Improving Software Maintainability through Automated Refactoring of Code Clones

ABSTRACT

Duplication in source code is often seen as one of the most harmful types of technical debt as it increases the size of the codebase and creates implicit dependencies between fragments of code. To remove such anti-patterns, the codebase should be refactored. Many tools aid in the detection process of such duplication problems but fail to determine whether refactoring a duplicate fragment would improve the maintainability.

We address this shortcoming by first analyzing what data should be gathered from an automatically refactored system. We then propose a tool to detect clones, analyze their context and automatically refactor a subset of them. We use a set of established metrics to determine the impact of the applied refactorings on the maintainability of the system. Based on these results, one could decide which clones would improve system design if refactored. We evaluate our approach over a large corpus of open source Java projects.

We analyze the overhead of applying a refactoring in terms of four factors: the size of the clone, the relation between the code fragments in a clone, whether clone fragments create, modify or return data and the amount of external data that cloned fragments use. We find that the combined volume of similar fragments in a clone is the biggest influencing factor: the majority of duplicates with a total size of 63 or more tokens improve maintainability when refactored. The amount of external data that needs to be passed to the merged location of the duplicate is the other important factor: the majority of clones with more than requiring more than one external parameter decreases maintainability.

KEYWORDS

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

ACM Reference Format:

. 2019. Improving Software Maintainability through Automated Refactoring of Code Clones. In *Proceedings of Technical Papers (ICSE 2020)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Duplicate fragments in source code (also named "code clones") are often seen as one of the most harmful types of technical debt [5]. Duplicate fragments create implicit dependencies that make the code harder to maintain as the resolution of erroneous behavior in one location may have to be applied to all the cloned code as well [16]. Apart from that, code clones can contribute up to 25% of the code size [3].

Current code clone 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 automatically refactored. We list a set of criteria that can obstruct refactoring opportunities. We evaluate the relation and location of clones to determine which refactoring techniques are best suited. We introduce a tool that automatically refactors clones by extracting new methods out of duplicated code.

We evaluate our approach by comparing the maintainability of before- and after snapshots of a codebase for each duplication problem that is refactored. We define which factors have the greatest influence on whether a clone improves maintainability when refactored. We find a set of thresholds by which clones can be detected that are more likely to improve maintainability when refactored. This allows for more a accurate suggestion of code clones for refactoring and can assist in the refactoring process.

2 BACKGROUND

Our study lies at the intersection of code clone and refactoring research. We use two concepts to argue about code clones [18]:
Clone instance: A single cloned fragment.
Clone class: A set of similar clone instances.

The most common technique for refactoring clones is "Extract Method" [5], which can be applied on code inside the body of a method. Applying this technique entails moving functionality in the body of a method to a new method. To reduce duplication with this technique, we extract the contents of a single clone instance to a new method and replace all clone instances in a clone class by a call to this method.

3 RELATED WORK

The state-of-the-art in code clone refactoring is the work by Mazinanian et al. [14]. The authors propose a tool named JDeodorant that can automatically refactor a subset of duplication problems. This tool is based on several studies that look into different aspects of code clone refactoring. In a study by Krishnan et al. [11] clone refactoring is approached as an optimization problem: how does variability between cloned fragments influence the refactoring techniques required and their implications on system design. Another study by Krishnan et al. [12] proposes a list of preconditions for clones to determine their refactorability. This list is revised by Tsantalis et al. [28], proposing a set of 8 preconditions that determine whether two duplicated fragments can be refactored.

ICSE 2020, ,

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

```java
package com.sb.game;

import java.util.List;

public class GameScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void addTen(int x) {
    x = x + 10; // Addition
    Notifier.notifyChanged(x);
    return x;
  }
}
```

```java
package com.sb.fruitgame;

import java.awt.List; // Different type

public class LoseScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void concatTen(String x) {
    x = x + 10; // String concat
    Notifier.notifyChanged(x);
    return x;
  }
}
```

Figure 1: Example of textually equal code fragments that differ in functionality.

A recent study [29] looks into how lambdas can assist the clone refactoring process. The main focus is to find out which clones **can** be refactored. We extend this work by looking into which clones **should** be refactored. We propose definitions for refactorable clones together with thresholds to be able to limit their negative impact on system design. We measure which clones improve maintainability when refactored. This results in a set of thresholds that can be used to detect and refactor clones that should be refactored.

