Using Refactoring Techniques to Reduce Duplication in Object-Oriented Programming Languages

Simon Baars
University of Amsterdam
Puiflijk, Netherlands
simon.mailadres@gmail.com

Ana Oprescu
University of Amsterdam
Amsterdam, Netherlands
A.M.Oprescu@uva.nl

Abstract

Duplication in source code can have a major negative impact on the maintainability of source code. There are several techniques that can be used in order to merge clones, reduce duplication and potentially also reduce the total volume of a software system. In this study, we look into the opportunities to aid in the process of refactoring these duplication problems for object-oriented programming languages. Measurements that have been conducted so far have indicated that more than half of the duplication in code is related to each other through inheritance, making it easier to refactor these clones in a clean way. More measurements will be conducted to get a detailed overview of in what contexts clones occur, and what this means for the refactoring processes of these clones. As a desired output, we strive to construct a model that automatically applies refactorings for a large part of the detected duplication problems and implement this model for the Java programming language.

1 Introduction

Refactoring is used to improve quality related attributes of a codebase (maintainability, performance, etc.) without changing the functionality. There are many methods that have been introduced to help with the process of refactoring [Fow18, Wak04]. However, most of these methods still require manual assessment of where and when to apply them. Because of this, refactoring takes up a signification portion of the development process [LST78, MT04], or does not happen at all [MVD$^+$03]. For a large part, refactoring requires domain knowledge to do it right. However, there are also refactoring opportunities that are rather trivial and repetitive to execute. In this thesis, we take a look at the challenges and opportunities in automatically refactoring duplicated code, also known as "code clones". The main goal is to improve maintainability of the refactored code.

Duplication in source code is often seen as one of the most harmful types of technical debt. In Martin Fowler's "Refactoring" book [Fow18], he exclaims that *"Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them."*. However, this statement is not accepted by everyone. Several papers argue that not each type of duplication is harmful [KG06].

In this research, we focus on formalizing the refactoring process of dealing with duplication in code. We will measure open source projects from the We will show the improvement of the metrics over various open source and industrial projects. Likewise, we will perform an estimation of the development costs that are saved by using the proposed solution. We will lay the main focus on the Java programming language as refactoring opportunities do feature paradigm and programming language dependent aspects [CYI$^+$11]. However, most practises used in this thesis will also be applicable with other object-oriented languages, like C#.

2 Clone Detection

As duplication in source code is a serious problem in many software systems, there are a lot of researches

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

that look into code clones. Many tools have been proposed to detect various types of code clones. Two surveys of modern clone detection tools [SK16, SR14] together show an overview of the most-popular clone detection tools up until 2016. To be able to refactor detected clones, it is useful to have the ability to rewrite the AST. We considered a set of clone detection tools for their ability to support the refactoring process automatically. None of the tools we consider seemed completely fit for this purpose, so we decided to implement our own clone detection tool: CloneRefactor[1].

2.1 CloneRefactor

A 2016 survey by Gautam [GS16] focuses more on various techniques for clone detection. For our tool we decided to combine AST and Graph based approaches for clone detection, similar to Scorpio (which is a clone detection tool that's part of TinyPDG: a library for building intraprocedural program dependency graphs for Java programs). We decided to base our tool on the JavaParser library [SvBT18], as it supports rewriting the AST back to Java code. We then collect each statement and declaration (ommitting their child statements and declarations) and compare those to find duplicates. This way we build a graph of each statement/declaration linking to each subsequent statement/declaration (horizontally) and linking to each of it's duplicates (verically). This is displayed in figure 1.

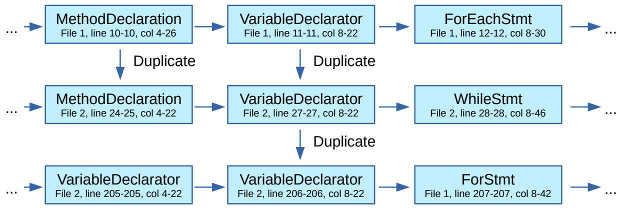
MethodDeclaration
File 1, line 10-10, col 4-26
VariableDeclarator
File 1, line 11-11, col 8-22
ForEachStmt
File 1, line 12-12, col 8-30
Duplicate
Duplicate
MethodDeclaration
File 2, line 24-25, col 4-22
VariableDeclarator
File 2, line 27-27, col 8-22
WhileStmt
File 2, line 28-28, col 8-46
Duplicate
VariableDeclarator
File 2, line 205-205, col 4-22
VariableDeclarator
File 2, line 206-206, col 8-22
ForStmt
File 1, line 207-207, col 8-42
...

Figure 1: Abstract figure of the graph representation built by CloneRefactor

3 Code Clone Context

To be able to refactor code clones, it is very important to consider the context of the clone. We define the following aspects of the clone as its context:

1. The relation of clone instances among each other (for example: two clone instances in a clone class are part of the same object).

2. Where a clone instance occurs in the code (for instance: a method-level clone is a clone instance that is (a part of) a single method).

3. The contents of a clone instance (for instance: the clone instance consists of a one method declaration, a foreach statement and two variable declarations).

Everything in the context of a clone has a big impact on how it has to be refactored. For this study we performed measurements on the context of clones in a large corpus of open source projects.

3.1 The corpus

For our measurements we use a large corpus of open source projects [AS13][2]. This corpus has been assembled to contain relatively higher quality projects (by filtering by forks). Also, any duplicate projects were removed from this corpus. This results in a variaty of Java projects that reflect the quality of average open source Java systems and are useful to perform measurements on.

