Towards Automated Merging of Code Clones in Object-Oriented Programming Languages

Simon Baars
simon.mailadres@gmail.com

30 June 2019, 23 pages

Research supervisor: Dr. Ana Oprescu, ana.oprescu@uva.nl
Host/Daily supervisor: Xander Schrijen, x.schrijen@sig.eu
Host organisation/Research group: Software Improvement Group (SIG), http://sig.eu/

University of Amsterdam
Faculty of Physics, Mathematics and Informatics
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

This should be done when most of the rest of the document is finished. Be concise, introduce context, problem, known approaches, your solution, your findings.

Duplication in source code can have a major negative impact on the maintainability of source code. There are several techniques that can be used in order to merge clones, reduce duplication, improve the design of the code and potentially also reduce the total volume of a software system. In this study, we look into the opportunities to aid in the process of refactoring these duplication problems for object-oriented programming languages.

We first look into redefinitions for different types of clones that have been used in code duplication research for many years. These redefinitions are aimed towards flagging only clones that are useful for refactoring purposes. Our definition defines additional rules for type 1 clones to make sure two cloned fragments are actually equal. We also redefined type 2 clones to reduce the number of false positives resulting from it.

We have conducted measurements that have indicated that more than half of the duplication in code is related to each other through inheritance, making it easier to refactor these clones in a clean way. Approximately a fifth of the duplication can be refactored through method extraction, the other clones require other techniques to be applied.

Contents

Chapter 1

Introduction

Context: what is the bigger scope of the problem you are trying to solve? Try to connect to societal/economical challenges. Problem Analysis: Here you present your analysis of the problem situation that your research will address. How does this problem manifest itself at your host organisation? Also summarises existing scientific insight into the problem.

Refactoring is used to improve quality related attributes of a codebase (maintainability, performance, etc.) without changing the functionality. There are many methods that have been introduced to help with the process of refactoring [**fowler2018refactoring**, **wake2004refactoring**]. However, most of these methods still require manual assessment of where and when to apply them. Because of this, refactoring takes up a signification portion of the development process [**lientz1978characteristics**, **mens2004survey**], or does not happen at all [**mens2003refactoring**]. For a large part, refactoring requires domain knowledge to do it right. However, there are also refactoring opportunities that are rather trivial and repetitive to execute. In this thesis, we take a look at the challenges and opportunities in automatically refactoring duplicated code, also known as "code clones". The main goal is to improve maintainability of the refactored code.

Duplication in source code is often seen as one of the most harmful types of technical debt. In Martin Fowler's "Refactoring" book [**fowler2018refactoring**], he exclaims that *"Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them."*.

In this research, we focus on formalizing the refactoring process of dealing with duplication in code. We will measure open source projects from the We will show the improvement of the metrics over various open source and industrial projects. Likewise, we will perform an estimation of the development costs that are saved by using the proposed solution. We will lay the main focus on the Java programming language as refactoring opportunities do feature paradigm and programming language dependent aspects [**choi2011extracting**]. However, most practises used in this thesis will also be applicable with other object-oriented languages, like C#.

1.1 Problem statement

1.1.1 Research questions

Code clones can appear anywhere in the code. Whether a code clone has to be refactored, and how it has to be refactored, is dependent on where it exists in the code (it's context). There are many different contexts in which code clones can occur (in a method, a complete class, in an enumeration, global variables, etc.). Because of this, we first must collect some information regarding in what contexts code clones exist. To do this, we will analyze a set of Java projects for their clones, and generalize their contexts. To come to this information, we have formulated the following research question:

Research Question 1:
How can we group and rank clones based on their harmfulness?

As a result from this research question, we expect a catalog of the different contexts in which clones occur, ordered on the amount of times they occur. On basis of this catalog, we have prioritized the further analysis of the clones. This analysis is to determine a suitable refactoring for the clone type that has been found at the design level. For this, we have formulated the following research question:

Research Question 2:
To what extend can we suggest refactorings of clones at the design level?