4 DEFINING REFACTORABLE CLONES

In literature, several clone type definitions have been used to argue about duplication in source code [18]. We discuss how we can define clones such that they can be automatically refactored without side effects on the source code.

4.1 Ensuring Equality

Most modern clone detection tools detect duplication by comparing the code textually together with the omission of certain tokens [21, 27]. Clones detected by such means may not always be suitable for refactoring, because textual 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 (Fig 1 shows an example of this). Such clones can be harder to refactor, if beneficial at all.

To detect automatically refactorable clones, 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.

4.2 Allowing variability in a controlled set of expressions

Often, duplication fragments in source code do not match exactly [10]. 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 look into what expressions we can allow variability in, 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 methods match (or are not used).

Often when allowing such variance, trade-offs come into play [11, 12]. 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.

4.3 Gapped clones

Sometimes, when fragments are duplicated, a statement is inserted or changed severely for the code to fit its new context [18]. When dealing with such a situation, there are several opportunities to refactor so-called “gapped clones” [30, 31]. “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 [29].

For both refactoring techniques, a trade-off is at play. This is because these solutions increase the complexity and volume of the source code in favor of removing a clone.

5 THE CLONEREFACTOR TOOL

To detect such clones that can be refactored, we surveyed a set of modern clone detection tools and techniques [6, 21, 25, 27]. We created a set of control projects[1] to determine the suitability of these tools to detect refactorable clones, either through configuration or postprocessing of their output.

None of the surveyed tools [8, 9, 14, 17, 19, 22–24] were suitable with our findings about refactorable clones, because of which we decided to implement a tool: CloneRefactor[2]. This tool goes through a 3-step process to automatically refactor clones as shown in Fig. 2. In the following sections, we explain these steps.

Figure 2: Simple flow diagram of CloneRefactor.

5.1 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. 4.2) by a threshold. This threshold is the percentage of different expressions against the total number of expressions in the source code:

$$\text{Variability} = \frac{\text{Different expressions}}{\text{Total expressions}} * 100 \tag{1}$$

After all clones have been detected, CloneRefactor determines whether clone classes can be merged into gapped clones (see Sec. 4.3). 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%:

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

To verify the correctness of all detected clones, we ran the tool over a large project and manually checked the output. We also created a set of control projects to test the correctness for many edge cases.

[1] Control projects for assessing clone detection tools: https://github.com/SimonBaars/CloneRefactor/tree/master/src/test/resources
[2] The source code of CloneRefactor is available on GitHub: https://github.com/SimonBaars/CloneRefactor

5.2 Context Mapping

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

5.2.1 Relation. Clone instances in a clone class may have a relation with each other through inheritance. This relation has a big impact on how the clone should be refactored [4]. 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, e.g. 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 classes 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 their parent.
  - **No Indirect Superclass**: All clone instances are in a hierarchy that contains a class that has the Object class as their parent.
  - **External Superclass**: All clone instances are in classes the same external class as their parent.
  - **Indirect External Ancestor**: All clone instances are in a hierarchy that contains a class that has an external class as their parent.

Based on these relations, CloneRefactor determines where to place the cloned code when refactored. These categories are mutually exclusive: a clone class will be flagged as the first relation in the above list that it applies to. 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*

```java
int a = getA();
while(a<1000) {
  a *= 5;
  doC();
}
```

```java
int a = getA();
while(a<1000) {
  a *= 5;
  doB(a);
}
```

Figure 3: A clone that spans a block partially.

can be placed in a newly created place: either a utility class, a new superclass abstraction or an interface.

5.2.2 Location. The location of a clone instance determines what refactoring techniques can be applied to refactor such clones. We define the following categories by which we analyze the location 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 Body:** 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.

5.2.3 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.

5.3 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.

5.3.1 Extract Method. Several influencing factors 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. An example of this is shown in Fig. 3.
- **Is Not A Partial Method**: If the clone does not fall in the "Partial method" category of Sec. 5.2.2, 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.
- **Top-Level Non-Statement**: If one of the top-level AST nodes of the clone is not a statement. For instance, if a (part of) an anonymous class is cloned.
- **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.*

5.3.2 AST Transformation. CloneRefactor uses JavaParser [26]: 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. 5.2.1). 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.

