EECS 4313: Bug Detection Tool Project

Group 7

Sanchit Duggal san249@my.yorku.ca

Kamsi Idimogu kidimogu@my.yorku.ca

Shamar Pryce spryce32@my.yorku.ca

Aanchal Sapkota aanchal0@my.yorku.ca

Jay Sharma jksharma@my.yorku.ca

Vrushank Vaghani vrushank@my.yorku.ca

Table of Contents

Abstract

This report outlines a holistic and thorough application of three diverse bug detection tools on five open-source health-based software projects. With an emphasis on detecting bugs for cost-saving purposes and enhancing software reliability and scalability, our objective was to identify and resolve bugs with each team member conducting individual tests in Java. Through this effort, we aimed to bolster the integrity of health software and contribute to bug detection methodologies. Our findings highlight the nuanced nature of bugs and underscore the critical role of bug detection in ensuring software efficacy and reliability.

Introduction

In the realm of software development, bug detection stands as a critical process, crucial for ensuring the functionality of software systems. Based on the shared experiences during the COVID-19 pandemic, the team decided to increase the integrity of health based software. The team researched a comprehensive list of bug detection tools and selected Findbugs, Checkstyle, and Randoop for diversity of style and Health Plus, Stop Coding, IBM Openshift, Health Care Service, and Kardio Master for their diversity of project size and scope.

Acknowledging the paramount importance of bug detection in saving costs and streamlining the maintenance phase of the software cycle, our objective is to adeptly apply, comprehend, and rectify bugs identified within five distinct health-based open-source projects with over 100 stars on GitHub each. The structure of our approach is meticulously organized, with each team member undertaking individual tests within the Java programming language domain.

Selecting Subject Projects

In the meticulous evaluation process aimed at identifying and assessing healthcare-related software projects, a deliberate and strategic approach was undertaken to ensure a comprehensive understanding of the landscape. The selection criteria were multifaceted, emphasizing not only the intrinsic relevance of each project to the healthcare sector but also its operational diversity and the degree of active engagement.

The first project selected was Health Plus which has a robust codebase comprising 21,488 lines and 54 classes [1], Health Plus signifies a sophisticated and comprehensive solution tailored to address a myriad of operational needs within healthcare facilities. The substantial size of its codebase suggests a rich array of functionalities designed to streamline various aspects of hospital operations. Furthermore, its moderate level of popularity, as indicated by 169 stars and 112 forks, underscores its relevance and utility within the developer community, hinting at its potential for further expansion and adoption. Secondly, the team selected Stop Coding, on the other hand, distinguishes itself through its unwavering commitment to safeguarding patient data privacy. Despite its medium-sized codebase of 3,571 lines and 53 classes [2], this open-source project has garnered significant attention within the community, as evidenced by its 155 stars and active engagement through bug report submissions. Such responsiveness to user feedback underscores the project's dynamic nature and its continuous efforts to enhance user security in healthcare. Thirdly, IBM Openshift was selected which has 1,665 lines and 15 classes [3] and is part of a larger multinational framework from IBM (International Business Machines Corp.) and 134 stars. Fourthly, there is Health Care Service, despite its modest size of 348 lines, 14 classes and 242 stars [4], plays a pivotal role in backend medical systems, underscoring the critical importance of efficient data management in healthcare operations. Lastly, Kardio Master addresses the user-facing aspect of healthcare technology through its direct-to-consumer health application. Despite its relatively compact codebase of 1,393 lines and 21 classes, this open-source project has garnered considerable acclaim, as evidenced by its 212 stars and contributions from 183 developers [5]. By adopting a strategic approach that encompasses projects of varying sizes, scopes, and developmental stages, this evaluation process provides a comprehensive and nuanced understanding of healthcare software.

Selecting Detection Tools

In the process of selecting bug detection tools for our projects, we established specific criteria to ensure the effectiveness and efficiency of these tools. The most important factors were their suitability for Java, user-friendliness, and popularity among developers, along with strong community backing.

The primary criteria was suitability for Java. Given that our project focuses solely on Java applications, it was crucial to choose tools specifically suited for the Java environment. This specificity is essential in identifying vulnerabilities and bugs. The next criteria that was just as crucial was the user-friendliness of the tools. Due to the varying levels of experience in our team, it was crucial for us to have user-friendly interfaces and easily understandable documentation. We preferred tools that could smoothly fit into our current workflow and could be easily implemented without need for extensive training. Finally, the tools' popularity and level of community support were important factors to take into consideration. Commonly used tools tend to be more dependable. Additionally, a lively community showcases a plethora of collective knowledge, resources, and assistance.

The process of selecting these tools was thorough, consisting of multiple steps. First, we put together a long list of bug detection tools that are well-known in the industry, which includes those suggested in our lectures. In order to simplify the selection process, we grouped these tools according to their main functions, such as static analysis, dynamic analysis, test generation, and AI-based analysis. This classification guaranteed an even mix of tools able to identify many different types of potential bugs.