As a result, we expect to have proposed refactorings for the most harmful clone patterns. On basis of these design level refactorings we will build a model, which we will proof using Java, that applies the refactorings to corresponding methods. For building this model, we have formulated the following research question:

Research Question 3:
To what extend can we automatically refactor clones?

As a result from this research question, we expect to have a model to be able to refactor the highest priority clones.

1.1.2 Research method

1.2 Contributions

Our research makes the following contributions:

1. We deliver several novel measurements regarding code clones on a large corpus of Java projects.
2. We deliver a novel clone detection tool that finds refactorable clones in Java.
3. We deliver a novel clone refactoring tool that suggests refactorings to be applied, and applies these refactorings.
4. We give further recommendations in how refactoring can be automatically applied to improve maintainability in software projects.

1.3 Scope

In this research we will look into code clones from a refactoring viewpoint. There are several methods that detect code clones using a similarity score to match pieces of code. This similarity is often based on the amount of tokens that match between two pieces of code. The problem with similarity based clones is that it is hard to assess the impact of merging clones that have different tokens, but what exactly this token is is unknown. Because of this, we will not focus on similarity based clone detection techniques, but rather on exact matches and predefined differences.

It is very disputable whether unit tests apply to the same maintainability metrics that applies to the functional code. Because of that, for this research, unit tests are not taken into scope. The findings of this research may be applicable to those classes, but we will not argue the validity.

1.4 Outline

In Chapter 2 we describe the background of this thesis. Chapter **??** describes ... Results are shown in Chapter 7 and discussed in Chapter 8. Chapter 9, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 10 together with future work.

Chapter 2

Background

This chapter will present the necessary background information for this thesis. Here, we define some basic terminology that will be used throughout this thesis.

2.1 Clone Class

As code clones are seen as one of the most harmful types of technical debt, they have been studied very extensively. A survey by Roy et al [**roy2007survey**] states definitions for various important concepts in code clone research. In this survey, he mentions the concept "clone pair", which is *a set of two code portions/fragments which are identical or similar to each other*. Furthermore, he defined "clone class" as *the union of all clone pairs*. Apart from this, we use the definition "clone instance", which is a single code portion/fragment that is part of either a clone pair or clone class.

Figure 2.1 displays an example of a clone pair or clone class. In this case, both cloned fragments, are found in the same class. Each of the cloned fragments can be defined as a "clone instance".

Figure 2.1: Example of a clone pair, as found in the ardublock project.

Figure 2.2 displays two clone classes. These clone classes are separated by a single line that is different. The first clone class spans over both the constructor and a method of this class.

Figure 2.2: Example of two clone classes.

2.2 Clone Types

Duplication in code is found in many different forms. Most often duplicated code is the result of a programmer reusing previously written code [**haefliger2008code**, **baxter1998clone**]. Sometimes this code is then adapted to fit the new context. To reason about these modifications, several clone types have been proposed. These clone types are described in Roy et al [**roy2007survey**]:

> **Type I:** Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.
> **Type II:** Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
> **Type III:** Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.
> **Type IV:** Two or more code fragments that perform the same computation but implemented through different syntactic variants.

A higher type of clone means that it's harder to detect and refactor. There are many studies that adopt these clone types, analyzing them further and writing detection techniques for them [**sajnani2016sourcerercc**, **kodhai2010detection**, **van2019novel**].

2.2.1 Type 2 clones

Relating this to the clone class example of 2.2, the complete class would be flagged as a type 2 clone. This is displayed in figure 2.3. Considering such clones displays the opportunity to refactor the class as a whole.

Figure 2.3: The clone displayed in figure 2.2 as a type 2 clone.

