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

Simon Baars
simon.mailadres@gmail.com

30 June 2019, 39 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 focus primarily on the Java programming language, as refactoring in general is very language-specific.

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**, 1]. 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 [2, 3], or does not happen at all [4]. 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 [5]. 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 6 and discussed in Chapter 7. Chapter 8, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 9 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 [6] 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 Valet project.

Figure 2.2 displays a clone class with three clone instances.

```
516     switch( numSigBytes )
517     {
518         case 3:
519             destination[ destOffset     ] = ALPHABET[ (inBuff >>> 18)        ];
520             destination[ destOffset + 1 ] = ALPHABET[ (inBuff >>> 12) & 0x3f ];
521             destination[ destOffset + 2 ] = ALPHABET[ (inBuff >>>  6) & 0x3f ];
522             destination[ destOffset + 3 ] = ALPHABET[ (inBuff       ) & 0x3f ];
523             return destination;
524
525         case 2:
526             destination[ destOffset     ] = ALPHABET[ (inBuff >>> 18)        ];
527             destination[ destOffset + 1 ] = ALPHABET[ (inBuff >>> 12) & 0x3f ];
528             destination[ destOffset + 2 ] = ALPHABET[ (inBuff >>>  6) & 0x3f ];
529             destination[ destOffset + 3 ] = EQUALS_SIGN;
530             return destination;
531
532         case 1:
533             destination[ destOffset     ] = ALPHABET[ (inBuff >>> 18)        ];
534             destination[ destOffset + 1 ] = ALPHABET[ (inBuff >>> 12) & 0x3f ];
535             destination[ destOffset + 2 ] = EQUALS_SIGN;
536             destination[ destOffset + 3 ] = EQUALS_SIGN;
537             return destination;
538
539         default:
```

Figure 2.2: Example of a clone class, as found in the Valet project.

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 [7, 8]. 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 [6]:

> **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 [9–11].

2.2.1 Type 4 clones

For this study 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 [12] 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 Background on refactoring methods

2.4 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 [13] 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 [14] 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.

2.4.1 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 [12] 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.

2.5 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."* [15]

However, the harmfulness of clones depends on a lot of factors. A paper by Kapser et al [16] 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 [17] 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.

2.6 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 [18] the authors look at the refactoring of class-level, method-level and constructor-level clones in Java.

[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`

Chapter 3

Defining refactoring-oriented clone types

In section **??** we introduced the four clone types as defined in literature. These simple definitions are suitable for analysis of a codebase. Their detection results in simple to understand numbers to argue about a codebase. However, these clone types have a few flaws which makes it hard to argue to what extend two fragments of code are functionally related. For each of type 1-3 clones [6] we list our solutions to their shortcomings to increase the chance that we can refactor the clone while improving the design.

We also look into clone detection tools for their suitability to support the proposed clone type definitions. We selected a few criteria Most clone detection tools support these definitions of clone types. However, many of these tools use a vastly different approach. A study by Saini et al [19] outlines different clone detection tools and compares their results for each of type 1-3 clones. Even though they operate on the same type definitions, the tools used in this study yield different results.

3.1 Shortcomings of clone types

Clone type 1-3 (further explain in section **??**) 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 [20]. However, these clone type definitions have shortcomings which makes 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. Because of these shortcomings, clones found by these definitions are often found to require additional judgment whether they should and can be refactored.

3.1.1 Type 1 clones

Type 1 clones are *identical clone fragments except for variations in whitespace and comments* [6]. 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 by most clone detection tools [21–25] 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 type 1 clones is bound to be error prone and can result in an uncompilable project or a difference in functionality.

GameWinningScene.java

```java
package com.simonbaars.fruitcollector;

import com.simonbaars.fruitcollector.fruits.winning.Apple;
import com.simonbaars.fruitcollector.images.winning.*;

import com.musiclibrary.music.highvolume.*;

public class GameWinningScene extends GameScene
{
    protected final Player player;

    public GameScene(Player player)
    {
        this.player = player;
    }

    public String checkAppleCollected(Apple collectedApple)
    {
        if(player.inventory.contains(collectedApple)){
            showImage(new PNGImage());
        }
        playSong(new MusicPlayer());
    }

    @Override
    public void playSong(MusicPlayer musicPlayer){
        musicPlayer.playSong("theme.mp3");
    }
}
```

