Towards Automated Refactoring of Code Clones in Object-Oriented Programming Languages - Work in Progress -

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

Ana Oprescu
University of Amsterdam
Amsterdam, Netherlands
AM.Oprescu@uva.nl

University of Amsterdam

Abstract

Duplication in source code can have a major negative impact on the maintainability of source code, as it creates implicit dependencies between fragments of code. Such implicit dependencies often cause bugs. In this study, we look into the opportunities to automatically refactor these duplication problems for object-oriented programming languages. We propose a method to detect clones that are suitable for refactoring. This method focuses on the context and scope of clones, ensuring our refactoring improves the design and does not create side effects.

Our intermediate results indicate 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 ten percent of the duplication can be refactored through method extraction without extra considerations required, while other clones require other refactoring techniques or further transformations. Similar future measurements will provide further insight into the contexts where clones occur and how this affects the automated refactoring process. Finally, we strive to construct a tool that automatically applies refactorings for a large part of the detected duplication problems.

Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2019 (sattose.org). 08-10 July 2019, Bolzano, Italy.

1 Introduction

Duplication in source code is often seen as one of the most harmful types of technical debt. In Martin Fowler's "Refactoring" book [Fow99], he claims 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."*.

Refactoring is used to improve the quality-related attributes of a codebase (maintainability, performance, etc.) without changing the functionality. Many methods were introduced to aid the process of refactoring [Fow99, Wak04], and are integrated into most modern IDE's. However, most of these methods still require a manual assessment of where and when to apply them. This means refactoring is either a significant part of the development process [LST78, MT04], or does not happen at all [MVD+03]. For a large part, proper refactoring requires domain knowledge. However, there are also refactoring opportunities that are rather trivial and repetitive to execute. Our goal is investigating to what extend code clones can be automatically refactored.

A survey by Roy et al. [RC07] describes various ways in which clones can be identified. Most clone detection tools focus on finding clones that align with these definitions. In this paper, we outline challenges with these clone type definitions when considered in a refactoring context. We next propose solutions to these problems that would enable the detection of clones that should be refactored, rather than fragments of code that are just similar.

Code clones can be found anywhere in a codebase. The location of a clone in the code has an impact on

how it can be refactored. Therefore, we first look at where and how often clones can be found, enabling the prioritization of refactoring opportunities.

Furthermore, a duplicate fragment in a codebase does not always have to be an exact match with another fragment to be considered a clone. Therefore, we also analyze the impact of the different definitions of clone types on refactoring opportunities.

We conduct our research on a large corpus of open source projects. We focus mainly on the Java programming language as refactoring opportunities feature paradigm and programming language dependent aspects [CYI+11]. However, most practices featured currently in our work will also be applicable to other object-oriented languages, like C#. This is because these programming languages share many similarities regarding refactoring opportunities.

Our end goal is to improve upon the current state-of-the-art in clone research [FZ15, Alw17] by building a clone refactoring tool that automatically applies refactorings to a large percentage of clones found. The design decisions for this tool are made on basis of data gathered from a large corpus of software systems together with our own experience and findings from literature.

Sec. 2 describes the background from literature that we used to conduct this study. Sec. 5 presents our clone refactoring tool proposal. Sec. 3 outlines challenges with clone type definitions and proposes solutions for them in a refactoring context. Sec. 6.4 outlines our analysis and measurements regarding the context of code clones. Sec. 6.8 shows our measurements regarding the amount of clones that can be refactored through method extraction. Sec. 7 shows the threats to validity of this study and Sec. 8 draws a conclusion based on our preliminary results.

1.1 Research questions

We have formalized the following research questions in order to improve upon the state-of-the-art in code duplication refactoring:

RQ1. How can we define clone types such that they **can** be automatically refactored?
RQ2. What are the discriminating factors to decide when a clone **should** be refactored?
RQ3. To what extent can we **automate** the process of refactoring clones?

For **RQ1** we look into current clone definitions and clone detection methods. We then assess their suitability for refactoring purposes. For **RQ2** we look into to what extent clones can be filtered based on their impact on the system design when refactored. We do this by both determining good thresholds for the clone detection and by assessing the design implications of applying refactorings to the code. **RQ3** is currently work in progress.

2 Background

As code clones are seen as one of the most harmful types of technical debt, they have been studied extensively. A survey by Roy et al. [RC07] states definitions of important concepts in code clone research. For instance, "clone pair"is defined as *a set of two code portions/fragments which are identical or similar to each other*; "clone class" as *the union of all clone pairs*; "clone instance" as a single code portion/fragment that is part of either a clone pair or clone class.

