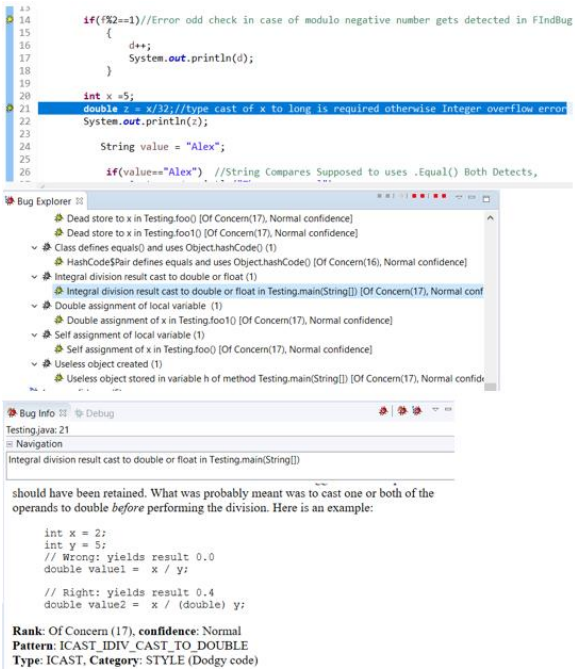

Figure 3

Improper User of equals() and hashCode:

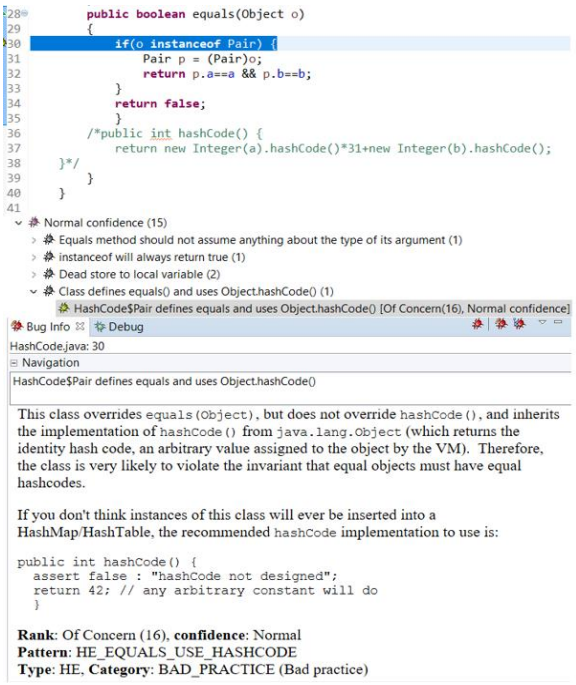

Figure 4

Array out of bound

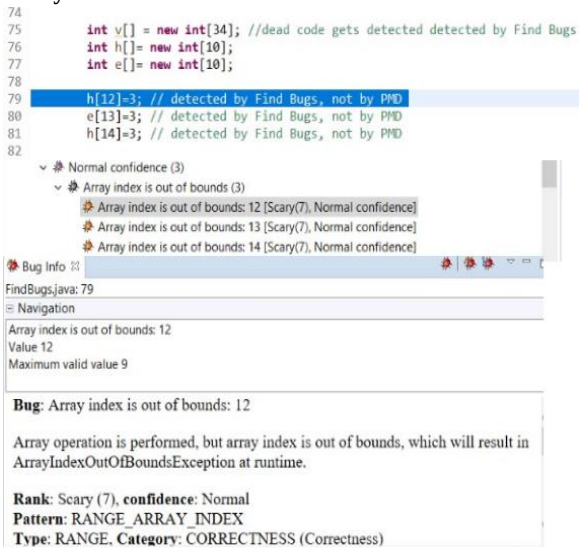

Figure 5

Self-Assignment

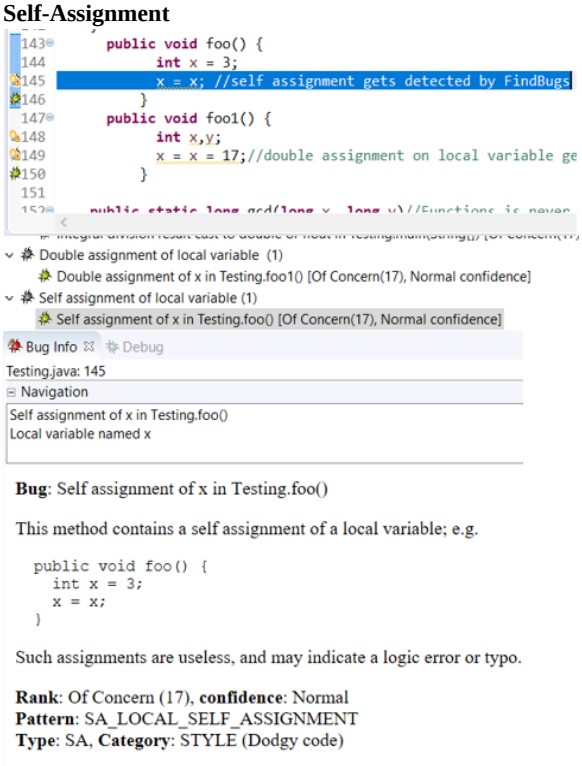

Figure 6

Infinite Loop

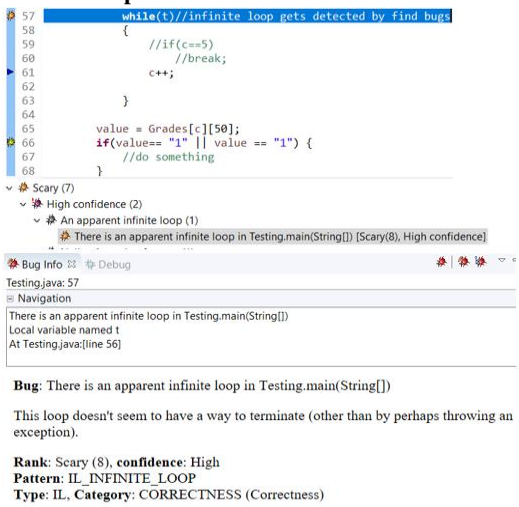

Figure 7

PMD

For PMD's analyze the same GitHub examples were used. The same 8 online source codes were used for the evaluation of PMD, however only one small size program was created to test and check some of the issues. The codes were ran using the same specs as FindBugs, i.e. Windows 10, 16 GB RAM Intel processor i7 7th generation CPU 2.70 GHz 64-bit processor and Eclipse Version: Oxygen.1a Release (4.7.1a). PMD's analyzation of the error was successful, and was created based on its Rule Set. Like FindBugs, PMD generated various warnings when it was running on the 8 programs extracted from GitHub, however unlike FindBugs, PMD generates warnings based on Rule Sets.

Following the same analyzing pattern that was used for FindBugs, after analyzing specific problem found on the GitHub projects, one manually written program was created, and unnecessary error were injected to observe other potential issue that PMD can detect.

While FindBugs showed a previously unknown category called "Experimental", PMD issues is that it has to many rulesets implemented that can be consider unnecessary and generally not helpful with regards to analyzing the code so these particulars ones where removed from rule set. A few examples of the rulesets that were removed were LawOfDemeter and SystemPrint. These two rules are generally used for removing statements that might have been used during debugging, but this is not the case either the GitHub or the manually written code.

Below are the screenshots that show some of the issues that were injected into the manual code as well as PMD finding and categorizing them.

Further analyzes can be seen with the table, section 4.4, containing the Rule Set and severity of the issues found with PMD, with regards to the 8 GitHub projects

Dead Code

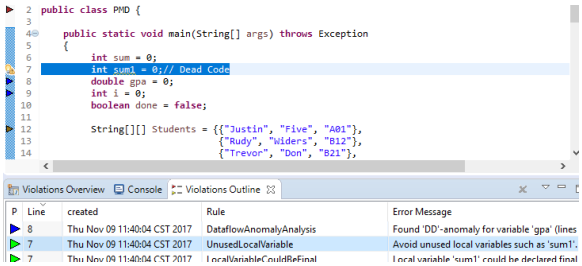

Figure 8

Using equals() instead of '=='

