Improving Software Maintainability through Automated Refactoring of Code Clones

1st Simon Baars
University of Amsterdam
Amsterdam, the Netherlands
simon.mailadres@gmail.com

2nd Ana Oprescu
University of Amsterdam
Amsterdam, the Netherlands
ana.oprescu@uva.nl

***Abstract*—Duplication in source code is often seen as one of the most harmful types of technical debt, because it increases the size of the codebase and creates implicit dependencies between fragments of code. To remove such antipatterns, a developer should refactor the codebase. There are many tools that assist in this process. The thresholds and prioritization of the clones detected by such tools can be improved by considering the maintainability of the codebase after a detected clone would be refactored.**

In this study, we define a technique to detect clones such that they can be refactored. We propose a tool to detect such clones and automatically refactor a subset of them. We use a set of metrics to determine the impact of the applied refactorings to the maintainability of the systems. These metrics are system size, cyclomatic complexity, duplication and number of method parameters. On basis of these results, we decide which clones improve system design and thus should be refactored. We identified a set of four factors that influence the maintainability impact of clones. The first is the size of the clone. The second is the relation between the code fragments in a clone. The third is whether the clone fragments create, modify or return data. The fourth is the amount of data that cloned fragments use. By using these four factors, we can suggest only clones that will improve maintainability when refactored and prioritize them accordingly.

Given these maintainability influencing factors and their effect on the source code, we measure their impact over a large corpus of open source Java projects. This results in a set of thresholds by which clones can be found that should be refactored to improve system maintainability.

***Index Terms*—code clones, refactoring, static code analysis, object-oriented programming**

I. Introduction

Duplication in source code is often seen as one of the most harmful types of technical debt, because it increases the size of the codebase and create implicit dependencies between fragments of code [**?**]. Bruntink et al. [**?**] show that code clones can contribute up to 25% of the code size.

Current code clone detection detection techniques base their thresholds and prioritization on a limited set of metrics. Often, clone detection techniques are limited to measuring the size of clones to determine whether they should be considered. Because of this, the output of clone detection tools is often of limited assistance to the developer in the refactoring process.

In this study, we define a technique to detect clones such that they can be refactored. We evaluate the context of clones to determine refactoring techniques are required to refactor clones in a specific context. We propose a tool for the automated refactoring of a subset of the detected clones, by extracting a new method out of duplicated code.

II. Background

This study researches an intersection of code clone and refactoring research.

A. Code clone terminology

Clone class collection, clone class, clone instance, cloned node, token.

B. Code clone definitions

Quick overview of clone type definitions

C. Refactoring techniques

In this section, we describe refactoring techniques that are relevant to refactoring code clones.

1) Extract Method: The most used technique to refactor duplicate code is "Extract Method" [**?**], which can be applied on code inside the body of a method. Several studies have already concluded that most duplication in source code is found in the body of methods [**?**], [**?**], [**?**]. The "Extract Method" technique moves functionality in method bodies to a new method. To reduce duplication, we extract the contents of a single clone instance to a new method and replace all clone instances by a call to this method.

2) Move Method: Often, "Extract Method" alone is not enough to refactor clones. This is because the extracted method has to be moved to a location that is accessible by all clone instances. To do this, we apply the "Move Method" refactoring technique [**?**]. In object-oriented programming languages, we often move methods up in the inheritance structure of classes, also called "Pull Up Method" [**?**].

III. Defining refactorable clones

In literature, several clone type definitions have been used to argue about duplication in source code [**?**]. In this section, we discuss how we can define clones such that they can be refactored without side effects on the source code.

A. Ensuring Equality

Most modern clone detection tools detect clones by comparing the code textually together with the ommission of certain tokens [**?**], [**?**]. Clones detected by such means may not always be suitable for refactoring, because textua comparison fails to take into account the context of certain symbols in the code. Information that gets lost in textual comparison is the referenced declaration for type, variable and method references. Equally named type, variable and method references may refer to different declarations with a different implementation. Such clones can be harder to refactor, if benificial at all.

To detect clones that can be refactored, we propose to:

- Compare variable references not only by their name, but also by their type.
- Compare referenced types by their fully qualified identifier (FQI). The FQI of a type reference describes the full path to where it is declared.
- Compare method references by their fully qualified signature (FQS). The FQS of a method reference describes the full path to where it is declared, plus the FQI of each of its parameters.

B. Allowing variability in a controlled set of expressions

Often, fragments in source code do not match exactly. Often when developers duplicate fragments of code, they modify the duplicated block to fit its new location and purpose. To detect duplicate fragments with minor variance, we looked into in what expressions we can allow variability while still being refactorable.

C. Gapped clones

Explain type 3, why it is not always refactorable, solution

IV. CloneRefactor

The tool: Detect Clones, Map Context, Refactor.

A. Clone Detection

Detecting refactorable clones, (clone graph?? do I want to explain my exact methods?)

B. Context Mapping

1) Relation: The rationale for our categories regarding clone relations.

2) Location: The rationale for our categories regarding clone locations.

3) Contents: The rationale for our categories regarding clone contents.

C. Refactoring

1) Extract Method: Show my categories to show what clones can be dealt with by method extraction.

2) AST Transformation: Explain what AST transformations I do to apply the refactorings.

3) The cyclic nature of refactoring: Explain how refactoring code clones might open up new refactoring opportunities. We refactor a project until there are no more open refactoring opportunities.

V. Results

A. Clone context

How many clones are there in certain contexts? Experiments for relation, location and context.

B. Clone refactorability

To what extent can found clones be refactored through method extraction, without requiring additional transformations.

C. Thresholds

I think the ultimate goal with this thesis is to do experiments with different clone thresholds. Which thresholds give clones that we should refactor? For this, we will measure the maintainability of the refactored source code over different thresholds. These thresholds range from minimum clone size, variability and gap size.

VI. Discussion

A. Clone Definitions

B. CloneRefactor

C. Experimental setup

VII. Conclusion

Acknowledgment

We would like to thank the Software Improvement Group for their continuous support in this project. In particular, we would like to thank Xander Schrijen for his invaluable input in this research. Furthermore, we would like to thank Sander Meester for his proofreading efforts and feedback.

References