2.1 Advantages of clone classes over clone pairs

Regarding clone detection, there is a lot of variability in literature whether clone pairs or clone classes should be considered for detection. We focus on clone classes, because of the advantages for refactoring. Clone pairs do not provide a general overview of all entities containing the clones, with all their related issues and characteristics [FZZ12]. Although clone classes are harder to manage, they provide all information needed to plan a suitable refactoring strategy, since this way all instances of a clone are considered. Another issue that results from grouping clones by pairs: clone reference amount increases according to the binomial coefficient formula (two clones form a pair, three clones form three pairs, four clones form six pairs, and so on), which causes a heavy information redundancy [FZZ12].

2.2 Clone types

In a 2007 survey by Roy et al. [RC07] he defines several types of clones:

Type 1: Identical code fragments except for variations in whitespace (may also be variations in layout) and comments.
Type 2: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
Type 3: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout, and comments.

A higher type of clone means that it is harder to detect. It also makes the clone harder to refactor, as more transformations would be required. Higher clone

types also become more disputable whether they actually indicate a harmful anti-pattern (as not every clone is harmful [JX10, KG08]).

There also exists a type 4 clone, denoting functionally equal code. We decided not to consider these clones in this study, because of the serious challenges in their detection and refactoring.

2.3 Related work in clone refactoring tools

The Duplicated Code Refactoring Advisor (DCRA) looks into refactoring opportunities for clone pairs [FZZ12, FZ15]. DCRA only focuses on refactoring clone pairs, with the authors arguing that *clone pairs are much easier to manage when considered singularly.* As intermediate steps, the authors measure a corpus of Java systems for some clone-related properties of the systems, like the relation (in terms of inheritance) between code fragments in a clone pair. We further look into these measurements in Sec. 6.5.

Aries [HKK+04, HKI08] focuses on the detection of refactorable clones. They do this based on the relation between clone instances through inheritance, similar to Fontana et al. [FZZ12]. Aries also proposes a refactoring: if two clone classes are siblings of each other (share the same superclass), they propose to perform "Extract method" and "Pull up method" sequentially. This tool only proposes the refactoring, and does not provide help in the process of applying the refactoring.

We investigated several research efforts that look into code clone refactoring [Alw17, CKS18, KN01]. However, all of these tools only support a subset of all harmful clones that are found. Also, these tools are limited to suggesting refactoring opportunities, rather than actually applying refactorings where suitable. Finally, all published approaches have limitations, such as false positives in their clone detection [CKS18] or being limited to clone pairs [HKI08].

3 Addressing problems with clone type definitions

For each of type 1-3 clones [RC07] we list our solutions to their shortcomings (see Sec. 2.2 for these definitions) to increase the chance that we can refactor the clone while improving the design.

4 Shortcomings of clone types

The clone definitions, as outlined in Sec. 2.2, allow reasoning about the duplication in a software system. Clones by these definitions can relatively easily and efficiently be detected. This has allowed for large scale analyses of duplication. However, these clone type definitions have shortcomings which make the clones detected in correspondence with these definitions less valuable for (automated) refactoring purposes.

In this section we discuss the shortcomings of the different clone type definitions which make them less suitable for (automated) refactoring. Because of these shortcomings, clones found by these definitions are often found to require additional judgement whether they should and can be refactored.

4.1 Type 1 clones

Type 1 clones are *identical clone fragments except for variations in whitespace and comments* [RC07]. This allows for the detection of clones that are the result of copying and pasting existing code, along with other reasons why duplicates might get into a codebase.

Type 1 clones are in most cases implemented as textual equality between code fragments (except for whitespace and comments). Although textually equal, method calls can still refer to different methods, type declarations can still refer to different types and variables can be of a different type. In such cases refactoring opportunities could be invalidated. This can make type 1 clones less suitable for refactoring purposes, as they require additional judgment regarding the refactorability of such a clone. When aiming to automatically refactor clones, applying refactorings to clones which might refer to different methods, types and variables is bound to be error prone and result in a uncompilable project or a difference in functionality.

Because of this, type 1 clones may not all be subject to refactoring. In section we describe an alternate approach towards detecting type 1 clones, which results in only clones that can be refactored.

4.2 Type 2 clones

Type 2 clones are *structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments* [RC07]. This definition allows for the reasoning about code fragments that were copied and pasted, and then slightly modified. For refactoring purposes, this definition is unsuitable; 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 can be harmful instead of helpful.

```java
public boolean containsOnlyRedCircles(List<Circle> circles){
    return circles.stream().allMatch(Shape::isRed);
}

public Apple getEdibleApple(FruitBasket<Apple> basket){
    return basket.getFruitContainer()
        .getAppleByCriterium(Fruit::hasNotYetBeenEaten);
}
```

Figure 1: Example of a type 2 clone.