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

- **Tokens**: The number of tokens in the body of the method.
- **Relation**: The relation category (Sec. 5.2.1) 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.

5.3.4 Impact on maintainability metrics. CloneRefactor measures 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. [7]. 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.

To determine whether the maintainability improves when refactoring a single clone, we need to measure the impact of small-grained changes. Because of that, we measure only a subset of the metrics [7] 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. [7] 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 [7] measures volume as lines of code. As with duplication, we use the number of tokens instead.
- **Complexity**: Heitlager et al. [7] use MCCabe complexity [15] to calculate their complexity metric. The MCCabe complexity is a quantitative measure of the number of linearly independent paths through a method.
- **Method Interface Size**: The number 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].

6 EXPERIMENTAL SETUP

In this section, we describe the setup of our experiments.

6.1 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 [1].

CloneRefactor requires all libraries of the projects it analyses, to find the full paths of all referenced symbols in the source code (see Sec. 4.1). 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[3] 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. The total size of all projects is 14.2M lines (11.3M when excluding whitespace) over a total of 100K Java files. This is an average of 6.3K lines over 44 files per project. The largest project in the corpus is *VisAD* with 502K lines.

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

6.2 Tool Validation

We have validated the correctness of CloneRefactor through unit tests and empirical validation. First, we created a set of 57 control projects to verify the correctness in many (edge) cases. These projects contain clones for each identified relation, location, and refactorability category to see whether they get correctly identified. Next, we run the tool over the corpus and manually verify samples of the acquired results. This way, we check both the correctness of the identified clones, their context, and their proposed refactoring.

We also test the correctness of the resulting code after refactoring. For this, we use a GitHub project named JFreeChart. JFreeChart has a high test coverage and working tests, which allows us to test the correctness of the program after running CloneRefactor.

6.3 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. However, when clones are very small, they may never be able to improve maintainability. The detrimental effect of the added volume of the newly created method exceeds the positive effect of removing duplication. Because of that, we perform all our experiments with a minimum clone size of 10 tokens, because smaller clones are very unlikely to improve maintainability when refactored.

6.4 Thresholds

Most clone detection tools can be configured using thresholds. These thresholds indicate the minimum number of lines, tokens and/or statements that must be spanned for duplicate fragments to be considered clones. Often, such thresholds are intuitively chosen [13, 20] or based on a quartile distribution of empirical data [2]. Using the maintainability score we can find support for which thresholds should be chosen to increase the chance to find clones that improve maintainability when refactored.

6.5 Calculating a maintainability score

In this study, we use four metrics to determine maintainability (see Sec. 5.3.4). For each of these metrics, we determine their value before and after each refactored clone class, resulting in a delta metric score. For our experiments, we aggregate the deltas obtained for 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.
- Normalizing each obtained metric delta over all deltas obtained for that metric in our dataset results in equally weighted scores.

We derived these assumptions from supporting evidence shown by Heitlager et al [7] and Alves et al. [2]. Using the resulting aggregated maintainability score, we can argue for each refactoring whether it increases or decreases the maintainability of the system.

We normalize each obtained metric delta using the "Standard score", which is calculated as follows:

$$N_{metric} = \frac{\Delta X - \mu}{\sigma} \tag{3}$$

Where ΔX is a metric delta, μ is the mean of all deltas for this metric and σ is the standard deviation of all deltas for this metric. This method works well for normalization of our data because as we divide by the standard deviation, outliers do not influence the resulting scores negatively.

We then calculate the maintainability for a specific refactoring as follows:

$$N_{duplication} + N_{complexity} + N_{volume} + N_{parameters} \tag{4}$$

7 RESULTS

In this section, we share the results of our experiments.

7.1 Clone context

To determine the refactoring method(s) that can be used to refactor most clones, we perform statistical analysis on the context of clones (see Sec. 5.2).

7.1.1 Relation. Table 2 displays the number of clone classes found for the entire corpus for the identified relations between clone instances (see Sec. 5.2.1).

