Improving Software Maintainability through Automated Refactoring of Code Clones

Simon Baars
University of Amsterdam
simon.mailadres@gmail.com

Ana Oprescu
University of Amsterdam
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.

KEYWORDS

code clones, refactoring, static code analysis, object-oriented programming

ACM Reference Format:
Simon Baars and Ana Oprescu. 2019. Improving Software Maintainability through Automated Refactoring of Code Clones. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 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.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

2 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.

2.1 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.

2.2 Refactoring techniques

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

2.2.1 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.

2.2.2 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" [**?**].

3 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.

3.1 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.

3.2 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.

3.3 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.

4 CLONEREFACTOR

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

Figure 1: Simple flow diagram of CloneRefactor.

In this section, we explain each of these steps.

4.1 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. 3.2) by a threshold. This threshold is the percentage of different expressions against the total number of expressions in the source code:

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

After all clones have been detected, CloneRefactor determines whether clone classes can be merged into gapped clones (see Sec. 3.3). 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%:

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

4.2 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.

4.2.1 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.

[1]The source code of CloneRefactor is available on GitHub: https://github.com/SimonBaars/CloneRefactor

  - **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.

4.2.2 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.

4.2.3 Full Method/Constructor/Class/Interface/Enumeration. The categories denote that a full declaration (method, class, etc.) often denote redundancy and are often easy to refactor: one of both declarations is redundant and should be removed. Clones in the "Partial Method/Constructor" category can often be refactored using the "Extract Method" refactoring technique. Clones consisting of *Several Methods* give a strong indication that cloned classes are missing some form of abstraction, or their abstraction is used inadequately. Clones consisting of *Only Fields* often indicate data redundancy: different classes use the same data.

4.3 Refactoring

CloneRefactor can refactor clones using the "Extract Method" refactoring technique. In this section we show which clones we refactor and how we apply these transformations.

4.3.1 Extract Method. There are several influencing factors that may obstruct the possibility to extract code to a new method:

- **Complex Control Flow**: This clone contains `break`, `continue` or `return` statements, obstructing the possibility of method extraction.
- **Spans Part Of A Block**: This clone spans a part of a block statement.
- **Is Not A Partial Method**: If the clone does not fall in the "Partial method" category of Sec. 4.2.2, the "Extract Method" refactoring technique cannot be applied.
- **Returns multiple values**: If a clone modifies or declares multiple pieces of data that it should return.
- **Can Be Extracted**: This clone is a fragment of code that can directly be extracted to a new method. Then, based on the relation between the clone instances, further refactoring techniques can be used to refactor the extracted methods (for instance "pull up method" for clones in sibling classes).

Clones that do not fall in the *Can Be Extracted* category may require additional transformations or other techniques to refactor. CloneRefactor only refactors the clones that *Can Be Extracted*.

4.3.2 AST Transformation. CloneRefactor uses JavaParser [**?**]: an AST-parsing library that allows to modify the AST and write it back to source code. To refactor clones, CloneRefactor creates a new method declaration and moves all statements from a clone instance in the clone class to the new method. This method is placed according to the relation between the clone instances (see Sec. 4.2.1). CloneRefactor analyzes the source code of the extracted method and populates it with the following properties:

- **Parameters**: For each variable used that is not accessible from the scope of the extracted method.
- **A return value**: If the method modifies or declares local data that is needed outside of its scope, or if the cloned fragments already returned data.
- **Thrown exception**: If the method throws an uncaught exception that is not a `RuntimeException`.

CloneRefactor then removes all cloned code and replaces it with a method call to the newly created method. In case of a return value, CloneRefactor either assigns the call result, declares it or returns it accordingly.

4.3.3 Characteristics of the extracted method. We define the following characteristics of the extracted method and/or the call:

- **Statements**: The number of statements in the body of the method.
- **Tokens**: The number of tokens in the body of the method.
- **Relation**: The relation category (Sec. 4.2.1) by which this methods' location was determined.

