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 anti-patterns, the codebase should be refactored. Many tools aid in the detection process of such duplication problems, but failed to determine whether a duplicate fragment would improve the maintainability of the codebase when refactored.

We perform an exploratory study into the data that can be gathered from comparing before- and after snapshots of an automatically refactored system. We propose a tool to detect clones, analyze their context 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 system. On the 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.

We have identified that the size of the duplication problem is the biggest influencing factor on system maintainability. We found that the majority of duplicates spanning 29 or more tokens improve maintainability when refactored. Another maintainability influencing factor is the amount of data that needs to be passed to the merged location of the duplicate.

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 Technical Papers (ICSE 2020).* ACM, New York, NY, USA, 8 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 [**?**]. Duplicate fragments create implicit dependencies that make the code harder to maintain as the resolution of erroneous behaviour in one location may have to be applied to all the cloned code as well [**?**]. Apart from that, code clones can contribute up to 25% of the code size [**?**].

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 refactored. We evaluate the context of clones to determine which 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.

We use automated refactoring to compare the maintainability of the before- and after snapshots of a codebase for each duplication problem that is refactored. This gives us insight into whether clones detected by certain thresholds should be refactored to improve the maintainability of the codebase. This allows for more accurate suggestion of code clones for refactoring and can provide assistance in the refactoring process.

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

ICSE 2020, May 23–29, Seoul, North Korea

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

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 omission of certain tokens [? ?]. 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. Such clones can be harder to refactor, if beneficial 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 methods match (or are not used).

Often when allowing such variance, trade-offs 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 is 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.

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

4 CLONEREFACTOR

To detect such clones that can be refactored, we surveyed a set of modern clone detection tools and techniques [? ? ? ?]. We created a set of control projects to determine the suitability of these tools to detect refactorable clones, either through configuration or post-processing of their output. None of the surveyed tools [? ? ? ? ? ? ?] seemed suitable, because of which we decided to implement our own tool.

To automate the process of refactoring clones, we propose a tool named CloneRefactor[1]. This tool goes through a 3-step process 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%:

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

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

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.

4.2 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 contents. We identify categories for each of these properties, to get a detailed insight into 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.
  - **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. 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.
- **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.
- **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*.

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.

We verified the correctness of the resulting refactorings manually. We ran the tool over a large software project and verified over 1.000 applied refactorings.

4.3.3 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. 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. 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. [**?**]. 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 [**?**] 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 number of tokens instead.
- **Complexity**: Heitlager et al. use MCCabe complexity [**?**] 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 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, to find the full paths of all referenced symbols in the source code (see Sec. 3.1). 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[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. 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.

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. 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 cannot improve maintainability when refactored.

5.3 Calculating a maintainability score

In this study, we use four metrics to determine maintainability (see Sec. 4.3.4). For our experiments, we 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:

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

- 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 derived these assumptions partly from Heitlager et al [**?**] and our own expert intuition. The validity of these metrics influences the validity of the conclusions we can draw from our data. Because of that, in our experiments we focus mainly on significant differences in maintainability to draw a conclusion.

By dividing the obtained metric scores by the average distance from zero of all obtained values for a specific metric, we ensure that each metric is weighted equally towards the final maintainability score. We calculate this average distance as follows:

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

Where M is the multiset of all change in a certain metric for all refactorings over all systems in 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. When comparing the maintainability of a set of refactorings we divide the sum of the maintainability score by the amount of clone instances of all refactored clones in the set.

6 RESULTS

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

6.1 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. 4.2).

6.1.1 Relation. Table 1 displays the number of clone classes found for the entire corpus for different relations (see Sec. 4.2.1).

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 1: Number of clone classes per clone relation

6.1.2 Contents. Table 2 displays the number of clone classes found for the entire corpus for different contents (see Sec. 4.2.2).

Category	*Contents*	*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 2: Number of clone instances for clone contents categories

6.2 Extract Method

Table 3 shows to what extent clone classes can be refactored by using the "Extract Method" refactoring technique. The second column shows our measurements for the complete systems (just like the former experiments). The third column shows our measurements when restricting our search to method bodies. The amount that can be extracted increases because, when restricting our search to method bodies, we do not exclude declarations that can obstruct the possibility of method extraction (for instance a cloned method signature).

Category	*All*	*Method Body*
Can Be Extracted	24,157	26,109
Is Not A Partial Method	21,625	0
Top-level AST-Node is not a Statement	19,887	4,607
Spans Part of a Block	12,964	13,460
Multiple Return Values	5,622	6,131
Complex Control Flow	1,106	1,216

Table 3: Number of clones that can be extracted using the "Extract Method" refactoring technique

6.3 Refactoring

In our corpus, CloneRefactor has refactored 12.710 clone classes and measured the change in indicated metrics (see Sec. 4.3.4). Using the presented formulas (see Sec. 5.3) we determine how the characteristics of the extracted method (see Sec. 4.3.3) influence the maintainability of the resulting codebase after refactoring. In this section, we explore the data received by comparing the before- and after snapshots of the system for each separate refactoring.

6.3.1 Clone Token Volume. Figure 2 shows the obtained results when plotting the clone size (in tokens) vs the maintainability increase/decrease. On the secondary axis the amount of refactorings that have refactored a clone with the specified amount of tokens is displayed (e.g. the amount of data points the data is based on). As the amount of data points decreases, the datapoints gain less statistical significance.

The volume of the clone is the dominating factor regarding the maintainability increase/decrease of cloned code. Because of that, for our further experiments, we filter out all refactorings with a token size smaller than 18 because otherwise the small clones dominate the results and turn all results towards unmaintainable.

