# IntelliComment

An IDE Plugin to Improve Java Source Code Using Comments

**Francisco Machado Duarte**

Thesis to obtain the Master of Science Degree in

**Information Systems and Computer Engineering**

Supervisors: Prof. João Fernando Peixoto Ferreira
Prof. Alberto Abad Gareta

## Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Dr. Rui Filipe Lima Maranhão de Abreu

**September 2020**

# Abstract

Code comments are an underrated source of information about the source code. Because they are written in natural language, their analysis is a non-trivial task. To make use of this hidden information we propose a tool in the form of a plugin for a widely used Java IDE, IntelliJ Idea. The plugin has core features such as a scope finder that returns the scope of a comment. These features can be used to create personalized analysis that take into consideration the comment content. Some analyses are already implemented, such as one that highlights the comment's scope. Another checks for comments that could possibly be better used as method names. This last one was suggested in another paper and data confirms the appearance of such cases in production code. The tools have been made available to facilitate the creation of new analysis.

# Keywords

Code Comments, Java, IntelliJ Plugin, Comment Scope, Analysis

# Contents

# 1 Introduction

**Contents**

## 1.1 Motivation

Comments are the second most used artifact for code understanding, behind only the code itself [1]. They facilitate code comprehension [2,3] and the lack of comments leads to misunderstandings and low maintainability of software [4, 5]. As programmers, we write a great amount of comments in our code, informing whoever reads the code (ourselves included) about a diverse range of information about our code.

But these comments are only ever used by humans and are discarded as soon as compilation occurs. What if we could utilize all the information contained in natural language comments for something more?

The bigger challenge to using the comments in an automatic way is their very nature: comments are written in Natural Language. This means that a regular comment does not follow any specific structure. It is very hard to automatically determine what part of the code the comment is referring to. It is also difficult to extract meaning from the contents of a comment.

Finding a way to not only extract information from a comment but also to use it for comparison with the code it refers to could lead to finding an inconsistency. This inconsistency could mean either something wrong with the comment (in which case we correct it to get better documentation) or something wrong with the code (in which case we found a bug). In either case, finding this inconsistencies would always lead to the betterment of our final product.

## 1.2 Objectives

As we will see in the next chapter, much work as already been done in understanding both the meaning of a comment as well as its scope (the section of the code it refers too).

Our main objective is to put this knowledge to practical work. To create an extensible tool, an IDE plugin, that provides an easy way for the programmer to create its own analysis to code-comment relations.

With comments being written in natural language, there is a high amount of possibilities for code-comment inconsistencies. We cannot possibly create analysis for each one, so the next best thing is a tool that allows the definition of more analysis to suit the programmers' needs.

## 1.3 Structure

In this section I will describe how this document is structured. In Chapter 2 we will review a series of works that are related to our project as well as some of the possible tools available. We will explore work already done in both the area of understanding and classifying code comments as well as the area of defining the scope of a comment.

Chapter 3 will be dedicated to Functional Design. We will discuss all that the plugin must be able to do and it should be organized without restricting it to a single language or IDE.

In Chapter 4 we will discuss our implementation. Although the concepts described in Chapter 3 are applicable to multiple combinations of languages and IDE's, in this chapter we will discuss our decisions in the making of a functioning IDE for the IDE IntelliJ, for Java.

Chapter 5 we will make an evaluation of our implementation, verifying how easily a new Analysis can actually be implemented.

Chapter 6 is reserved for our conclusions and reflections on possible future work.

# 2 Background and Related Work

**Contents**

## 2.1 Background

For this section we decided to dived it into two different main aspects.

First, we will be dealing with code comments as our study subject, so we need to learn more about them. What work has already been done in this regard?

Second, other tools we might need in order to create our plugin. Examples of this would be Natural Language tools to interpret the comments, IDE's available that could be the base of our implementation or machine learning tools that could help uncover what the scope of a comment is.

### 2.1.1 Code Comments

We have proposed to use code comments as a source of useful information that can assist us while writing code. However, we are well aware that many comments do not carry information that can be used with this objective. With this in mind comes the need to separate comments based on their usefulness to us, to categorize them.

There have been made many different efforts to come to a set of code comments categories but their very nature (being written in natural language with no rules on how they should be made) made this a very difficult task. Bellow I will present some of these approaches and some reasons why these papers might be of great use to this project.

Code comment quality was typically measured quantitatively by the ratio of code to comments. But quantity did not always meant quality. To better assess the quality of a comment, classification system was created for code comments [6]. This system had in consideration the following features:

1. length. The number of words of a comment.
2. copyright. Whether or not the comment contains the words "copyright" or "license".
3. braces. How many braces in the code are open at the position of the code.
4. declaration distance. Distance in lexical tokens to the next method/field declaration.
5. frame. whether or not the comment contains a separator string multiple times (e.g., "***", "—", "///".
6. task tag. Whether or not the comment contains the words "task", "fixme" or "hack".
7. followed. Whether or not the comment was followed by another comment.
8. special characters. Whether or not the comment contains special characters (e.g., ";", "=", "(", ")").
9. code snippet. Whether or not the comment contains code snippets.

These features were then used to train a classifier with machine learning which would assign a category for each comment. The categories proposed by the authors were:

1. Copyright Comments. Comments include information about the copyright or license of the code.
2. Header Comments. Overview of the class and provide some metadata about the class (author, revision number, peer review status, etc).
3. Member Comments. Description of the functionality of a method or field.
4. Inline Comments. Description of implementation decisions within a method body.
5. Section Comments. Comments used as a delimiter for sections of code. Usually something like this:

   ```
   // ----- Getter and Setter Methods ----
   ```

6. Code Comments. Commented code.
7. Task Comments. Developer note for future work, such as TODOs or a bug that needs to be fixed.

Some of these are of little use to us, such as the Code Comments, because the information that is contained in those comments is not usable for our objectives.

On the other hand, Member Comments can be of great use. To help determine the coherence between a member comment and a method name a new metric was created: the coherence coefficient. This metric is a number from 0 to 1 given to a pair of a member comment and a method name. The closer to 0, the less related the two are. So if a comment has a coherence coefficient of 0 (it has nothing to do with the method name) the authors established that it meant one of two things: either the comment was not useful enough and should be enhanced or removed or we were in the presence of a bad identifier, in which case the identifier itself should be improved. This is an example of the possible usefulness of Member Comments, as the comment provides utilizable information.

Bellow is a real example of a Member Comment.

```
public void calc(double[][] A) {
    for (int i = 0; i < n; i++) {
     for (int j = 0; j < n; j++) {V[i][j]= A[i][j];}
    }
    tred2(); // Tridiagonalize.
    tql2(); // Diagonalize.
    ...
}
```

This excerpt of code is a good example of something our plugin could do to help the programmer. Upon analyzing this case, the plugin could give the user the option to rename the method with the comment being the new name.