We then narrowed our focus to tools specific to Java in order to meet the needs of our projects. This process consisted of testing these tools to see if they were compatible with our project codebases and evaluating their ease of use. Tools that were difficult to integrate or had a high learning curve were given lower priority. We also closely examined the level of community support and approval for these tools, assessing them by the thoroughness of their documentation, and how often patches were released. Some of the tools we considered were FindBugs, Randoop, Checkstyle, PMD, DeepCode, SonarQube.

The final selection of tools after the selection process were Randoop, Checkstyle, and FindBugs. FindBugs, a vital static analysis tool, was essential for its thorough Java-specific pattern-based analysis [6], necessary for detecting subtle yet important bugs. Checkstyle was chosen because of its emphasis on code quality [7] and consistent style, which are both essential for the project's long-term maintainability [8]. Randoop was selected for its capability of generating high-quality test cases automatically, offering potential time and efficiency savings [9].

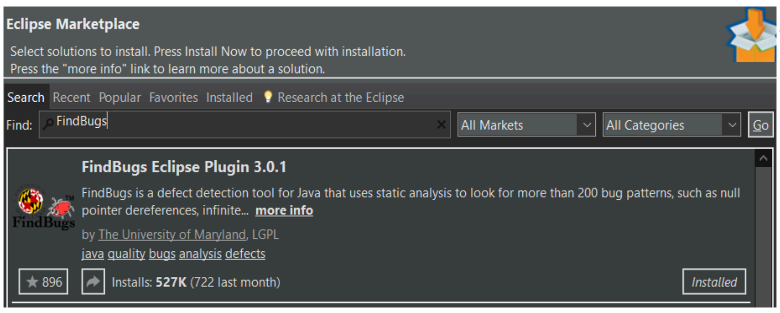
Tool	Purpose	Description	Reason for selection
Randoop	Test Generation	Automatically generates unit tests for Java classes by using feedback-directed random test generation	Excellent for creating high-coverage test cases with minimal manual effort, enhancing our testing efficiency
Checkstyle	Code Quality	Analyzes Java code to ensure it adheres to coding standards and practices	Helps maintain code consistency and readability, which is vital for long-term maintainability of projects
FindBugs	Static Analysis	Scans Java bytecode to identify potential bugs based on patterns of known bug definitions	Popular for its effectiveness in catching common Java bugs, offering deep static analysis capabilities

Figure 1: List of tools used with qualitative aspects of each bug detection tool selected

Applying Tools

FindBugs

Since FindBugs can be used as a plugin in the Java Eclipse IDE, it can be found in the “Eclipse Marketplace” after selecting the “Help” tab at the top of the application. After typing “FindBugs” into the search, it will appear and may then be installed as shown in Fig 2.



Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Search Recent Popular Favorites Installed Research at the Eclipse

Find: FindBugs All Markets All Categories Go

FindBugs Eclipse Plugin 3.0.1

FindBugs is a defect detection tool for Java that uses static analysis to look for more than 200 bug patterns, such as null pointer dereferences, infinite... more info

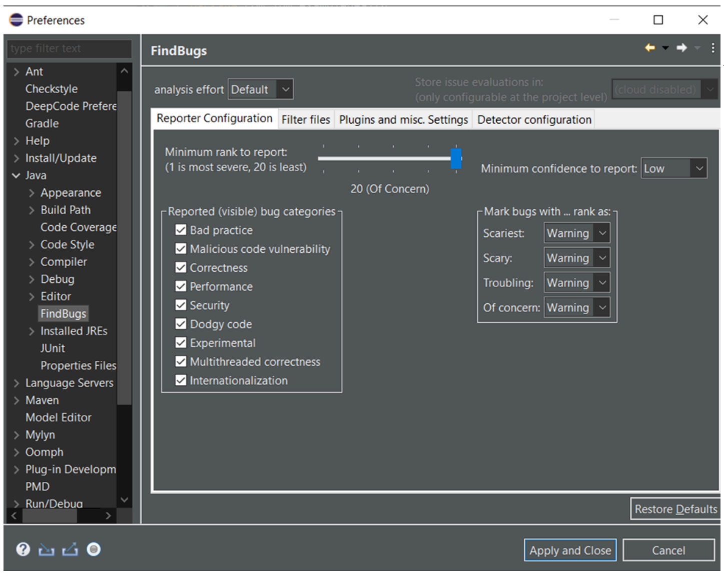
by The University of Maryland, LGPL

java quality bugs analysis defects

896 Installs: 527K (722 last month) Installed

Figure 2: FindBug plugin image

Once the installation is completed, a filter can be found in the Window -> Preferences tab in the IDE. Here, after selecting FindBugs, a window will appear that allows users to choose which types of bugs they want to be flagged, the sensitivity of the bugs they want to be flagged, and how confident the tool is that what they flag is an actual bug as shown in Fig 3. To make sure bugs appeared during our testing, we chose to flag every kind of bug and set sensitivity and confidence of bugs found to the lowest.