Category	*Relation*	*Clone Classes*	*Total*
Common Class	Same Class	22,893	31,848
	Same Method	8,955	
Common Hierarchy	Sibling	15,588	20,342
	Superclass	2,616	
	First Cousin	1,219	
	Common Hierarchy	720	
	Ancestor	199	
Unrelated	No Direct Superclass	10,677	20,314
	External Superclass	4,525	
	External Ancestor	3,347	
	No Indirect Superclass	1,765	
Common Interface	Same Direct Interface	7,522	13,074
	Same Indirect Interface	5,552	

Table 2: Number of clone classes per clone relation

Our results show that most clones (37%) are in a common class. 24% of clones are in a common hierarchy. Another 24% of clones are unrelated. 15% of clones are in an interface.

7.1.2 Location. Table 3 displays the number of clone classes found for the entire corpus for different locations (see Sec. 5.2.2).

Category	*Location*	*Clone instances*	*Total*
Partial	Partial Method	219,540	229,521
	Partial Constructor	9,981	
Full	Full Method	12,990	13,173
	Full Interface	64	
	Full Constructor	58	
	Full Class	37	
	Full Enum	24	
Other	Several Methods	22,749	53,773
	Only Fields	17,700	
	Other	13,324	

Table 3: Number of clone instances for clone location categories

From these results, we see that 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods. 6% of clones span only global variables. Only 4% of clones span a full declaration (method, class, constructor, etc.).

7.2 Refactorability

Table 4 shows to what extent clone classes can be refactored by using the "Extract Method" refactoring technique (see Sec. 5.3.1).

Category	*All*	*%*
Can Be Extracted	24,157	28.2%
Is Not A Partial Method	21,625	25.3%
Top-level AST-Node is not a Statement	19,887	23.2%
Spans Part of a Block	13,181	15.5%
Multiple Return Values	5,622	6.6%
Complex Control Flow	1,106	1.3%

Table 4: Number of clones that can be extracted using the "Extract Method" refactoring technique.

These results indicate that, given our refactorability criteria, 28% of clones can be automatically refactored. Clones in other categories may require other refactoring techniques or further transformations to be automatically refactorable.

7.3 Thresholds

In our corpus, CloneRefactor has refactored 12.710 clone classes and measured the change in indicated metrics (see Sec. 5.3.4). Using the presented formulas (see Sec. 6.5) we determine how the characteristics of the extracted method (see Sec. 5.3.3) influence the maintainability of the resulting codebase after refactoring. In this section, we explore the data received by comparing the before- and

Relation	*Duplication*	*Complexity*	*Parameters*	*Volume*	*#*	*Score*
Common Hierarchy	**-66.33**	**0.73**	**1.20**	**-8.85**	**2,202**	**0.23**
Superclass	-64.48	0.79	0.94	-7.22	229	0.42
Sibling	-70.07	0.69	1.28	-10.97	1,722	0.23
Same Hierarchy	-44.18	0.95	0.89	1.54	87	0.10
First Cousin	-42.69	0.89	0.93	4.86	144	0.02
Ancestor	-32.75	1.00	0.75	11.00	20	-0.03
Common Interface	**-47.06**	**0.83**	**1.04**	**4.50**	**1,044**	**-0.02**
Same Indirect Interface	-37.08	0.93	0.82	9.96	487	-0.01
Same Direct Interface	-55.79	0.75	1.24	-0.28	557	-0.02
Common Class	**-52.42**	**0.87**	**1.13**	**1.47**	**7,239**	**-0.02**
Same Class	-51.85	0.86	1.03	3.36	4,874	0.04
Same Method	-53.60	0.90	1.32	-2.44	2,365	-0.15
Unrelated	**-45.86**	**0.88**	**1.08**	**9.56**	**2,198**	**-0.15**
No Direct Superclass	-52.24	0.84	1.12	6.04	811	-0.06
External Superclass	-47.09	0.87	1.13	8.77	697	-0.17
External Ancestor	-35.73	0.93	0.95	14.58	586	-0.21
No Indirect Superclass	-44.89	0.84	1.18	14.08	104	-0.30
Grand Total	**-53.26**	**0.84**	**1.12**	**1.33**	**12,683**	**0.00**

Table 1: Influence on maintainability of refactoring clones with the specified relation categories.

after snapshots of the system for each separate refactoring. Using this data, we find supporting evidence regarding which thresholds are most likely to find clones that should be refactored to improve maintainability.

