Detecting and Fixing Common Code Style Issues in Java

Alexander Magnusson
D18, Lund University, Sweden
al0173ma-s@student.lu.se

Abstract

Code style is an important aspect of programming. While two different pieces of code may perform the same task logically one might be simpler to read and understand than the other. Code written in an unnecessarily complex way might show that the author might not have fully understood a certain concept. In addition to reducing the complexity of the code itself, adhering to a common code style help multiple developers collaborating on the same code base understand code written by others quicker.

1 Introduction

In real world applications there are hundreds of developers working on the same codebase it is essential to have a coherent code style to allow for efficient maintainability of the code. Code style can be as simple as choosing between tabs or spaces for indentation: syntactic style. To more complex refactoring based on semantics such as refactoring the if-statement in Figure 1. There exist similar tools, such as Checkstyle for Java [1]. The focus of this paper will be on semantic style, which concerns how certain features of a programming language is used.

While code style is an important concept there is a tendency for it to be ignored as long as the program works correctly. In university courses it is often encouraged to follow conventions for code style, but it is rarely followed up on. Comments on code style are often seen as recommendations rather than requirements, which are easily ignored unless trivial to fix [2]. Seeing how adhering to code style is often only a recommendation, spending time and energy modifying a logically correct program might feel unmotivated and hence ignored. It is therefore motivated to develop an integrated and automated tool for detecting and fixing common code style problems. Reducing the friction in finding and fixing code style violations increases the probability that the programmer will follow the conventions.

Therefore the goal of this project is to develop a tool that detects and suggests fixes to common code style violations in Java. The tool will be in the form of a VSCode extension built upon the Magpiebridge framework [5] using Reference Attributed Grammars extending the ExtendJ compiler [3]. MagpieBridge wraps the Language Server Protocol and prodivdes an interface for the communication between Code Style Improver and the text editor. The proposed tool will be evaluated in the form of a Proof Of Concept, where the tool is applied on interesting examples.

The contribution of this paper is the development of the analyses that detect code style violations and compute suggested fixes, as well as integrating them into the framework. One big difference between Code Style Improver and other tools such as Checkstyle is that Code Style Improver provides real-time feedback and allows the user to select exactly which analyses should be enabled [1].

1.1 Components - JastAdd, ExtendJ, MagpieBridge

JastAdd is a Meta-compiler framework designed to aid developers building compilers. JastAdd is based on an object-oriented approach to the Abstract Syntax Tree. JastAdd support the Reference Attribute Grammars and it is implemented in Java [4]. The functinality built using JastAdd can be divided into aspects representing modules that can easily be modified or extended [4]. ExtendJ is an extensible Java compiler, it allows developers to modify the compiler, adding new features. The ExtendJ compiler makes use of JastAdd when working with Reference Attribute Grammars, evaluating attributes on-demand enabling the analysis to be performed in real time [3]. `MagpieBridge` is a Language Server Protocol framework that intends to simplify the creation of language servers by providing an interface between the language server and the client. This allows developers to focus on implementing the language and client specific features. One important feature is the modularity of the framework, it enables developers to modify the language server without having to rewrite it entirely. Using this approach reduces the overhead needed to connect Code Style Improver with a text editor [2].

2 Motivating Examples

The examples below show a pair consisting of a code style violation and a proposed improvement. Violations of code style like in the examples below might indicate that the programmer has not sufficiently understood the concepts of the programming language.

Course paper, EDAN70, Lund University, Sweden
January 11, 2023.

Listing 1. Before refactoring.

```
if(condition == true) { ... }
```

Listing 2. After refactoring.

```
if(condition) { ... }
```

Figure 1. A common mistake for someone who do not fully understand boolean statements.

Listing 3. Before refactoring.

```
String[] coll = { "a", "b", "c" };
for (int i = 0; i < coll.length; i++) {
  System.out.println(coll[i]);
}
```

Listing 4. After refactoring.

```
for(String s : coll) {
  System.out.println(s);
}
```

Figure 2. If the only use of an index is to access elements in an array, it is clearer to use an enhanced for-loop.

3 Code Style Improver

The code style improver is a tool that enables detection and fixing of common code style violations in Java. The proposed solution uses ExtendJ, an extensible Java compiler [3]. Using ExtendJ it is possible to create a semantic analysis extension for the purpose of our tool. The analysis is done on the AST using RAGs. By incorporating the compiler extensions with the MagpieBridge framework it is easy to add an analysis into a text editor or an IDE [5].

There is a big advantage of having the analysis in the editor, namely the fact that the feedback is in the same place as the code it refers to. When the tool detects a code-style violation a visual warning will highlight the relevant lines with a message explaining the reason behind the warning and a suggested quick-fix allowing the developer to fix the code with a simple keyboard command or IDE interaction. This creates a system with a quick feedback loop, the developer writes code, receives instant feedback and if a problem is detected it is trivial to apply the fix.

