Evaluating code quality

Simon Hayoz

Supervised by
Solal Pirelli
Prof. George Candea

Dependable Systems Laboratory, DSLAB
Ecole Polytechnique Fédérale de Lausanne, EPFL
June 6 2020

1 Introduction

In current programming languages, it is pretty straightforward to evaluate the correctness of code with tests, but what about the style of the code? Indeed, it is also necessary to produce code of high quality for both readability and maintainability.

Evaluating code quality is very often a manual process, which takes time and introduces the possibility of human error. The goal of this project is to create a tool to evaluate code quality automatically. This project is implemented in the context of the exams of a Software Engineering class where the students have to write code and the quality of their code is one of the grading criteria.

As everybody has their own definition of what a code of good quality is it may be useful to first define what could be automatically checked to quantify the quality of a code.

One solution could be of course to use a coding standard and impose a unique style for everybody with strict rules. This solution is used in a lot of companies and seems to be the best solution when multiple developers have to work on the same project. Indeed, it would be hard to work on the same code with really different coding styles. The developers would have to adapt to each other's style all the time. And what about small changes and refactoring? Should one use the creator's style while adding a small part to an existing code or keep one's own style? This would not scale at all and cause many problems.

In the context of an exam, on the other hand, it is not very relevant because students work on their own project and do not have to share their code with others. Thus, using their own style should not be a problem as nobody would have to maintain, understand their code or add some code on their own. All of which could cause a problem of style. Moreover, as the used approach is more portable, it can be used on the fly on any project and will adapt to the style of the project and only report wrongly styled parts. It is also a more modular solution, as it does not force a change to the configuration every time a part of the style changes.

The chosen solution is as follows: it will not force a unique pre-chosen style, but will infer the style "at runtime" by analysing the code and inferring any possible style. By analysing and reporting errors this way, students are free to use their own style and the analyser will only enforce the consistency of their style. They could for instance choose the camel case variable naming style and if the analyser detects that this style was mostly used, then it will just report any other found naming styles that is not camel case (or all lower which is a super type of camel case, see subsection 3.2).

The analyser has four different, fully automatic, checks:

- Blank lines: checks that there are no more than one blank line in a row
- Braces: checks curly braces position and one-liner block style
- Indentation: checks the indentation difference of any blocks
- Naming: checks the variable name style

Additionally to these four checks, there are three different specific design pattern checks that are implemented:

- Singleton pattern
- Builder pattern for *product* and *builder* classes
- Visitor pattern for *parent*, *children* and *visitor* classes

2 User guide

2.1 Checks

The analyser was implemented in Java 13, thus requiring Java 13 and also Gradle to run. It can be found and downloaded at https://github.com/simhayoz/eval-code-quality.

2.1.1 Terminology

Before the different checks, it may be good to first define different terms used in the following sections. Using the following example:

```
1 if(true) {
2     System.out.println("This is a statement");
3 } else if(false) {
4     System.out.println("This is another statement");
5 } else {
6     System.out.println("This is the else statement");
7 }
```

Parent: a parent element is the header part before any block. In the previous example, `if(true)` is the parent of:

- `else if(false)`
- `else`
- the statement on line 2
- both opening and closing braces on line 1 and 3

Moreover, `else if(false)` is the parent of the statement on line 4 and the opening and closing braces on line 3 and 5.
Statement: a statement is any Java element. In the previous example, the full `if` structure is a statement as well as the element on line 2, 4 and 6.
Child: a child element is one of `else if(...)`, `else`, `while(...);` of a do-while loop or `catch`. In the example above, `else if(false)` and `else` are children of `if(true)`.
Block: a block is any part of a code starting by an opening curly brace and finishing by a closing curly brace **or** for special cases, it can also be a one line block without braces, such as:

```
1 if(true)
2     *single statement*
```

2.1.2 Blank lines

The blank lines check will just check and report if there is more than one blank line in a row.

2.1.3 Braces

The braces check will check the different property regarding the position of the opening and closing curly braces, namely:

- The closing brace should be vertically aligned with its parent element
- All opening braces position should follow one of the two following possibilities:
  - The opening brace is on the same line as its parent:

    ```
    1 if(true) { /*...*/
    ```

  - The opening brace is on the line after its parent:

    ```
    1 if(true)
    2 {
    3     /*...*/
    ```

Moreover, if this style is chosen, the opening brace should also be vertically aligned with its parent.

