University of Amsterdam

Faculty of Science

Automatic Refactoring of Code Clones in Object-Oriented Programming Languages

Improving a systems maintainability with the push of a button

Master Thesis Software Engineering

Author:
Simon Baars

Student Number:
12072931

Academic Supervisor:
dr. Ana Oprescu

Company Supervisor:
Xander Schrijen, MSc.

Company:
Software Improvement Group
Schrijen, MSc.

April 10, 2019

Contents

Chapter 1

Abstract

This should be done when most of the rest of the document is finished.

Chapter 2

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 [**?**, **?**]. 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 [**?**, **?**], or does not happen at all [**?**]. Refactoring mostly requires some domain knowledge to do it right, but 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.

There are several models which describe ways to measure maintainability. None of these are sufficient to make a full assessment of the maintainability of a software project, but they strive to give a good indication. For this thesis, we will make use of the *SIG maintainability model* [**?**], as it is based on a lot of experience in the field of software quality assessment. This maintainability model is independent of programming language. For this thesis we will lay the main focus on the Java programming language as refactoring opportunities do feature paradigm and programming language dependent aspects [**?**]. However, most practises used in this thesis will also be applicable with other object oriented languages, like C#.

Improving the maintainability metrics does not automatically lead to a better maintainable codebase [**?**]. For instance, in general, a bigger codebase (in volume) is harder to maintain. However, refactoring a big method into smaller methods can definitely improve the maintainability of the codebase (but still increase the volume metric). Because of this, it is important that refactorings focus on the resolution of harmful anti-patterns [**?**] rather than just the improvement of the metrics. This will be one of our main focus points in this thesis.

For this research, we will focus on formalizing the refactoring process of dealing with duplication in code. To validate this approach, we will validate the refactored results with domain experts. Apart from that, 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.

2.1 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.

2.2 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.

Chapter 3

Background

3.1 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 [**?**] 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 [**?**] 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 og 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 focussing 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.

3.2 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 [**?**, **?**]. 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 [**?**]:

> **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 less subtle. There are many studies that adopt these clone types, analyzing them further and writing detection techniques for them [**?**, **?**, **?**].

3.2.1 Relevance to clone refactoring

Higher type clones are not only harder to detect, but also harder to refactor. How to refactor clones is heavily dependent on it's context and type. Method-level Type I clones can commonly be refactored by using the "Extract method" refactoring [**?**].

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

Chapter 4

Clone detection

Chapter 5

Analyzing clone harmfulness

Chapter 6

Analyzing clone refactorings

6.1 Catalogue of clones and their refactorings

Bibliography

[1] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[2] 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.

[3] Norman E Fenton and Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149–157, 1999.

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

[5] Stefan Haefliger, Georg Von Krogh, and Sebastian Spaeth. Code reuse in open source software. *Management science*, 54(1):180–193, 2008.

[6] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007.

[7] 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.

[8] E Kodhai, S Kanmani, A Kamatchi, R Radhika, and B Vijaya Saranya. Detection of type-1 and type-2 code clones using textual analysis and metrics. In *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, pages 241–243. IEEE, 2010.

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

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

[11] 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.

[12] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[13] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. IEEE, 2016.

[14] Brent van Bladel and Serge Demeyer. A novel approach for detecting type-iv clones in test code. In *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, pages 8–12. IEEE, 2019.

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