Robot Comment Fixed Detector

Patrik Fjellstedt
D19 Lund University, Sweden
pa2747fj-s@student.lu.se

Joakim Svensson
D20 Lund University, Sweden
jo3337sv-s@student.lu.se

Abstract

Today there exists a lot of tools to analyze existing code to generate useful feedback for the programmer. When these tools are used with code review tools, such as Gerrit, this feedback is also known as robot comments. Generating feedback upon code review is a useful workflow to prevent bugs and improve code quality before accepting changes into a project's main branch. A problem that arises with the use of such tools though is the amount of generated comments and their accuracy for them to be useful. This points to a need for a pre-processing step where some comments could be filtered out before presenting them to the user. We focused our attention on one such tool, ErrorProne, which is a static analysis tool for Java that helps to identify errors during compilation. Using this tool we tried to detect whether a robot comment was fixed between its head version and the revised version.

1. Introduction

Program analysis tools can give useful warnings and errors, but it can be difficult to interpret what is solved and how useful the information is from the tools. In the worst case the tools produce so much output that working with them becomes impractical. This was noted in previous work, such as Ljungberg et al. [2], where they found that some analyzers reported so many results it hampered the developer's work, as well as ignoring up to 90 % of the generated comments. They also mentioned their use of a way to filter out duplicated robot comments based on line numbers and error types to reduce the output generated. We use this strategy as a baseline in our exploration of a detector system in this paper.

In the paper by Avgustinov et al. [3] they tracked code violations over time and tried to match them to specific users to give useful feedback to the developers about common mistakes they make. This identification can be as simple as checking the line number that an error was at between the versions, akin to the MEAN approach to filter out duplicated comments. In addition to this, their identification approaches included a way to hashing the code line where the error occurs in the two versions and comparing them. This approach is also used in our system where we refer to it as just "hash".

This paper tries to build on the identification approach from Avgustinov et al but forgo the matching of developers and instead focus on how we can emphasize identifying if a robot comment between versions has been fixed or not. To do this we built a detector system containing three detectors and compared their results. In addition to a baseline and hash-based detector, we developed a detector based on the Abstract Syntax Tree (AST) version of the source file to see if we could improve the hash-based version.

1.1 Motivating Example

To motivate the proposed solution, we look towards a simple example that shows how we can automatically detect and reduce the amount of robot comments that get forwarded to the reviewer. In the listing below a case is presented where an error has not been fixed between the main file, which we call *head*, and a checked-in changed version which we will call *revision*. Instead, the user only inserted a comment into the file, pushing the error down by one line. Here it would be useful to filter out this error and note to the reviewer that the error has not been fixed since nothing has been changed.

```java
// SubStringErrorHead.Java
package epf;

public class SubStringErrorHead {
    public static void main(String[] args) {
        String s = "Hello World";
        // This should generatee an error.
        String error = s.substring(0);
        error.concat("\n");
        System.out.println(s + " " + error);
    }
}
```

Figure 1. Head version of file.

```java
// SubStringErrorLineMove.Java
package epf;

public class SubStringErrorLineMove {
    public static void main(String[] args) {
        // Inserted comment.
        String s = "Hello World";
        // This should generatee an error.
        String error = s.substring(0);
        System.out.println(s + " " + error);

    }
}
```

Figure 2. Revision, inserting a comment.

Course paper, EDAN70, Lund University, Sweden
January 10, 2024

1.2 Used tools

In this project we used one static analysis tool, **ErrorProne** [4]. To use the output from ErrorProne a layer between the static analysis tool and the proposed detector system was developed. This would allow for more tools to be used in the future without making it dependent on the tool itself. ErrorProne checks the compilation for typical errors in the code that would compile but have unintended behavior and has obvious replacement code also typically called anti-patterns. Examples could be an if statement always being true (self equals), using a method substring with 0 as an argument i.e. always returning the same string (SubstringOfZero), etc. These errors are noticed during compilation and the easiest way to use ErrorProne is through a build system. With the help of either of these three build tools Gradle, Bazel, and Maven you can invoke the ErrorProne and produce the desired output.