So, from this work there is much that can be used. Be it either the categories defined and the features used for comment categorization (that would help greatly with the code comment classifying part) or the several examples of specific conditions that could lead to the assumption of something based on the comments (like the example above) that in turn could be used as example analysis for our plugin.

The previous work dealt more with code comment quality. But the exercise of pure classification is also important and further developed in the work of Luca Pascarella and Alberto Bacchelli [7].

Once again, classification is of great importance for our project as not all comments provide usable information and we need a way decide between useful comments and not so useful ones.

In this work, the first thing that was done was defining the categories for comments. These are too high level to be used on specific analysis but are good enough to differentiate between a good and a bad comment (for our purposes). The categories defined were these:

- A. **Purpose**
  - A.1 Summary
  - A.2 Expand
  - A.3 Rationale
- B. **Notice**
  - B.1 Deprecation
  - B.2 Usage
  - B.3 Exception
- C. **Under Development**
  - C.1 TODO
  - C.2 Incomplete
  - C.3 Commented Code
- D. **Style and IDE**
  - D.1 Directive
  - D.2 Formatter
- E. **Metadata**
  - E.1 License
  - E.2 Ownership

  - E.3 Pointer
- F. **Discarded**
  - F.1 Automatically Generated
  - F.2 Noise

A code comment from any of the F categories should be instantly ignored (as it does not provide any usable information for our scope) while one from category A will undergo extra analysis.

For this classification to be possible, the authors of this paper developed a web application and manually classified over 2000 files. Both the application and data set are publicly available, both things that can contribute a lot to our work.

Yet another taxonomy is possible and studied. Instead of a comment being able to fit to one and only one category (like in the above classifications), the comments could analyzed in 4 different dimensions, given a variable number of classification tags [8]. These are the dimensions:

- Object of the Comment. This dimension dealt with the relative position of the object of the comment in relation with the comment itself. It had these subcategories:
  - Follow. Comment concerns following instruction.
  - Block. Comment concerns following block of instructions.
  - Nocode. Comment does not concern any code in its vicinity.
  - Other. For all others situations. For example, when the code is after the instruction it refers to.
- Comment Type. It is possible for a comment to be of these types:
  - Explanation. When it describes functionality of the related code.
  - Working. Comments describing future tasks (TODO) or with code still being made. Does not explain functionality.
  - Code. Just plain commented code.
  - Other. Any other possible type, such as licensing and credit.
- Style. This dimension is only in regard to the Explanation type comments. It only has two kinds:
  - Explicit. Description with instruction keywords and identifiers.
  - Implicit. Description with abstract terms.
- Quality. Another dimension only applicable to comments of type Explanation. A subjective measure that describes how well the comment explains the related code. It can be:

  – Fair+. Presents the functionalities of related code and other information.

  – Fair. Presents only the functionalities of the related code.

  – Poor. Only a few or none of the functionalities are described.

Bellow is a real example of a code comment:

```
//if the document was an auxiliary file, remove it from the list
if (doc.isAuxiliaryFile())
 removeAuxiliaryFile(doc);
```

To classification of this example, using the taxonomy explained above, would be the tags:

- Follow, from the dimension Object of Comment because the comment is about the code that follows it.
- Explanation, from dimension Comment Type, as it explains what the following code is doing.
- Explicit, from dimension Style. Because it has the tag Explanation, the dimension Style is relevant to this comment. It is explicit as the explanation uses the identifiers of the method (auxiliary file).
- Fair, from dimension Quality. Same as Style, because the comment has the tag Explanation, this dimension is also relevant. This

The biggest difference from this taxonomy to the others we have already discussed is that this one focus more on the spatial relation between the code and the code comments. The dimension Object of Comment is a really important factor in trying to automatically understand to what code is a specific comment related to.

There is another paper following the same approach (classifying comments on different dimensions) with the big difference of using OS code comments instead of the usual OOP and Java comments, this paper still provides useful information in regards to the code classification, mostly due to the more specific examples and fine classification. For example, one category they provide is IntRange. Comments that belong here are specific about the range of the integers used in the code, information that can be used to verify that code [9].

### 2.1.2 Other Tools

There are many different tools available that could prove useful for what we will try to accomplish.

- IntelliJ/Eclipse/Visual Studio. All of these are IDE's for Java with the capability of creating plugins and could be the mainframe to support our implementation of the plugin.

- Infer.AI[1]. This is a tool that belongs to Facebook, and used to write and run Java code analysis with many analysis already available.
- Opal[2]. This tool utilizes Java bytecode to conduct static analysis and has a specific module dedicated to Abstract Interpretation, something we may be specially interested considering our approach to Value Range Analysis.
- WALA[3]. Watson Libraries for Analysis, another tool that makes use of Java bytecode. There is a number of more specialized tools[4] based off of this tool.
- Spoon[5]. This tool is usable for source to source transformations, working on the Abstract Syntax Tree of the source code. Useful for when the plugin finds an inconsistency to which the solution is a little bit of code rewriting.
- Core NLP[6]. A very useful tool to deal with all the natural language problems that will arise from working with natural language written comments.
- Weka 3[7]. Contains a collection of visualization tools and algorithms for data analysis and predictive modeling, usable to discover the scope of a comment.

## 2.2 Related Work

We will start this section by addressing a very pertinent question: how does this work reconciles with the notions of experts like Martin Fowler that code should be able to be read and understood without comments? How can a work that deals mostly with code comments can be important in a world that is walking towards the norm of no comments in code?

We have two arguments to promote the value of our work.

The first one comes from Martin Fowler himself. In an article by Fowler [8] he explains that the objective is not to just stop writing comments. The objective is to shift the view that the programmer has that code is not a type of documentation. When code is not documentation, presentation loses value. There is not a need to understand the code just with the code, because that is what the documentation is for. But my making code also documentation, a programmer needs to make the code do what it should, but also be perceptible and self explained. This would lead to better code overall which would also lead to

[1] https://fbinfer.com/
[2] http://www.opal-project.de/
[3] http://wala.sourceforge.net/wiki/index.php/Main_Page
[4] https://github.com/wala/WALA/wiki/WALA-Based-Tools
[5] https://github.com/INRIA/spoon
[6] https://stanfordnlp.github.io/CoreNLP/
[7] https://www.cs.waikato.ac.nz/ml/weka/index.html
[8] https://martinfowler.com/bliki/CodeAsDocumentation.html

less need of comments. But not zero need. Comments will still exist and be important, as other types of documentation. It is just that code now has even more responsibilities.

So even in the world idealized by Fowler, our work still has a place. Perhaps even more so. With enough flexibility, the plugin can me made to search for possible situations where less comments would be needed and help the user towards Fowler's objectives.

The second argument to be made is a more tangible one. Martin Fowler has been sharing his ideas for years now. If the community was in fact shifting in that direction, little to no work would be made in this aspect. But this is not the case. As we will see next, many teams are working with comments in code trying to get more uses out of them. There is a very current and sizeable community interested in this topic and working on it.