Preferences

FindBugs

analysis effort: Default

Store issue evaluations in: (only configurable at the project level) (cloud disabled)

Reporter Configuration | Filter files | Plugins and misc. Settings | Detector configuration

Minimum rank to report: (1 is most severe, 20 is least) 20 (Of Concern)

Minimum confidence to report: Low

Reported (visible) bug categories
- [x] Bad practice
- [x] Malicious code vulnerability
- [x] Correctness
- [x] Performance
- [x] Security
- [x] Dodgy code
- [x] Experimental
- [x] Multithreaded correctness
- [x] Internationalization

Mark bugs with ... rank as:

Scariest:	Warning
Scary:	Warning
Troubling:	Warning
Of concern:	Warning

Figure 3: Altering preferences image

Once the filter is selected, to run FindBugs, right click on the chosen Maven project and select the "FindBugs" option as shown in Fig 4. This will open a tab at the bottom of the IDE called the "Bug Explorer". Located here are all the bugs the tool found in the given project separated by confidence level. Each bug will show the type of bug it is, its sensitivity level, confidence level, and once double clicked, its location inside the project as shown in Figure 5.

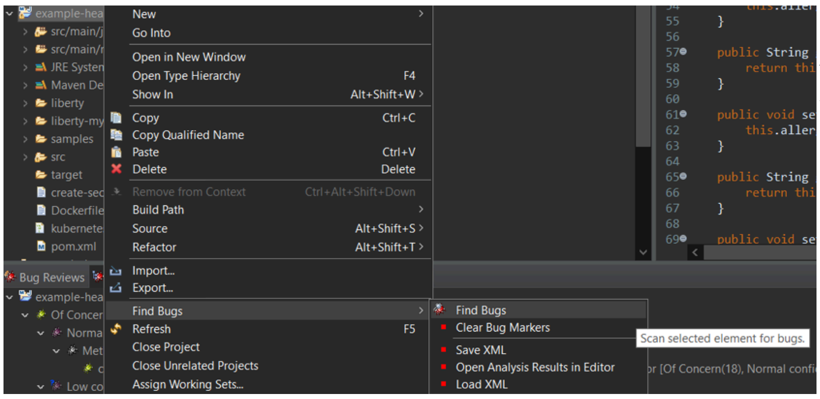
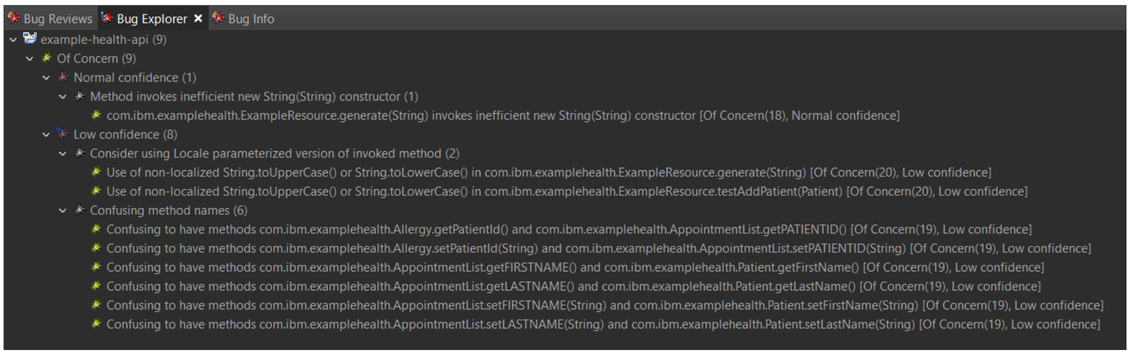