Gradle is a build tool to make projects have more consistency in what order things are executed and how they are executed. This way there is no need for long user manuals as the build tool will build it according to the instructions with things like a specified Java version and tools on top of the compiler like ErrorProne and a testing framework like JUnit running bundles of tests to ensure functionality in the code. The problem with build tools like Gradle is that the first execution takes a long time and it isn't always fast after the first build either. It can be a minor inconvenience to wait for 3 minutes+ for a build tool to run a larger project which is the setback of these tools. The benefit is that it streamlines the process of execution and makes the burden on the project creator and user a lot smaller while forcing checks to be run every time something is changed making sure there are no unintended side effects but at the cost of time.

Finally, to implement a detector based on the AST of the source file **ExtendJ** [5] was integrated into the build chain to produce an AST output for the files being analyzed. ExtendJ is a Java compiler but the compiler provides information on a tree of nodes that builds the code, the AST which can be interesting as a tool to analyze the code. ExtendJ has support for Java versions up to 8 so it compiles Java code up to Java 8.

For more details on the tools used and the implementation see the source code [6].

1.3 Research Questions

To improve the workflow of using robot comments we seek out answers to the following research questions(**RQ**):

- **RQ1** How do we create metadata that can be used to evaluate if a robot comment has been solved?
- **RQ2** How can we implement a detector that use this metadata to evaluate if a robot comment has been solved ?
- **RQ3** Can we use tools, like the ExtendJ compiler, to generate more in-depth information, like an AST, in order to get more context about where a robot comment is generated in order to improve our detector?
- **RQ4** Out of a baseline, hash and AST detector which one performs the best ?

Through our experiments we found that developing a middle layer for static analysis tools was useful to separate the detector system from a specific tool. Also the most useful detector out of the three we experimented with the hash based seems to be the most generic and useful out of the three.

2. Implementation/Solution

2.1 Base of solution

The solution consists of output from program analysis tools to generate useful errors from Java projects or small example programs. As ErrorProne is a tool that is integrated with build tools like Bazel or Gradle, we used Gradle to build our project and took the output from that build to analyze what happened in said project. Once we got an output file we created a script that would generate an output file and then read that information to do our analysis by reading in that file in a Java program. In that Java program, we would then run the main analysis on the information by comparing the errors, in the form of checking what error ErrorProne generated, on what line number it occurs on, and in what file it is in.

2.2 ErrorProne example and test files

The small Java programs were taken from the examples of ErrorProne for simple examples of Java programs illustrating errors like SubstringOfZero. ErrorProne had instructions on how to easily create basic programs with the errors in them on their webpage [11]. So we created folders with different alterations and line number changes to a test Java file with errors from the site. This generally consisted of a head file, one fixed file, one unfixed file, one with the line number of the error moved and one with an additional error.

2.3 Using ErrorProne to generate information

The next step was using the Gradle build file to incorporate ErrorProne in the compilation of the project. So we added dependencies and specified what version of ErrorProne to use so it would be compatible with Java 8 to integrate with ExtendJ. We then saved the output of the Gradle file in an output file. The next step was then to extract the relevant information from the said file to be able to use the information we wanted from ErrorProne. Error type, line number error occurred on and file the error happened in, this file was then the base of the detectors.

2.4 Base script to build project

Once we had this in place we built a small script to help test our program. It consisted of recompiling Java files, saving the Gradle output to a file running the reformatting program on that output, and then running the detectors.

2.5 Implementation and improvement on detectors

The detectors are based on reading in these files and saving the error data above. At the core, all the different detectors compare the old file (head file) to all the updated files and try to figure out if it's fixed or not based on the error information. One of these detectors is based on the AST.