In 2019, there was a large-scale empirical study made on code-comment inconsistencies [10]. It was just a study, so nothing was done yet with the information collected, but is a great step in order to automatize the checking of code comment discrepancies in order to find bugs or possible bad documentation.

In the same year, another paper was published with a way to generate and propagate comments automatically [11]. This would be a great help on maintaining a good documentation with comments without much effort from the programmer.

Also in 2019, there was work done in order to automatically detect the scope of source code comments [12]. As we already discussed, this is a very challenging theme and a machine learning solution has already been provided.

From this year, 2020, a machine learning model capable of suggesting comment locations was developed [13]. Again, in great help with assisting programmers with creating better, more useful, comments by suggesting where new ones should be made.

As we can see, considerable effort is being put into this are at the moment . Our work is the next step, trying to utilize knowledge obtained to create a day to day tool capable of helping the programmer with their activities.

# 3

# Functional Design

**Contents**

In order to create a plugin that does what we proposed there is a set of requirements that must be met. These are not coupled with the technical implementation, so they maintain value independently of what IDE we are using.

This chapter addresses all that the plugin needs to be able to access, process and modify and give a general look on how its architecture is structured.

## 3.1 Requisites

### 3.1.1 Process Language Elements

First of all, if the goal is to suggest code improvements, there needs to be a way access the code being written by the user. We thus need a way to extract any element of the code that might be relevant for our analysis, being the comments themselves or code structures like cycles, method calls, variable declarations, etc.

An IDE that allows the creation of Plugins usually will provide some model of the code to be access and manipulated.

### 3.1.2 Process Text from Comments

Besides having access to the code and comments, there is also the need to interpret the contents of a comment. Because a comment is written in Natural Language, some sort of Natural Language processing tool will be required, capable of interpreting meaning of the words in the comment as well as the overall message it tries to convey. Also it is important to be able to distinguish between types of comments. A comment might be trying to convey the purpose of the referenced code, inform that some part of the code is still under development or just be plain noise like commented code. This categorization might be important to understand the value of a comment [7].

### 3.1.3 Find Comment Scope

If we are trying to use the comments to get some sort of knowledge about the referenced code it will be of great importance to know what portion of the code the comment is referencing.

We thus need a way to infer the scope of a given comment. This is not a trivial task and we considered two possible approaches for it. We can implement one of the many scope identifier options reviewed in the related work, either only one or a set of them for different possible situations. The other less intelligent approach would be putting the work on the client side. The client would be responsible for annotations in the code such as:

```
//Comment
//------Start------
    *Snippet of Code*
//------End-------
```

In this case, the first comment would be referring to that specific snippet of code and the Plugin would be able to utilize that information because of the formal structure imposed.

### 3.1.4 Create New Analysis

Having a way to access the code elements, to understand the meaning of a comment and to know exactly what part of the code is being commented would do no good if we could not use this to our advantage. The ways in which this could be used are many and that is the point: different programmers on different projects will also need possibly very specific analysis.

We wish to support the most generic of analysis. Because of the huge amount of comment types, the amount of possible analysis is also very big. Then the plugin should be extensible, and the process of this extension should be possible and easy for any programmer.

### 3.1.5 Modify Language Elements

As previously pointed, the IDE should give us some way of accessing the model it has of the code so that we can work on it. Using all the information from the comment and its scope and running it through an analysis will output a set of possible changes that could be made to improve the code. Having the capability of actually executing such changes and refactoring some of the code or add some extra lines is again an important aspect that should be present.

### 3.1.6 User Communication

Communication with the user is key. For example, if the plugin finds some discrepancy between a comment and the code it refers to, it should not change the code immediately. It is important to keep the user in the loop, since analyses relying on natural language will never be 100% accurate. What should be done is to present an array of options to the user. Maybe the user just wants to ignore this case because it was a false positive. Maybe the problem is in the comment part of the relation or maybe it is on the code part. The user should be given full control on how to deal with the situations that the Plugin finds.

## 3.2 Architecture

Considering all these aspects we can now create an architecture for our Plugin. The design we achieved was as represented on figure 3.1.

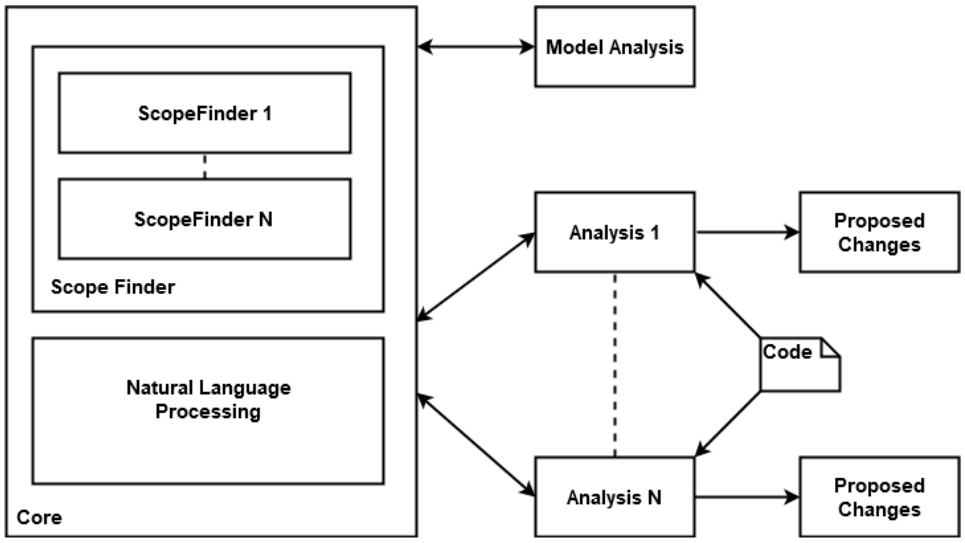


**Figure 3.1:** Plugin Architecture

Analyses are independent from each other. They scan the code individually and then each possibly suggests a set of changes to be made. The logic behind the analysis is contain within it, but some general functionalities will be made available to all via the core, with general stuff as finding the scope of a comment or some Natural Language Processing.

# 4 Technical Design

## Contents

## 4.1 Choosing Language and IDE - IntelliJ

With the objective of our work being the conceptualization of an IDE plugin, having an heavy discussion about which one is the best is not in our best interest. The idea is that, given the first part of the project, we should be able to implement the plugin in any IDE given that it satisfies a minimum set of requirements. Nevertheless, we should discuss what options exist.

First of all, we need to chose which language to work with. The initial proposed language was Java as other work had been made by colleagues in this area with Java. It was a language that we were comfortable with and there is much work already done with comments with Java. There were no reasons to change our target language, and so Java was decided.

With Java decided, three main IDE's come to mind: Eclipse, Visual Studio and IntelliJ. From these three, all of them complied with the requirements from the previous chapter and so, the plugin could be implemented in any of them.