Looking at the example in Fig. 1, we see an example of a type 2 clone that poses no harm to the design of the system. Both methods are, except for their matching structure, completely different in functionality. They operate on different types, call different methods, return different things, etc. Having such a method flagged as a clone does not provide much useful information.

When looking at refactoring, type 2 clones can be very difficult to refactor. For instance if we have variability in types, the code can describe operations on two completely dissimilar types. Type 2 clones do not differentiate between primitives and objects, which further undermines the usefulness of clones detected by this definition.

4.3 Type 3 clones

Type 3 clones are *copied fragments with further modification (having added, removed or changed statements)* [RC07]. Detection of clones by this definition can be hard, as it may be hard to detect whether a fragment was copied in the first place if it was severely changed. Because of this, most clone detection implementations of type 3 clones work on basis of a similarity threshold [RC08, RK19, JMSG07, SYCI17]. This similarity threshold has been implemented in different ways: textual similarity (for instance using levenshtein distance) [**?**], token-level similarity [SSS$^+$16] of statement-level similarity [KS17].

Having a definition that allows for any change in code poses serious challenges on refactoring. A levenshtein distance of one can already change the meaning of a code fragment significantly, for instance if the name of a type differs by a character (and thus referring to different types).

4.4 Refactoring-oriented clone types

To resolve the shortcomings of clone types as outlined in the previous section, we propose alternative definitions for clone types to be directed at detecting clones that can and should be refactored. We have named these clones T1R (type 1R), T2R and T3R clones. These definitions share similarities with the literature definitions, the number of each type corresponds with the clone type it is modeled after. The "R" stands for refactoring-oriented (and may be less suitable for other analyses).

4.4.1 T1R clones

To solve the issues identified in Sec. 4.1, we introduce an alternative definition: cloned fragments have to be both textually *and* functionally equal. Therefore, T1R clones are a subset of type 1 clones.

We check functional equality of two fragments by checking the equality of the fully qualified identifier (FQI) for referenced types, methods and variables. If an identifier is fully qualified, it means we specify the full location of its declaration (e.g. `com.sb.fruit.Apple` for an `Apple` object). For method calls, we also compare the equality of the FQI of the type of each of its arguments, to differentiate between overloaded method variants. This way we can check whether two identifiers are functionally equal.

4.4.2 T2R clones

To solve the issues identifier in Sec. 4.2, we introduce an alternative definition. All rules that apply to T1R clones also apply to T2R clones. Additionally, T2R clones allow variability in literals, variables and method calls. Furthermore, T2R clones allow variability in method names and class/enum/interface names.

When refactoring two fragments that differ by literals, called methods or used variables, we are faced with a design tradeoff. When replacing the cloned fragments by a new method, we need an extra argument for each literal, called method or used variable that differs. This can be done as long as the type of used literals/variables and the signature of called methods are equal.

To limit the negative impact of this tradeoff on the design of the system, we formalized as threshold for the variability between fragments. The formula by which this threshold is calculated is displayed in equation 1. In this formula, *different expressions* refers to the amount of literals, variables and method calls that differ from other clone instances in a clone class. We divide this by the total number of tokens in the clone instance. Based on this threshold, we decide whether a clone should be considered for refactoring.

$$\text{T2R} = \frac{\text{Different expressions}}{\text{Total tokens}} * 100 \qquad (1)$$

4.4.3 T3R clones

Type 3 clones allow any change in statements (added, removed and changed statements). When looking at how we can refactor a statement that is not included by one clone instance but is in another, we find that

we require a conditional block to make up for the difference in statements. This is a tradeoff, as an added conditional block increases the complexity of the system. Because of that, we defined T3R clones in such a way that they are directed towards finding clones that are worth this tradeoff.

T3R clones allow, different from T2R clones, a gap between two clone classes of statements that are not cloned. The following rules apply to this gap:

- **The difference in statements must bridge a gap between two clones that were valid by the original thresholds**. This entails that, different from type 3 clones, the difference in statements cannot be at the beginning or the end of a cloned block. It is rather somewhere within, as it must bridge two existent clones. This holds the most value when looking at refactoring such clones. A difference at the beginning or end of a cloned block can better not be refactored.
- **The size of the gap between two clones is limited by a threshold.** This threshold is calculated by taking the percentage of the amount of statements in the gap over the amount of statements that both clones that are being bridged span. This is displayed in equation 2.
- **The gap may not span a partial block.** To make sure that the T3R clone can be refactored, we do not allow the gap to span a part of a block, for instance the declaration and a part of the body of a for-loop. The reason for this is that it is not possible to wrap a partially spanned block in a single conditional statement. We could however use multiple conditional blocks (one for each block spanned), but due to the detrimental effect on the design of the code (as each conditional block adds a certain complexity), we decided not to allow this for T3R clones.

$$\text{T3R Gap} = \frac{Statements(C_{gap})}{Statements(C_{above} \cup C_{below})} * 100 \quad (2)$$

$Statements$ = Statements in a code fragment

C_{gap} = The gap between two clones

C_{above} = The clone instance above the gap

C_{below} = The clone instance below the gap

4.4.4 The challenge of detecting these clones

To detect each type of clone, we need to parse the FQI of all types, method calls and variables. This comes with challenges, regarding both performance and implementation. To trace the declarations of variables, methods and types, we might need to follow cross file references. The referenced types/variables/methods might even not be part of the project, but rather of an external library or the standard libraries of the programming language. All these factors need to be considered for the referenced entity to be found, on basis of which an FQI can be created.