2.2.2 Type 4 clones

For this thesis we have chosen not to consider type 4 clones for refactoring, because they are both hard to detect and hard to refactor (how to choose the best alternative for a certain computation could be a thesis in itself). A study by Kodhai et al [**kodhai2013method**] looks into the distribution of the

different types of clones in several open source systems (see table 6 of his study). It becomes apparent that type 4 clones exist way less in source code than all of the other types of clones. For instance, for the J2sdk-swing system he finds 8115 type 1 clones, 8205 type 2 clones, 11209 type 3 clones and only 30 type 4 clones. Because of that, we can conclude that type 4 clones are relatively less relevant to study.

2.3 Clone Contexts

Code clones can be found anywhere in the code. The most commonly studied type of clone is the method-level clone. Method-level clones are duplicated blocks of code in the body of a method. Many clone detection tools only focus on method-level clones (like CPD[1], Siamese[2], Sysiphus[3]). The reason for this is that with method-level clones it's most likely that the clones are harmful, and they are more straight-forward to refactor.

A paper by Lozano et al [**lozano2007evaluating**] discusses the harmfulness of cloning. In this paper the author argues that 98% are produced at method-level. However, the paper that is cited to support this claim [**bergman2004ethnographic**] does not conclude this same information. First of all, the study that is referenced uses a very small dataset (460 copy & paste instances by 11 participants). Secondly, the group of subject only consists of IBM researchers (selection bias). Thirdly, it only focuses on copy and paste instances, as opposed to other ways clones can creep into the code. Finally, the "98%" is not stated explicitly, but is vaguely derivable from one of the figures (figure 1) in this paper. Because of this, there is no reliable overview of how many clones there are in different contexts.

This thesis will focus on measuring how many clones there are per context. This way we can determine the impact of focusing our search on a specific context, like the analysis of only method-level clones. Our hypothesis is that the 98% claim is not true (we think this should be far less). We also hypothesize that clones in different contexts than method-level are less likely to be harmful and less straight forward to refactor.

2.3.1 Clone refactoring in relationship to its context

How to refactor clones is highly dependent on their context. Method-level clones can be extracted to a method [**kodhai2013method**] if all occurrences of the clone reside in the same class. If a method level clone is duplicated among classes in the same inheritance structure, we might need to pull-up a method in the inheritance structure. If instances of a method level clone are not in the same inheritance structure, we might need to either make a static method or create an inheritance structure ourselves. So not only a single instance of a clone has a context, but also the relationship between individual instances in a clone class. This is highly relevant to the way in which the clone has to be refactored.

2.4 Code clone harmfulness

There has been a lot of discussion whether code clones should be considered harmful.

Most papers view clones as harmful regarding program maintainability. *"Clones are problematic for the maintainability of a program, because if the clone is altered at one location to correct an erroneous behaviour, you cannot be sure that this correction is applied to all the cloned code as well. Additionally, the code base size increases unnecessarily and so increases the amount of code to be handled when conducting maintenance work."* [**ostberg2014automatically**]

However, the harmfulness of clones depends on a lot of factors. A paper by Kapser et al [**kapser2006cloning**] describes several patterns of cloning that may not be considered harmful. In this paper Kapser names examples where eliminating clones would compromise other important program qualities. Another study by Jarzabek et al [**jarzabek2010clones**] categorized "Essential clones": clones that are essential because of the solution that is being modelled by the program. Overall, many of the benefits of code clones do not apply to most modern object-oriented programming languages.

[1] CPD is part of PMD, a commonly used source code analyzer: `https://github.com/pmd/pmd`

[2] Siamese is an Elasticsearch based clone detector: `https://github.com/UCL-CREST/Siamese`

[3] Sisyphus crawls the Java library for existing implementations of parts of a codebase: `https://github.com/fruffy/Sisyphus`

2.5 Related work

There have been some papers that take some steps towards code clone refactoring. Most research towards refactoring code clones has been conducted by Y. Higo et al. In a 2008 study [**higo2008metric**] the authors look at the refactoring of class-level, method-level and constructor-level clones in Java.