We chose IntelliJ due to it's large community, lots of online support on forums, great resources in documentation, specially to create plugins, and lots of tools that it provides and would be of extreme importance to us, such as the PSI or the Inspections mechanism. Also, it was the one I was most comfortable with as I have used it before.

Before explaining our implementation of the Plugin, we will start with describing some functionalities that IntelliJ possesses that will be used later on.

### 4.1.1 PSI

First of all, the PSI. It stands for Program Structure Interface and is the layer of IntelliJ responsible for parsing files and creating the syntactic and semantic code model used by many of it's other features. This is the mechanism that allows the access and modification of the actual code written by the user and as such, of extreme importance for our work.

### 4.1.2 Inspections

Inspections are another feature of IntelliJ. It is a piece of software ran constantly on the background. It uses visitors of specific PSI elements to find specific things on the code. For example, in our case, we could make a visitor that visits PsiComment elements making it visit every comment in the code. With this access to the code we can now decide if something needs to be done (Inspection logic) and give the option to the user to call a "Fix" class that will resolve the issue found.

Inspections will be further explained as we showcase how our plugin works and give specific Inspection examples we made.

## 4.2 Implementation of the Requirements

We can already see how many of the requirements from Chapter 3 will be solved just by learning about specific IntelliJ functionalities. Even so, we will go one by one and show how the requirements were met.

### 4.2.1 Process Language Elements - Psi and Visitors

IntelliJ does indeed give us access to a model of the written code. Through Visitors, we are able to visit any specific element of the code (for example commentaries), check it's content and all of its surroundings and references.

### 4.2.2 Process Text from Comments - CoreNLP

Nothing from IntelliJ itself allows us to Natural Language Processing of the comments. But there are libraries that can be used for this effect. The one we chose for this project is CoreNLP [14]. Created by the Stanford NLP Group, this library as a big multitude of tools to be able to process natural language such as:

- Token and Sentence Boundaries
- Parts of Speech
- Named Entities
- Numeric and Time Values
- Dependency and Constituency Parses
- Coreference
- Sentiment
- Quote Attributions
- Relations

With this tool, our NLP needs should be mostly met, depending on what the Inspection needs to function.

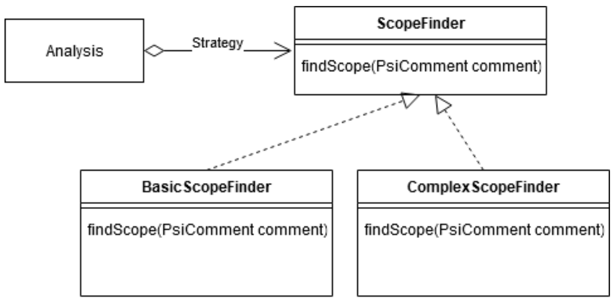
For the work that has been done, mostly the part-of-speech tagger from the CoreNLP library was used. This function receives a sentence and returns all the tags of the words on the sentence, allowing the identification of the nouns and verbs.

Any other library capable of such thing could be considered a viable replacement.

### 4.2.3 Find Comment Scope - ScopeFinder

The main core functionality that almost all Inspections will feel the need to use is the ScopeFinder.

As it is now, the ScopeFinder has a very simplistic way to determine the scope of a comment. However, everything was set up using the Strategy Pattern (as seen in fig. 4.1), allowing for new, more complex ways of determine a comment's scope to be implemented and used on already existing Analysis seamlessly and with little effort. Examples of such ScopeFinder implementations were referenced in the related work.



Analysis
Strategy
ScopeFinder
findScope(PsiComment comment)
BasicScopeFinder
findScope(PsiComment comment)
ComplexScopeFinder
findScope(PsiComment comment)

**Figure 4.1:** UML: ScopeFinder

As to the existing version of the ScopeFinder it makes use of the PSI structure to return a scope. Given a comment, it checks what is the code structure after it and returns that element. The return is always a PsiElement, as a PsiElement can contain other elements and be a structure of different dimensions.

### 4.2.4 Ease of New Analysis Creation - New Analysis Procedure

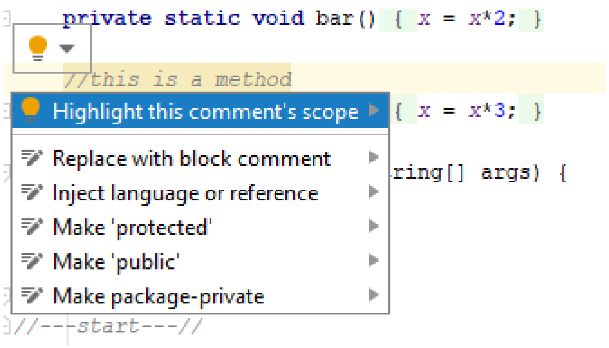
For us and in this particular case, we were happy with a clear example of how an Analysis is supposed to be created that could be followed while creating our own. We have this example not only on the IntelliJ Documentation but also another one in Chapter 5. The examples show and explain the different steps needed to create a new Analysis in detail and also the various elements that an Analysis needs to have in order to function.

### 4.2.5 Modify Language Elements - “Fix” Classes

Every time a warning is issued in IntelliJ, we are given a set of options to deal with said warning. Each of these options are mapped to a ”Fix” Class. This class implements an IntelliJ class called LocalQuickFix and has to define three specific methods (more detail in example in chapter 5). It receives a problem as input and applies some sort of fix to it, resolving the issue. To do so, we need the capability to edit the code as we might need to remove, add or just change part of it. The PSI helps again in this regard, providing the ability to access the element related to the ”problem”, all the elements that reference it and all those around it and allowing for the manipulation of all.

### 4.2.6 User Communication - JPanel and Multiple ”Fix” Classes

Communication with the user here is a simple set of messages that can be given. If the plugin finds an issue it communicates it to the user with a warning. Hovering the warning, the user gets to see a description of the problem as well as a set of possible solutions to it, differing in size depending on the Analysis. This visual communication with the warnings and the options is made possible by another key element of the Inspections tool provided by IntelliJ called JPanel, show in fig. 4.2 Depending on the number of possible ”fixes” for a given problem, the user should be able to call the multiple corresponding ”Fix” Classes, each one being an option on the JPanel.



```
    private static void bar() { x = x*2; }
    //this is a method
                                { x = x*3; }
                                ring[] args) {
//---start---//
```

Highlight this comment's scope
Replace with block comment
Inject language or reference
Make 'protected'
Make 'public'
Make package-private

**Figure 4.2:** JPanel

## 4.3 Install and Run

There are two ways for a user to install and utilize the plugin.

The first one is still in progress at the time of writing. It will be through the IntelliJ Plugin Marketplace where the plugin is currently waiting approval to be added. When available, all the user needs to do is install it through the plugins interface inside IntelliJ and it is done.

The second way is the manual way and will be need if the user intends on adding another Inspection to the plugin. Just download the plugin zip, and manually install it in the plugins interface.