5 Clone Detection

As duplication in source code is a serious problem in many software systems, many tools have been proposed to detect various types of code clones [SK16, SR14]. However, these tools were not yet assessed in terms of automatically refactoring clones.

In this section, we first assess a set of modern clone detection tools for their applicability to this domain. Next, we introduce our own tool geared towards automatic clone refactoring, CloneRefactor.

5.1 Survey on Clone Detection Tools

We conducted a short survey on (recent) clone detection tools that we could use to analyze refactoring possibilities. The results of our survey are displayed in table 1. We chose a set of tools that are open source and can analyze a popular object-oriented programming language. Next, we formulate the following four criteria by which we analyze these tools:

1. **Should find clones in any context.** Some tools only find clones in specific contexts, such as only method-level clones. We want to perform an analysis on all clones in projects to get a complete overview.
2. **Finds clone classes in control projects.** We assembled a number of control projects to assess the validity of clone detection tools. On basis of this, we checked whether clone detection tools can correctly find clones in diverse contexts.
3. **Can analyse resolved symbols.** When detecting clones for refactoring purposes, it is important that clone instances can be merged. Sometimes, textual equality between code fragments does not imply that these can be merged (this is described more elaborately in Sec. 4.1). Because of this, we want to use a clone detection tool that can analyze such structures.
4. **Extensive detection configuration.** We aim to exclude expressions/statements from matching (more about our rationale in section 3). To achieve this, the tool needs to be able to allow those threshold changes. This can be either

through simple changes of the source code, or by using some configuration file.

Table 1: Our survey on clone detection tools.

Clone Detection Tool	(1)	(2)	(3)	(4)
Siamese [RK19]				✓
NiCAD [RC08, CR11]	✓	✓		
CPD [RCK09]	✓	✓		
CCFinder [KKI02] D-CCFinder [LHMI07]	✓	✓		
CCFinderSW [SYCI17]	✓			✓
SourcererCC [SSS+16] Oreo [SFL+18]	✓			✓
BigCloneEval [SR16]	✓	✓		
Deckard [JMSG07]	✓		✓	
Scorpio [HMK13, KS17]	✓		✓	✓

5.2 CloneRefactor

None of the state-of-the-art tools we identified implement all our criteria, so we decided to implement our own clone detection tool: CloneRefactor[1]. In this tool, we implemented both the literature clone types [RC07] and our refactoring-oriented clone types as described in Sec. 4.4.

Our tool is based on the JavaParser library [SvBT18]. This library can parse Java source code to an AST representation. This AST can then be analyzed, modified and eventually written back to source code. This allows us to perform AST-based clone detection and apply transformations to the AST based on the clones found.

CloneRefactor walks the AST and collects each declaration and statement (similar to the Scorpio clone detection tool [HMK13]). It then builds a graph representation on basis of this AST, in which each declaration and statement becomes a node. This graph representation maps the following relations for each declaration and statement:

- The declaration/statement preceding it
- The declaration/statement following it
- The previous statement/declaration that is cloned

Whether a node is considered a clone depends on the clone type that is being analyzed. On basis of this graph we detect clone classes. We compare clone classes against thresholds, and remove the clone classes that do not pass the test.

[1]CloneRefactor (WIP) is available on GitHub: `https://github.com/SimonBaars/CloneRefactor`. This repository contains all scripts that were used to retrieve the data that is displayed in this paper.

6 Experiments

This section outlines the conducted experiments to determine refactoring opportunities for code clones. First, we look into the thresholds that determine whether duplicated fragments are considered a clone in section 6.2. Next, we show the corpus on which we have performed our measurements in section 6.1. We then perform an analysis on the context of code clones in section 6.4. Finally, we map refactoring opportunities for clones in section 6.8.

6.1 The corpus

For our experiments we use a large corpus of open source projects [AS13]. This corpus has been assembled to contain relatively higher quality projects. 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 filtered this corpus further to only include projects that use Maven, which is a build tool which is mainly used to manage dependencies in the Java ecosystem. We then filtered the corpus further to only include projects for which all external dependencies are available. This resulted in 1.343 projects of varying sizes averaging at 980 source lines of code (omitting whitespace, comments) per project.