Figure 4: Executing findbugs screenshot image in Eclipse



Bug Reviews | Bug Explorer | Bug Info

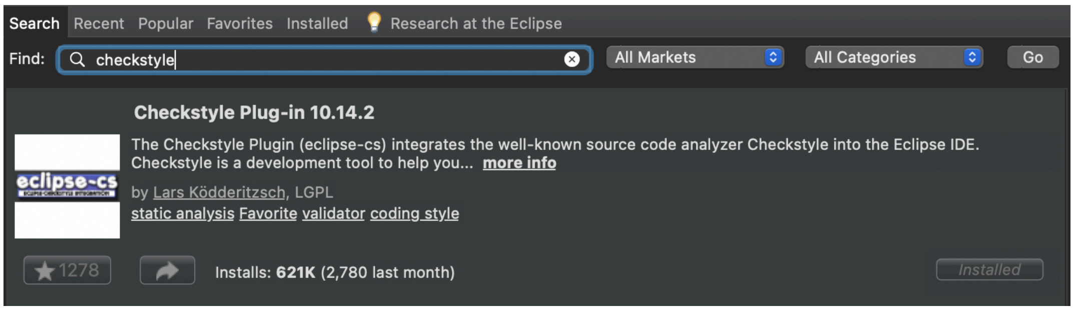
```
example-health-api (9)
  Of Concern (9)
    Normal confidence (1)
      Method invokes inefficient new String(String) constructor (1)
        com.ibm.examplehealth.ExampleResource.generate(String) invokes inefficient new String(String) constructor [Of Concern(18), Normal confidence]
    Low confidence (8)
      Consider using Locale parameterized version of invoked method (2)
        Use of non-localized String.toUpperCase() or String.toLowerCase() in com.ibm.examplehealth.ExampleResource.generate(String) [Of Concern(20), Low confidence]
        Use of non-localized String.toUpperCase() or String.toLowerCase() in com.ibm.examplehealth.ExampleResource.testAddPatient(Patient) [Of Concern(20), Low confidence]
      Confusing method names (6)
        Confusing to have methods com.ibm.examplehealth.Allergy.getPatientId() and com.ibm.examplehealth.AppointmentList.getPATIENTID() [Of Concern(19), Low confidence]
        Confusing to have methods com.ibm.examplehealth.Allergy.setPatientId(String) and com.ibm.examplehealth.AppointmentList.setPATIENTID(String) [Of Concern(19), Low confidence]
        Confusing to have methods com.ibm.examplehealth.AppointmentList.getFIRSTNAME() and com.ibm.examplehealth.Patient.getFirstName() [Of Concern(19), Low confidence]
        Confusing to have methods com.ibm.examplehealth.AppointmentList.getLASTNAME() and com.ibm.examplehealth.Patient.getLastName() [Of Concern(19), Low confidence]
        Confusing to have methods com.ibm.examplehealth.AppointmentList.setFIRSTNAME(String) and com.ibm.examplehealth.Patient.setFirstName(String) [Of Concern(19), Low confidence]
        Confusing to have methods com.ibm.examplehealth.AppointmentList.setLASTNAME(String) and com.ibm.examplehealth.Patient.setLastName(String) [Of Concern(19), Low confidence]
```

Figure 5: Bug explorer screenshot of FindBugs in Eclipse image

Overall, due to FindBugs being a plugin in Eclipse, installing and applying it to projects was simple and did not provide any difficulties.

Checkstyle

Installing Checkstyle on an IDE is a straightforward process. For Eclipse, the process begins by accessing the Eclipse Marketplace, a platform where you can find various software development plugins and tools. Within the marketplace, search for 'Checkstyle' in the search bar. Once located, click on 'Install' to install the plugin which is followed by agreeing to the terms and conditions before the plugin is accessible in the eclipse environment. After installing, the next step consists of configuring the plugin by aligning the settings to the desired coding convention. For our project, the coding convention that was tested against is, Google checks.



Find: checkstyle

Checkstyle Plug-in 10.14.2

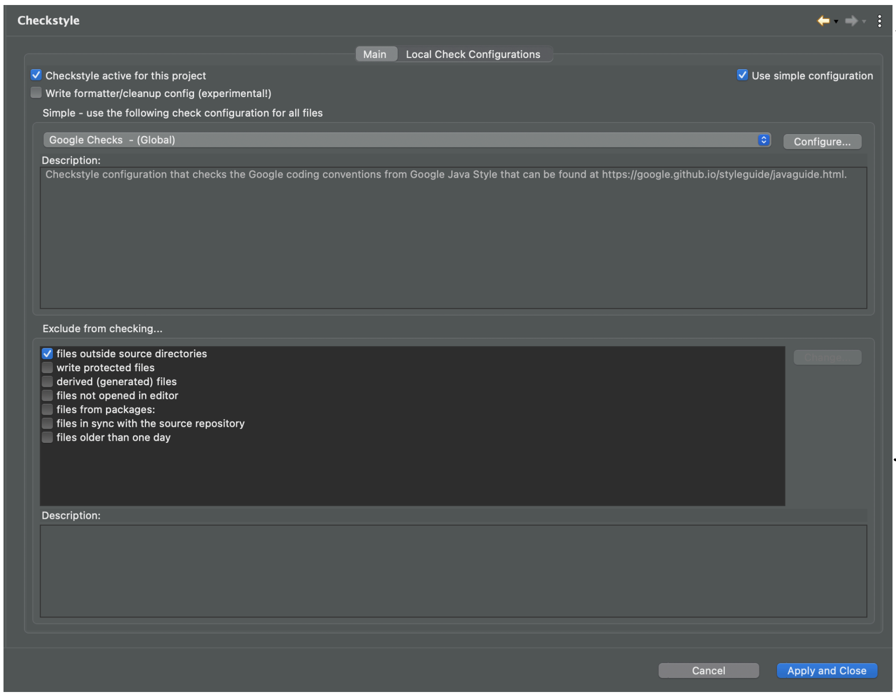
The Checkstyle Plugin (eclipse-cs) integrates the well-known source code analyzer Checkstyle into the Eclipse IDE. Checkstyle is a development tool to help you... more info

by Lars Ködderitzsch, LGPL

static analysis Favorite validator coding style

Installs: **621K** (2,780 last month)

