Finding and Resolving Common Code Style Errors in Java

Mark Lundager
D19, Lund University
Sweden
mark_06_20@outlook.com

Andreas Berqvist
E18, Lund University
Sweden
andreas.bergquist92@gmail.com

Abstract

Verbose code with highly specialized `if-statements` containing `return-statements` for the different cases of the conditions is harder to maintain when compared to a single `return-statement` containing the condition. It is commonly referred to as a code style error. In this report we present a tool which can be used as an extension in your IDE, and performs on-demand analysis of your code to discover, and repairs to your style violations. All while using the same variables, constants and general structure of your original code. There is no change in functionality of the written code. It is also able to preserve the comments you may have in your original code. With this tool we hope to help people who are prone to these style errors to kick that habit.

1 Introduction

Verbose code is common amongst new programmers. As a beginner, it is intuitive to write code which reflects the programmer's thought process. However, what a beginner considers intuitive and self-evident may not always align with the most clear and concise coding practices. An example of this is the `If Return Condition (IFRC)` semantic style issue [5]. Instead of checking a condition with an if statement to then return either true or false in respective blocks, one can simply return the condition since the condition itself represents the value that we wish to return. The second approach brings several benefits, including simpler code, reduced complexity, and improved clarity. By returning the condition directly, the code becomes more straightforward, requiring fewer lines and making it easier for developers to understand. In the coding industry where there are multiple software engineers that are working on the same codebase, it is vital to have a coherent code style and to avoid redundant code as they are a prime example of Wayne's Babich's double maintenance problem [1]. Regardless, it is not uncommon for such practices to be ignored even if they are encouraged. It is also difficult to motivate a reason for performing these refactors if the code works as intended. Hence, it would be ideal if a tool could refactor such instances of semantic style issues to the improved ones in order to help beginners improve more quickly and foster a better understanding of clean coding practices.

In this project, we aim to develop a Java Code Style Improver using the Declarative Referenced Attribute Grammar `JastAdd`[3] in conjunction with the Java compiler `ExtendJ` [2] to detect some of the more common semantic style issues and refactor them. Due to the nature of `JastAdd`, this linter will be an on-demand analyzer, meaning the user has the choice which cases should be ran, and which should not. Only computing what is necessary for the analyzes being ran. This differs from other popular linters such as `Programming Mistake Detector` (PMD), `SonarLint`, and `Checkstyle`. Moreover, for a linter to be useful, it must offer convenience. Otherwise, it would defeat its purpose. Consequently, our tool will be implemented as an extension for Integrated Development Environments (IDEs), such as `Visual Studio Code`. This will be accomplished with `Magpiebridge` which offers an interface to communicate with IDEs through the Language Server Protocol (LSP).

2 Motivating Examples

Listed below are examples of verbose code and their respective proposed solution. One is the `Empty If Body (EIFB)` style issue [5]. In which the programmer has a condition, and the intended result is to only perform actions when the condition does not hold. If inexperienced, this may result in the programmer leaving the if-body empty and solely adding code in the else-body. As illustrated in the code-snippet below:

Listing 1. `EIFB` style error

```
public boolean eifb(int y, int x){
    if( x < y ){
    }else{
        return false;
    }
}
```

However, the same result can be achieved by simply replacing the `if-else statement` with an `if-statement` checking the inverted condition. This way, there is no redundant statement with an empty block. This action has been performed in the repaired code below.

Listing 2. `EIFB` style error repaired

```
public boolean eifb (int y, int x){
    if(! ( x < y ) ){
```

Course paper, EDAN70, Lund University, Sweden
January 8, 2024.

```
        return false;
    }
}
```

Another example with is the previously mentioned `IFRC`[5]. In this example, the only code contained in the if-, and else-bodies is a return-statement either returning true or false depending on the condition in the `if-statement`.

Listing 3. `IFRC` style error

```java
public boolean ifrc(int y, int x){
    if( x < y  ){
        return true;
    }else{
        return false;
    }
}
```

Essentially, either the value, or the inverted value of the condition is returned. Which is equivalent to returning the condition, or the inverted condition. Consequently, this makes the else statement redundant.