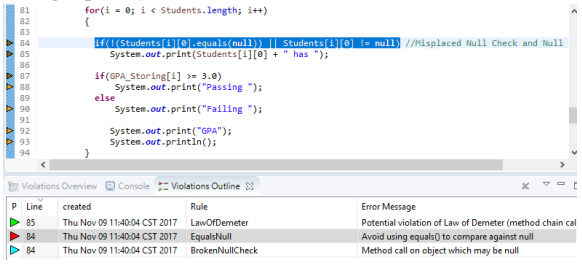

Figure 9

Overcomplicated Expressions

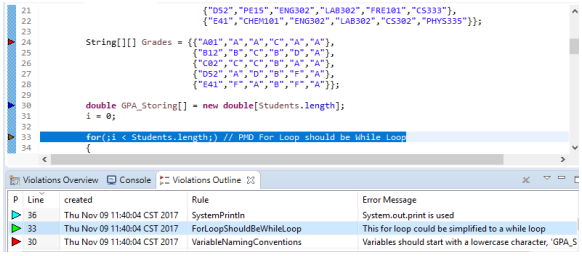

Figure 10

Violation of Naming Conventions

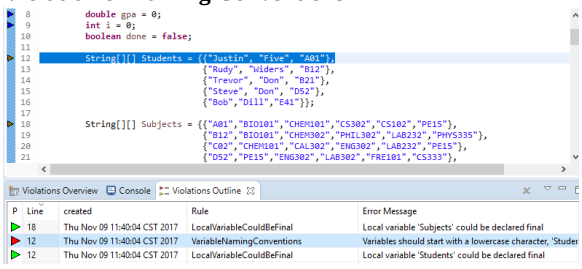

Figure 11

Lack of Curly Braces

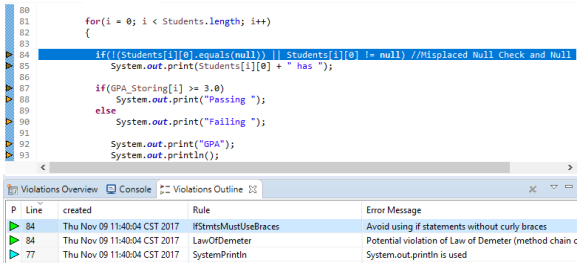

Figure 12

Misplaced Null Check

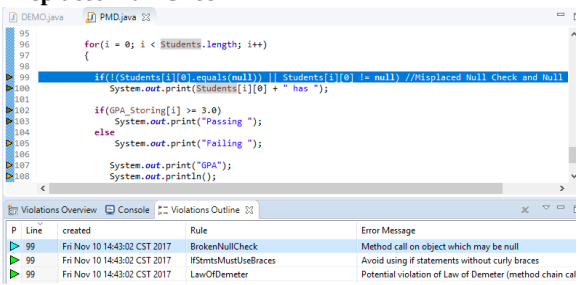

Figure 13

Missing Break in Switch Statement

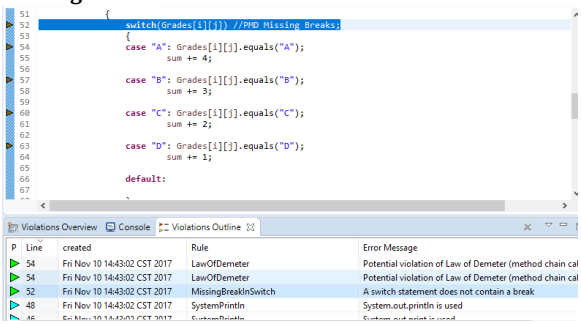

Figure 14

4.4 Analysis of Results

For the deeper analysis of the results in both the tool, our main aim was to answer six analysis question:

- **AQ1: What does FindBugs detect and PMD cannot?**
- **AQ2: What does PMD detects and FindBugs cannot?**
- **AQ3: What can both detect?**
- **AQ4: What cannot both detect?**
- **AQ5: Strength and weakness of FindBugs**
- **AQ6: Strength and weakness of PMD.**

Since, FindBugs can detect bugs such as unicast, self-assignment, infinite loop, synchronization and multithreading whereas PMD cannot. Likewise, PMD can detect issues related to style such as if/else must use braces, no package but FindBugs cannot. These gives answer for AQ1 and AQ2. Similarly, both can detect missing break in switch cases, string comparison (use .equals()to compare strings), dead code and something both cannot detect such as functions that is never used , double semicolon, divide by zero, repeated conditional test which then answer AQ3 and AQ4.

The Static Code Analyze Tools, both analyze code, but in different ways. FindBugs searches the byte code for issues, while PMD detects issues based on the Source Code and Style of the source code. If an issue affects both the source code and byte code, both FindBugs and PMD will

detect it. However, if an issue doesn't cause problems with either the source code or in the byte code, then neither PMD nor FindBugs will detect them.

FindBugs

Upon analyzing the projects with FindBugs, there were total 43 bugs found in 7 GitHub projects, 4 bugs in HashCode and 21 in manually written code. Those errors were analyzed and resulted in those categories. These bugs results giving us the information about how severe it was and, its pattern of error type, its confidence and how can it be solved.

Categories\Severity	Scariest	Scary	OfConcern	Troubling
Correctness	3	10	0	2
Style (Dodgy Code)	0	0	2	0
Bad Practices	0	1	1	2
Dodgy Code	0	0	8	0
Security	0	2	0	1
Performance	0	0	3	0
MT-Correctness	0	0	0	3
Experimental	0	0	3	0
Total	3	13	17	10

Table 3 Number Bugs found based on Categories and Severities in GitHub Projects using FindBugs

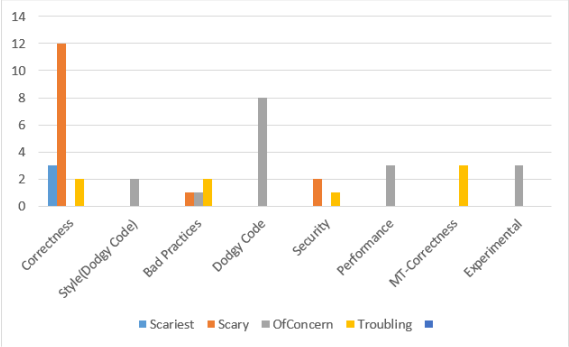

Figure 15

Hash Code that had altogether 4 bugs out of which 1 was in correctness category (scariest), 2 was in Style category (Of Concern and 1 was Bad Practice (Of Concern). And, manually written code most of the bugs were in either correctness or Style (Dodgy Code) category.