6.2 Thresholds

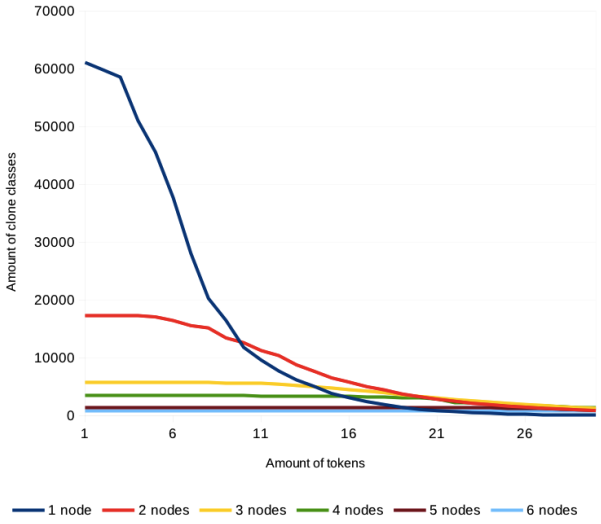
Thresholds are a tool that aid in the process of deciding whether a clone *should* be refactored. Many clone detection tools focus on either the number of lines [KKI02, SR16], number of tokens [RC08, SSS+16, RK19] and/or number of nodes (declarations/statements) [HMK13] to decide whether code fragments should be considered clones of each other. A comparison of clone detection tools by Bellon et al. [BKA+07] shows that most clone detection tools choose a minimum of **6** lines of code to be duplicate to consider code fragments a clone. The Scorpio clone detection tool uses this same number as minimum number of nodes [HMK13]. Token based tools mostly operate on basis of a minimum of **50** tokens for fragments to be considered a clone [SSS+16].

We would argue that going with this "magic number 6" eliminates a lot of harmful clones that should be refactored. For instance, a single 100-token statement will not be considered by such a threshold, which can still be harmful for the system design when cloned. Because of that, we decided to perform our measurements using thresholds that will include most clones that should be refactored, while eliminating most of the noise. With "noise" we mean duplication that has

no relevancy towards refactoring, like a single token that is duplicated elsewhere.

To find such a threshold, we measured the amount of clone classes found for a certain number of tokens. We then looked at the amount of nodes (statements/declarations) that these clones span. The resulting chart is displayed in Fig. 2.

Figure 2: Amount of times a clone with a certain amount of tokens has a certain amount of nodes.

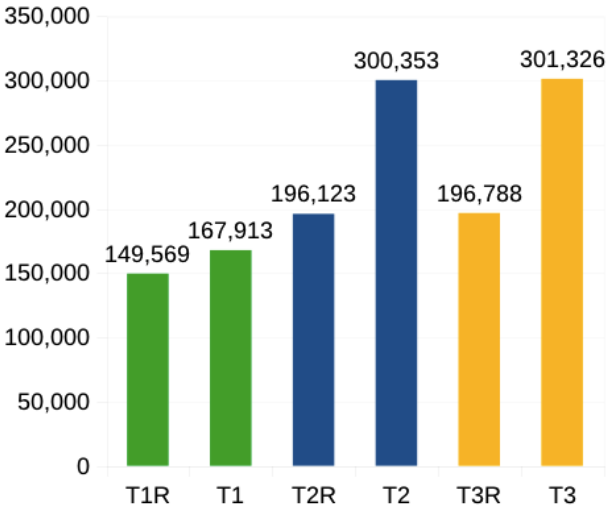


In this chart we see that when seeking clones with a minimum of 10 tokens, there are more clones that span 2 statements than clones that span one statement. We manually assessed these clones, and mainly the two-statement clones contain many clones that we classified as "should be refactored". Because of this, we decided to go with a minimum of 10 tokens threshold for our experiments.

6.3 Clone types

In this chapter we display the differences between clone type 1-3 [RC07] and type 1R-3R as proposed in section 4.4. When running our clone detection script over the corpus, we get the results displayed in Fig. 3.

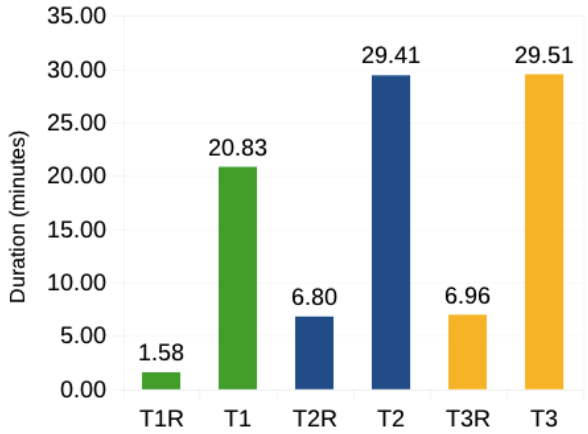
Figure 3: Number of cloned declarations/statements.