7.3.1 Clone Token Volume. Figure 4 shows the obtained results when plotting the clone volume vs the average delta maintainability score. We define the *token volume* as the combined number of tokens in all clone instances in a refactored clone class. For higher token volume numbers we have fewer refactorings that refactor such clones, because of which we represent the x-axis as a logarithmic scale. The trendline intersects the "zero" line (maintainability does not increase nor decrease) at a token volume of 63.

7.3.2 Relation. Table 1 shows our data regarding how different relations influence maintainability. We have marked rows based on less than 100 refactorings red, as their result does not have statistical significance.

The displayed relations are ordered by their scores. Overall, the scores do not deviate much (-0.15 to 0.23), indicating that the relation between clones has a minor impact on maintainability. Overall, we see that common hierarchy clones have the highest maintainability, whereas unrelated clones have the lowest maintainability.

7.3.3 Return Value. Table 5 shows how the return value of the extracted method influences the maintainability of the resulting system.

The displayed return categories are ordered by their scores. Overall, the scores do not deviate much (-0.18 to 0.19), indicating that the return category has a minor impact on maintainability. When the result of the call to the extracted method is directly returned, the maintainability score is the highest. When no value is returned, the maintainability score is the lowest. The main reason that no return value ends up lowest, is that it is linked to a higher number of parameters required for the extracted method.

7.3.4 Parameters. Fig. 5 shows how an increase in parameters lowers the maintainability of the refactored code. On the primary x-axis, the maintainability is displayed. The secondary x-axis shows the number of refactorings. The y-axis shows the number of parameters.

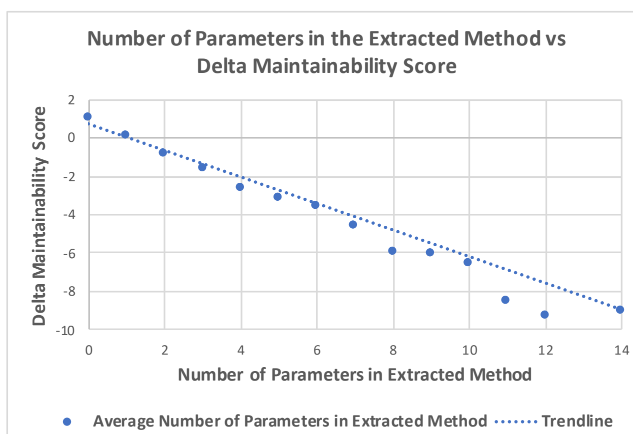

Figure 5: Influence of number of method parameters on system maintainability.