6.3.2 Relation. Table 4 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.

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

Return Value	*Maintainability Score*	*Number of Refactorings*
Return	0.20	421
Void	0.19	2052
Declare	0.03	1318
Assign	-1.29	3

Table 5: Maintainability scores for different return values

6.3.4 Parameters. Fig. 3 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.

Relation	*Maintainability Score*	*Number of Refactorings*
Common Hierarchy	**0.51**	**792**
Sibling	0.57	637
Same Hierarchy	0.55	22
Superclass	0.20	74
First Cousin	-0.02	53
Ancestor	-0.65	6
Common Class	**0.01**	**2,025**
Same Method	0.01	762
Same Class	0.01	1,263
Unrelated	**-0.02**	**688**
No Direct Superclass	0.08	289
External Superclass	-0.02	225
No Indirect Superclass	-0.04	30
External Ancestor	-0.26	144
Common Interface	**-0.11**	**283**
Same Direct Interface	0.02	160
Same Indirect Interface	-0.31	123

Table 4: Influence on maintainability of refactoring clones by certain relations.

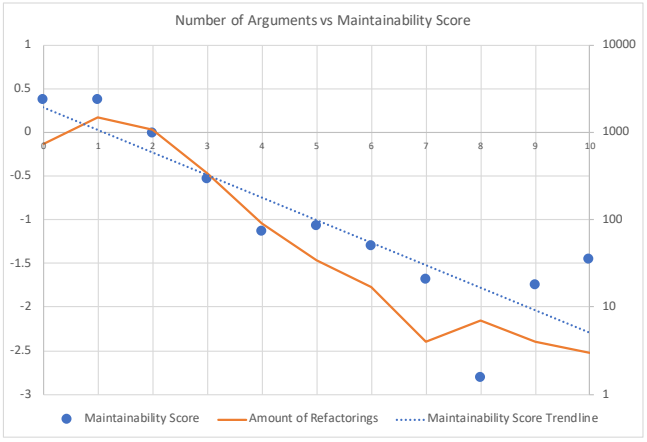

Figure 3: Influence of number of method parameters on system maintainability.

7 DISCUSSION

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

7.1 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.

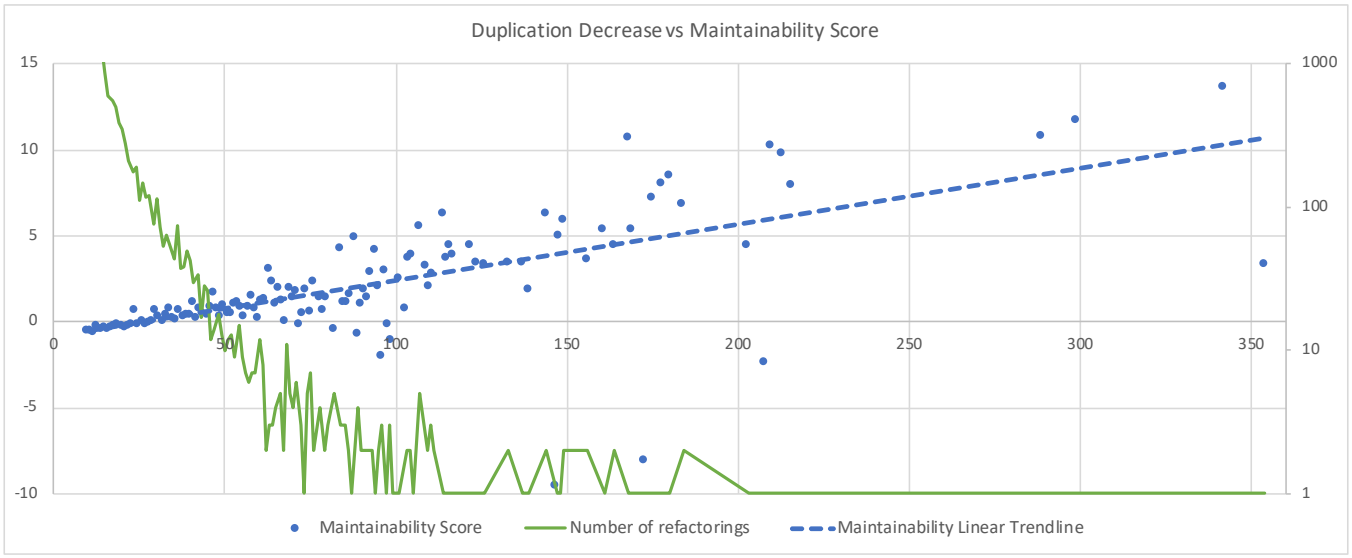

Figure 2: A graph that shows how the size in tokens of the refactored clone affects maintainability. Maintainability on the primary axis and amount of refactorings on the secondary axis.

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 contents, 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.).

7.2 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 clones in anonymous functions or anonymous classes. About 15% of clones span only part of an AST-Node.

7.3 Refactoring

In Fig. 2 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 4 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 1. Meanwhile, the number of refactored unrelated clones is smaller than the number reported in Table 1 (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 4 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; there is no indication that relations have a major 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. 3 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 trend of the graph in Fig. 3 decreases relatively rapidly.

8 CONCLUSION

We defined 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 contents. Regarding relations, we found that most clones are found in the same class, over 37%. About 24% of clones are in the same inheritance hierarchy. Another 24% of clones are unrelated. The final 7% of clones have the same interface. Regarding contents, 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 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, for instance 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.