## 4.4 Creating and Utilizing New Inspections

For the user that wishes to extend the plugin with his own personalized Inspection, a few steps must be taken.

First, the source code for the Plugin needs to be downloaded. Within it, the user creates a new Inspection, following the steps that will be described in detail in Chapter 5.

Once the Inspection is done and since the project is built using Gradle, just run the command ”build-Plugin”. This will create a zip with the new extended Plugin in build/distributions which can then be utilized to install the plugin in the local IntelliJ.

# 5 Evaluation

Contents

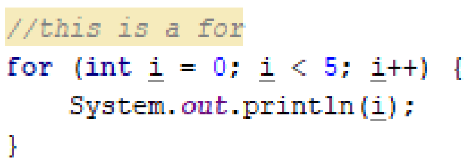
The objective of this project was an extensible plugin where the user could use some provided functionalities like the ScopeFinder to create personalized Analysis. How do we evaluate this then? The natural answer would be: in the end, how capable where we of creating a new Analysis?

For this we needed to follow our model analysis to create a new one with a specific purpose. This chapter will be divided in two parts. First, the explanation of the Model Analysis and the process of creating a new one. And after, the implementation of a new personalized Analysis, the One Word Comment Analysis.

## 5.1 Model Analysis

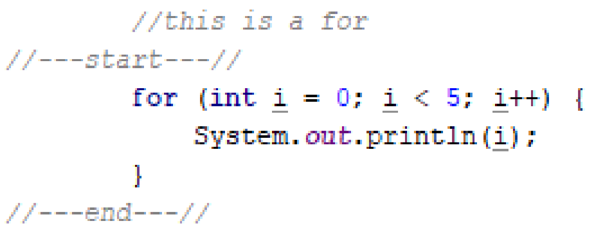
For this one we decided to make a very simplified analysis that would utilize all of the functionalities the plugin provides. For this, we chose a scope highlighter.

This analysis would find all comments on the code, ask if the user would want to highlight the scope of each one and, if yes, go highlight it. The code should go from something like fig. 5.1 to fig. 5.2.



```
//this is a for
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

**Figure 5.1:** Before Scope Highlight



```
        //this is a for
//---start---//
        for (int i = 0; i < 5; i++) {
            System.out.println(i);
        }
//---end---//
```

**Figure 5.2:** After Scope Highlight

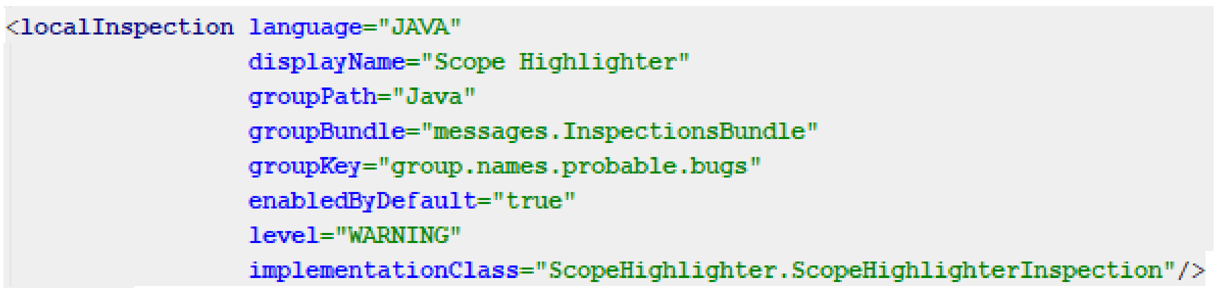
In order to create a new Analysis there are several steps that need to be taken.

### 5.1.1 Step 1 - plugin.xml

The first step to create a new Analysis is to edit the plugin.xml in order to include it.

On the ”extensions” section we need to had a new ”localInspection” field such as:

All of its sub fields should be filled appropriately in accord to the Analysis being implemented.

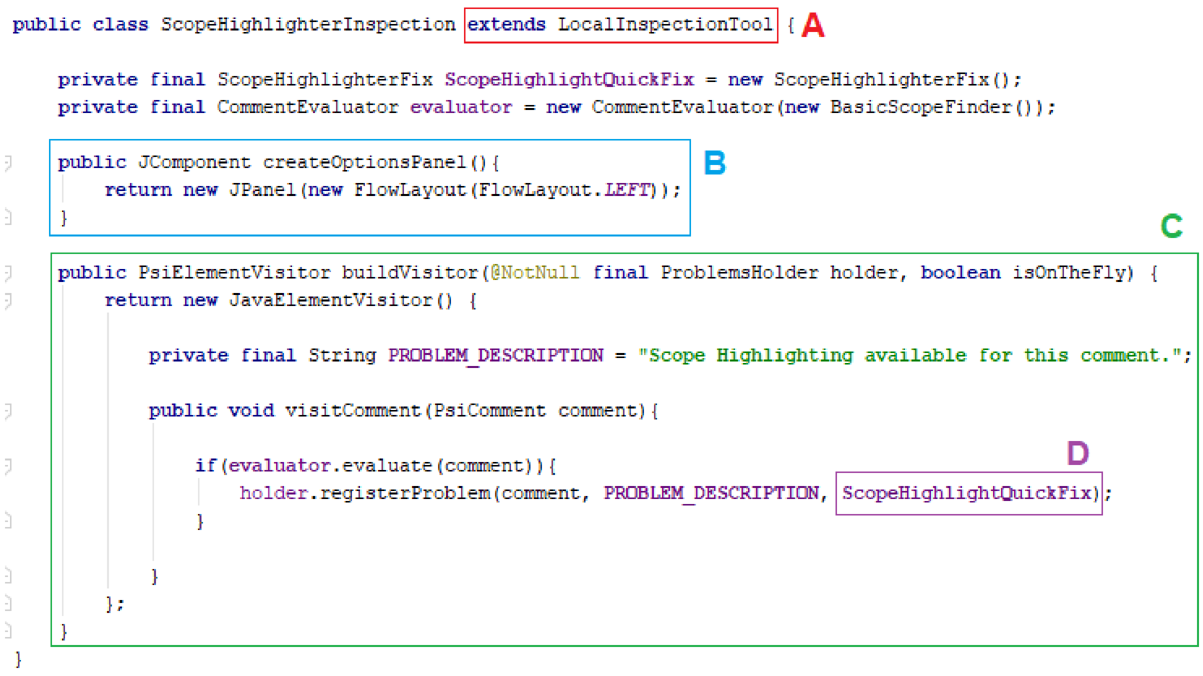


```
<localInspection language="JAVA"
                 displayName="Scope Highlighter"
                 groupPath="Java"
                 groupBundle="messages.InspectionsBundle"
                 groupKey="group.names.probable.bugs"
                 enabledByDefault="true"
                 level="WARNING"
                 implementationClass="ScopeHighlighter.ScopeHighlighterInspection"/>
```

**Figure 5.3:** localInspection field in plugin.xml

### 5.1.2 Step 2 - Create a new Inspection class

The last camp to fill on step 1 is the implementation of the Analysis we want to do. There is a very concrete set of characteristics that the Inspection class should have.