Chapter 3

Redefining type 1, 2 and type 3 clones

In the background section we discussed the different types of clones, as defined by Roy et al [**roy2007survey**]. However, when detecting clones by these definitions, we will detect many clones that are not immediately useful for refactoring. In this chapter, we will discuss the shortcoming of each of the clone types, and propose solutions for them.

3.1 Type 1 clones

Type 1 clones are identical clone fragments except for variations in whitespace and comments. However, when two clone fragments are textually identical, does not yet indicate that they are actually identical. Even when textually equal, method calls can refer to different methods, type declarations can refer to different types and even variables can be of a different type.

This could invalidate a refactoring opportunity for such a code fragment. Because of that, we recommend redefining type-1 clones for refactoring. We propose to:

- **Compare the equality of the fully qualified method signature for method references.** This way we can validate whether two method references, like method calls, are actually equal. In the method signature, not only the fully qualified identifier of the method should be considered, but also of the type of all its arguments. This way we can be sure that two potentially cloned method references do not point to overloaded variants (in a case that the data type of arguments is overloaded).
- **Compare the equality of the fully qualified identifier for type references.** This way we can be sure that two referenced types are actually equal, and that they are not just two types with the same name.
- **Compare the equality the fully qualified identifier for variable usages.** Two cloned lines might use a variable with the same name, but different types. This might pose serious challenges on refactoring, as the variables might not concern the same object or primitive. To check this, we need to track the declaration of variables and from this infer the fully qualified identifier of its type.

3.2 Type 2 clones

Type 2 clones, by definition, allow any change in identifiers, literals, types, layout, and comments. For refactoring purposes, this doesn't always make a lot of sense. If we allow any change in identifiers, literals, and types, we cannot distinguish between different variables, different types and different method calls anymore. This could render two methods that have an entirely different functionality as clones. Merging such clones, if possible at all, might only prove to be harmful.

Because of that, for this research, we have looked into a redefinition of type 2 clones to be able to detect such clones that can and should be merged. Our definition ensures functional similarity by applying the following changes to type 2 clones:

- **Considering types:** The definition of type 2 clones states that types should not be considered. We disagree because types can make a significant change to the meaning of a code segment and

thus whether this segment should be considered a clone.

- **Having a distinction between different variables:** By the definition of type 2 clones, any identifiers would not be taken into account. We agree that a difference in identifiers may still result in a harmful clone, but we should still consider the distinction between different variables. For instance, if we call a method like this: *"myMethod(var1, var2)"*, or call this method like this: *"myMethod(var1, var1)"*. Even if the variables have the same type, the distinction between the variables is important to ensure the functionality is the same after merging.
- **Defining a threshold for variability in literals:** By the definition of type 2 clones any literals would not be taken into account. We agree, as when merging the clone (for instance by extracting a method), we can simply turn the literal into a method parameter. However, we would argue that thresholds matter here. How many literals may differ for the segment still to be considered a clone with another segment? We need to define a threshold to be sure that, by merging, we are not replacing a code fragment by a worse maintainable design.
- **Consider method call signatures and define a threshold for variability in method calls:** As type-2 clones allow changes in identifiers, also the names of called methods may vary. However, because of this, completely different methods can be called in cloned fragments as a result. This poses serious challenges on refactoring and makes it more disputable whether such a clone is actually harmful. This is because different method identifiers can describe a completely different functionality. Because of that, we recommend to keep considering the call signatures of cloned methods when they are compared. We can allow variability in the rest of method identifiers by passing the function as a parameter. To limit the amount of parameters required we also recommend defining a threshold for variability in method call expressions, so only a limited number of method calls can vary.

3.3 Type 3 clones

Type 3 clones are even more permissive than type 2 clones, allowing added and removed statements. Thresholds matter a lot here to make sure that not the whole project is detected to be cloned. The main question for this study regarding type 3 clones is: "how can we merge type 3 clones while improving the design?".