Figure 6: Checkstyle installation process screenshot



☑ Checkstyle active for this project
☐ Write formatter/cleanup config (experimental!)
☑ Use simple configuration

Simple - use the following check configuration for all files

Google Checks - (Global)

Description:
Checkstyle configuration that checks the Google coding conventions from Google Java Style that can be found at https://google.github.io/styleguide/javaguide.html.

Exclude from checking...

☑ files outside source directories
☐ write protected files
☐ derived (generated) files
☐ files not opened in editor
☐ files from packages:
☐ files in sync with the source repository
☐ files older than one day

Description:

Figure 7: Checkstyle configuration screenshot

Once the settings have been configured, the activation process is quite simple, right click on the project and click on 'Activate Checkstyle'. Upon activation, Checkstyle is integrated to the project and begins analyzing the java code in real time. The results of the analysis are presented at the bottom of the ide on a tab called 'Checkstyle Violations' where the plugin provides the 'Violation Type', the occurrences and where the violation occurs. This allowed for a seamless identification of all the areas of code where the violations that occurred against the coding conventions.

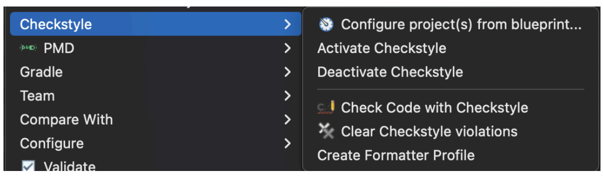
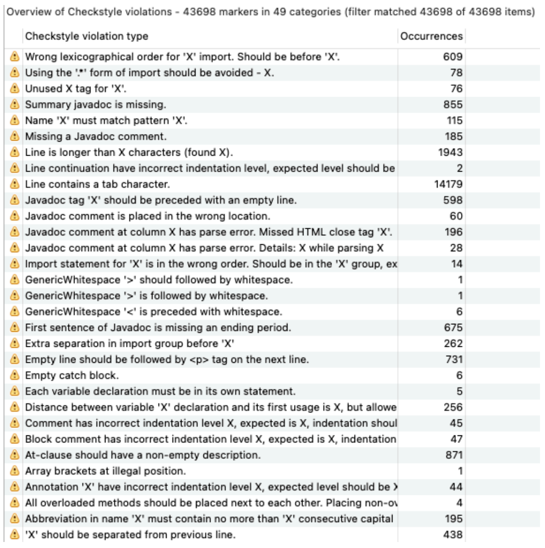


Figure 8: Checkstyle execution screenshot



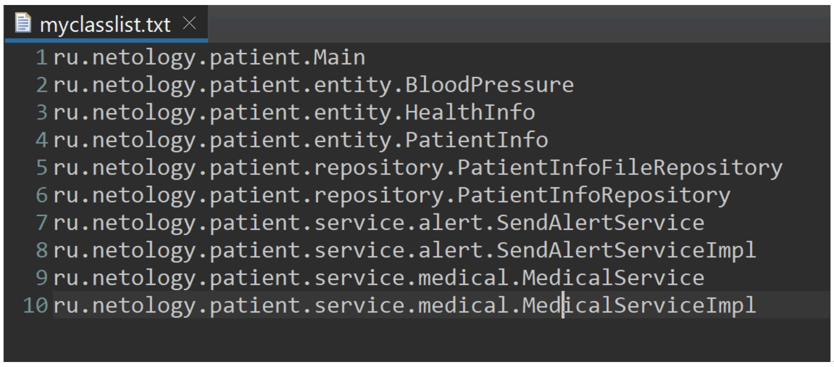
Overview of Checkstyle violations - 43698 markers in 49 categories (filter matched 43698 of 43698 items)