In this figure the amount cloned statements per clone type are displayed. The difference between T1R and T1 is small (10.9%), because most often textually equal code is also functionally equal. The difference between T2R and T2 is bigger (34.7%), mainly because we allow no variability in used types. T3R and T3 are similar to T2R and T2, because our dataset does not have so many gapped clones for the thresholds used.

We also measured the duration of finding clones by the different clone types. Fig. 4 shows the duration of detecting all clones in the corpus using CloneRefactor for different clone types. Although this data is dependent on the implementation, there is a notable difference between the refactoring-oriented clone types and the literature clone types. Most of the time that the refactoring-oriented clone types take longer is spent resolving symbols (to infer FQIs).

Figure 4: Duration in minutes of identifying clones for clone type definitions.



6.4 Context Analysis of Clones

To be able to refactor code clones, it is 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 through inheritance (for example: a clone instance resides in a superclass of another clone instance in the same clone class).

2. Where a clone instance occurs in the code (for example: a method-level clone is a clone instance that is in a method).

3. The contents of a clone instance (for example: the clone instance spans several methods).

Figure 5: Abstract representation of clone classes and clone instances.

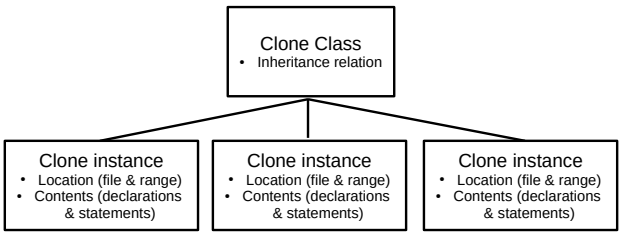


Fig. 5 shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is measured on clone class level: it involves all child clone instances. The location and contents of clones is measured on clone instance level. A clone's location involves the file it resides in and the range it spans (for example: line 6 col 2 - line 7 col 50). A clone instance contents consists of a list of all statements and declarations it spans.

All data shown in this section is measured using the T1R clone definition. We have performed the same measurements for the other type definitions and found that they follow similar trends. Because of that, we decided not to further show them in this section.

6.5 Relations Between Clone Instances

When merging code clones in object-oriented languages, it is very important to consider the inheritance relation between clone instances. This relation has a big impact on how a clone should be refactored.

6.5.1 Categorizing Clone Instance Relations

Fontana et al. [FZ15] describe 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 shown in Fig. 6. 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.

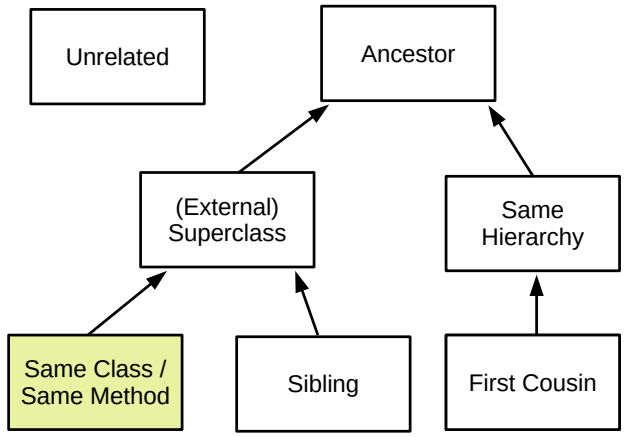
6. **First cousin class**: All instances of the clone class have the same grandparent class.

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.

Figure 6: Abstract figure displaying some relations of clone classes. Arrows represent superclass relations.



We use a similar setup to that used by Fontana et al. (Table 3 of Fontana et al. [FZ15]). Fontana et al. measure clones using their own tool (DCRA). As explained in section 5.1, we chose to implement our own tool, CloneRefactor. Therefore, the setup for our measurements differs as follows from Fontana et al.:

- We consider clone classes rather than clone pairs. The rationale for this is given in Sec. 2.1.
- We use different thresholds regarding when a clone should be considered. Fontana et al. seek clones that span a minimum of 7 source lines of code (SLOC). We seek clones with a minimum size of 6 statements/declarations. This is explained detail in Sec. 6.2.
- We seek duplicates by statement/declaration rather than SLOC. This makes our analysis depend less on the coding style (in terms of newline usage) of the author of the software project.
- We test a broader range of projects. Fontana et al. use a set of 50 relatively large projects. We use the corpus as explained in 6.1, which contains a diverse set of projects (diverse both in volume and code quality).

