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

Academic Supervisor:
dr. Ana Oprescu

Author:
Simon Baars

Student Number:
12072931

Company Supervisor:
dr. Xander Schrijen

Company:
Software Improvement Group
Schrijen

April 4, 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 [1, 5]. 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 [2, 3], or does not happen at all [4]. 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. As some refactoring methods, like “extract method”, require user input (the name of the method in the case of “extract method”), this model will probably require some user input. Because of this, we have formulated the following research question:

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

Bibliography

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

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

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

[4] 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.

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