- Child elements like `else if(...)`, `else`, `while(...);` of a do-while loop and `catch` statements position related to the closing brace of the previous block should respect the same style, namely one of:
  - The child element is on the same line as the previous block closing brace:

```
} else /* ... */
```

  - The child element is on the line after the previous block closing brace:

```
}
else /*...*/
```

    Moreover, if this style is chosen, it should also be aligned with the closing brace.

Furthermore, the braces check will also enforce single-line block style. There are four different possible styles choosing from any combination of the two following properties:

- Regarding the statement position:
  - The statement is on the same line than the parent element:

```
if(true) return true;
```

  - The statement is on the line following the parent element:

```
if(true)
    return true;
```

- Regarding block with/without braces:
  - The block has braces:

```
if(true) {
    return true;
}
```

  - The block has no braces:

```
if(true)
    return true;
```

2.1.4 Indentation

The indentation check will inspect the alignment of elements related to their parent and children statements. Specifically, it will check the following properties:

- It will check that the following elements in a java code are aligned left:
  - `package ...;`
  - `import ...;`
  - `class AnyClass {` (but only if the class is **not** an inner class)
- It will check that all statements within the same block are evenly indented and that they are more indented than their parent. For example, the following block is misaligned:

```
public static void example() {
    int i = 0;
      i = 1;
    i = 2;
    i = 3;
}
```

  The indentation check would report line 3 as misaligned in the previous block.

- It will check that the indentation difference of every block is the same

2.1.5 Naming

The naming check will examine the naming convention of different elements of the code. It will check separately the class name, variable name, etc. by group of modifiers (`private`, `public`, etc.) in the following way:

- It will check that both start characters are of the same type (i.e. both a letter, a digit, a dollar or an underscore)
- It will check that both end characters are neither a dollar nor an underscore, or that they are equal
- It will check that the rest of the name respects the same convention (see the possible convention in Table 1)). Moreover, two conventions can be equivalent if they are in the same tree path (see Figure 1 for the naming convention tree representation). See subsection 3.2 for more information regarding the different naming convention and its implementation.

2.2 Design pattern

Full examples of the following pieces of code can be found in the appendix of the corresponding sections.

2.2.1 Singleton pattern

The singleton pattern is a pattern that limits the instantiation of an object to one single instance. This instance can then be used instead of creating a new object every time.

To ensure that a class follows the singleton design pattern, the following checks are done on the given class (full examples: Example 2 and Example 3):

- There exists a unique private constructor:

```
5     private Connection() {
6     }
```

- There exists a unique static variable of type of the object:

```
3     private static Connection INSTANCE;
```

- The static variable is initialized only once
- The variable can be accessed:
  - It has the `public static` modifier and it was statically initialized:

```
7     public static final Stack INSTANCE = new Stack();
```

  OR

  - It is accessible through a `public static` method that either lazily initializes the variable or it was statically initialized before:

```
18    public static Connection getOrInitConnection() {
19        if (INSTANCE == null) {
20            INSTANCE = new Connection();
21        }
22        return INSTANCE;
23    }
```

Warning: to force "good" and machine-understandable lazy initialization for the variable, the lazy initialization should be done inside an if-statement as depicted above in the `getOrInitConnection()` method. If the initialization is not done inside an `if(INSTANCE == null)` or `if(null == INSTANCE)`, it will be considered incorrect. Indeed, as the analyser cannot check the `if` condition for more complex expressions, it cannot ensure whether multiple instances have been created every time the `getOrInitConnection()` method is called. Thus it will report an error of lazy variable initialization.

2.2.2 Builder pattern

The builder pattern is a software design pattern that offers a flexible solution for object creation. There exists two classes in the builder pattern: the product, which is the final class to build, and the builder, which is the helper class to build the product class.

