Towards Automated Merging of Code Clones in Object-Oriented Programming Languages

Simon Baars
simon.mailadres@gmail.com

30 June 2019, 14 pages

Research supervisor: Dr. Ana Oprescu, ana.oprescu@uva.nl
Host/Daily supervisor: Xander Schrijen, x.schrijen@sig.eu
Host organisation/Research group: Software Improvement Group (SIG), http://sig.eu/

University of Amsterdam
Faculty of Physics, Mathematics and Informatics
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

This should be done when most of the rest of the document is finished. Be concise, introduce context, problem, known approaches, your solution, your findings.

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, improve the design of the code 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.

We first look into redefinitions for different types of clones that have been used in code duplication research for many years. These redefinitions are aimed towards flagging only clones that are useful for refactoring purposes. Our definition defines additional rules for type 1 clones to make sure two cloned fragments are actually equal. We also redefined type 2 clones to reduce the number of false positives resulting from it.

We have conducted measurements that 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. Approximately a fifth of the duplication can be refactored through method extraction, the other clones require other techniques to be applied.

Contents

1 Introduction **3**
- 1.1 Problem statement . . . 3
  - 1.1.1 Research questions . . . 3
  - 1.1.2 Research method . . . 4
- 1.2 Contributions . . . 4
- 1.3 Scope . . . 4
- 1.4 Outline . . . 4

2 Background **5**
- 2.1 Clone Class . . . 5
- 2.2 Clone Types . . . 6
- 2.3 Clone Contexts . . . 6
  - 2.3.1 Clone refactoring in relationship to its context . . . 7
- 2.4 Code clone harmfulness . . . 7
- 2.5 Related work . . . 7

3 Prioritizing code clones for refactoring **8**
- 3.1 The corpus . . . 8

4 Results **9**

5 Discussion **10**

6 Related work **11**

7 Conclusion **12**
- 7.1 Future work . . . 12

Appendix A Non-crucial information **14**

Chapter 1

Introduction

Context: what is the bigger scope of the problem you are trying to solve? Try to connect to societal/economical challenges. Problem Analysis: Here you present your analysis of the problem situation that your research will address. How does this problem manifest itself at your host organisation? Also summarises existing scientific insight into the problem.

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 [**fowler2018refactoring**, **wake2004refactoring**]. 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 [**lientz1978characteristics**, **mens2004survey**], or does not happen at all [**mens2003refactoring**]. 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 [**fowler2018refactoring**], 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."*.

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 [**choi2011extracting**]. However, most practises used in this thesis will also be applicable with other object-oriented languages, like C#.

1.1 Problem statement

1.1.1 Research questions

Code clones can appear anywhere in the code. Whether a code clone has to be refactored, and how it has to be refactored, is dependent on where it exists in the code (it's context). There are many different contexts in which code clones can occur (in a method, a complete class, in an enumeration, global variables, etc.). Because of this, we first must collect some information regarding in what contexts code clones exist. To do this, we will analyze a set of Java projects for their clones, and generalize their contexts. To come to this information, we have formulated the following research question:

Research Question 1:
How can we group and rank clones based on their harmfulness?

As a result from this research question, we expect a catalog of the different contexts in which clones occur, ordered on the amount of times they occur. On basis of this catalog, we have prioritized the further analysis of the clones. This analysis is to determine a suitable refactoring for the clone type that has been found at the design level. For this, we have formulated the following research question:

Research Question 2:
To what extend can we suggest refactorings of clones at the design level?

As a result, we expect to have proposed refactorings for the most harmful clone patterns. On basis of these design level refactorings we will build a model, which we will proof using Java, that applies the refactorings to corresponding methods. For building this model, we have formulated the following research question:

Research Question 3:
To what extend can we automatically refactor clones?

As a result from this research question, we expect to have a model to be able to refactor the highest priority clones.

1.1.2 Research method

1.2 Contributions

Our research makes the following contributions:

1. We deliver several novel measurements regarding code clones on a large corpus of Java projects.
2. We deliver a novel clone detection tool that finds refactorable clones in Java.
3. We deliver a novel clone refactoring tool that suggests refactorings to be applied, and applies these refactorings.
4. We give further recommendations in how refactoring can be automatically applied to improve maintainability in software projects.