```
public class ScopeHighlighterInspection extends LocalInspectionTool {

    private final ScopeHighlighterFix ScopeHighlightQuickFix = new ScopeHighlighterFix();
    private final CommentEvaluator evaluator = new CommentEvaluator(new BasicScopeFinder());

    public JComponent createOptionsPanel(){
        return new JPanel(new FlowLayout(FlowLayout.LEFT));
    }

    public PsiElementVisitor buildVisitor(@NotNull final ProblemsHolder holder, boolean isOnTheFly) {
        return new JavaElementVisitor() {

            private final String PROBLEM_DESCRIPTION = "Scope Highlighting available for this comment.";

            public void visitComment(PsiComment comment){

                if(evaluator.evaluate(comment)){
                    holder.registerProblem(comment, PROBLEM_DESCRIPTION, ScopeHighlightQuickFix);
                }

            }
        };
    }
}
```

**Figure 5.4:** Inspection class

As we can see in fig. 5.4 there are 4 main aspects, classified with the letters A, B, C and D.

#### 5.1.2.A LocalInspectionTool

The new Inspection class must extend from LocalInspectionTool. This in itself will force some other paramethers to be met.

#### 5.1.2.B JPanel

These two lines of code are responsible for the creation of the little window with options that appears when we hover over an highlighted par of the code.

#### 5.1.2.C Visitor

This is where we decide what in the code we actually want to inspect. In our case, we create a visitor for PsiComment. This will mean that every time the inspection comes across a comment in the code, the piece of code inside will run.

In our case, we will evaluate the comment and depending on what that evaluation returns we decide between doing nothing or calling the ”fix” class. The evaluator will also need to be implemented.

#### 5.1.2.D ”Fix” Class

This is the class capable of refactoring the code in order to fix the problem that the inspection might have found. Will also need to be implemented.

### 5.1.3 Step 3 - Extra logic for the Inspection

Having found the parts of the code we are interested now we need to do something about them. This is the part that will differ completely from Analysis to Analysis. In this simple case, we just want to evaluate the comment, and to do so, we use the evaluator class. This is a class also made by us that will look at the content of the comment and check two things.

First, is this comment an actual original comment, or is it one of the ”//——start——” that the Analysis is capable of adding?

Second, is the scope of this comment already highlighted?

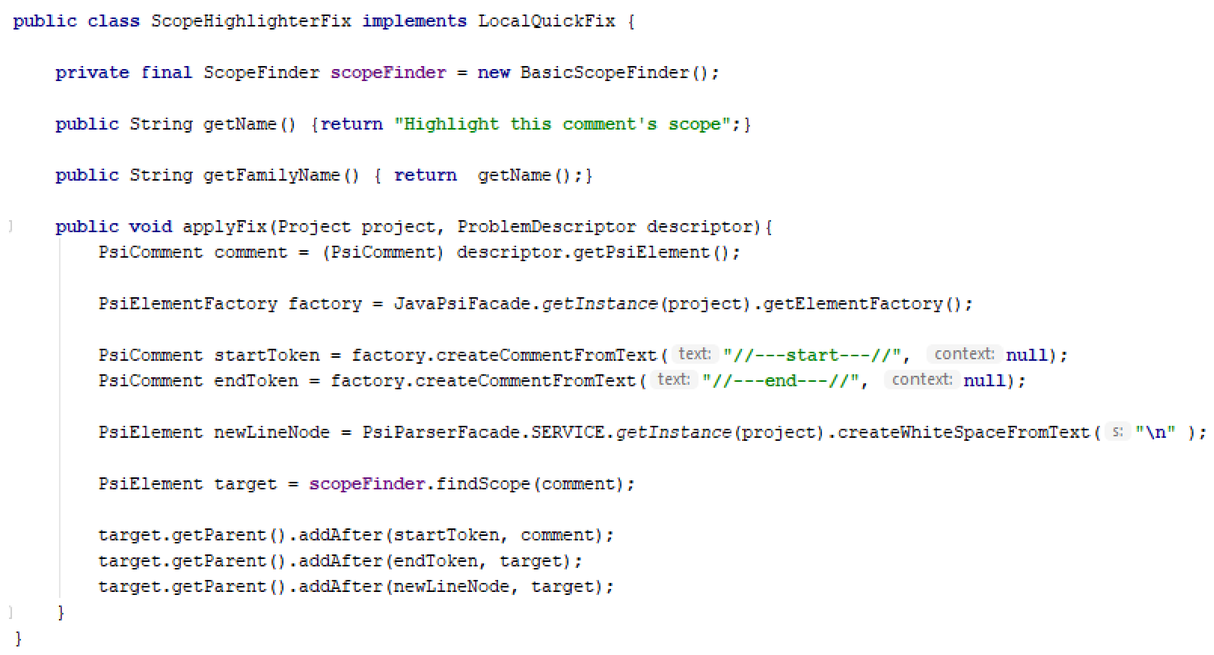
By answering these two questions, only not yet highlighted original comments will be flagged for a possible change.

### 5.1.4 Step 4 - The ”Fix” Class

Having discovered what we might want to change, we need a way to do so. In our example, only one ”fix” class is given as an option but it could have been many if the Analysis needed so.

The ”fix” class we use for this case is shown on fig. **??**.

The first important aspect is the need to implement LocalQuickFix. Then at least three diferent methods also need to be implemented. The getName() and GetFamilyName() will be used to fill in the option of the JPanel with the user hovers over marked code. The most important one will be the



```java
public class ScopeHighlighterFix implements LocalQuickFix {

    private final ScopeFinder scopeFinder = new BasicScopeFinder();

    public String getName() {return "Highlight this comment's scope";}

    public String getFamilyName() { return  getName();}

    public void applyFix(Project project, ProblemDescriptor descriptor){
        PsiComment comment = (PsiComment) descriptor.getPsiElement();

        PsiElementFactory factory = JavaPsiFacade.getInstance(project).getElementFactory();

        PsiComment startToken = factory.createCommentFromText( text: "//---start---//",  context: null);
        PsiComment endToken = factory.createCommentFromText( text: "//---end---//",  context: null);

        PsiElement newLineNode = PsiParserFacade.SERVICE.getInstance(project).createWhiteSpaceFromText( s: "\n" );

        PsiElement target = scopeFinder.findScope(comment);

        target.getParent().addAfter(startToken, comment);
        target.getParent().addAfter(endToken, target);
        target.getParent().addAfter(newLineNode, target);
    }
}
```

**Figure 5.5:** ”Fix” Class

applyFix() method. This method always receives the project and the descriptor. With these two things we are able to know where the problem was found and what we actually need to fix in there.

After this, is just the logic specific to this Analysis. We create new PsiElements for the tokens we will put to mark the start and ending of the scope and then insert them in the appropriate place.

In here we also make use of a core functionality, the ScopeFinder. As of now, the only implementation we have for the ScopeFinder is the BasicScopeFinder. But as soon as another one is made it can be easily used in this fix instead of the basic one by modifying the field.

### 5.1.5 Resume

In quick resume, first we need to modify the plugin.xml in order to include our new analysis. Then we create a new Inspection class which will search for the pieces of code we are interested in and provide the panel with the options once it finds something. In conjunction with this, we might have to create an extra class to include a bit of logic for what we are looking for, such as the evaluator. And finally, we need at least one ”fix” class capable of receiving the problem found and apply a fix to it. This is the structure of an Analysis and what we will follow every time we wish to create a new one. In the next chapter we will exemplify the creating of another more complex Analysis with this same structure.

## 5.2 One Word Comment Analysis

We decided to implement a second, more useful Analysis. For this one we took inspiration from an example of a paper [7]. The example they gave there was is represented in excerpt of code next.