To ensure that a product class and a builder class follow the builder design pattern, the following checks are done (full example: Example 4):

- The builder has at least one "construction" method that returns `this`:

```
56          public Builder addToDebt(int amount) {
57              studentDebt += amount;
58              return this;
59          }
```

- The builder has a "build" method that returns an object of type of the product:

```
61          public Employee build() {
62              return new Employee(uuid, fullName, studentDebt);
63          }
```

- It will also throw a warning if the build method is not called "build" or "construct". It is not an error to use other names, but should be good practice nonetheless.

2.2.3 Visitor pattern

The visitor pattern is a software design pattern that helps separate an algorithm from an object.

To ensure that a parent class, one or more children classes and a visitor class follow the visitor pattern, the following checks are done (full example, see subsection 5.3):

- Visitor has at least one "visit" method per child with child as argument of the method:

```
1 public interface Visitor {
2     public void visit(Book book);
3     public void visit(Fruit fruit);
4 }
```

- Parent has an "accept" method to be implemented by children **or** every children implement "accept" method with visitor as argument of the method:

```
1 public interface Item {
2     public void accept(Visitor visitor);
3 }
```

- It will also throw a warning if the "visit" method is not called visit and the "accept" method is not called accept. Same as builder pattern, this is not an error to use other names, but should be good practice nonetheless.

2.3 Cannot infer error

A not-so-common error is the *Cannot infer unique property* error. This error is thrown when the analyser found multiple possible styles and cannot infer a unique one due to the fact that there is more than one property with the maximum number of occurrences in the code. In the following example, there are two such errors, one on the variable name and one on the indentation difference of the different blocks:

```
1  package example;
2
3  public class CannotInferExample {
4  <4->private int withCamelCase;
5  <4->private String with_underscore;
6  <4->private boolean WITH_UPPER;
7
8  <4->public static void forIndentation() {
9      <8----->System.out.println("is more indented than previous block");
10     }
11 }
```

The naming check will report line 4 to 6 as it cannot infer a unique naming convention for field declaration with the `private` modifiers from the properties: *CamelCase*, *AllLowerUnderscore* and *AllUpperUnderscore*. Moreover, as the block from line 4 to 10 is of indentation difference of 4 and the block line 9 is of indentation difference of 8 (the "<...-...>" are here to help denoting the number of spaces there is), the indentation check will report that it cannot infer a unique property for indentation difference between 4 and 8.

2.4 Creating a test suite

The analyser is separated in multiple checks (see subsection 2.1) and can be launched on any local file/folder, even on multiple files and directories. To configure the test suite two solutions exist: using the command line or a JSON configuration file. The two differences are that the command line can specify the JSON file and it cannot create a design pattern check, only the JSON can do such check.
Warning: the command line overrides JSON configuration file parameter if both are defined.
This project is based on gradle and command line argument should be inside the gradle parameter: `--args="..."`:

2.4.1 Using the command line

The command line can be used to create a test suite using the following command:

```
-c,--check <arg>        Set the checks to do from the following
                        possibilities: 'braces', 'indentation', 'naming',
                        'blank lines', if not set or contains the value
                        'all' will run all the checks
-d,--directory <arg>    Path to directory containing Java file to analyze
-f,--file <arg>         Path to a Java file to analyze
-h,--help               Display this help message and exit
-hjson,--help-json      Display help message for JSON config and exit
-j,--json <arg>         Set the JSON config file (warning terminal
                        arguments will override config file if defined in
                        both)
-n,--name <arg>         Required: set the name of the current test suite
-o,--output <arg>       Specify output XML file
-s,--sysout             If present will print report to the terminal
```

Only the name is required as long as it was not defined in a JSON configuration. The *check*, *directory* and *file* arguments can take more than one argument, the argument must be separated by a comma.
Example:

```
./gradlew run --args="-d assets/manual/,src/ -f assets/tests/ExampleFile.java -
    n nameOfTestSuite -s"
```

The command above will run all the checks (as they are not explicitly defined) on all the Java files in the *assets/manual* and *src/* directories and sub-directories and on the *assets/tests/ExampleFile.java* file, is named *nameOfTestSuite* and should output the results of the test suite to the terminal result due to the *-s*.

