IntelliComment

An IDE Plugin to Improve Java Source Code Using Comments

Francisco Machado Duarte
francisco.duarte1995@gmail.com

Instituto Superior Técnico

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.

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

1 Introduction

1.1 Motivation

Comments are the second most used artifact for code understanding, behind only the code itself [de Souza et al., 2005]. They facilitate code comprehension [Tenny, 1988,Woodfield et al., 1981] and the lack of comments leads to misunderstandings and low maintainability of software [Hartzman and Austin, 1993,Lientz, 1983]. 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

As background for this work, the main focus are studies made around code comments. As examples, we have quality analysis of a source code comment [Steidl et al., 2013], classification of comments in Java OpenSource Software [Pascarella and Bacchelli, 2017] or classification with multiple categories instead of one [Haouari et al., 2011,**?**]. All of these works are important to understand comments, their role in the comment and possible uses in the future.

As for Related Work, multiple recent papers have been made recently exploring these areas, such as:

- Large-scale empirical study made on code-comment inconsistencies[Wen et al., 2019].
- How to generate and propagate code automatically [Zhai et al., 2019].
- Automatically detect the scope of source code comments [Chen et al., 2019].
- A machine learning model capable of suggesting comment locations[Louis et al., 2020].

These works give support to the claim that our work is relevant in today's programming world and the next step into transitioning these concepts into application.

3 Functional Design

There is a set of requirements that must be met in order to create a plugin that extracts and uses comment information. We should be able to implement the plugin in any OOP language and any IDE that supports the requirements.

3.1 Requirements

3.1.1 Process Language Elements The plugin needs to be able to access and process language elements, like comments or method declarations or cycles.

3.1.2 Process Text from Comments Some kind of Natural Language processing tool needs to be used here.

3.1.3 Find Comment Scope In order to compare comment with code, we need to be able to associate the comment to the specific code. The lose structure of comments with the fact that they are written in natural language makes this task very difficult to accomplish.

3.1.4 Create New Analysis The amount of possibilities for analyses is enormous. The thing about the plugin is that it should be able to be easily extendable with new personalized analysis that a user might need.

3.1.5 Modify Language Elements In addition to accessing language elements, the plugin need to be able to actually create, delete and modify elements, so that when an incoherence is detected the plugin can fix it.

3.1.6 User Communication Communication with the user is key. The user needs to be in control if changes will happen or not and the plugin should be able to communicate all the possibilities.

3.2 Architecture

Considering all these aspects we can now create an architecture for our Plugin. The design we achieved was as represented on 1.

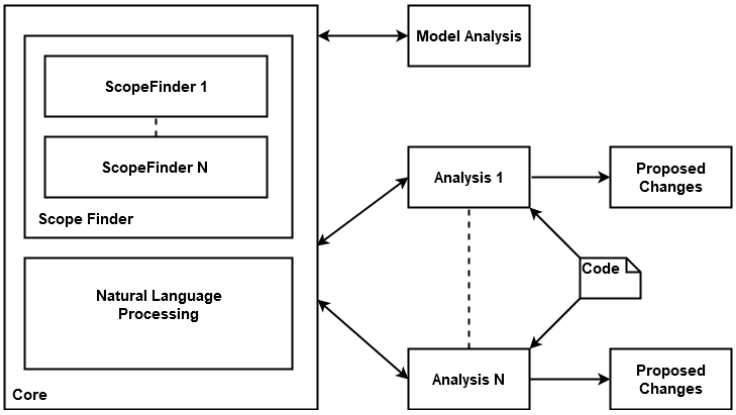

Fig. 1. Plugin Architecture

4 Technical Design

The language chosen for this implementation was Java and the IDE IntelliJ. However, any other combination could be used as long as it met the requirements.

4.1 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.

4.2 Implementation of the Requirements

- Process Language Elements - Psi and Visitors
- Process Text from Comments - CoreNLP
- Find Comment Scope - ScopeFinder
- Ease of New Analysis Creation - New Analysis Procedure
- Modify Language Elements - "Fix" Classes
- User Communication - JPanel and Multiple "Fix" Classes

5 Evaluation

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 will follow the creation of a new Analysis. First, we will consider this example:

```
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.
    ...
}
```

This example was taken from a real project [Pascarella and Bacchelli, 2017] and we can see what could be changed: "tridiagonalize" is a better, more legible name for tred2().

So our objective is an Analysis that searches and fixes this kind of problems.

5.1 plugin.xml

First, we need to edit the plugin.xml in order to add the new analysis.

5.2 Inspection class

Then comes the creation of an Inspection class, that extends LocalInspectionTool and has a very rigid form. Part of this class will be the JPanel and visitors for the elements we are interested in.

5.3 Extra Logic

Here comes the logic behind this specific Inspection. It is an auxiliary class that will process each comment and check whether or not it is a one English verb comment and that what it references is a method call or declaration.

5.4 The Possible "Fix" Classes

In this cases, the plugin provides two options for the user when a code comment pair is flagged. Either it is ignore, or the switch from the original id of the method to the comment happens.

5.5 The Fix

If the user does chose to make the switch, the plugin will search all the references to the considered method and change them all plus the declaration to the new identifier. The comment is then deleted.

5.6 Conclusion

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.

5.7 Future work

We identify two main areas worthy of future investment.

5.7.1 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.

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.

References

[Chen et al., 2019] Chen, H., Huang, Y., Liu, Z., Chen, X., Zhou, F., and Luo, X. (2019). Automatically detecting the scopes of source code comments. *Journal of Systems and Software*, 153:45–63.

[de Souza et al., 2005] de Souza, S. C. B., Anquetil, N., and de Oliveira, K. M. (2005). 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*, pages 68–75. ACM.

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

[Hartzman and Austin, 1993] Hartzman, C. S. and Austin, C. F. (1993). 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*, pages 138–170. IBM Press.

[Lientz, 1983] Lientz, B. P. (1983). Issues in software maintenance. Technical report, CALIFORNIA UNIV LOS ANGELES GRADUATE SCHOOL OF MANAGEMENT.

[Louis et al., 2020] Louis, A., Dash, S. K., Barr, E. T., Ernst, M. D., and Sutton, C. (2020). 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).

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

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

[Tenny, 1988] Tenny, T. (1988). Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279.

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

[Woodfield et al., 1981] Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. (1981). The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering*, pages 215–223. IEEE Press.

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