2.6 Extending script to support AST building

For the AST detector to be implemented we decided to work on the JavaDumpTree output for easier integration with the other code. So we updated the script to generate an AST with the help of a cloned template project for static analysis [12] and the jar file for the said project was used to run Java -cp "template.jar" "org.extendj.JavaDumpTree" fileName which we used to generate an AST for each of the files in the script.

2.7 AST detector implementation and further work

The AST was then used to find variable names and check if the tree structure had the same line numbers for the errors. Then we manually added checks based on trends in the AST to see if that tree looks like expected or identifies a certain combination of nodes for error checks. The AST was also introduced as an extra version of error generation for the code to notice missed typical errors. The

AST detector is a lot less general as it was harder to make it apply to several errors without taking the case into account and therefore hard-coding a lot of situations which ended up being out of the scope of the project.

3. Evaluation

3.1 Setup and base detector

To evaluate our detectors the following experimental setup was used. First, we created a set of simple files with the errors, our head versions with a set of revisions where the error would have either been fixed or not. Then we ran the ErrorProne static analyzer on the test set to produce ErrorProne output for our files. This produced the necessary metadata about the files we wanted to analyze and could be done separately from the detectors. This output was then fed into a metadata builder that extracted the ErrorProne errors for a file, and their line numbers and hashed the line on which the error occurred. This data is then used for the baseline and hash-based detectors. To evaluate a robot comment the detector outputs the following:

- Yes, if the error had been fixed.
- No, if the error was still present.

This answers our initial research question RQ1 that a metadata set was producible for our detectors from standard ErrorProne output.

With this metadata output, we could then implement the following three detectors, each trying to improve on the previous implementation:

- Baseline detector, comparing line numbers.
- Hashing detector, compares hash ids of the error line.
- AST detector, using the AST output from ExtendJ for additional context around the error.

3.2 Baseline detector

The baseline detector simply compared the head version's line number with the revision file for the same type of error, like the MEAN approach to detect duplicated errors. If it was unable to find a related error type it determined that the error had been fixed. Otherwise, if the same error appeared on the same line in the revision file it would note that this robot comment was the same and produce a no.

3.3 Hash-based detector

In order to improve the baseline number detector a hash-based approach was implemented. The code line where the error occurred was hashed and stored in the metadata file and then compared with the new revision file if the same type of error was found in the file. If two errors of the same type had the same hash code the detector would output a No, since it could then determine that the same error was present in both files. In this case, compared with the baseline detector the error was no longer tied to a line number in the code and solves the introductory motivating example of inserting a comment.

3.4 AST detector

The AST has information about values stored in variables and the related context. So we decided to, on an error by error basis, implement support for handling errors with extra AST analysis. This detector will if it doesn't support the error noticed in ErrorProne just send out maybe solved on all errors relating to it. If there are no errors all of the detectors will put out the same answer as ErrorProne doesn't see an error in the file. It currently looks for the variable names so if the only thing about the hash that has changed is the naming convention or an extra space it will fairly clearly point out for some errors that this is the case. The information is stored in different places and the interesting information to give back to the user will be different on a case-by-case basis for the errors.

4. Results

Here we list the output of the detectors on our test data.

4.1 Baseline detector

As we can see from the figures the baseline detector is only capable of detecting errors that have not moved or that no longer appear between head and revision. This makes it unsuitable for files where the number of lines changed between head and revisions. Since most files that incur a revision change is the result of adding or removing lines in practice leads us to suspect that such a detector is impractical.

4.2 Hash-based detector

The hash detector improves on the baseline version, especially when the line moves. Since we hash the error line, if the same hash is detected on a different line it would still have the same hash value. Since we pair hash with error type the detector is capable of notifying that the error is still present. This makes it generic and can be applied to any type of error with no additional work, as can be seen from the different figures.