Listing 4. `IFRC` style error repaired

```java
public boolean ifrc (int y, int x){
    return x < y;
}
```

The final style violation from DeRuvo our tool detects is the `If Return True (IFRT)`[5]. The `IFRT`-antipattern consists of a `if-body` containing a `return` statement, no `else-block` and immediately after the `if-body` there is a `return` statement.

Listing 5. `IFRT` style error

```java
public boolean ifrt (int y, int x){
    if( x < y ){
       return true;
    }
    return false;
}
```

For all intents and purposes, given that the two return statements return two different boolean literals one may replace the entire violation with one return statement. Due to their similarities, one may regard the `IFRT`-antipattern as a special case of the `IFRC`-antipattern. Therefore, the repair is identical to the `IFRT` style error. Due to time restraints, there was no opportunity of correcting the indentation of the code style improvements. We use the pretty print introduced by `ExtendJ` which does not have support for indentation. However, there are multiple extensions that solves this issue. Therefore, extending the pretty print in `ExtendJ` was determined out of scope.

3 Java Code Style Improver

Our solution, the Java Code Style Improver, is a tool designed to identify verbose code in Java, and provide suggestions for improvements based on these detections. The Java Code Style Improver is available either as a standalone program or as an IDE extension. This provides a convenient way for the user to interact with the linter. By letting the static analysis being integrated directly into the development environment, the user does not have to go out of the way to utilize the tool. This reduces the risk of a user not utilizing the tool due to the code working. To integrate our tool as an extension, we opted for `Magpiebridge`, an LSP framework, making the process quick and straightforward. The Java Code Style Improver is based on `ExtendJ`, an extensive Java Compiler developed using `JastAdd`. Due to `ExtendJ` using `JastAdd`, it was simple to extend the Java compiler by adding attributes which defined the characteristics of the semantic style issues. For each such issue, there is a corresponding `JRAG`-file, making the analyzing system modular, allowing the user to decide which patterns to look out for. To see the lines of code of each `RAG`-file see Table 1, These `RAG` files also provide the improvement suggestions using the information from the anti-patterns, such as the relevant variables, values, and even comments present in the code. This leads to no information loss when one decides to refactor the code according to the proposed fix. The overall architecture of our solution can be seen in figure 1.

The first step to implementing the Java Code Style Improver was analyzing the Abstract Syntax Tree (AST) generated by using `ExtendJ` to compile written code. This was done with the help of `Code-Prober`[7] which facilitated the process of identifying the troublesome sub-trees that reflected the code-style issues we wished to detect. `Code-prober` is a tool which provides a graphical interface to interact with the AST directly in real time. Using this information, we were able to determine in which nodes we should introduce new attributes. Additionally, it allowed us to understand when a false-positive might emerge, a code-style improvement which changes the behaviour of the code. This was especially important as our approach to developing this tool was to minimize fixes that would alter the behaviour of the code. Changing the behaviour of a program is in our opinion worse than missing a potential refactoring improvement.

To demonstrate the process, we can use the `IFRC` as an example. In figure 2 we can see the `AST` representation of the `IFRC` example code presented in listing 3. By looking at this visual representation, it becomes clear that we have to define an attribute for `if-statement` to determine whether it itself violates the `IFRC` issue. We know that in order for an `IFRC` issue to be present, there has to be an else block in the `if-statement`. So we write an attribute which checks

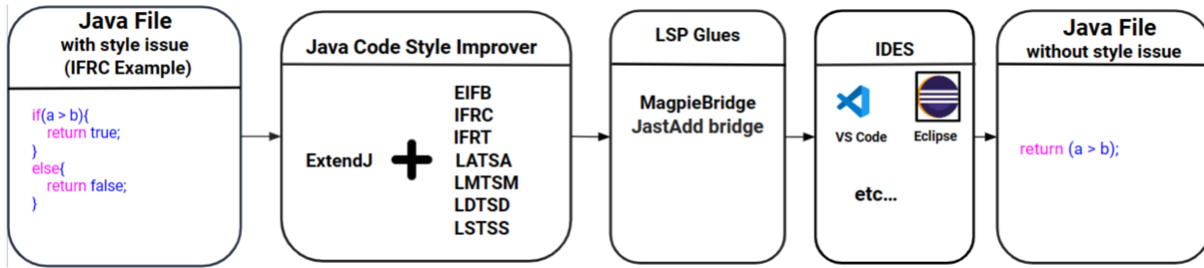

Figure 1. Architecture of Java Code Style Improver

whether this is the case. Then in a similar manner we continue looking and developing attributes for characteristics which define IFRC. However the complexity increases as the number of nodes increase.