Clone instances in type 3 clones are almost always different in functionality. As we have to ensure equal functionality after merging the clone, we have to wrap the difference in statements between the clone instances in conditional blocks (either if-statements or switch-statements). We can then pass a variable as to which path should be taken through the code (either a boolean or an enumeration). Such a refactoring would make added statements that are contiguous less harmful for the design then added statements than added statements that are separated by statements that both clone instances have in common.

We also want to argue that statements that are not common between two clone instances, should not count towards the size of the clone (and thus towards the threshold which determines whether the clone will be taken into account). Also, clones should not start and not end with an added statement (as that would be nonsense: such a thing could be done for any clone).

As for the detection of type 3 clones, we think the easiest opportunity to detect these clones is to consider it as a postprocessing step after clone detection. By trying to find short gaps between clones, we can find opportunities to merge clone classes into a single type 3 clone class. The amount of statements that this "short gap" can maximally span should be dependent on a threshold value.

Chapter 4

Clone Detection

As duplication in source code is a serious problem in many software systems, there are a lot of researches that look into code clones. Many tools have been proposed to detect various types of code clones. Two surveys of modern clone detection tools [**sheneamer2016survey**, **svajlenko2014evaluating**] together show an overview of the most-popular clone detection tools up until 2016. We have considered a set of these tools. However, most of these tools have a vastly different purpose than this thesis. Most of the tools are focussed on efficiency and detecting all of the types of clones. Efficiency is lesser of a concern to us, correctness is the most important aspect.

To be able to refactor detected clones, it is useful to have the ability to rewrite the AST. We considered a set of clone detection tools for their ability to support the refactoring process automatically. None of the tools we consider seemed completely fit for this purpose, so we decided to implement our own clone detection tool: CloneRefactor[1].

4.1 CloneRefactor

A 2016 survey by Gautam [**gautam2016various**] focuses more on various techniques for clone detection. For our tool, we decided to combine AST- and Graph-based approaches for clone detection, similar to Scorpio (which is a clone detection tool that's part of TinyPDG: a library for building intraprocedural program dependency graphs for Java programs). We decided to base our tool on the JavaParser library [**tomassetti2017javaparser**], as it supports rewriting the AST back to Java code and is useable with all modern Java versions (Java 1-12). We collect each statement and declaration and compare those to find duplicates. This way we build a graph of each statement/declaration linking to each subsequent statement/declaration (horizontally) and linking to each of its duplicates (vertically). This is displayed in figure 4.1.

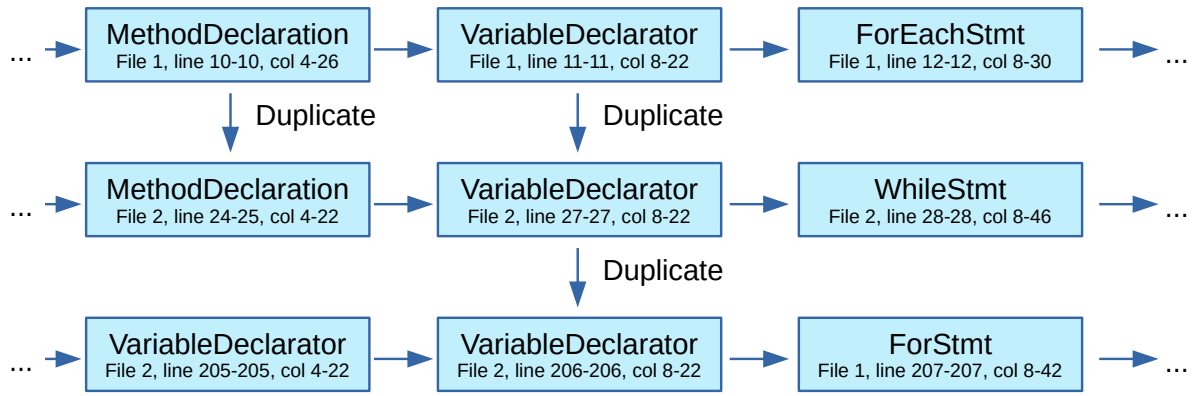

Figure 4.1: Abstract figure of the graph representation built by CloneRefactor

[1] CloneRefactor (WIP) is available on GitHub: `https://github.com/SimonBaars/CloneRefactor`