2.4.2 Using the JSON configuration

The JSON configuration can use the same parameters as the command line and can additionally define a check for each design pattern. The configuration is as follows:

```
{
  "name": "required: name of the current test",
  "check": ["list of checks to do from the following possibilities: 'blank
    lines', 'indentation', 'naming', 'braces', if not set or contains the value
    'all' will run all the checks"],
  "directory": ["path/to/directory", "..."],
  "file": ["path/to/file.java", "..."],
  "output": "specify output XML file",
  "designPattern": {
    "singleton": "name of the singleton class (for nested class use '$' for
    separation)",
```

```
    "builder": {
      "builder": "name of the builder class (for nested class use '$' for
    separation)",
      "product": "name of the product class (for nested class use '$' for
    separation)"
    },
    "visitor": {
      "parent": "name of the parent class (for nested class use '$' for
    separation)",
      "children": ["list of name of children class (for nested class use '$'
    for separation)"],
      "visitor": "name of the visitor class (for nested class use '$' for
    separation)"
    }
  },
  "sysout": true or false (if true will print report to the terminal)
}
```

The only required parameter is the "name" one, if not defined on the command line.

2.5 Getting results of the test suite

There are two ways of getting the results of a test suite, as depicted on subsection 2.4. The first one is seeing results as a simple output on the terminal. The second one is to specify an XML output file. The resulting XML file will then have the following structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<testSuite name="name_of_test_suite">
    <check name="naming">
        <report number_errors="3" number_warnings="0">
            <errors>
                <description>
                    <positionDescription>
                        <namedPosition name="ForExample.java">
                            <position col="5" line="6"/>
                        </namedPosition>
                        <descriptor>
                            <description>naming convention...</description>
                            <was>{start:Upper,property:...}</was>
                            <expected>{start:Lower,property:...}</expected>
                        </descriptor>
                    </positionDescription>
                </description>
                ...
            </errors>
            <warnings/>
        </report>
    </check>
    <check name="braces">
        <report number_errors="1" number_warnings="0">
            ...
        </report>
    </check>
   ...
    <check name="singleton pattern for class <IvoryTower>">
        <report number_errors="0" number_warnings="0">
            <errors/>
            <warnings/>
        </report>
    </check>
    ....
</testSuite>
```

This output is more machine-readable and adds multiple layers for different depths of error reporting.

3 Implementation

This section will explain some tricky or useful implementation details.
The code can be found at: https://github.com/simhayoz/eval-code-quality.

3.1 Parsing java code

For static analysis of a java source file, it is mandatory to get a data representation that is easy to use and analyse. To get a good representation, in many compiled languages, the compiler will transform the source code into an Abstract Syntax Tree (AST). This tree representation of the different parts of the java source file makes it easier to iterate over data, to understand whether an element is contained inside another and to get overall information on the structure of the code.
For example, the following code:

```java
package example;

public class test {
    public boolean test() {
            System.out.println("Return true");
            return true;
    }
}
```

Example 1: Example code for a simple class

will be transformed into the following AST (simplified for readability):

```
root(Type=CompilationUnit):
    packageDeclaration(Type=PackageDeclaration):
        name(Type=Name): identifier: "example"
    types:
        - type(Type=ClassDeclaration):
            name(Type=SimpleName): identifier: "test"
            members:
                - member(Type=MethodDeclaration):
                    body(Type=BlockStmt):
                        statements:
                            - statement(Type=ExpressionStmt):
                                expression(Type=MethodCallExpr):
                                    name(Type=SimpleName):
                                        identifier: "println"
                                    scope(Type=FieldAccessExpr):
                                        name(Type=SimpleName):
                                            identifier: "out"
                                        scope(Type=NameExpr):
                                            name(Type=SimpleName):
                                                identifier: "System"
                                    arguments: value: "Return true"
                            - statement(Type=ReturnStmt):
                                expression(Type=BooleanLiteralExpr): "true"
                    type(Type=PrimitiveType):  type: "BOOLEAN"
                    name(Type=SimpleName): identifier: "test"
                    modifiers: keyword: "PUBLIC"
            modifiers: keyword: "PUBLIC"
```

It is easy to see from this tree that the method `test()` is a child of the class `Test`, which should be the case. Thus, it is easier for a machine to iterate over the children of any statement as there is a logical hierarchical tree between statements. Each node also contains useful information concerning its type, position and other object specific details.

Many different parser for Java exist, from the basic javac, which is the default java compiler, to EclipseJDT which is the java development kit from the popular IDE Eclipse. But finally, JavaParser[1] was chosen for parsing java code. JavaParser is one of the most popular Java parsers. It has great advantages compared to other parsers. Indeed, the way JavapParser represents the AST using Java

[1] JavaParser site

objects makes it an easy way to get the position of any element. JavaParser also uses recent Java version which makes use of many post Java 1.8 functional programming oriented techniques. All of these helped a lot for iterating and easily checking different conditions on different statements. Moreover, JavaParser was created partially for code analysis, whereas javac is more compiler oriented and EclipseJDT is more IDE warning and general IDE helper oriented. Thus, JavaParser was the best possible choice.
For example, for getting all `if`s condition inside a program, using JavaParser, you could simply do:

```
List<IfStmt> ifStmts = compilationUnit.findAll(IfStmt.class);
```

And then it is easy, for example, to check if any of these conditions has an else-if part:

```
ifStmts.forEach(ifStmt -> {
    if(ifStmt.hasCascadingIfStmt()) {
        System.out.println("Has else-if part: " + ifStmt);
    }
});
```

3.2 The naming convention implementation

To define a naming convention, every variable is split in two to three parts:

$$aCamelCaseVariable\_ \rightarrow \underbrace{\text{a}}_{\text{start}}\overbrace{\text{CamelCaseVariable}}^{\text{property}}\underbrace{\_}_{\text{end}}$$

$$aCamelCaseVariable \rightarrow \underbrace{\text{a}}_{\text{start}}\overbrace{\text{CamelCaseVariable}}^{\text{property}}$$

As depicted above, the implemented naming convention has two or three properties: one for the first letter, optionally one for the last letter and one for the rest of the word. The starting and ending properties are used to detect some naming conventions, such as starting or ending variables with an underscore when the variable is private. If a word is too small (i.e. one character long), then the starting and ending properties will just be equal.

The starting property is always defined, but to ensure that variables named like "*mId*" are not detected as an *AllUpper* property (see possible variable properties on Table 1), the ending property is defined if, and only if, the end character is an underscore or a dollar sign. Otherwise, the ending character is part of the "full" property. In the case of "*mId*", the naming convention will then be: *start:Lower,property:CamelCase*, which seems to be the most logical possible convention.

Two naming conventions are said to be equivalent, when both starting properties are equal, both ending properties are equal, or both do not exist and the full property is equivalent as described below.

The following full properties are possible for a variable:

Property name	Description	Example
AllUpper	all characters are upper letters	THISISAVARIABLE
AllUpperUnderscore	all characters are upper letters and they contain at least one underscore	THIS_IS_A_VARIABLE
AllUpperDollar	all characters are upper letters and they contain at least one dollar sign	THISISA$VARIABLE
CamelCase	camel case convention	thisIsAVariable
AllLower	all characters are lower letters	thisisavariable
AllLowerUnderscore	all characters are lower letters and they contain at least one underscore	this_is_a_variable
AllLowerDollar	all characters are lower letters and they contain at least one dollar sign	thisisa$variable
Underscore	It contains at least one underscore and has upper and lower letters	this_Is_A_Variable
Dollar	It contains at least one dollar sign and has upper and lower letters	thisIsA$Variable
Digit	It contains at least one digit	thisIs1Variable
Empty	the variable is empty (i.e. length is 0)	-
None	It follows none of the previous conventions	-

Table 1: Table representation of the different naming conventions

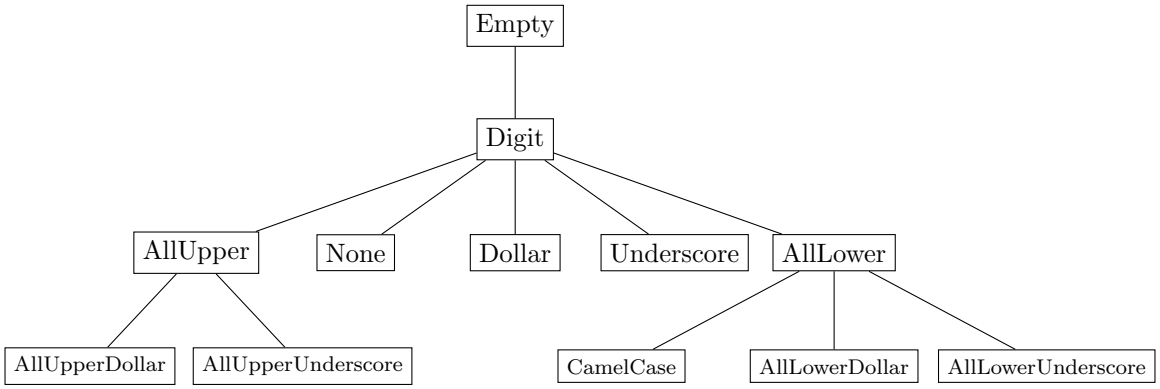

Figure 1: Tree representation of the different naming conventions

The tree representation (see Figure 1) helps understanding how different styles are linked. Two elements are linked if there is a direct tree traversal up or down from the node of the first element to the node of the second. Linked nodes are said to be equivalent.

For example, *AllLower* is the super type of *CamelCase*, *AllLowerDollar* and *AllLowerUnderscore*. Thus, `name` and `longerVariable` are using different styles, but are equivalent and the final style of the variable `name` could be inferred as *CamelCase*.

One problem of these inferred types is when there is multiple style on child and parent. For example, finding 20 *CamelCase* variables, 3 *AllLower* and 7 *AllLowerUnderscore*:

The logical solution is to report the 7 *AllLowerUnderscore* and infer the *CamelCase* style to the 3 *AllLower*. To do so, the following algorithm was implemented using the above tree:

Algorithm 1 Infer used variable property

Input: L list of (key, value) pairs of (style, elements), ordered decreasingly by the size of elements, where elements is a list of statements following the key style