There are two main problems solved by the tool. Firstly there is the problem when the developer does not know that the code they are writing could be written in a better way. The detection helps the programmer to become aware of the problem. The second part is actually fixing the problem. Some code style violations might be trivial enough that when one is made aware it is easy to fix with minimal effort, for instance the example in Listing 1. Another example is a for-loop using indices to iterate thorugh an array, which can be refactored to an enhanced for-loop as seen in Listing 2. This is a more complex operation, that requires limiting the types of for-loops we can refactor and it comes with edge cases that needs to be considered. A novice programmer unaware of the concept of enhanced for-loops might not understand how to perform the refactoring only from a warning message and will therefore also benefit from help applying the fix. An experienced programmer aware of the alternative will still benefit from getting help from the tool the same way that they would benefit from regular code completion.

3.1 Challenges

The main challenges developing Code Style Improver were:

- Dealing with complex checks where there exists a lot of cases to consider. For the for-loop check this means that the scope of cases that the tool can handle had to be tightened, the challenge is then to decide how much the problem can be simplified while still being general enough to be useful.
- Generating new code is another big challenge, it is important to ensure that the new code is properly indented. Especially important is the case when generated code introduces new variables, it must be ensured that the newly introduced variables do not conflict with other variables in the namespace.

3.2 Architecture

The architectural overview in Figure 3 shows a high-level overview of the components of the tool. From left to right, there is the input code that is to be analyzed. The input flows into Code Style Improver. ExtendJ and JastAdd allows one to extend the compiler with new attributes used to detect possible code style violations. The next level contains warnings and quick-fixes, when a violation is detected, a warning is constructed containing an error message, the location and actual value of the faulty code. Additionally a suggested quick-fix is computed and bundled with the warning. This package with the warning and quick-fix is then sent towards the text-editor. There is an interface between the Code Style Improver and the text-editor, which consists of MagpieBridge that in turn wraps the LSP, or Language Server Protocol. On the far right is the output after applying the quick-fix.

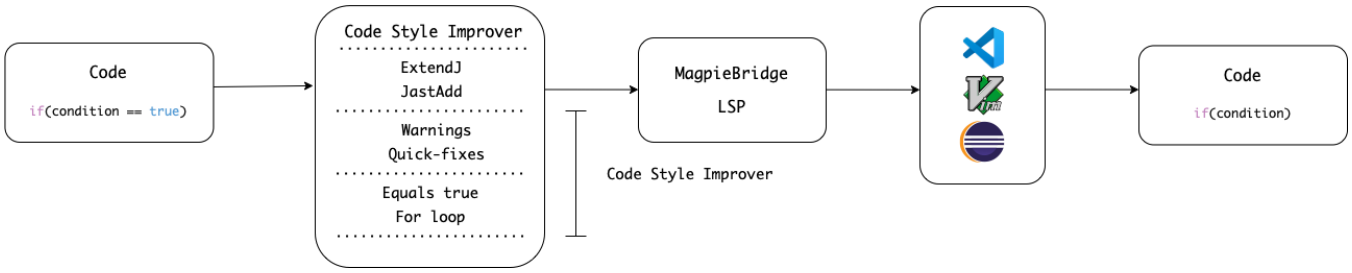

Figure 3. Architectural overview.

3.3 Implementation

The code style improver is implemented by modifying the AST using RAGs. Using the grammar it is possible to analyze code to find structures that might be potential style violations. We can add attributes for these structures to detect and fix the violations. The starting framework contains a type warning, which is the kind of message passed to the IDE. Warnings contain information computed in the attributes, such as information where the faulty code is located, a human-readable error message to show in the IDE, additional information to be shown in the IDE and it may also contain a fix for the violation as seen in the example in Listing 5.

Let us consider the example in Listing 1. The first if-statement consists of an expression within the parenthesis followed by a block of code. In this case the expression is of the type `EQExpr`, which in turn consists of two expressions separated by the equality operator, "==". New attributes are created in order to detect the potential violation. Firstly there is a boolean synthesized attribute called `isTrueLiteral`, that checks whether any of the operators actually equals true. There is a collection attribute that collects warnings when `isTrueLiteral` evaluates to true. As Listing 1 shows, the first if-statement can be simplified by only keeping the operand that is not a true literal. Therefore, there is a synthesized attribute of type `Expr` that returns the operand that is not equal to true. Working with the AST it is possible to generate new strings by replacing nodes in the tree. In this case the `EQExpr` is replaced by an `Expr`, the operand that was supopsed to be kept. The AST tree has now been fixed and the string containing the improved code can now be passed to the IDE as part of the warning message.

Listing 5. Example of a warning message sent to the IDE when detecting the type of violation shown in Listing 1.

```
EQExpr contributes warning(
          getCompilationUnit().pathName(),
          String.format("'%s' simplifies to '%s'",
              this.prettyPrint(),
              this.newPrettyPrintedEqTrue()),
          Analysis.AvailableAnalysis.IFEQTR,
          this.newPrettyPrintedEqTrue(),
```

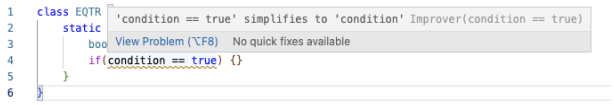

Figure 4. Warning message as shown in the IDE.