1.3 Scope

In this research we will look into code clones from a refactoring viewpoint. There are several methods that detect code clones using a similarity score to match pieces of code. This similarity is often based on the amount of tokens that match between two pieces of code. The problem with similarity based clones is that it is hard to assess the impact of merging clones that have different tokens, but what exactly this token is is unknown. Because of this, we will not focus on similarity based clone detection techniques, but rather on exact matches and predefined differences.

It is very disputable whether unit tests apply to the same maintainability metrics that applies to the functional code. Because of that, for this research, unit tests are not taken into scope. The findings of this research may be applicable to those classes, but we will not argue the validity.

1.4 Outline

In Chapter 2 we describe the background of this thesis. Chapter **??** describes ... Results are shown in Chapter 4 and discussed in Chapter 5. Chapter 6, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 7 together with future work.

Chapter 2

Background

This chapter will present the necessary background information for this thesis. Here, we define some basic terminology that will be used throughout this thesis.

2.1 Clone Class

As code clones are seen as one of the most harmful types of technical debt, they have been studied very extensively. A survey by Roy et al [**roy2007survey**] states definitions for various important concepts in code clone research. In this survey, he mentions the concept "clone pair", which is *a set of two code portions/fragments which are identical or similar to each other*. Furthermore, he defined "clone class" as *the union of all clone pairs*. Apart from this, we use the definition "clone instance", which is a single code portion/fragment that is part of either a clone pair or clone class.

Figure 2.1 displays an example of a clone pair or clone class. In this case, both cloned fragments, are found in the same class. Each of the cloned fragments can be defined as a "clone instance".

Figure 2.1: Example of a clone pair, as found in the ardublock project.

Figure 2.2 displays two clone classes. These clone classes are separated by a single line that is different. The first clone class spans over both the constructor and a method of this class.

Figure 2.2: Example of two clone classes.

2.2 Clone Types

Duplication in code is found in many different forms. Most often duplicated code is the result of a programmer reusing previously written code [**haefliger2008code**, **baxter1998clone**]. Sometimes this code is then adapted to fit the new context. To reason about these modifications, several clone types have been proposed. These clone types are described in Roy et al [**roy2007survey**]:

> **Type I:** Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.
> **Type II:** Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
> **Type III:** Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.
> **Type IV:** Two or more code fragments that perform the same computation but implemented through different syntactic variants.

A higher type of clone means that it's harder to detect and refactor. There are many studies that adopt these clone types, analyzing them further and writing detection techniques for them [**sajnani2016sourcerercc**, **kodhai2010detection**, **van2019novel**].

Relating this to the clone class example of 2.2, the complete class would be flagged as a type 2 clone. This is displayed in figure 2.3. Considering such clones displays the opportunity to refactor the class as a whole.

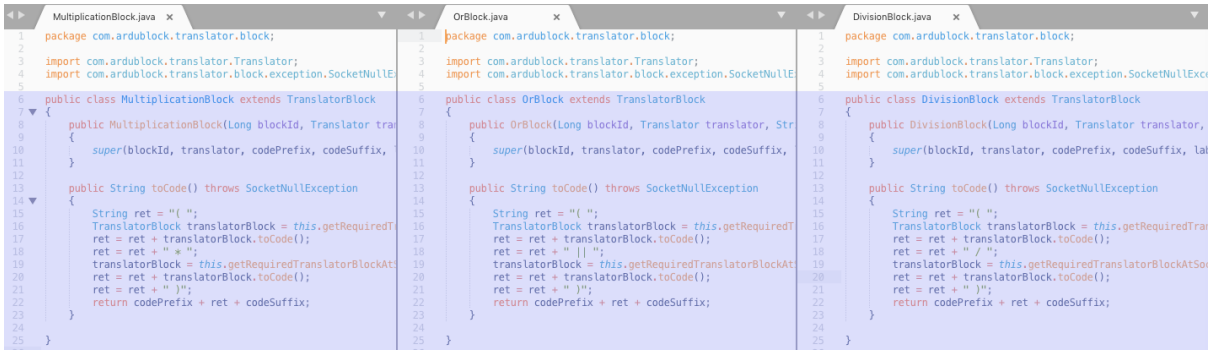

Figure 2.3: The clone displayed in figure 2.2 as a type 2 clone.

For this thesis we have chosen not to consider type 4 clones for refactoring, because they are both hard to detect and hard to refactor (how to choose the best alternative for a certain computation could be a thesis in itself).