4.3 AST detector

The AST detector required more work than we anticipated and was only tested for one of the errors, namely the SubStringOfZero which can be seen in Figure 3. It performs just as well as the hash detector. The real benefit from this type of detector comes through the additional test case where an alteration in the testing set was introduced. The alteration was introducing an indirection for the value 0 that was passed to the function substring through a variable. The ErrorProne tool was unable to determine that this was still an error but by looking at the AST we could find the value of a variable that was used for the function and tell the user that the error was in fact still present. However, this type of work was time consuming and was not generic but used contextual information from the type of error that was investigated leading us to believe that such work, although very powerful, being to expensive to implement.

5. Related Work

One way to handle robot comments would be to manually compare the comments. They would check what errors the build tool generates and if the test management tool runs on the build. This would not be replaced by automating part of the comparison but the goal of the solution would be to create less work and make the process of code review shorter by filtering out useless information. The main problem the tool has to solve is making sure it saves time for the user by creating a better output of errors than the initial tool with more useful information.

One such work that is currently in use is SonarQube which is a continuous code inspection platform that allows users to have their source code checked for errors [10]. This platform checks for more than robot comments from static analysis tools, like security vulnerabilities, coding standards defined by the maintainers etc. The work presented in this paper could potential be useful for such a platform to automatically remove non-useful robot comments in their bug reporting subsection.

6. Conclusion

Being able to reduce the amount of robot comments can lead to an improved experience for code reviewers by reducing the amount of

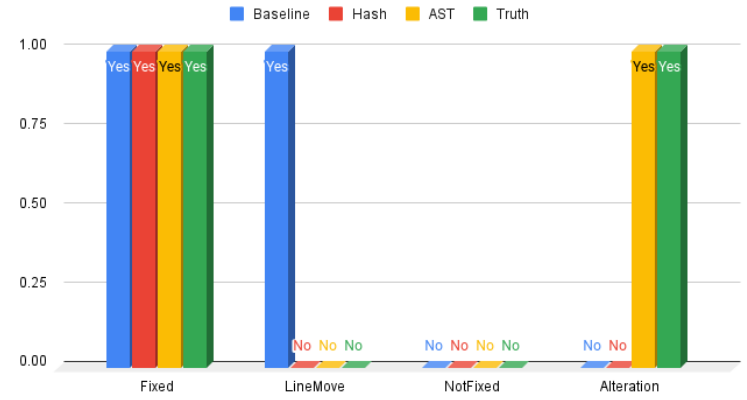

Figure 3. Single Error of Substring for the different detectors.

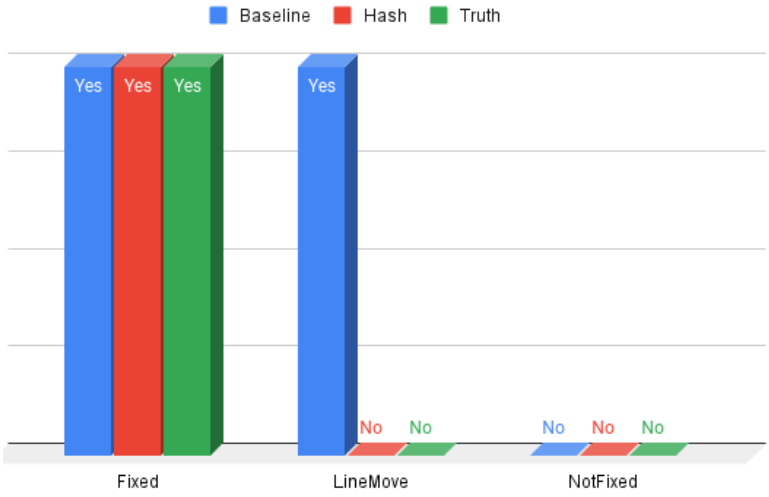

Figure 4. Single Error of ArrayHashCode for the different detectors.