GameOverScene.java

```java
package com.simonbaars.fruitcollector;

import com.simonbaars.fruitcollector.fruits.rotten.Apple;
import com.simonbaars.fruitcollector.images.losing.*;

import com.musiclibrary.music.lowvolume.*;

public class GameOverScene extends GameScene
{
    protected final Player player;

    public GameOverScene(Player player)
    {
        super(player);
    }

    public String checkAppleCollected(Apple collectedApple)
    {
        if(player.inventory.contains(collectedApple)){
            showImage(new PNGImage());
        }
        playSong(new MusicPlayer());
    }

    @Override
    public void playSong(MusicPlayer musicPlayer){
        musicPlayer.playSong("theme.mp3");
    }
}
```

Figure 3.1: Example of a type 1 clone that is functionally different.

In the example in figure 3.1, we see a type 1 clone consisting of two methods. However, these clones might still be very hard to refactor as we cannot see by this example whether they are functionally equal. Both code fragments use different imported types, some of which imported via a wildcard. Because of this, it is hard to verify which of the used types have the same underlying implementation.

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.

3.1.2 Type 2 clones

Type 2 clones are *structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments* [6]. This definition allows for the reasoning about code fragments that were copied and pasted, and then slightly modified. However, the definition does not adequately differentiate between slight modification and completely different fragments that just happen to have the same structure.

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> listOfCircles){
    return listOfCircles.stream().allMatch(Shape::isRed);
}

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

Figure 3.2: Example of a type 2 clone.

The example in figure 3.2 shows 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 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 reference types, which further undermines the usefulness of clones detected by this definition.

3.1.3 Type 3 clones

Type 3 clones are *copied fragments with further modification (with added, removed or changed statements)* [6]. 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 [22, 23, 26, 27]. This similarity threshold has been implemented in different ways: textual similarity (for instance using Levenshtein distance) [28], token-level similarity [9] or statement-level similarity [29].

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).

3.2 Refactoring-oriented clone types

To resolve the shortcomings of clone types as outlined in the previous section, we propose alternative definitions for clone types directed at detecting clones that can and should be refactored. We have named these clones T1R (type 1R), T2R and T3R clones. These definitions address problems of the corresponding literature definitions. The "R" stands for refactoring-oriented (and may be less useful for other analyses).

3.2.1 Type 1R clones

To solve the issues identified in Sec. 3.1.1, 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 validating 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).

3.2.1.1 Referenced Types

Many object-oriented programming languages (like Java, Python and C#) require the programmer to import a type (or the class in which it is declared) before it can be used. Based on what is imported, the meaning of the name of a type can differ. For instance, if we import `java.util.List`, we get the interface which is implemented by all list datastructures in Java. However, importing `java.awt.List`, we get a listbox GUI component for the Java Abstract Window Toolkit (AWT). To be sure we compare between equal types, type 1R clones compare the FQI for all referenced types.

3.2.1.2 Called methods

A codebase can have several methods with the same name. The implementation of these methods might differ. When we call two methods with an identical name, we can in fact call different methods. This is another reason that textually identical code fragments can differ functionally.

Because of this, for type 1R clones, we compare the fully qualified method signature for all method references. A fully qualified method signature consists of the fully qualified name of the method, the fully qualified type of the method plus the fully qualified type of each of its arguments. For instance, an `eat` method could become `com.sb.AppleCore com.sb.fruitgame.Apple.eat(com.sb.fruitgame.Tool)`.

3.2.1.3 Variables

In typed programming languages, each variable declaration should declare a name and a type. When we reference a variable, we only use its name. If, in different code fragments, we use variables with the same name but different types, the code can be functionally unequal but still textually equal. As an example, see the code in figure 3.3.

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

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

Figure 3.3: Variables with different types but the same name.

The body of both methods in figure 3.3 is equal. However, their functionality is not. The first method adds two numbers together and the other concatenates an integer to a String.

For type 1R clones variable references should be compared by both type and name.

3.2.2 Type 2R clones

Type 2R clones are modelled after type 2 clones, which allow any change in identifiers, literals, types, layout, and comments. 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 (as shown in figure 3.2 previously). Refactoring such clones can be harmful instead of helpful.