Checkstyle violation type	Occurrences
Wrong lexicographical order for 'X' import. Should be before 'X'.	609
Using the '.*' form of import should be avoided - X.	78
Unused X tag for 'X'.	76
Summary javadoc is missing.	855
Name 'X' must match pattern 'X'.	115
Missing a Javadoc comment.	185
Line is longer than X characters (found X).	1943
Line continuation have incorrect indentation level, expected level should be	2
Line contains a tab character.	14179
Javadoc tag 'X' should be preceded with an empty line.	598
Javadoc comment is placed in the wrong location.	60
Javadoc comment at column X has parse error. Missed HTML close tag 'X'.	196
Javadoc comment at column X has parse error. Details: X while parsing X	28
Import statement for 'X' is in the wrong order. Should be in the 'X' group, ex	14
GenericWhitespace '>' should followed by whitespace.	1
GenericWhitespace '>' is followed by whitespace.	1
GenericWhitespace '<' is preceded with whitespace.	6
First sentence of Javadoc is missing an ending period.	675
Extra separation in import group before 'X'	262
Empty line should be followed by <p> tag on the next line.	731
Empty catch block.	6
Each variable declaration must be in its own statement.	5
Distance between variable 'X' declaration and its first usage is X, but allowe	256
Comment has incorrect indentation level X, expected is X, indentation shoul	45
Block comment has incorrect indentation level X, expected is X, indentation	47
At-clause should have a non-empty description.	871
Array brackets at illegal position.	1
Annotation 'X' have incorrect indentation level X, expected level should be X	44
All overloaded methods should be placed next to each other. Placing non-ov	4
Abbreviation in name 'X' must contain no more than 'X' consecutive capital	195
'X' should be separated from previous line.	438

Figure 9: Checkstyle violations bugs results

Randoop

In the process of applying Randoop, I followed a systematic approach to ensure effective test case generation. Firstly, I obtained the Randoop JAR file from the official website, ensuring I had the latest version for optimal functionality. Then, I compiled a text file containing the specific class names for which I intended to generate test cases using Randoop's automated capabilities. Note that, for healthcare-service and HealthPlus, we combined classes into a `.txt` file and with the use of command `-classlist`, we generated test cases collectively. However, for pg-health-index, healthcare-data-harmonization and IBM-Openshift, we generated tests individually for each class.

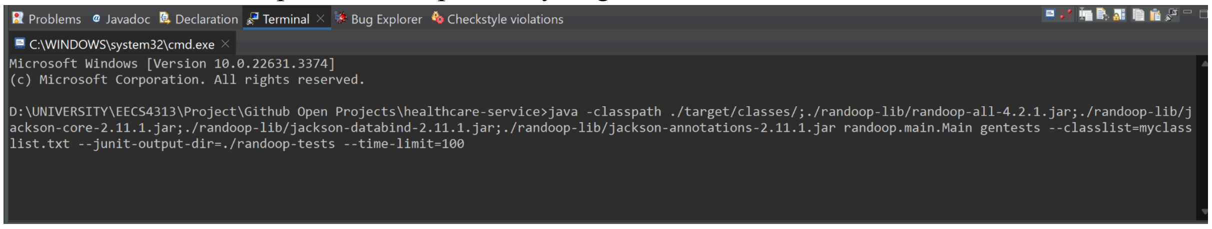


myclasslist.txt

```
ru.netology.patient.Main
ru.netology.patient.entity.BloodPressure
ru.netology.patient.entity.HealthInfo
ru.netology.patient.entity.PatientInfo
ru.netology.patient.repository.PatientInfoFileRepository
ru.netology.patient.repository.PatientInfoRepository
ru.netology.patient.service.alert.SendAlertService
ru.netology.patient.service.alert.SendAlertServiceImpl
ru.netology.patient.service.medical.MedicalService
ru.netology.patient.service.medical.MedicalServiceImpl
```

Figure 10: Randoop output

One significant challenge encountered during this process was related to dependency management, particularly in Maven-based projects. To address this, I manually downloaded all the external or third-party libraries required by the Java projects sourced from GitHub. This step was crucial to prevent "`ClassNotFoundException`" and "`NoClassDefFoundError`" errors, which could impede Randoop's ability to generate accurate test cases.



```
Microsoft Windows [Version 10.0.22631.3374]
(c) Microsoft Corporation. All rights reserved.

D:\UNIVERSITY\EECS4313\Project\Github Open Projects\healthcare-service>java -classpath ./target/classes/;./randoop-lib/randoop-all-4.2.1.jar;./randoop-lib/j
ackson-core-2.11.1.jar;./randoop-lib/jackson-databind-2.11.1.jar;./randoop-lib/jackson-annotations-2.11.1.jar randoop.main.Main gentests --classlist=myclass
list.txt --junit-output-dir=./randoop-tests --time-limit=100
```

Figure 11: Randoop terminal command screenshot in Eclipse

After downloading and adding the required libraries to the project, I ran the command line in the project directory. I made sure the external libraries and the Randoop JAR file were on the classpath by using the Java command. I also gave Randoop a time restriction of 100 seconds using command `--time-limit` to generate test cases and provided the class text file with the targeted class names. By using this systematic approach, Randoop was able to create a complete set of test cases .