PMD

For PMD the total amount of issues that were 97 for the GitHub project and for the manual code 23. These issues were analyzed and sorted into the Rule Sets that they violate, as seen in Table 4.

Rule Set/Severity	High	Medium High	Medium	Medium Low	Low
Basic	0	0	5	0	0
Braces	0	0	2	0	0
CodeSize	0	0	2	0	0
Comment	0	0	1	0	0
Controversial	0	0	15	0	0
Design	0	0	19	0	0
Empty	0	0	19	0	0
Imports	0	1	0	0	0
JavaBeans	0	0	2	0	0
Strict Exceptions	0	0	13	0	0
Strings	0	0	13	0	0
Style	9	0	0	0	0
Unnecessary	0	0	1	0	0
Unused Code	0	0	5	0	0
Total	9	1	97	0	0

Table 4 Number of Issues found based on Rule Sets and Severities in GitHub using PMD

Based on the above information, it is obvious that FindBugs and PMD both are useful tool to apply to projects to analyze code and catch errors which otherwise can be missed with code reviews and inspections. While using FindBugs and PMD we realized that static analysis is a very useful method for finding defects in code. Trying to find these defects manually or by code inspection would have taken a lot longer than the automated checking that FindBugs and PMD provided. It also helps users to find common pieces of bad code and avoid them in the future. But as it only works for code and cannot find runtime issues. Also upon inspections it has been noticed that even eclipse will find issues/errors (like obsolete methods). It is difficult to decrypt FindBugs errors and PMD issues if you are not a developer, reason being is that the both FindBugs and PMD is geared specifically towards developers and helping to improve upon developers coding methodologies and as well as ensuring code consistency. Another feature in PMD is that we can add/modify our own rulesets, run mixtures of rulesets for each run, use optional arguments to run PMD on codebases using new (and old, for regression testing) JDKs, etc., but to add code to create a new rule, it requires an understanding of ASTs.

4.5 SUGESSTED IMPROVEMENTS

FindBugs

FindBugs has its own defined coding style (naming conventions of methods etc.). There should be a way to customize the coding style so that it can analyze the code against a custom coding style specified by the user. It will not always be the case that the user will be coding in the style defined in FindBugs.

PMD

PMD, only use the source code to find and detect issues, and the issues that it generally detects are those that are considered style issues within the code. So even if an issue that is found in the byte code is detect, it means that the issues is also a style problem, and even then, the styling issues might not be relevant to the byte code. A way to improve PMD would be to incorporate a way for PMD to

detect issues in both the source code and the byte code, instead of just limiting the detection capability to the source code.

5. CONCLUSION

In conclusion both tools show their value in they help increase the quality of code a programmer is working on. FindBugs is better at eliminating false positives and catching bugs and issues that occur during compiling. While PMD is better for styles and structural code consistency. Both tools while they have their strengths are better used in tandem, when using them to analyze the code. This helps to eliminate issues that could occur in both the byte code and the source code.

REFERENCES

[1] http://continuousdev.com/2015/08/checkstyle-vs-pmd-vs-findbugs/

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

[3] http://www.methodsandtools.com/tools/findbugs.php

[4] http://www.sw-engineering-candies.com/blog-1/findbugstmwarningsbysample-parti

[5] https://www.javatips.net/blog/pmd-in-eclipse-tutorial

[6] https://pmd.github.io/pmd-5.7.0/pmd-java/rules/index.html

[7] http://thewebplant.com/findbugs-description-of-the-categories-the-error-codes/

[8] http://www.baeldung.com/intro-to-findbugs

[9] http://www.sw-engineering-candies.com/blog-1/findbugstmwarningsbysamplenonnullandcheckreturnvalueofjsr-305

[10] http://fbenoit.blogspot.com/2016/12/eclipse-annotation-based-null-analysis.html

[11] https://gist.github.com/aishwaryarana01/6b30b372baa6dca44edc1e2fc1122659

[12] https://github.com/MarkusSprunck/findbugs-warnings-by-example/tree/master/findbugs_samples/src/com/sw_engineering_candies