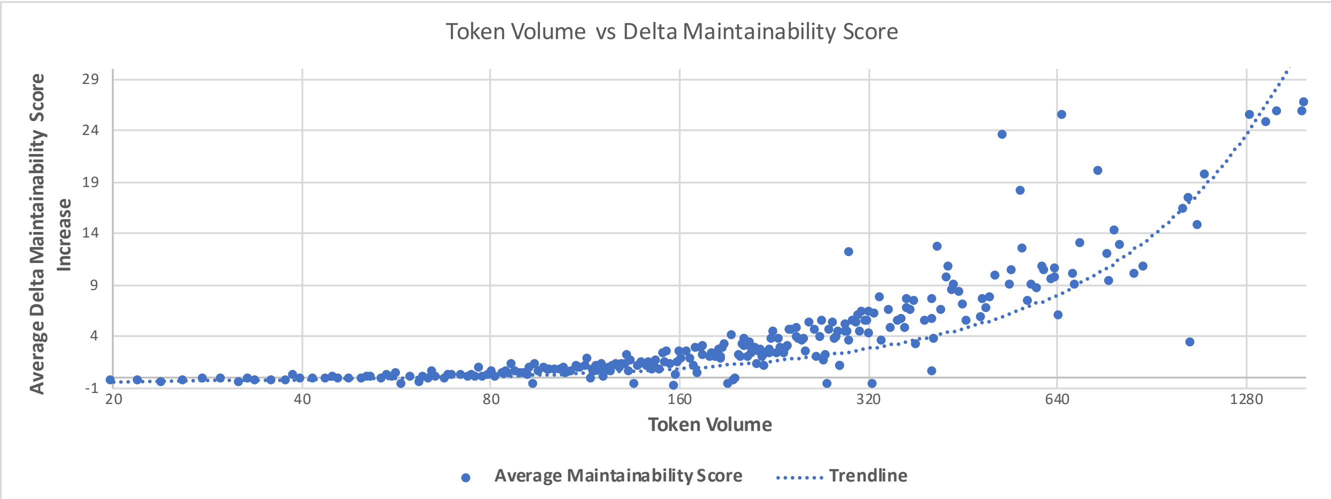

Figure 4: A graph that shows how the size in tokens of the refactored clone affects maintainability.

Return Category	*Complexity*	*Parameters*	*Size*	*Duplication*	*#*	*Score*
Return	0.85	1.02	-3.84	-55.00	1,571	0.19
Declare	0.94	0.74	11.11	-49.19	5,177	0.15
Assign	0.79	1.07	0.43	-56.29	14	0.12
Void	0.76	1.49	-5.85	-56.35	5,921	-0.18
Grand Total	**0.84**	**1.12**	**1.33**	**-53.26**	**12,683**	**0.00**

Table 5: Average metric values for refactorings of clone classes with the specified return category

8 DISCUSSION

8.1 Clone Context

Regarding clone context, our results indicate that most clones (37%) are in a common class. This is favorable for refactoring because the extracted method does not have to be moved after extraction. 24% of clones are in a common hierarchy. These refactorings are also often favorable. Another 24% of clones are unrelated, which is often unfavorable because it often requires a more comprehensive refactoring. 15% of clones are in an interface.

Regarding clone locations, 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods, which often require refactorings on a more architectural level. 6% of clones span only global variables, requiring an abstraction to encapsulate these data declarations. Only 4% of clones span a full declaration (method, class, constructor, etc.).

8.2 Extract Method

28% of clones can be refactored using the “Extract Method” refactoring technique (50% if we limit our searching scope to method bodies). About 25% of clones do not span part of a method, because of which they cannot be refactored. Many clones (23%) do not have a statement as top-level AST-Node. Upon manual inspection, we noticed that the main reason for this is when clones are found in lambda functions or anonymous classes. About 15% of clones span only part of an AST-Node.

8.3 Refactoring

In Fig. 4 we see an increase in maintainability for refactoring larger clone classes. The tipping point, between a better maintainable refactoring and a worse maintainable refactoring, seems to lie around 28 tokens. There are fewer large clones than small clones, resulting in a very limited statistical significance on our corpus when considering clones larger than 100 tokens.

In Table 1 we see the results regarding refactorings that are applied to clones with diverse relations. We see that most refactored clones are in a common class, over 54%. This is significantly more than the percentage of clones in the common class relation as reported in Table 2. Meanwhile, the number of refactored unrelated clones is smaller than the number reported in Table 2 (24% -> 18%). The main reason for this is that refactoring unrelated clones can change the relation of other clones in the same system. If we create a superclass abstraction to refactor an unrelated clone, other clones in those classes that were previously unrelated might become related.

The maintainability scores displayed in Table 1 show that the most favorable clones to refactor are clones with a Sibling relation. The most unfavorable is to refactor clones to interfaces. However, the differences in maintainability in this table are generally small;

according to our data relations have only a minor impact on the maintainability of clones.

Regarding the return type of refactored clones, we see in Table 5 that this has no major impact on maintainability. A method call to the extracted method that is directly returned and no return type extracted methods are slightly more favorable than the others. We think the main reason that the "Return" category is on top is that when a variable is declared at the end of the cloned fragment, CloneRefactor directly returns its value and removes the declaration. This decreases the volume slightly.

A higher number of parameters directly influences the corresponding metric. Because of this, we see in Fig. 5 that more parameters negatively influence maintainability. Not only the number of parameters metric is negatively influenced, but more method parameters also increase volume for the extracted method and each of the calls to it. Because of that, we see that the trend of the graph in Fig. 5 decreases relatively rapidly.