```
public void calc(double[][] A) {
    for (int i = 0; i < n; i++) {
     for (int j = 0; j < n; j++) {V[i][j]= A[i][j];}
    }
    tred2(); // Tridiagonalize.
    tq12(); // Diagonalize.
    ...
}
```

What we can as humans identify in this code excerpt is that the names tred2() and tq12() might not be the best names for those two methods. The word ”Tridiagonalize” seems a much suitable name for the method that it is commenting and we would be able to improve the legibility of the code.

Our goal is then to make an Analysis that will search for cases such as this. To better define our goal we will say the following about the target comments:

- Only one word comments.
- The word will have to be a valid English verb.
- The comment will have to be referencing a method call or a method declaration.

Our logic behind this criteria is that if a comment only has one word, there is not much information that it is actually providing. If there is information provided, being only word one, if the method was named after the method it would be cleaner code. Since we are looking for one word comments to replace method names, it makes sense that the word is actually a verb since methods are typically actions.

When we do find such a combination of comment-method we should provide the user with two options. The first one, to just ignore this comment as it is in fact intended to be that way. The second one, we want to fix the situation and change the method name to the more appropriate one, there and everywhere on the code where the method is used.

Having defined what the plugin should be doing, we start following the steps detailed in the section above.

### 5.2.1 plugin.xml

First of all, we include it in the plugin.xml file, filling in correctly all of the fields.

### 5.2.2 Inspection class

Next, we create the main Inspection class, that extends the LocalInspectionTool. In here we create our JPanel to show our options later on and then a visitor for the PsiComment elements so we can check them all.

Inside the visitor for the PsiComment elements, we need to filter which comments we actually want to flag according to our specifications. For this we create an auxiliary class, once again the evaluator class. This class will receive a comment as input and decide whether or not it will be flaged for a possible change.

Also we create the two "fix" options, one for the ignore and another one for the actual fix. There needs to be two of these, one for method declarations and another one for method calls.

### 5.2.3 Extra Logic

The extra logic we needed here was the evaluator class. This class is responsible for checking two things, the comment's contents and also the comment's context.

#### 5.2.3.A Comment's Contents

In here we need to check if the comment's contents are a single word and if that word is a valid English verb. With the PsiComment element we can request its contents. Then we need to clear this result from the "//" or "/*" that are present in comments to get the actual content.

Now that we have the content, we have to check whether or not the comment only has one word, done so by checking for the existence of white spaces. Now that we filtered all the non one-word comments, is this word an actual English word?

For this, we used a dictionary with English words with a total of 370103 words. If the word of was part of that list it would pass to the next stage.

The next step would be to make sure it was a verb. For that, we used a functionality of CoreNLP, the Parts of Speech Tagger. To use this, we simply gave our word to the tagger and it would provide a classification for that word. However, we found a problem here. The PoS function was supposed to be used with full sentences and use the context around a word to classify it but we were only providing a single word. In order to get the best results we did a study to find out what we should provide the PoS: the single word or the word inserted in a sentence and if so, which sentence.

**Table 5.1:** Tag Study

| Tag | % |
|---|---|
| VB (Verb, base form) | 53.5% |
| VBN (Verb, Past Participle) | 18% |
| VBG (Verb, Gerund / Present Participle) | 13% |
| VBD (Verb, Past Tense) | 9% |
| VBZ (Verb, 3rd ps. sing. present) | 5.5% |
| VBP (Verb, non-3rd ps. sing. present) | 1% |

#### 5.2.3.B The Sentence Study

Here we should explain the classification that the PoS used for words. It is based on the Penn Treebank POS tagset. From this set, 6 tags are referred to verbs in different tenses. Some of them are of interest for us and some are not. To define which ones we are interested in to classify a comment as a verb for our analysis we made a simple study first.

To make this study we used the Muse data set (with millions of entries of real comment/code/source triplets from C, C++, C#, Java and Python. From this set we used a Python script that would extract all the one word comments from Java files.

This resulted in a file with 333931 entries. Many of these were not valid English words. So we had to filter those next. For that, I used the same dictionary I had before and a simple Java program removed all the non English words from the file, resulting in a new file with 24600 entries. Many of these words were duplicated, so next we cleared the file by eliminating repeating words. This resulted in a new file with 2185 entries.

So now we had 2185 different words that were actual comments in real code. From here we could try to classify all the words and check which of the six verb tags were the most relevant.

Because we had not yet decided on the best phrase to provide the POS tagger, we used a combination of proposed phrases to classify each of these words. For this we used four different phrases:

- just the word.
- ”Do not ____”.
- ”____ that”.
- ”I will ____ this”.

The average of the results can be seen in table 5.1.

From these results, we can see that the most common would be VB and it does make sense in the context of our problem. The word ”process” is a verb in its base form and could easily be the name of a method.

**Table 5.2:** Phrase Study

| Phrase | True Positives | False Negatives | % of True Positives | False Positives | True Negatives | % of False Positives |
|---|---|---|---|---|---|---|
| ____ | 1385 | 1508 | 47.87% | 3242 | 51284 | 5.95% |
| I will ____ this. | 2506 | 387 | 86.62% | 25532 | 28994 | 46.83% |
| Do ____ that. | 1274 | 1619 | 44.04% | 3139 | 51387 | 5.76% |
| ____ that. | 1539 | 1354 | 53.20% | 3573 | 50953 | 6.55% |
| I am thinking about ____ . | 551 | 2342 | 19.05% | 2399 | 52127 | 4.40% |
| I am about to ____ . | 2190 | 703 | 75.70% | 5130 | 49396 | 9.41% |
| I would like to ____ . | 2072 | 821 | 71.62% | 4310 | 50216 | 7.90% |
| I ____ | 3 | 2890 | 0.10% | 2093 | 52433 | 3.84% |

From the next two, VBN and VBG, only VBG makes sense in our context, as a method name would not make much sense in the past tense. However, Gerund / Present Participle do make sense.

So VB and VBG would be the most appropriate tags to search and words that the POS tagger deemed to have one of those two classifications would be considered a verb for us.

So now, what is left to decide is which phrase do we insert our one word into in order to get the best results from the POS tagger, if any?

For this, we created two new data sets. These data sets were created with a Python script using the NLTK library. From the brown corpus existent in this library we separated the corpus into two different data sets: one with all the words tagged with VB and VBG and the other with everything else.

With these two corpus we can feed each word inserted in a phrase to the CoreNLP and check which phrase yields the best results. So another Java class was made in order to test this in such a way that new phrases that we can think of can easily be added and tested as well. The results of this test can be seen in table 5.2. True positives are the cases where a VB or VBG is identified as such, false negatives when it is not. False positives are when a non VB or VBG is identified as such and true negatives when it does not. With this in consideration, we want to maximize our True Positive percentage while minimizing our False Positive percentage. From the candidates tested, the one with the best results is ”I am about to ____.” And so, this was the phrase chosen for our analysis.