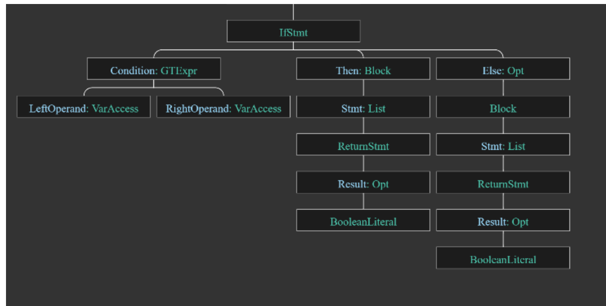

Figure 2. IFRC AST structure

Table 1. The RAG-files for antipattern analysis and their respective lines of code.

RAG-file	Lines of code
IFRC	61
EIFB	50
IFRT	101
Long add. to short add.	70
Long sub. to short sub.	57
Long mult. to short mult.	57
Long div. to short div.	57
CommentChecker	52

3.1 Limitations

It was vital for the proposed solution to be able to discern if an anti-pattern was genuine. Therefore we opted to limit the search to unambiguous code patterns. Therefore the analyser has a narrow scope of what constitutes an anti-pattern, which in fact does lead to false negatives. Our decision to choose this narrow definition of an anti-pattern is due to the An example would be in the IFRC case where the if-body and else-body does not solely consist of a return statement.

This perfectly displays how quickly defining the characteristics for style issues becomes complex. In order to determine whether such an `if-statement` presents an IFRC issue, we would need the ability to know if the code contained in the blocks have side effects. More specifically, whether a statement produces a side-effect. This problem is related to Purity Analysis, which is the problem of determining the side effects of methods [6]. This is a known difficult problem and would have to be a project of its own. In addition not every edge-case was considered for every antipattern. Instead we chose to go the sound route[4], with a functional tool performing several analyses rather than a highly specialized tool covering every case of a single antipattern. Therefore, in order to enforce our approach of avoiding as many false positives as possible, we implemented very strict requirements to discern whether a style issue is presented. An example would be the requirement of no other statements than a return statement in both blocks of an if statement for the IFRC issue.

4 Evaluation

The evaluation was performed on 27 repositories from the course Compilers (EDAN65) at Lunds Tekniska Högskola (LTH). These repositories contained all the necessary code to solve all issues presented in the course. This includes code such as visitor implementations, Test code, name analysis, type analysis and code generation. This was done by creating a script which checks for all Java files within a repository, compiles them and then runs the analysis. This was performed once. Then we collected the amount of code style issues detected by the Java Code Style Improver.

Table 2. Analysis and number of triggers for each antipattern

Antipattern	Triggers
IFRC	2
EIFB	0
IFRT	0
Long operation to short operation	0

As shown in Table 2, out of 27 repositories, only 2 code style violations were found. Since the compiler course is an advanced programming course, students are more likely to

be familiar with concepts and code practice. Consequently, the result of solely detecting 2 IFRC instances does not come as a surprise. In one way, this demonstrates that the even experienced programmer can use linters in order to learn. This does not come as a surprise as just because one may have experience with one programming language, does not necessarily mean that those skills translates to another programming language. An example would be Python and Assembly. The few detections can also be attributed to the very strict analysis and the few code style issues implemented. In order to better understand the practicality of The Java Code Style Improver, further evaluation would have to be carried out on code produced by beginners.

In addition to the evaluation of the repositories of the compiler course, manual testing were performed on all the different style issues. This was done to ensure that the refactoring suggestions did not change the behaviour of the code. We did this by experimenting with code which should be detected and introducing small changes to that code. During this testing, we were unable to find any instances of the tool suggesting a fix which changed the behaviour of the code. Due to the nature of coding, we cannot claim that we have tested every scenario but this shows some promise to our approach of avoiding false positives.