```
inferredStyles empty list of (style, elements)
i ← 0
while i < L.size do
  wasFound ← false
  for each inferredStyles as inStyle do
    if !wasFound then
      if inStyle key is a parent of L[i] key then
        wasFound ← true
        add all position of L[i] to inStyle
      else if inStyle key is a child of L[i] key then
        wasFound ← true
        add all position of L[i] to inStyle and rename inStyle key to L[i] key
      end if
    end if
  end for
  if !wasFound then
    add L[i] to inferredStyles
  end if
  order inferredStyles decreasingly by the size of elements
  i ← i + 1
end while
return inferredStyles
```

This algorithm iterates over the list of properties and nodes in order of decreasing number of elements and will try to merge equivalent properties together. Doing so will ensure that the final inferred property (i.e. the property with the maximum number of elements in *inferredStyles*) is the right property and that it can report any other properties in the list as they cannot be equivalent to the inferred property. Moreover, that ensures that the chosen property is the right one. For example, if the following properties were found:

- 15 *CamelCase*
- 12 *AllUpper*
- 10 *AllUpperUnderscore*

this distribution should logically report the 15 *CamelCase* and choose the *AllUpperUnderscore* as the overall inferred style even if the most found naming convention was *CamelCase*. This is due to the fact that *AllUpper* and *AllUpperUnderscore* are equivalent and combining them gives $22 > 15$ elements. The algorithm above would generate a list with two elements:

- 22 *AllUpperUnderscore*
- 15 *CamelCase*

Thus, rightly report the 15 *CamelCase* and infer the *AllUpperUnderscore* property.

The implemented version of this algorithm uses an iterator instead of a list and iterates recursively with an accumulator instead of the *inferredStyles* list. This algorithm may have two main problems. The first one is that it could infer a property on the most often found property, even if multiple children together could represent the most often found property. For example:

- 10 *Empty*
- 9 *CamelCase*
- 6 *AllUpperUnderscore*
- 5 *AllUpper*

On the first two iterations, it will have a unique property of *CamelCase* with 19 elements and will finish the algorithm with the following properties:

- 19 *CamelCase*
- 11 *AllUpperUnderscore*

Thus, it will report the *AllUpperUnderscore* even if $11 > 9$ elements for the *AllUpperUnderscore* property. To palliate to this problem, one solution could be to not add parent properties to the list, but it may cause other problems. The solution as depicted above was kept as it should be good enough and does not add too much complexity to this algorithm. Indeed, similarly to what a compiler may do, this algorithm ensures that if there is an error, it will report it. Even if this is not exactly the right error to report. For example, a compiler may report the line after the one missing a semicolon and it is still easy to troubleshoot the error and find where it really appeared and how to fix it. The second problem, very similar to the first one, is when there are multiple properties with the same number of elements. If these properties are both child of the same element, it may cause inconsistencies between run and will report one or the other property randomly. It should report a *Cannot infer error* (see subsection 2.3) if this was the most found property. This should not be a problem if none was chosen as the inferred property but could nevertheless be if the tree was deeper.

3.3 Multi-line parent header problem

One problem of using JavaParser is the way it is handling positions. An element always has a start and an end position. For example, see Example 1 `test()` method position will be:

```
(line 4,col 5)-(line 7,col 5)
```

There is no information about the end of the parent position (the position of the closing brace of `public boolean test()`). This is not a problem in this example, but it can cause some problem for multi-line headers such as:

```
1     @Override
2     public String toString()
3     {
4         /* ... */
```

```
1     if(true
2         && true
3         && true)
4     {
5         /* ... */
```

In both examples, the start position is at (line 1,col 5) and there is no information regarding the end position of the header (i.e. respectively (line 2,col 28) and (line 3,col 16)). This is a problem when estimating if the open curly braces are on the same line than their parent, on the line after their parent or if it should report an error because it was more than one line after their parent.

To palliate to this problem, a custom heuristic for each kind of statement was used. All of them use nearly the same technique, namely to start at the last known position inside the header of a parent (for a method declaration, it could be the last parameter inside the parenthesis) and try to find the last non-null element before the curly brace, keeping tracks of the position.

To find the last non-null element, it will either use the String content of the file or the tokens of the parent element generated by JavaParser and iterate until the last non null element is found. Tokenization is another step in the compilation. It breaks the string content into different tokens by matching them with known tokens. For example, `this` is a keyword token and `thisIsAVariable` is an identifier, because it does not correspond to any known keyword.

4 Conclusion

The project's goal was to find a way to evaluate code quality without any prior. It was done using different techniques of static analysis to point out error of styling in Java code. The tool report error nearly always right and efficiently, even for bigger project and can give a good insights of small error that are easy to miss.

To go further, one could try and implement easy case scenario of complexity analysis. This, to enforce a certain complexity on an implemented algorithm but such analysis could be a project on its own, even for easy cases. Of course, due to the halting problem, it could not be done for all cases and could for example throw a warning for too complex algorithms.

5 Appendix

5.1 Singleton pattern

```
import java.util.ArrayList;
import java.util.EmptyStackException;
import java.util.List;

public class Stack {

    public static final Stack INSTANCE = new Stack();
    private final List<String> list;

    private Stack() {
        list = new ArrayList<>();
    }

    public static void main(String[] args) {
        Stack stack = Stack.INSTANCE;
        stack.add("this")
                .add("is")
                .add("an example");
        System.out.println(stack.pop());
        System.out.println(stack.pop());
    }

    /**
     * Add to the stack.
     *
     * @param newString the string to add
     * @return this
     */
    public Stack add(String newString) {
        list.add(newString);
        return this;
    }

    /**
     * Pop the last String added in the list.
     *
     * @return the last String in the list
     */
    public String pop() {
        int size = list.size();
        if (size > 0) {
            String last = list.get(size - 1);
            list.remove(size - 1);
            return last;
        }
        throw new EmptyStackException();
    }
}
```

Example 2: Singleton pattern

```
public class Connection {

    private static Connection INSTANCE;

    private Connection() {
    }

    public static void main(String[] args) {
        Connection currentConnection = Connection.getOrInitConnection();
```

```
        currentConnection.send("Hello World!");
    }

    /**
     * Get current connection or init it if it was never created.
     *
     * @return the current connection
     */
    public static Connection getOrInitConnection() {
        if (INSTANCE == null) {
            INSTANCE = new Connection();
        }
        return INSTANCE;
    }