2.3 Clone Contexts

Code clones can be found anywhere in the code. The most commonly studied type of clone is the method-level clone. Method-level clones are duplicated blocks of code in the body of a method. Many

clone detection tools only focus on method-level clones (like CPD[1], Siamese[2], Sysiphus[3]). The reason for this is that with method-level clones it's most likely that the clones are harmful, and they are more straight-forward to refactor.

A paper by Lozano et al [**lozano2007evaluating**] discusses the harmfulness of cloning. In this paper the author argues that 98% are produced at method-level. However, the paper that is cited to support this claim [**bergman2004ethnographic**] does not conclude this same information. First of all, the study that is referenced uses a very small dataset (460 copy & paste instances by 11 participants). Secondly, the group of subject only consists of IBM researchers (selection bias). Thirdly, it only focuses on copy and paste instances, as opposed to other ways clones can creep into the code. Finally, the "98%" is not stated explicitly, but is vaguely derivable from one of the figures (figure 1) in this paper. Because of this, there is no reliable overview of how many clones there are in different contexts.

This thesis will focus on measuring how many clones there are per context. This way we can determine the impact of focusing our search on a specific context, like the analysis of only method-level clones. Our hypothesis is that the 98% claim is not true (we think this should be far less). We also hypothesize that clones in different contexts than method-level are less likely to be harmful and less straight forward to refactor.

2.3.1 Clone refactoring in relationship to its context

How to refactor clones is highly dependent on their context. Method-level clones can be extracted to a method [**kodhai2013method**] if all occurrences of the clone reside in the same class. If a method level clone is duplicated among classes in the same inheritance structure, we might need to pull-up a method in the inheritance structure. If instances of a method level clone are not in the same inheritance structure, we might need to either make a static method or create an inheritance structure ourselves. So not only a single instance of a clone has a context, but also the relationship between individual instances in a clone class. This is highly relevant to the way in which the clone has to be refactored.

2.4 Code clone harmfulness

There has been a lot of discussion whether code clones should be considered harmful.

Most papers view clones as harmful regarding program maintainability. *"Clones are problematic for the maintainability of a program, because if the clone is altered at one location to correct an erroneous behaviour, you cannot be sure that this correction is applied to all the cloned code as well. Additionally, the code base size increases unnecessarily and so increases the amount of code to be handled when conducting maintenance work."* [**ostberg2014automatically**]

However, the harmfulness of clones depends on a lot of factors. A paper by Kapser et al [**kapser2006cloning**] describes several patterns of cloning that may not be considered harmful. In this paper Kapser names examples where eliminating clones would compromise other important program qualities. Another study by Jarzabek et al [**jarzabek2010clones**] categorized "Essential clones": clones that are essential because of the solution that is being modelled by the program. Overall, many of the benefits of code clones do not apply to most modern object-oriented programming languages.

2.5 Related work

There have been some papers that take some steps towards code clone refactoring. Most research towards refactoring code clones has been conducted by Y. Higo et al. In a 2008 study [**higo2008metric**] the authors look at the refactoring of class-level, method-level and constructor-level clones in Java.

[1] CPD is part of PMD, a commonly used source code analyzer: `https://github.com/pmd/pmd`

[2] Siamese is an Elasticsearch based clone detector: `https://github.com/UCL-CREST/Siamese`

[3] Sisyphus crawls the Java library for existing implementations of parts of a codebase: `https://github.com/fruffy/Sisyphus`

Chapter 3

Prioritizing code clones for refactoring

Where a clone instance is located in the code, and how clones in a clone class are related, has a big impact on how this clone should be refactored. Because of this, we have performed measurements on a big corpus of open source projects.

3.1 The corpus

For our measurements we use a large corpus of open source projects [**githubCorpus2013**][1]. 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). Of the 14.436 projects in the corpus over 3.853 remained, which is plenty for our purposes. The script to filter the corpuses in included in our GitHub repository [2].

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

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

Chapter 4

Results

In this chapter, we present the results of our experiment.

Chapter 5

Discussion

In this chapter, we discuss the results of our experiment(s) on ...

Finding 1: Highlight like this an important finding of your analysis of the results.

Refer to Finding 1.

Chapter 6

Related work

We divide the related work into ... categories: ...

Chapter 7

Conclusion

7.1 Future work

Acknowledgements

Thanks to you, for reading this :)

Appendix A

Non-crucial information