4.2 Thresholds

CloneRefactor works on basis of three thresholds for finding clones:

1. **Number of statements/declarations:** The number of statements/declarations that should be equal/similar for it to be considered a clone.
2. **Number of tokens:** The number of tokens (excluding whitespace, end-of-line terminators and comments) that should be equal/similar for it to be considered a clone.
3. **Number of lines:** The minimum amount of lines (excluding lines that do not contain any tokens, for tokens same exclusions apply) that should be equal/similar for it to be considered a clone.

Of course, if we put any of these thresholds to zero it won't be taken into account anymore, so not all thresholds have to be used at all times. We consider "number of lines" to be the least important, as it highly depends on the programmer of the codebase (and we do not want this kind of dependence!). On basis of manual assessment, we have determined that setting the "number of statements/declarations" to 6 ensures that most non-harmful clones are filtered out. On the downside, this also filters out some harmful antipatterns, for instance, if a cloned line has many tokens we might want to consider a clone if it spans less than 6 statements (as a cloned line with many tokens is more harmful than one with few).

For the measurements in the next chapter we have chosen to apply the following thresholds:

- **Number of statements/declarations:** 6
- **Number of tokens:** 10
- **Number of lines:** 6

Chapter 5

Code Clone Context

To be able to refactor code clones, it is very important to consider the context of the clone. We define the following aspects of the clone as its context:

1. The relation of clone instances among each other (for example two clone instances in a clone class are part of the same object).
2. Where a clone instance occurs in the code (for instance: a method-level clone is a clone instance that is (a part of) a single method).
3. The contents of a clone instance (for instance: the clone instance consists of a one method declaration, a foreach statement, and two variable declarations).

Everything in the context of a clone has a big impact on how it has to be refactored. For this study, we performed measurements on the context of clones in a large corpus of open source projects.

5.1 The corpus

For our measurements we use a large corpus of open source projects [**githubCorpus2013**][1]. This corpus has been assembled to contain relatively higher quality projects (by filtering by forks). Also, any duplicate projects were removed from this corpus. This results in a variety of Java projects that reflect the quality of average open source Java systems and are useful to perform measurements on.

We then filtered the corpus further to make sure we are not including any test classes or generated classes. Many Java/Maven projects use a structure where they separate the application and its tests in the different folders ("/src/main/java" and "/src/test/java" respectively). Because of this, we chose to only use projects from the corpus which use this structure (and had at least a "/src/main/java" folder). To limit the execution time of the script, we also decided to limit the maximum amount of source files in a single project to 1.000 (projects with more source files were not considered, which filtered only 5 extra projects out of the corpus). Of the 14.436 projects in the corpus over 3.848 remained, which is plenty for our purposes. The script to filter the corpus in included in our GitHub repository [2].

Running our clone detection script, CloneRefactor, over this corpus gives the results displayed in table 5.1.

[1] The corpus can be downloaded from the following URL: `http://groups.inf.ed.ac.uk/cup/javaGithub/java_projects.tar.gz`

[2] The script we use to filter the corpus: `https://github.com/SimonBaars/CloneRefactor/blob/MeasurementsVersion1/src/main/java/com/simonbaars/clonerefactor/scripts/PrepareProjectsFolder.java`

Table 5.1: CloneRefactor results for Java projects corpus [githubCorpus2013].

Amount of projects	3,848
Amount of lines	8,284,140
Amount of lines (excluding whitespace, comments, etc.)	8,163,429
Amount of statements/declarations	6,863,725
Amount of tokens	66,964,270
Amount of lines cloned	1,341,094
Amount of lines cloned (excluding whitespace, comments, etc.)	815,799
Amount of statements/declarations cloned	747,993
Amount of tokens cloned	9,800,819
Amount of clone classes	34,367

5.2 Relations Between Clone Instances

When merging code clones in object-oriented languages, it is very important to consider the relation between clone instances. This relation has a big impact on how a clone should be merged, in order to improve the software design in the process.