    /**
     * Send a message through the current connection.
     *
     * @param message the message to send
     */
    public void send(String message) {
        System.out.println(message);
    }
}
```

Example 3: Singleton pattern with lazy initialization

5.2 Builder pattern

```
public class Employee {
    private String uuid;
    private String fullName;
    private int previousDebt;

    public Employee(String uuid, String fullName, int previousDebt) {
        this.uuid = uuid;
        this.fullName = fullName;
        this.previousDebt = previousDebt;
    }

    public static void main(String[] args) {
        Builder builder = new Builder()
                .setName("My Example")
                .setUuid("this_is_a_uuid")
                .addToDebt(25000)
                .addToDebt(5000)
                .addToDebt(5000);
        Employee employee = builder.build();
    }

    public static class Builder {

        private String uuid;
        private String fullName;
        private int studentDebt = 0;

        /**
         * Set the student uuid.
         *
         * @param uuid the student uuid
         * @return this
         */
        public Builder setUuid(String uuid) {
            this.uuid = uuid;
```

```
            return this;
        }

        /**
         * Set the full name of the student.
         *
         * @param fullName the full name of the student
         * @return
         */
        public Builder setName(String fullName) {
            this.fullName = fullName;
            return this;
        }

        /**
         * Add to the current debt.
         *
         * @param amount the amount to add to the debt
         * @return this
         */
        public Builder addToDebt(int amount) {
            studentDebt += amount;
            return this;
        }

        public Employee build() {
            return new Employee(uuid, fullName, studentDebt);
        }
    }
}
```

Example 4: Builder pattern, with product class: `Employee` and builder class: `Builder`

5.3 Visitor pattern

This example was adapted from: https://github.com/csoulakian/Visitor-Pattern-Java.

Visitor classes:

```
public interface Visitor {
    public void visit(Book book);
    public void visit(Fruit fruit);
}
```

Example 5: Parent visitor

```
public class PriceVisitor implements Visitor {
    @Override
    public void visit(Book book) {
        System.out.println("Price: " + book.getPrice());
    }

    @Override
    public void visit(Fruit fruit) {
        System.out.println("Price: " + fruit.getPricePerKg() * fruit.getWeight
    ());
    }
}
```

Example 6: Visitor implementation for price

```
public class NameVisitor implements Visitor {
    public void visit(Book book) {
        System.out.println("Book: " + book.getTitle());
    }

    public void visit(Fruit fruit) {
```

```
        System.out.println("Fruit: " + fruit.getName());
    }
}
```

Example 7: Visitor implementation for name

Parent class:

```
public interface Item {
    public void accept(Visitor visitor);
}
```

Example 8: Parent class

Children classes:

```
public class Book implements Item {
    private String title;
    private double price;

    public Book(String title, double price) {
        this.title = title;
        this.price = price;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

Example 9: Book item

```
public class Fruit implements Item {
    private String name;
    private double pricePerKg;
    private double weight;

    public Fruit(String name, double pricePerKg, double weight) {
        this.name = name;
        this.pricePerKg = pricePerKg;
        this.weight = weight;
    }

    public String getName() {
        return name;
    }

    public double getPricePerKg() {
        return pricePerKg;
    }

    public double getWeight() {
        return weight;
    }

    @Override
    public void accept(Visitor visitor) {
```

```
        visitor.visit(this);
    }
}
```

Example 10: Fruit item

Main class example:

```
import java.util.List;

public class App {
    public static void main(String[] args) {
        List<Item> shoppingList = List.of(new Book("To Kill a Mockinbird",
    14.95),
                new Book("Of Mice and Men", 7.89),
                new Fruit("peach", 3.95, 2),
                new Book("The Old Man and The Sea", 9.99),
                new Fruit("banana", 3.30, 0.5));
        NameVisitor nameVisitor = new NameVisitor();
        PriceVisitor priceVisitor = new PriceVisitor();
        for(Item item : shoppingList) {
            item.accept(nameVisitor);
            item.accept(priceVisitor);
        }
    }
}
```

Example 11: Main class example