5 Related work

5.1 AI assisted coding

As technology advances rapidly, the landscape of learning to program is undergoing a shift. AI-assisted coding tools, such as `ChatGPT`, `GitHub Copilot`, and `OpenAI Codex`, are redefining the way developers approach programming tasks. The traditional model of learning to code by manually writing every line from scratch may not seem as important anymore with the help of these tools. This makes linters such at the Java Code Style Improver seem less useful. Tools like GitHub Copilot are not as limited as static analyzers which are crafted manually [8].

5.2 Linters

A linter is a static code analysis tool, which are used to flag errors. There are various linters, some examples are Checkstyle, Programming Mistake Detector (PMD) and Sonarcube.

Checkstyle is a highly configurable tool designed to assist programmers in adhering to a set coding standard. This standard can be designed from a preset list of checks. These checks can range from limiting the numbers of parameters in a method to having a set character-limit per line[1].

PMD primarily focuses on unused variables, empty catch blocks, unnecessary object creation. It has a set of rules one can use, as well as support for writing ones own rules[2].

[1]https://checkstyle.org/, visited 20 December 2023
[2]https://pmd.github.io/ , visited 20 December 2023

SonarLint is a product sold by the sonar company. It features over 5000 rules and provides real-time feedback[3]. Due to it being proprietary, there is no room for users to write their own rules.

6 Conclusion

In this project we have developed The Java Code Style Improver, a linter for Java available as an extension in IDEs. This was done by extending `ExtendJ` with attributes defining code style violations with `JastAdd` and utilizing `Magpiebridge` as the LSP interface to communicate with the IDE:s.

Since the evaluation was performed by running the tool on code written by experienced programmers it is difficult to assess the effectiveness of the linter. However, since the tool was still able to detect code style issues in such code, it means that the potential benefits of linters may not be limited to beginners. It is also important to note that even if one may have experience in developing code, there are various programming languages which vary in coding practices. Meaning an experienced developer being introduced to a new programming language may still find a linter useful. Additionally, we were unable to write code which produced false positives, showcasing our intention of avoiding false positives.

6.1 Future work

The Java Code Style Improver can be improved and expanded upon. One could add additional antipatterns so that the tool becomes more extensive in its analysis. An example for an antipattern would be for the for-loop. Depending on the contents of a for-loop, it can be converted to a for-each-loop without changing or breaking the code.

Additionally, one can enhance the existing checks such as the `IFRC` issue by implementing an attribute for every statement node which defines whether it produces side effects. An approach to this implementation would be to introduce an attribute for the abstract `AST` node "statement". This attribute would be of boolean value and return false by default. Then step by step iterate through the different statement nodes and writing code to determine whether each node of type statement produces side-effects. A potential risk with this improvement is introducing false positives which we aimed to avoid.

Finally, one may consider modifying the prettyprinting to have support for indentation. As it currently stands, the linter does not conform to standard practices in regards to indentation. Since the information is available from the AST, it is possible to implement a fix for this issue.

[3]https://www.sonarsource.com/products/sonarlint/, visited 20 December 2023

Acknowledgments

We would like to thank Idriss Riouak for the starting skeleton code, the `Magpiebridge` integration and guidance.

References

[1] Wayne Babich. 1986. *Software Configuration Management - Coordination for Team Productivity.* Addison-Wesley Publishing Company.

[2] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (oct 2007), 1–18. https://doi.org/10.1145/1297105.1297029

[3] Görel Hedin and Eva Magnusson. 2003. JastAdd—an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0 Special Issue on Language Descriptions, Tools and Applications (L DTA'01).

[4] Ben Livshits. 2015. In Defense of Soundiness: A Manifestp. *Commun. ACM* (2015).

[5] Andrew et al. Luxton-Reilly. 2018. Understanding semantic style by analysing student code. (2018).

[6] David J. Pearce. 2011. JPure: A Modular Purity System for Java. *International Conference on Compiler Construction (CC)* (2011). https://doi.org/10.1007/978-3-642-19861-8_7

[7] Anton Risberg Alaküla, Görel Hedin, Niklas Fors, and Adrian Pop. 2022. Property Probes: Source Code Based Exploration of Program Analysis Results. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 148–160. https://doi.org/10.1145/3567512.3567525

[8] Muhammad Ali Babar Triet H. M. Le, Hao Chen. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges. *ACM Comput. Surv.* (2020). https://doi.org/10.1145/3383458