We then filtered the corpus further to make sure we are not including any test classes or generated classes. Many Java/Maven projects use a structure where they separate the application and it's tests in the different folders ("/src/main/java" and "/src/test/java" respectively). Because of this, we chose to only use projects from the corpus which use this structure (and had at least a "/src/main/java" folder). To limit the execution time of the script, we also decided to limit the maximum amount of source files in a single project to 1.000 (projects with more source files were not considered, which filtered only 5 extra projects out of the corpus). Of the 14.436 projects in the corpus over 3.848 remained, which is plenty for our purposes. The script to filter the corpus in included in our GitHub repository [3].

3.2 Relations Between Clone Instances

When merging code clones in object-oriented languages, it is very important to consider the relation between clone instances.

3.2.1 Categorizing Clone Instance Relations

A paper by Fontana et al [FZZ15] performs measurements on 50 open source projects on the relation of clones to each other. To do this, they first define several categories for the relation between clone instances in object-oriented languages. These categories are as follows:

[1] CloneRefactor (WIP) is available on GitHub: `https://github.com/SimonBaars/CloneRefactor`

[2] The corpus can be downloaded from the following URL: `http://groups.inf.ed.ac.uk/cup/javaGithub/java_projects.tar.gz`

[3] The script we use to filter the corpus: `https://github.com/SimonBaars/CloneRefactor/blob/MeasurementsVersion1/src/main/java/com/simonbaars/clonerefactor/scripts/PrepareProjectsFolder.java`

1. **Same method**: All instances of the clone class are in the same method.

2. **Same class**: All instances of the clone class are in the same class.

3. **Superclass**: All instances of the clone class are children and parents of each other.

4. **Ancestor class**: All instances of the clone class are superclasses except for the direct superclass.

5. **Sibling class**: All instances of the clone class have the same parent class.

6. **First cousin class**: All instances of the clone class have the same grandparent class.

7. **Common hierarchy class**: All instances of the clone class belong to the same hierarchy, but do not belong to any of the other categories.

8. **Same external superclass**: All instances of the clone class have the same superclass, but this superclass is not included in the project but part of a library.

9. **Unrelated class**: There is at least one instance in the clone class that is not in the same hierarchy.

Please note that no of these categories allow external classes (except for "same external superclass"). So if two clone instances are related through external classes but do not share a common external superclass, it will be flagged as "unrelated". The main reason for this is that it is (often) not possible to refactor to external classes.

The ranking of the previous list of categories also matters, as it shows the different levels in which clones were assessed. For instance, if two clone instances of a clone class belong to the "same method" category but the third belongs to the "same class", we will always chose the item lowest on the list.

3.2.2 Our measurements

We have recreated the same table as Fontana et al [FZZ15], but with the following differences:

- We consider clone classes rather than clone pairs

- We use different thresholds regarding when a clone should be considered.

- We seek by statement/declaration rather than SLOC.

- We test a broader range of projects (they a set of 50 relatively large projects, we use a large corpus that was assembled by a machine learning algorithm testing java projects on GitHub for quality, which contains projects of all sizes and with differing code quality)

As a result, we got the following table:

3.3 Clone instance location

After mapping the relations between individual clones, we looked at the location of individual clone instances. A paper by Lozano et al [LWN07] discusses the harmfulness of cloning. In this paper the author argues that 98% are produced at method-level. However, this claim is based on a very small dataset and a questionable way of measurement. We validated this claim over our corpus. For this, we chose the following categories:

1. **Method Level:** A clone instance that does not exceed the boundaries of a single method (optionally including the declaration of the method itself).

2. **Class Level:** A clone instance in a class, that exceeds the boundaries of a single method or contains something else in the class.

3. **Interface Level:** A clone that is (a part of) an interface.

4. **Enumeration Level:** A clone that is (a part of) an enumeration.

3.4 Clone instance contents

Finally, we looked at the contents of individual clone instances: what kind of

3.4.1 Tables

All tables must be centered, neat, clean and legible. Do not use pencil or hand-drawn tables. Table number and title always appear before the table.

One line space before the table title, one line space after the table title and one line space after the table. The table title must be initial caps and each table numbered consecutively.

Table 1: Sample Table

A	B	1
C	D	2
E	F	3

3.4.2 Handling References

Use a first level heading for the references. References follow the acknowledgements.

3.4.3 Acknowledgements

We would like to thank the Software Improvement Group (SIG) for their continuous support in this project.

References

[AS13] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[CYI$^+$11] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, pages 7–13. ACM, 2011.

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[FZZ15] Francesca Arcelli Fontana, Marco Zanoni, and Francesco Zanoni. A duplicated code refactoring advisor. In *International Conference on Agile Software Development*, pages 3–14. Springer, 2015.

[GS16] Pratiksha Gautam and Hemraj Saini. Various code clone detection techniques and tools: A comprehensive survey. pages 655–667, 08 2016.

[KG06] Cory Kapser and Michael W Godfrey. “cloning considered harmful” considered harmful. In *Reverse Engineering, 2006. WCRE’06. 13th Working Conference on*, pages 19–28. Citeseer, 2006.

[LST78] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[LWN07] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007)*, pages 18–18. IEEE, 2007.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[MVD$^+$03] Tom Mens, Arie Van Deursen, et al. Refactoring: Emerging trends and open problems. In *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[SK16] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[SR14] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330. IEEE, 2014.

[SvBT18] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser, 05 2018.

[Wak04] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.