- **Returns**: Whether the method calls return, declare, assign or don't use any data from the extracted method.
- **Parameters**: The number of parameters the extracted method has.

We hypothesize that these characteristics are the main factors influencing the impact on the maintainability of the system as a result of refactoring the clone.

4.3.4 Impact on maintainability metrics. We measure the impact on maintainability metrics of the refactored source code for each clone class that is refactored. These metrics are derived from Heitlager et al. [**?**]. This paper defines a set of metrics to measure the maintainability of a system. For each of these metrics, risk profiles are proposed to determine the maintainability impact on the system of a whole.

In this case we are dealing with single refactorings and we want to measure the maintainability impact of such a small change. Because of that, we measure only a subset of the metrics [**?**] and focus on the absolute metric changes (instead of the risk profiles). The subset of metrics we decided to focus on are all metrics that are measured on method level (as the other metrics show a lesser impact on the maintainability for these small changes). These metrics are

- **Duplication**: In Heitlager et al. [**?**] this metric is measured by taking the amount of duplicated lines. We decided to use the amount of duplicated tokens part of a clone class instead, to have a stronger reflection of the impact of the refactoring by measuring a more fine-grained system property.
- **Volume**: The more code a system has, the more code has to be maintained. The paper [**?**] measures volume as lines of code. As with duplication, we use the amount of tokens instead.
- **Complexity**: Heitlager et al. use MCCabe complexity [**?**] to calculate their complexity metric. The MCCabe complexity is a quantative measure of the number of linearly independent paths through a method.
- **Method Interface Size**: The amount of parameters that a method has. If a method has many parameters, the code may become harder to understand and it is an indication that data is not encapsulated adequately in objects [**?**].

5 EXPERIMENTAL SETUP

In this section we describe the setup of our experiments.

5.1 Corpus

We ran all our experiments using CloneRefactor on a corpus of open source Java projects. This corpus is derived from the corpus of a study that uses machine learning to determine the suitability of Java projects on GitHub for data analysis purposes [**?**].

CloneRefactor requires all libraries of the projects it analyses, in order to find the full paths of all referenced symbols in the source code (see Sec. **??**). Because of that requirement, we decided to filter the corpus to only projects using the Maven build automation system. We created a set of scripts[2] to prepare such a corpus with all dependencies included.

This procedure results in 2.267 Java projects including all their dependencies. The projects vary in size and quality.

5.2 Minimum clone size

In this study, we want to find out what thresholds to use to improve maintainability if clones by those thresholds are refactored. This implies that

5.3 Calculating a maintainability score

In this study we use four metrics to determine maintainability **??**. For our experiments, we want to aggregate the scores obtained by these metrics to draw a conclusion about the maintainability increase or decrease after applying a refactoring. We base our aggregation on the following assumptions:

- All metrics are equal in terms of weight towards system maintenance effort.
- Higher values for the metrics imply lower maintainability.
- The obtained increase of the metric divided by the average distance from zero over our corpus weights the metric equally against the other metrics.

We calculate the average deviation as follows:

$$A = \frac{\sum_{x \in M} abs(x)}{|M|} \tag{3}$$

Where M is the multiset of all values obtained for a certain metric over our corpus and $|M|$ is the cardinality of this set. We then calculate the maintainability for a specific refactoring as follows:

$$A_M = \frac{dup}{A_{dup}} + \frac{com}{A_{com}} + \frac{par}{A_{par}} + \frac{vol}{A_{vol}} \tag{4}$$

Where dup is the decrease in duplication, com is the decrease in complexity, par is the decrease is method parameters and vol is the decrease in system volume after the refactoring is applied.

6 RESULTS

6.1 Clone context

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

6.2 Clone refactorability

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

6.3 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.

[2]All scripts to prepare the corpus are available on GitHub: https://github.com/SimonBaars/GitHub-Java-Corpus-Scripts

7 DISCUSSION

7.1 Clone Definitions

7.2 CloneRefactor

7.3 Experimental setup

8 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 proof-reading efforts and feedback.