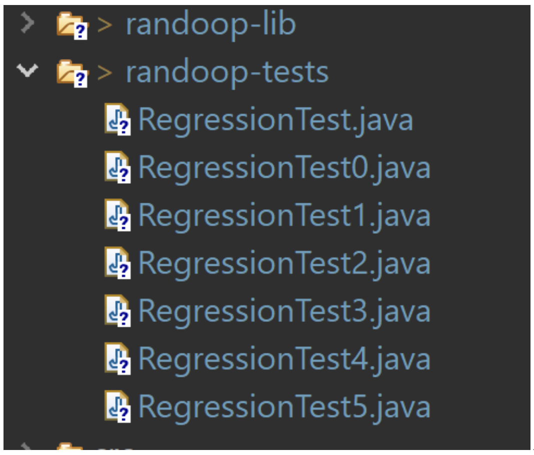


Figure 12: Randoop regression tests screenshot

Evaluating Results & Improving

FindBugs

After applying FindBugs to each project, the number of detected bugs in each were found and displayed in Fig 13. HealthPlus was the largest and most complex project we tested, so by nature it had the most detected bugs. This made testing the HealthPlus project the hardest to test overall.Due to having to manually go through each bug and check if they were false positives, the testing for the HealthPlus project was the most time consuming. On the other hand, the Healthcare-service project had the least amount of bugs detected with only 4 which made testing it completely simple.

Project	**HealthPlus**	**Healthcare-service**	**IBM-Openshift**	**StopCoding**	**Kardio**
Number of bugs detected	42	4	9	12	10

Figure 13: Findbugs bug detection results table

To manually evaluate the “bugs” detected for false positives, we went through each bug individually and determined if they were actually bugs. As explained before, by double clicking on each bug in the Bug Explorer, Eclipse will show the exact line of code that was flagged which made the process easier.

Depending on the filter used for the kinds of bugs, sensitivity of bugs, and confidence in bugs being displayed, the false positive rate changed. Decreasing the sensitivity and level of confidence to their lowest values respectively like we did during testing, will detect more bugs, but it will also detect more false positives. One of the kinds of low confidence bugs that were generated while testing was “confusing method names”. However, during our manual inspection of these types of bugs, we noticed that they made sense in the context of the class they were found in. In these cases they were declared as false positives.

This is the main disadvantage of using static analysis tools like FindBugs and one of the main reasons false positives are generated. It is harder for them to understand the context of given codes which makes them flag things that should not be flagged. In these scenarios it is easier for humans to understand the context and relationship of classes and methods which made identifying the false positives simple.

Although there are disadvantages to using FindBugs to test projects, there are also many advantages as well. As explained before, since it is a plugin, installing and using it to detect bugs early on in development is easy. FindBugs also comes with a comprehensive filter that allows users to choose the kinds of bugs they want to focus on, FindBugs' confidence in them actually being a bug, and the seriousness of each.

A lot of the bugs detected by FindBugs were either false positives or not worth reporting to their developers because they were only flagged for inefficient methods of performing actions. For example "invoking inefficient new String(string) constructor" and "reliance on default encoding" were both "bugs" found, but since they didn't alter the performance of these projects we decided they weren't worth reporting.

The bugs we did end up reporting were bugs that had the possibility of being problems in the future or under certain circumstances. The bugs that fell under this category were unused variables, comparing string references rather than string contents, and possible null pointer dereferences.

The process of reporting these bugs consisted of creating Issue tickets on each of their respective github repositories. Inside the issue ticket, the type of bug was specified, where to find them, and what is usually expected in this case versus what was actually used.

Checkstyle

In our analysis of various projects, Kardio exhibited the highest number of violations, totaling 43,698. This high number of violations suggests a non-negligible false positive rate

Project	HealthPlus	Healthcare-service	IBM-Openshift	StopCoding	Kardio
Checkstyle Violations	27480	256	16124	46	43698

Figure 14: Checkstyle bug detection results table

Checkstyle's primary advantage lies in its ability to enforce consistent coding guidelines. However, one of the hurdles that were faced during the testing of a tool like Checkstyle was that a lot of configuration was required based on project-specific requirements and coding standards. This process was deemed to be time consuming and required a lot of fine tuning. Another challenge that was faced after the generation of violations was that the balance between false positive and false negative rates required an in depth review of the 43,698 violations.

As part of the evaluation process, an attempt was made to review all the violations manually, but soon the realization was made that it would be too time consuming and repetitive. Based on the data that was reviewed manually, it was discovered that 14,179 out of 43,698 violations were simply, “line contains a ‘tab’ character”. Checkstyle encourages programmers to adhere to strict coding standards and practices, which may initially pose a learning curve. This proved that the repetitiveness of the violations that are generated because of minute differences in coding conventions can be used as an additional filter while using Checkstyle to refine the violations and eliminate false positives.

Randoop

Upon evaluating the results of Randoop's test case generation, I observed varying outcomes across different projects. In four out of five projects (healthcare-service, pg-health-index, IBM-Openshift, and healthcare-data-harmonization), Randoop successfully produced passing test cases. However, it encountered significant challenges with HealthPlus, where the generated regression test cases (which are supposed to pass as per definition of Randoop) exhibited a high failure rate. Notably, only HealthPlus saw the generation of error-revealing tests, whereas the other four projects did not.