Table 2 contains our results regarding the relations between clone instances.

Table 2: Clone relations

Relation	#	%
Unrelated	4,762	38.14
Same Class	3,131	25.07
Sibling	1,949	15.61
Same Method	1,685	13.49
External Superclass	558	4.47
First Cousin	197	1.58
Superclass	118	0.94
Common Hierarchy	73	0.58
Ancestor	14	0.11

The most notable difference when comparing it to the results of Fontana et al. [FZ15] is that in our results most of the clones are unrelated (38.14% with T1R), while for them it was only 15.70%. 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, uses more classes from outside the project (which are marked "Unrelated" if they do not have a common external superclass). About a fourth of all clone classes have all instances in the same class, which is generally easy to refactor. On the third place come the "Sibling" clones, which can often be refactored using a pull-up refactoring. There are no noteworthy differences between type 1 and T1R clones.

6.6 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. [LWN07] discusses the harmfulness of cloning. The authors argue 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 our corpus. The results for the clone instance locations are shown in Table 3. 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.

Please note that these results are measured over each clone instance rather than each clone class, hence the higher total amount in comparison to the results of Sec. **??**.

Table 3: Clone instance locations

Location	#	%
Method Level	19,075	58.23
Class Level	12,207	37.27
Constructor Level	1,080	3.30
Interface Level	247	0.75
Enum Level	147	0.45

Our results indicate that 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. Regarding the differences between type 1 and T1R, it seems that there are relatively less method level clones and more class level clones for T1R. This is probably due to that the main reason for variability between type 1 and T1R is variable references, which occur more at method level than class level.

6.7 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 4.

Table 4: Clone instance contents

Contents	**# T1**	**% T1**	**# 1R**	**% 1R**
Partial Method	32,214	64.72	18,791	57.37
Several Methods	10,542	21.18	8,514	25.99
Includes Constructor	1,772	3.56	1,213	3.70
Includes Field	1,681	3.38	1,487	4.54
Partial Constructor	1,389	2.79	1,078	3.29
Only Fields	962	1.93	888	2.71
Full Method	647	1.30	284	0.87
Includes Class Declaration	263	0.53	258	0.79
Other Categories	304	0.61	243	0.74

Unsurprisingly, most clones span a part of a method. More than a quarter of the clones (for T1R) span over several methods, which either requires more advanced refactoring techniques or indicates a non-harmful clone.

6.8 Method extraction opportunities

The most used technique to refactor clones is method extraction (creating a new method on basis of the contents of clones). However, method extraction cannot be applied in all cases. 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 statements like `return`, `break`, `continue`. In these instances, more conditions may apply to be able to conduct a refactoring, if beneficial at all.

We measured the amount of clones that can be refactored through method extraction (without additional transformations being required). Our results are displayed in Table 5. In this table we use the following categories:

- **Can be extracted:** This clone is a fragment of code that can directly be extracted to a 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).

- **Complex control flow:** This clone contains `break`, `continue` or `return` statements.
- **Spans part of a block:** This clone spans a part of a statement.
- **Is not a partial method:** If the clone does not fall in the "Partial method" category of Table 4, the "extract method" refactoring technique cannot be applied.

Table 5: Refactorability through method extraction

	# T1	% T1	# 1R	% 1R
Is not a partial method	5,917	34.22	4,806	38.49
Complex control flow	5,511	31.87	3,158	25.29
Spans part of a block	3,989	23.07	3,152	25.24
Can be extracted	1,874	10.84	1,371	10.98

From Table 5, we can see that approximately ten percent of the clones can directly be refactored through method extraction (and possibly other refactoring techniques based on the relation of the clone instances). For the other clones, other techniques or transformations will be required. Looking into these techniques and transformations will be one of our next steps.

7 Threats to validity

We noticed that, when doing measurements on a corpus of this size, the thresholds that we use for the clone detection have a big impact on the results. There does not seem to be one golden set of thresholds, some thresholds work in some situations but fail in others. We have chosen thresholds that, according to our assessments, seemed optimal. However, by using these, we definitely miss some harmful clones.

8 Conclusion and next steps

In the research we have conducted so far we have made three novel contributions:

- We proposed a method with which we can detect clones that can/should be refactored.
- We mapped the context of clones in a large corpus of open source systems.
- We mapped the opportunities to perform method extraction on clones this corpus.

We have looked into existing definitions for different types of clones [RC07] and proposed solutions for problems that these types have with regards to automated refactoring. We propose that FQIs of method call signatures and type references should be considered instead of their plain text representation, to ensure refactorability. Furthermore, we propose that one should define thresholds for variability in variables, literals and method calls, in order to limit the number of parameters that the refactored unit shall have.