And now, our content evaluation can be made. Just insert the word in our phrase, feed it to the POS tagger and it will return its classification. If it returns something different form VB or VBG, we can abort the rest of the evaluation as the comment is no longer a viable candidate. If it returns VB or VBG, we advance to the next phase.

#### 5.2.3.C Comment's Context

For the context we needed the comment to be referencing a method call or declaration. For this we used the core function, ScopeFinder. Knowing what the scope of the comment was and what PSIElement the comment was referring to made it possible to check whether or not it was a method call or declaration.

#### 5.2.3.D Ignore option and Non Original Methods

We considered an Ignore option as one of the actions the user could take. But as the system is now, even if the user ignore a pair of code/comment, it would be flagged again the next time the analysis visited it.

Another problem that we have is that as of now, all the method calls are considered, even System ones like System.out.println().

Both of these aspects were solved in the same way. The Analysis would visit not only PsiComment elements, but also PsiMethod elements. In these visits, two lists of methods were updated. First, a list of the user declared methods. Second, a list of the methods the user chose to ignore. This way, the first list could be checked to see if a method should be flagged or not. The second would stop user declared methods from being considered if the user had chosen to ignore them. If we simply removed ignored methods from the first list, they would end up there again the next time the Analysis visited them, so the second list was necessary.

### 5.2.4 The Possible ”Fix” Classes

We had as an objective two ”fix” classes, one to actually change the identifier for the comment and the other one to ignore that case. But with the difference between a method call and a method declaration, we actually needed 4 ”fix” classes, two for each.

#### 5.2.4.A The Ignore Fix

The Ignore ”Fixes” were the most simple. The only thing they did was adding the method being considered to the blacklisted methods and then return, doing nothing else.

#### 5.2.4.B The Actual Fix

These were a little more complex. We ca not only change the identifier of the method we found in that specific part of the code. We need to search for every usage of the method and change them, as well as its declaration.

Each case (declaration or call) had us begin in a different part of the code, but the process was the same. Produce a new ID with the contents of the comment, use the PSI to reach all the times the method was referenced, change its ID in every one and then change the ID on its declaration as well.

And with this, our new Analysis was implemented, capable of finding possible bad method names and suggesting a better one based on comments.

# 6

# Conclusion

**Contents**

Work had already been done in the field of comment understanding, both in the comments meaning as well as in what it refers to. However, the knowledge obtained from this work was not yet put to practical use. This was our objective for this project.

We wanted to create an extensible IDE plugin that provides an easy way for the programmer to create its own analysis to code-comment relations.

The final result was an IntelliJ plugin for Java. As part of this plugin we have some core features such as: the framework and one implementation for the scope finder (to return what part of the code a specific comment is referring), a way to do some natural language analysis with CoreNLP and two functional analysis. From these two analysis, one is simpler with the objective to serve as a model for the programmer to create new ones, highlighting the various aspects of the plugin. The other one is the implementation of a suggestion for this type of analysis made in a related work that tries to improve code understanding by searching for possible poorly named methods and suggesting a correction based on the comment that accompanies it.

In addition to our concrete implementation, we also set a number of requirements that such type of tool must follow. By adhering to these, it is possible to create the same functional tool for other languages and IDE's besides Java with IntelliJ.

To utilize our tool it is possible to just utilize the analysis already implemented. It is also possible to create new analysis as needed. To help create new ones we provided a step by step guide on the important components that make an analysis.

## 6.1 Future work

We identify two main areas worthy of future investment.

### 6.1.1 Scope Finder

Our current implementation of the scope finder is a simple one. It can be improved with more comment scope knowledge, which is already being worked on as we saw in the related work section. A better scope finder means better results form the plugin as it will be better at finding correct comment code pairs.

### 6.1.2 New Analyses

The ground work is all done, but the plugin now needs more functionality, and that means more Analyses. Imagination and need are the limit here. For example, an analysis that uses comment information to determine the range of a variable and then checked the following code to see if this range is being respected. Or an example from one of the papers mentioned in the related code, to find how where in the code there is a need for a comment to help the user not forget and make comments in appropriate places. It could something that helps the user avoid certain types of not so useful comments, or that tracks nonsensical comments such as commented code for the user to delete.

Bottom line is that the plugin exists as a tool for the programmer to make personalized use based on their needs.

# Bibliography

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*. ACM, 2005, pp. 68–75.

[2] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.

[3] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 215–223.

[4] C. S. Hartzman and C. F. Austin, "Maintenance productivity: Observations based on an experience in a large system environment," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*. IBM Press, 1993, pp. 138–170.

[5] B. P. Lientz, "Issues in software maintenance," CALIFORNIA UNIV LOS ANGELES GRADUATE SCHOOL OF MANAGEMENT, Tech. Rep., 1983.

[6] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*. Ieee, 2013, pp. 83–92.

[7] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 227–237.

[8] D. Haouari, H. Sahraoui, and P. Langlais, "How good is your comment? a study of comments in java programs," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2011, pp. 137–146.

[9] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers taxonomies and characteristics of comments in operating system code," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 331–341.

[10] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 53–64.

[11] J. Zhai, X. Xu, Y. Shi, M. Pan, S. Ma, L. Xu, W. Zhang, L. Tan, and X. Zhang, "Cpc: Automatically classifying and propagating natural language comments via program analysis," 2019.

[12] H. Chen, Y. Huang, Z. Liu, X. Chen, F. Zhou, and X. Luo, "Automatically detecting the scopes of source code comments," *Journal of Systems and Software*, vol. 153, pp. 45–63, 2019.

[13] A. Louis, S. K. Dash, E. T. Barr, M. D. Ernst, and C. Sutton, "Where should i comment my code? a dataset and model for predicting locations that need comments," in *Proceedings of the 42nd International Conference on Software Engineering (New Ideas and Emerging Results)(ICSE NIER 2020)*. Association for Computing Machinery (ACM), 2020.

[14] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.