9 CONCLUSION

We defined automatically refactorable clones and created a tool to detect and refactor them. We measured statistical data with this tool over a large corpus of open-source Java software systems to get more information about the context of clones and how refactoring them influences system maintainability.

We defined two aspects as part of the context of a clone: relation and location. Regarding relations, over 37% of clones are found in the same class. About 24% of clones are in the same inheritance hierarchy. Another 24% of clones are unrelated. The final 15% of clones have the same interface. Regarding location, over 74% of clones span part of a method. About 8% span several methods. Only 4% of clones span a declaration (method, class, etc.) fully.

We built a tool that can automatically apply refactorings to 28% of the clones in our corpus using the "Extract Method" refactoring technique. The main reason our tool could not refactor all clones is that many clones span certain statements that obstruct method extraction, e.g., when code outside a method is part of a clone.

We measured four maintainability metrics before- and after applying each refactoring to determine the impact of each refactoring on system maintainability. We found that the most prominent factor influencing maintainability is the size of the clone. We found that the threshold lies at a clone volume of 29 tokens per clone instance for system maintainability to increase after refactoring the clone. Another factor with a major impact on maintainability is the number of parameters that the extracted method requires to get all required data. We noticed that the inheritance relation of the clone and the return value of the extracted method has only a minor impact on system maintainability.

ACKNOWLEDGMENTS

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

[1] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*. IEEE, 207–216.

[2] Tiago L Alves, Christiaan Ypma, and Joost Visser. 2010. Deriving metric thresholds from benchmark data. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[3] Magiel Bruntink, Arie Van Deursen, Remco Van Engelen, and Tom Tourwe. 2005. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering* 31, 10 (2005), 804–818.

[4] Francesca Arcelli Fontana and Marco Zanoni. 2015. A duplicated code refactoring advisor. In *International Conference on Agile Software Development*. Springer, 3–14.

[5] Martin Fowler. 2018. *Refactoring: improving the design of existing code* (second ed.). Addison-Wesley Professional.

[6] Pratiksha Gautam and Hemraj Saini. 2016. Various Code Clone Detection Techniques and Tools: A Comprehensive Survey. In *SmartCom 2016: Smart Trends in Information Technology and Computer Communications*. 655–667. https://doi.org/10.1007/978-981-10-3433-6_79

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

[8] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[9] CM Kamalpriya and Paramvir Singh. 2017. Enhancing program dependency graph based clone detection using approximate subgraph matching. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE, 1–7.

[10] E Kodhai and S Kanmani. 2013. Method-Level code clone modification using refactoring techniques for clone maintenance. *Advanced Computing* 4, 2 (2013), 7.

[11] Giri Panamoottil Krishnan and Nikolaos Tsantalis. 2013. Refactoring clones: An optimization problem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 360–363.

[12] Giri Panamoottil Krishnan and Nikolaos Tsantalis. 2014. Unification and refactoring of clones. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 104–113.

[13] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.

[14] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. 2016. JDeodorant: clone refactoring. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 613–616.

[15] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[16] J. Ostberg and S. Wagner. 2014. On Automatically Collectable Metrics for Software Maintainability Evaluation. In *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*. 32–37. https://doi.org/10.1109/IWSM.Mensura.2014.19

[17] Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering* (2019), 1–49.

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

[19] Chanchal K Roy and James R Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*. IEEE, 172–181.

[20] Chanchal K Roy and James R Cordy. 2009. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 157–166.

[21] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.

[22] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–365.

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

[24] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2017. CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 654–659.

[25] Abdullah Sheneamer and Jugal Kalita. 2016. A survey of software clone detection techniques. *International Journal of Computer Applications* 137, 10 (2016), 1–21.

[26] N Smith, D van Bruggen, and F Tomassetti. 2017. JavaParser: Visited; Analyse, transform and generate your Java code base.

[27] Jeffrey Svajlenko and Chanchal K Roy. 2014. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 321–330.

[28] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090.

[29] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. 2017. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 60–70.

[30] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. On detection of gapped code clones using gap locations. In *Ninth Asia-Pacific Software Engineering Conference, 2002.* IEEE, 327–336.

[31] Jun Zhao. 2018. *Automatic Refactoring for Renamed Clones in Test Code.* Master's thesis. University of Waterloo.