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 clones that 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.

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. In this background section, we will explain the required background knowledge for terms used throughout this study.

A. Code clone terminology

In this study we use two concepts used to argue about code clones [**?**]:
Clone instance: A single cloned fragment.
Clone class: A set of similar clone instances.

B. 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, duplication 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.

We define the following expressions as refactorable when varied:

- **Literals**: Only if all varying literals in a clone class have the same type.
- **Variables**: Only if all varying variables in a clone class have the same type.
- **Method references**: Only if the return value of referenced method match (or are not used).

Often when allowing such variance, tradeoffs come into play. For instance, variance in literals may require the introduction of a parameter to an extracted method if the "Extract Method" refactoring method is used, increasing the required effort to comprehend the code.

C. Gapped clones

Sometimes, when fragments are duplicated, a statement in inserted or changed severely for the code to fit its new context. When dealing with such a situation, there are several opportunities to refactor so-called "gapped clones". "Gapped clones" are two clone instances separated by a "gap" of non-cloned statement(s). We define the following methods to refactor such clones:

- We wrap the difference in statements in a conditionally executed block, one path for each different (group of) statement(s).
- We use a lambda function to pass the difference in statements from each location of the clone.

Again, a tradeoff is at play, as these solutions increase the complexity in favor of removing a clone.

IV. CloneRefactor

To automate the process of refactoring clones, we propose a tool named CloneRefactor. This tool goes through a 3-step process in order to refactor clones. This process is displayed in figure 1.

Fig. 1. Simple flow diagram of CloneRefactor.

In this section, we explain each of these steps.

A. Clone Detection

We use an AST-based method to detect clones. Clones are detected on a statement level: only full statements are considered as clones. In this process, we limit the variability between indicated expressions (see Sec. III-B) by a threshold. This threshold is the percentage of different expressions against the total number of expressions in the source code:

$$\text{Variability} = \frac{\text{Different expressions}}{\text{Total expressions}} * 100 \qquad (1)$$

After all clones have been detected, CloneRefactor determines whether clone classes can be merged into gapped clones (see Sec. III-C). The maximum size of the gap is limited by a threshold. This threshold is the percentage of (not-cloned) statements in the gap against the sum of statements in both clones surrounding it. Unlike the expression variability threshold, this threshold can exceed 100%:

$$\text{Gap Size} = \frac{\text{Statements in gap}}{\text{Statements in clones}} * 100 \qquad (2)$$

B. Context Mapping

After clones are detected, we map the context of these clones. We have identifier three properties of clones as their context: relation and contents. We identify categories for each of these properties, to get a detailed insight in the context of clones.

1) Relation: Clone instances in a clone class can have a relation with each other through inheritance. This relation has a big impact on how the clone should be refactored [**?**]. We define the following categories to map the relation between clone instances in a clone class. These categories do not map external classes (classes outside the project, for instance belonging to a library) unless explicitly stated:

- **Common Class**: All clone instances are in the same class.
  - **Same Method**: All clone instances are in the same method.
  - **Same Class**: All clone instances are in the same class.
- **Common Hierarchy**: All clone instances are in the same inheritance hierarchy.
  - **Superclass**: Clone instances reside in a class or its parent class.
  - **Sibling Class**: All clone instances reside in classes with the same parent class.
  - **Ancestor Class**: All clone instances reside in a class, or any of its recursive parents.
  - **First Cousin**: All clone instances reside in classes with the same grandparent class.

  - **Same Hierarchy**: All clone instances are part of the same inheritance hierarchy.
- **Common Interface**: All clone instances are in classes that have the same interface.
  - **Same Direct Interface**: All clone instances are in a class that have the same interface.
  - **Same Class**: All clone instances are in an inheritance hierarchy that contains the same interface.
- **Unrelated**: All clone instances are in classes that have the same interface.
  - **No Direct Superclass**: All clone instances are in classes that have the Object class as parent.
  - **No Indirect Superclass**: All clone instances are in a hierarchy that contains a class that has the Object class as parent.
  - **External Superclass**: All clone instances are in classes the same external class as parent.
  - **Indirect External Ancestor**: All clone instances are in a hierarchy that contains a class that has an external class as parent.

Based on these relations, we determine where to place the cloned code when refactored. The code of clones that have a *Common Class* relation can be refactored by placing the cloned code in this same class. The code of clones with a *Common Hierarchy* relation can be placed in the intersecting class in the hierarchy (the class all clone instances have in common as an ancestor). The code of clones with a *Common Interface* relation can be placed in the intersecting interface, but in the process has to become part of the classes' public contract. The code of clones that are *Unrelated* can be placed in a newly created place: either a utility class, a new superclass abstraction or an interface.

2) Contents: The contents of a clone instance determine what refactoring techniques can be applied to refactor such clones. We define the following categories by which we analyze the contents of clones:

1) **Full Method/Constructor/Class/Interface/Enumeration:** A clone that spans a full class, method, constructor, interface or enumeration, including its declaration.
2) **Partial Method/Constructor:** A clone that spans (a part of) the body of a method/constructor.
3) **Several Methods:** A clone that spans over two or more methods, either fully or partially, but does not span anything but methods.
4) **Only Fields:** A clone that spans only global variables.
5) **Other:** Anything that does not match with above-stated categories.

3) Full Method/Constructor/Class/Interface/Enumeration: These categories denote that a full declaration, including its body, is cloned with another declaration. These categories often denote redundancy and are often easy to solve: one of both declarations is redundant and should be removed. All usages of the removed declaration should be redirected to the clone instance that was not removed. We check for clones in this category by checking that the first node in a clone instance is a declaration and the last node in the clone instance is the last node in the body of that declaration.

4) Partial Method/Constructor: These categories describe clone instances which are found in the body of a method or constructor. These clones can often be refactored by extracting a new method out of the cloned code. We detect such clones by checking that each node in the clone instance has a ancestor node that is a MethodDeclaration, but none of the nodes is a method declaration. We describe the procedure by with we recursively seek the parent nodes of a node in Section **??**.

5) Several Methods: Several methods cloned in a single class is a strong indication of implicit dependencies between two classes. This increases the chance that these classes are missing some form of abstraction, or their abstraction is used inadequately. We detect such clones by checking that each cloned node has a parent node that is a MethodDeclaration. This is the same process as described in equation **??**. Additionally, we check that the MethodDeclaration ancestor of the first node in the clone instance differs from the last, to be sure that the clone instance spans over at least two methods.

6) Only Fields: This category denotes that the clone spans over only global variables, fields that are declared outside of a method. This is to indicate data redundancy: pieces of data have an implicit dependency. In such cases, these fields may have to be encapsulated in a new object. Or, the fields should be somewhere in the inheritance structure where all objects containing the clone can access them. We check for this category by validating that all nodes in the clone instance are a VariableDeclarator. Because VariableDeclarators in methods would already fall into the "Partial Method" category, this includes only globally defined fields.

7) Other: The "Other" category denotes all configurations of clone contents that do not fall into above categories. Often, these are combinations of above states concepts. For instance, a combination of constructors and methods is clones, or a combination of fields and methods. Such clones indicate, like the "Several Methods", the requirement of performing a more architectural-level refactoring. These are often more complicated, especially when aiming to automate this process.

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.

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