5.2.1 Categorizing Clone Instance Relations

A paper by Fontana et al [**fontana2015duplicated**] performs measurements on 50 open source projects on the relation of clone instances to each other. To do this, they first define several categories for the relation between clone instances in object-oriented languages. These categories are as follows:

1. **Same method**: All instances of the clone class are in the same method.
2. **Same class**: All instances of the clone class are in the same class.
3. **Superclass**: All instances of the clone class are children and parents of each other.
4. **Ancestor class**: All instances of the clone class are superclasses except for the direct superclass.
5. **Sibling class**: All instances of the clone class have the same parent class.
6. **First cousin class**: All instances of the clone class have the same grandparent class.
7. **Common hierarchy class**: All instances of the clone class belong to the same hierarchy, but do not belong to any of the other categories.
8. **Same external superclass**: All instances of the clone class have the same superclass, but this superclass is not included in the project but part of a library.
9. **Unrelated class**: There is at least one instance in the clone class that is not in the same hierarchy.

Please note that no of these categories allow external classes (except for "same external superclass"). So if two clone instances are related through external classes but do not share a common external superclass, it will be flagged as "unrelated". The main reason for this is that it is (often) not possible to refactor to external classes.

The ranking of the previous list of categories also matters as it shows the different levels in which clones were assessed. For instance, if two clone instances of a clone class belong to the "same method" category but the third belongs to the "same class", we will always choose the item lowest on the list.

5.2.2 Our measurements

We have recreated table 3 of Fontana et al [**fontana2015duplicated**], but with the following differences in our setup:

- We consider clone classes rather than clone pairs.
- We use different thresholds regarding when a clone should be considered.

- We seek by statement/declaration rather than SLOC.
- We test a broader range of projects (they use a set of 50 relatively large projects, we use a large corpus that was assembled by a machine learning algorithm testing java projects on GitHub for quality, which contains projects of all sizes and with differing code quality).

Table 5.2 contains our results regarding the relations between clone instances.

Table 5.2: Clone relations

Relation	Amount	Percentage
Unrelated	13,529	39.37
Same Class	8,341	24.28
Sibling	5,978	17.40
Same Method	2,456	7.15
External Superclass	2,402	6.99
First Cousin	695	2.02
Superclass	489	1.42
Common Hierarchy	442	1.29
Ancestor	28	0.08

Comparing it to the results of Fontana et al [**fontana2015duplicated**], we find way more unrelated clones. This might be due to the fact that we consider clone classes rather than clone pairs, and mark the clone class "Unrelated" even if just one of the clone instances is outside a hierarchy. It could also be that the corpus which we use, as it has generally smaller projects, use more classes from outside the project (which are marked UNRELATED if they do not have a common external superclass). On the second place, we have the "Same Class" clones. On the third place come the "Sibling" clones.

5.3 Clone instance location

After mapping the relations between individual clones, we looked at the location of individual clone instances. A paper by Lozano et al [**lozano2007evaluating**] discusses the harmfulness of cloning. In this paper, the author argues that 98% are produced at method-level. However, this claim is based on a small dataset and based on human copy-paste behavior rather than static code analysis. We validated this claim over the corpus we use. For this, we chose the following categories:

1. **Method/Constructor Level:** A clone instance that does not exceed the boundaries of a single method or constructor (optionally including the declaration of the method or constructor itself).
2. **Class Level:** A clone instance in a class, that exceeds the boundaries of a single method or contains something else in the class (like field declarations, other methods, etc.).
3. **Interface Level:** A clone that is (a part of) an interface.
4. **Enumeration Level:** A clone that is (a part of) an enumeration.

The results for the clone instance locations are shown in table 5.3.