Project	Number of classes tested	Regression tests generated	Error-revealing tests generated
healthcare-service	10	2997	0
HealthPlus	14	1974	223
IBM-Openshift	15	2904	0
pg-index-health	1	39	0
healthcare-data-harmonization	2	21	0

Figure 15: Randoop bug detection results table

To validate the effectiveness of the generated test cases, I executed the test files within the respective projects' `test` source packages. In the case of HealthPlus, all test cases failed, with only a minimal number (2-3 out of 2000) passing. Further investigation revealed that these failures were false positives, primarily attributed to either incomplete inclusion of external libraries or supporting/helping classes alongside the test class. Additionally, failures occurred in methods such as `toString()` or `equals()`, which were not even overridden, suggesting that the failing test cases may be false positive results.

Discussion of Detected Bugs and Tool Pros and Cons

Each of the bug detection tools presents distinct pros and cons. FindBugs simplifies early bug detection with their user-friendly interface and comprehensive filtering options which allows for easy identification of potential bugs. One of the main cons for FindBugs would be manual handling of false positives which can be time consuming, particularly in complex projects such as HealthPlus. Checkstyle enforces coding standards effectively, promoting consistent code quality. Even Though, it offers flexibility with their customizable rule sets, it is challenging to configure to project-specific standards and it may not cover all the bug types. Randoop on the other hand, automates unit test generation efficiently, saving time and integrating seamlessly with JUnit. However, its inability to control test inputs and sensitivity to false positives in complex projects like HealthPlus highlights the importance of considering dependencies and interactions during testing. It is also worth noting that configuring Randoop proves challenging, especially in projects with unique structures like Stop Coding, where generating test cases was not possible due to missing dependencies. It was not possible to generate test cases even though we downloaded external libraries and dependencies. Each tool has their strengths in bug detection, mitigating false positives, and maintaining code quality, addressing their challenges is crucial for maximum optimization. Finding a balance between automation as well as manual verification, adapting to project complexity and enhancing the overall quality of these tools is important in ensuring software reliability and quality. During our verification process for FindBugs and Checkstyle, it was evident that different IDEs resulted in different results. While running the healthcare-service project using checkstyle in Intellij we recorded a total of 54 bugs whereas, running the same project in Eclipse we recorded 256 bugs. Similarly, the healthplus project using FindBugs plugin in Intellij resulted in 0 bugs however, the same project had 42 bugs recorded using the Eclipse plugin.

Conclusion & Future Steps

In our project, we've encountered common bugs and explored the effectiveness of bug detection tools. These tools play a crucial role in early issue identification, therefore saving time and resources while ensuring software reliability. When selecting these tools, prioritizing compatibility and seamless integration with our project workflows is essential. Tools should seamlessly fit into our existing processes and adapt to our project's unique requirements. Customization is key, as bug detection tools should be customizable to meet our needs, allowing us to tailor them for optimal effectiveness. Additionally, tools such as Randoop should prioritize enhanced dependency handling and adapt to diverse project environments to ensure accurate bug detection across different project structures. Regular updates are crucial for these tools to stay effective, evolving to combat emerging issues and provide reliable solutions. It's important to modify settings according to the project when using different bug detection tools, ensuring optimization for our specific requirements and accurate results. Looking ahead, future development should focus on refining bug detection methodologies and addressing challenges such as false positives and complex project structures to enhance the effectiveness of bug detection tools.

References

[1] Shanmugathasan, Heshan. "HealthPlus." https://github.com/heshanera/HealthPlus/. Accessed 11 Mar. 2024.
[2] Jogeen. "StopCoding." https://github.com/jogeen/StopCoding. Accessed 11 Mar. 2024.
[3] IBM. "example-health-jee-openshift." https://github.com/IBM/example-health-jee-openshift. Accessed 12 Mar. 2024.
[4] Ne. "healthcare-service." https://github.com/neee/healthcare-service. Accessed 13 Mar. 2024.
[5] T-Mobile. "kardio." https://github.com/tmobile/kardio. Accessed 7 Mar. 2024.
[6] FindBugs. "FindBugs™ - Find Bugs in Java Programs." *SourceForge*, https://findbugs.sourceforge.net/. Accessed 16 Mar. 2024.
[7] Baeldung. "Checkstyle – Introduction." *Baeldung*, https://www.baeldung.com/checkstyle-java. Accessed 23 Mar. 2024.
[8] The Checkstyle Project. "Checkstyle." *SourceForge*, https://checkstyle.sourceforge.io/. Accessed 30 Mar. 2024.
[9] Randoop. "Randoop." https://randoop.github.io/randoop/. Accessed 18 Mar. 2024.