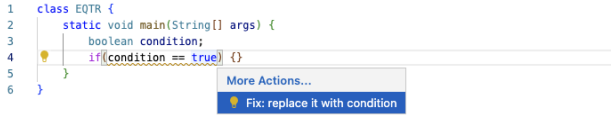

Figure 5. Suggested quick-fix as shown in the IDE.

```
        getRelatedInfo())
    when this.isTrueLiteral()
    to CompilationUnit.IFEQTR();
```

4 Evaluation

The tool was evaluated by two different approaches. The first approach is systematic tests, at every build a test suite will run checking that the tool produces the correct logical results for the given test cases. The second approach is manual testing of the tool using VSCode. The tool is manually tested on a subset of the test cases that were used in the systematic testing. In addition to testing that the tool is logically correct it tests if the warnings and fixes are correctly computed and displayed visually in the editor.

4.1 Constraints and Edge Cases

When evaluating the tool it is important to think about the constraints and edge cases concerning the problems that the tool is applied on. Some of the problems are by themselves more or less complex which influences the number of edge cases needed to taken into account and to which degree the covered cases of the problem must be constrained. The two examples provided in Figures 1 and 2 provide two problems with constrasting complexity illustrating the difference. The example shown in Figure 1 is simple, it concers few components of the language, it applies only to `EQExpr`, which in turn means that the applicable operands must be `Boolean`.

Examples of edge cases to consider are what happens if both operands are true-literals, or if all true-literals are covered, not just the string `true`.

In contrast the example of refactoring a for-loop to an enhanced for-loop as per the example in Figure 2, comes with a much wider array of possible cases. There are a couple key factors explaining why. For one, for-loop such as the one in Figure 2, makes use of a larger portion of the language compared to an `EQExpr`, each additional part of the language comes with edge cases on its own, but the combinations of different language constructs add further edge cases to take into account. Secondly, the nature of the problem is more general, iterating through a container is a broader problem compared to checking if one of two operands of an `EQExpr` has a specific value. Lastly, a big difference between the two examples is that when refactoring the for-loop a new local variable is introduced. It is therefore important to consider possible conflicts associated.

Due to the sheer volume of cases allowed by the language it is necessary to constrain the number of cases covered. One constraint that has been placed on the problem is that the refactoring is only possible when the index variable is only used for array access in the same collection that is used to determine how many iterations of the loop is to be performed. If the index variable is used for anything else it is not possible to refactor to an enhanced for-loop. Examples of edge cases to consider are what happens if the value of the index variable uses another convetion than indexing from zero, or what happens if the value of the index variable is offset by some integer value at the indexing of the collection. When refactoring the for-loop a new variable is introduced, it is important to consider possible conflicts that could arise.

5 Related work

There are other projects that are similar, such as Checkstyle [1]. Checkstyle is an open-source static analysis tool for Java. It automates the task of adhering to a predefined code style. Checkstyle contains a set of checks. An example is the check `BooleanExpressionComplexity` that checks if a boolean expression contains at most the specified max amount of boolean operators.

Similar to Code Style Improver, Checkstyle uses a modular approach where it is possible to extend the tool with more checks. Checkstyle also works with the AST in order to perform the analysis. One difference though is that Checkstyle uses the visitor pattern, while Code Style Improver takes an aspect oriented approach.

Another key difference between Checkstyle and Code Style Improver is the fact that the latter provides real-time analysis when a file with a .java extension is opened or saved. Checkstyle is run as a command line tool or as an Ant task.

6 Conclusion and Future work

Code Style Improver has room for improvement and expansion. In the future it can be improved to better support the for-loop so that it support all Java collections, in addition to arrays that are supported right now. There is also the possibility of adding new types of analyses. A concrete example for analyses is to take the Java version of the user into account, so that the tool would not suggest refactorings that are only available in newer versions, but it could also suggest refactorings that are new to the current version in the event that a user upgrades to a newer version of the language that contain new features. If one were to create a more complete product rather than the prototype of today, there are two main considerations. Firstly the amount of analyses available. As of right now there are too few analyses to provide any value to a user. It would also be wise to consider the target audience and concentrate and tailor the type of analyses. As an example, the equals-true example in Figure 1 is trivial enough that it would really only help a novice programmer, while it would not be much help to a professional programmer who knows better. On the other hand more complex refactorings might not be that useful to a novice whose main objective is to learn, if they do not understand why a certain refactoring was performed. To conclude, Code Style Improver has a lot of potentital. The protype prestented in this paper represent a very small subset of the possible analyses. While the examples presented do not provide much value alone they hint about the possibilities.

Acknowledgments

Thank you to Idriss Riouak for all your help and feedback.

References

[1] Checkstyle. 2022. Checkstlye. (2022). https://github.com/checkstyle/checkstyle

[2] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B. Rowe, and Nasser Giacaman. 2018. Understanding Semantic Style by Analysing Student Code. In *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3160489.3160500

[3] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029

[4] 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

[5] Linghui Luo, Julian Dolby, and Eric Bodden. 2019. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 21:1–21:25. https://doi.org/10.4230/LIPIcs.ECOOP.2019.21