Table 5.3: Clone instance locations

Location	Amount	Percentage
Method Level	19,818	57.68
Class Level	13,259	38.59
Constructor Level	1,099	3.20
Interface Level	110	0.32
Enum Level	74	0.22

From these results, we can see that the claim of 98% of clones being produced at method-level is nowhere near correct. In fact, around 58% of the clones are produced at method-level. About 39% of clones either span several methods/constructors or contain something like a field declaration. Another 3% of the clones are found in constructors. The amount of clones found in interfaces and enumerations is very low.

5.4 Clone instance contents

Finally, we looked at the contents of individual clone instances: what kind of declarations and statements do they span. We selected the following categories to be relevant for refactoring:

1. **Full Method/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 method partially, optionally including its declaration.
3. **Several Methods:** A clone that spans over two or more methods, either fully or partially, but does not span anything but methods (so not fields or anything in between).
4. **Only Fields:** A clone that spans only global variables.
5. **Includes Fields/Constructor:** A clone that spans a combination of fields and other things, like methods.
6. **Method/Class/Interface/Enumeration Declaration:** A clone that contains the declaration (usually the first line) of a class, method, interface or enumeration.
7. **Other:** Anything that does not match with above-stated categories.

The results for these categories are displayed in table 5.4.

Table 5.4: Clone instance contents

Contents	Amount	Percentage
Partial Method	19,264	56.07
Several Methods	9,076	26.41
Includes Constructor	1,528	4.45
Includes Field	1,149	3.34
Partial Constructor	1,098	3.20
Only Fields	565	1.64
Full Method	554	1.61
Includes Class Declaration	445	1.30
Other	369	1.07
Full Class	192	0.56
Includes Enum Constant	52	0.15
Includes Enum Declaration	47	0.14
Includes Interface Declaration	10	0.03
Full Interface	6	0.02
Full Enumeration	4	0.01

Unsurprisingly, most clones span a part of a method. Just 1.6% of the methods are cloned fully. More than a quarter of the clones spans over several methods, which is surprising. Simply extracting methods won't work in a case where a clone spans over several methods. About 4.5% of clones include a constructor. About 3.3% of clones include a global variable defined in a class.

Chapter 6

Merging duplicate code through refactoring

Now we have mapped the contexts in which clones occur, we can start looking at refactoring opportunities. Regarding refactoring, we separate clones in two categories: easy and difficult refactoring opportunities. Easy refactoring opportunities are clones that can easily be automatically refactored. Examples of these opportunities are fully cloned methods or a set of fully cloned statements. According to the relation between the clone instances, we can propose a refactoring automatically.

However, not always can a clone easily be merged. Sometimes a clone spans a statement partially (like a for-loop of which only it's declaration and a part of the body is cloned). Merging the clones can be harder in such instances. Also, the cloned code can contain a complex control structure, like labels, return, break, continue, etc. In such instances, more conditions apply to be able to conduct a refactoring, if advisable at all.

The most trivial way to merge a clone is through method extraction. However, method extraction is not always possible. For instance, if a part of a subtree (in a programs' AST) is matched as a clone. Because of this, we chose to measure what percentage of "Partial Method" clones are refactorable using method extraction. Our results are displayed in table 6.1.

Table 6.1: Clone instance contents

Refactorability	**Amount**	**Percentage**
Cannot directly be extracted	10,990	41.91
Is not a partial method	9,444	36.01
Can be extracted	5,791	22.08

From this table, we can see that approximately 22 percent of the clones can be refactored through method extraction. For the other clones, we must first build a catalog of appropriate refactorings.

Chapter 7

Results

In this chapter, we present the results of our experiment.

Chapter 8

Discussion

In this chapter, we discuss the results of our experiment(s) on ...

Finding 1: Highlight like this an important finding of your analysis of the results.

Refer to Finding 1.

Chapter 9

Related work

We divide the related work into ... categories: ...

Chapter 10

Conclusion

10.1 Future work

Acknowledgements

Thanks to you, for reading this :)

Appendix A

Non-crucial information