The research that we have conducted so far analyzes the context of different kinds of clones and prioritizes their refactoring. Firstly, we looked at the inheritance relation of clone instances in a clone class. We have found that more than a third of all clone classes are flagged unrelated, which means that they have at least one instance that has no relation through inheritance with the other instances. For about a fourth of the clone classes all of its instances are in the same class. About a sixth of the clone classes have clone instances that are siblings of each other (share the same superclass).

Secondly, we looked at the location of clone instances. Most clone instances (58 percent) are found at method level. About 37 percent of clone instances were found at class level. We defined "class level clones" as clones that exceed the boundaries of a single method or contain something else in the class (like field declarations, other methods, etc.). Thirdly, we looked at the contents of clone instances. Most clones span a part of a method (57 percent). About 26 percent of clones span over several methods.

We also looked into the refactorability of clones that span a part of a method. Over 10 percent of the clones can directly be refactored by extracting them to a new method (and calling the method at all usages using their relation). The main reason that most clones that span a part of a method cannot directly be refactored by method extraction, is that they contain `return`, `break` or `continue` statements.

8.1 Next steps

Our next step is to implement the "Extract Method" refactoring for the identified automatic refactoring opportunities. On basis of the resulting code, we can perform experiments comparing the maintainability index of the refactored code to the original code. On basis of these results, we can find better thresholds in order to determine thresholds. We can assess the impact of

our work on code volume, complexity, etc. This helps to verify the usability of the proposed type definitions towards our goal of improving the quality of a software system.

Acknowledgements

We would like to thank the Software Improvement Group (SIG) for their continuous support in this project.

References

[Alw17] Asif Alwaqfi. *A Refactoring Technique for Large Groups of Software Clones*. PhD thesis, Concordia University, 2017.

[AS13] Miltiadis Allamanis and Charles Sutton. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE, 2013.

[BKA+07] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

[CKS18] Zhiyuan Chen, Young-Woo Kwon, and Myoungkyu Song. Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution. *Journal of Software: Evolution and Process*, 30(10):e1951, 2018.

[CR11] James R Cordy and Chanchal K Roy. The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE, 2011.

[CYI+11] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, and Tateki Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones*, pages 7–13. ACM, 2011.

[Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FZ15] Francesca Arcelli Fontana and Marco Zanoni. A duplicated code refactoring advisor. In *International Conference on Agile Software Development*, pages 3–14. Springer, 2015.

[FZZ12] Francesca Arcelli Fontana, Marco Zanoni, and Francesco Zanoni. *Duplicated Code Refactoring Advisor (DCRA): a tool aimed at suggesting the best refactoring techniques of Java code clones*. PhD thesis, UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA, 2012.

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[HKK+04] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, and K Words. Aries: Refactoring support environment based on code clone analysis. In *IASTED Conf. on Software Engineering and Applications*, pages 222–229, 2004.

[HMK13] Yoshiki Higo, Hiroaki Murakami, and Shinji Kusumoto. Revisiting capability of pdg-based clone detection. Technical report, Citeseer, 2013.

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[JX10] Stan Jarzabek and Yinxing Xue. Are clones harmful for maintenance? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 73–74, New York, NY, USA, 2010. ACM.

[KG08] Cory J Kapser and Michael W Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645, 2008.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[KN01] Georges Golomingi Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. *Master's thesis, University of Bern*, 2001.

[KS17] CM Kamalpriya and Paramvir Singh. Enhancing program dependency graph based clone detection using approximate subgraph matching. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, 2017.

[LHMI07] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *29th International Conference on Software Engineering (ICSE'07)*, pages 106–115. IEEE, 2007.

[LST78] Bennet P Lientz, E. Burton Swanson, and Gail E Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[LWN07] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 18–18. IEEE, 2007.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[MVD+03] Tom Mens, Arie Van Deursen, et al. Refactoring: Emerging trends and open problems. In *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[RC07] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[RC08] Chanchal K Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th iEEE international conference on program comprehension*, pages 172–181. IEEE, 2008.

[RCK09] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[RK19] Chaiyong Ragkhitwetsagul and Jens Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, pages 1–49, 2019.

[SFL+18] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 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*, pages 354–365. ACM, 2018.

[SK16] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[SR14] Jeffrey Svajlenko and Chanchal K Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330. IEEE, 2014.

[SR16] Jeffrey Svajlenko and Chanchal K Roy. Bigcloneeval: A clone detection tool evaluation framework with bigclonebench. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 596–600. IEEE, 2016.

[SSS+16] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. IEEE, 2016.

[SvBT18] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser, 05 2018.

[SYCI17] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. Ccfindersw: Clone detection tool with flexible multilingual tokenization. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 654–659. IEEE, 2017.

[Wak04] William C Wake. *Refactoring workbook*. Addison-Wesley Professional, 2004.