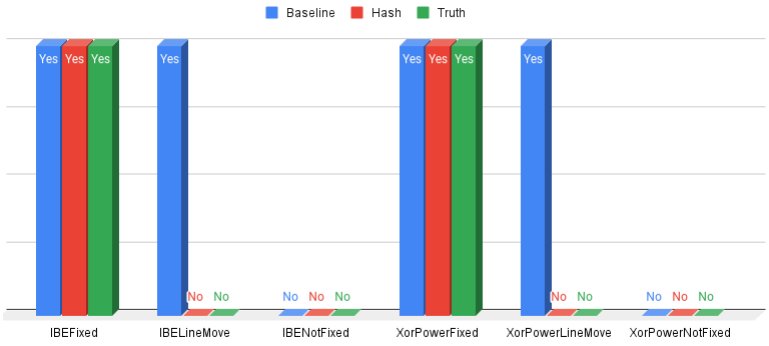

Figure 5. Multiple Errors of XorPower for the different detectors.

non helpful robot comments that can be generated. In our work we have experimented with three detectors to produce output whether or not a robot comment has been fixed or if they are the same between revisions. In summary for our initial research questions, we found the following answers:

- **RQ1** Yes, we could produce a metadata set of the original output for our detectors.
- **RQ2** We found that the baseline strategy of looking at only line numbers led to imperfect precision at best whereas a hash approach improved this to some extent but still had some weaknesses.
- **RQ3** Yes, the information generated from the AST was useful in specific cases and could do things ErrorProne could not and improved the detector.
- **RQ4** From our experiments we found the baseline detector to be mostly unusable and the hash detector to be the most useful detector in practice. Altough the AST detector can use context surrounding the error line to produce powerful feedback this type of work is time consuming and not practical.

6.1 Threats to Validity

This work is only a proof of concept where time became a limiting factor. Here are some suggestions for future work that could improve this detector system:

6.1.1 Larger dataset

In this initial work, our dataset is fairly small and only tests a handful of robot comments produced by ErrorProne. To strengthen the validity of this tool a larger test set of more robot comments would be good.

6.1.2 More static analyzers

We focused our efforts on using the output from ErrorProne, and whilst extending the amount of robot comments that this produces would be good adding more static analyzers to our metadata output would be an interesting continuation

7. Acknowledgments

We extend our thanks to Emma Söderberg for all help and feedback.

References

[1] Sadowski, C. et al. (2015) 'Tricorder: Building a program analysis ecosystem', 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering [Preprint]. doi:10.1109/icse.2015.76.

[2] Ljungberg, A. et al. (2021) 'Case study on data-driven deployment of program analysis on an open tools stack', 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) [Preprint]. doi:10.1109/icse-seip52600.2021.00030.

[3] Avgustinov, P. et al. (2015) 'Tracking static analysis violations over time to capture developer characteristics', 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering [Preprint]. doi:10.1109/icse.2015.62.

[4] Error Prone. Available at: https://ErrorProne.info/ (Accessed: 09 November 2023).

[5] ExtendJ. Available at: https://ExtendJ.org/ (Accessed: 29 November 2023).

[6] Robot Comment Fixed Detector. Available at: https://git.cs.lth.se/nex/robot-comment-detector (Accessed: 29 November 2023).

[7] Gerrit code review - robot comments. Available at: https://gerrit-review.googlesource.com/Documentation/config-robot-comments.html (Accessed: 06 December 2023).

[8] Aftandilian, E. et al. (2012) 'Building useful program analysis tools using an extensible Java compiler', 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation [Preprint]. doi:10.1109/scam.2012.28.

[9] Öqvist, J. (2018) 'ExtendJ: Extensible java compiler', Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming [Preprint]. doi:10.1145/3191697.3213798.

[10] Code quality tool & secure analysis with SonarQube. Clean Code: Writing Clear, Readable, Understandable & Reliable Quality Code. Available at: https://www.sonarsource.com/products/sonarqube/ (Accessed: 03 January 2024).

[11] Errorprone bugpatterns. Available at: https://errorprone.info/bugpatterns (Accessed: 03 January 2024).

[12] Static analysis template on gitbucket. Available at: https://bitbucket.org/extendj/analysis-template/src/master/ (Accessed: 03 January 2024).