We tackle these problems with type 2R clones to be able to detect such clones that can and should be refactored. Type 1R clones are a subset of type 2R clones. All rules that apply to type 1R clones also apply to type 2R clones. Additionally, type 2R clones allow variability in literals, variables and method calls. This variability however is constrained by a threshold. Type 2R do not allow any variability in types, as opposed to type 2 clones which do allow variability in types. As type 2R clones are per definition more permissive than type 1R clones, type 1R clones are a subset of type 2R clones.

3.2.2.1 A threshold for variability in literals, variables and method calls

Type 2 clones allow any variability in literals, variables and method identifiers. However, this information tells a lot about the meaning of the code fragment. Most clone detection tools do not differentiate between a type 2 clone that differs by a single literal/identifier and one that differs by many. However, this does have a big impact on the meaning of the code fragment and thus the harmfulness of the duplication being there.

For type 2R clones we define a threshold for variability in literals, variables and method calls. We calculate the variability in literals, variables and method calls using the following formula:

$$\text{T2R Variability Percentage} = \frac{Diff(l) + Diff(v) + Diff(m)}{Total(t)} * 100$$

$Diff$ = Amount that differ from other clone instances in the clone class

$Total$ = Total number in the clone instance

l = Literals in clone

v = Variables in clone

m = Method calls in clone

t = Tokens in clones

Equation 1: Type 2R variability threshold formula

3.2.2.1.1 Literal and variable variability We allow only variability in the value of literals and variables, but not in their types. This is because a difference in literal/variable type may have a big impact on the refactorability of the cloned fragment. When we refactor different literals/variables that have both the same type, in case of an "Extract Method" refactoring, we have to create a parameter for this literal and pass the corresponding literal/variable from cloned locations. However, if two literals have different types, this might not be possible (or will have a negative effect on the design of the system).

This is because a lot of variability in literals will result in more parameters required in the extracted method, which is detrimental for the design of the system.

Consider the example in figure 3.4. In this example, the two methods have two literals that differ between them. We can perform an "Extract method" refactoring on these to get the result that is displayed on the right. In this process, we create a method parameter for the corresponding literal.

```java
// Original
public void printNice(String s){
    System.out.println("====================");
    System.out.println(s);
    System.out.println("====================");
}

public void printNice2(String s){
    System.out.println("--------------------");
    System.out.println(s);
    System.out.println("--------------------");
}
```

```java
// Refactored
public void printNice(String s){
    printNice(s, "====================");
}

public void printNice2(String s){
    printNice(s, "--------------------");
}

public void printNice(String s, String decoration){
    System.out.println(decoration);
    System.out.println(s);
    System.out.println(decoration);
}
```

Figure 3.4: Literal variability refactored.

3.2.2.1.2 Method call variability Most modern programming languages (like Java, Python and C#) allow to pass method references as a parameter to a method. This helps reducing duplication, as it is possible to refactor two code fragments which differ only by a method call. However, a method call does often not consist of a single token (like variables and literals). For instance, a method call `System.out.println()` consists of several segments: a type reference to the `System` type, a reference to the static `out` field and a call of the `println()` method.

Type 2R clones allow called methods to vary as long as they have the same argument types and return type. As with type 1R clones, these types are compared using their fully qualified identifiers. An example of this is shown in figure 3.5. In this example, we have two methods (`System.out.println` and `myFancyPrint`). We use the "Extract Method" refactoring method to extract a new method and use a parameter to pass the used method.

The method call variability property of type 2R clones imply that type 2R clones are not a subset of type 2 clones. Because methods calls can have a different structure, type 2R clones can be structurally slightly different. The example as shown in figure 3.5 can be a type 2R clone (dependent on the thresholds used), but is not a type 2 clone.

```java
// Original
public void printNice(){
    System.out.println("====================");
    myFancyPrint("The weather is nice");
    System.out.println("====================");
}

public void printNice2(){
    System.out.println("====================");
    System.out.println("The weather is nice");
    System.out.println("====================");
}

public void myFancyPrint(String s){
    Arrays.stream(s.toCharArray()).boxed()
        .map(e -> "("+e+")")
        .collect(Collectors.joining);
}
```

```java
// Refactored
public void printWeather(){
    printNice(this::myFancyPrint);
}

public void printNice2(){
    printNice(System.out::println);
}

public void printNice(Consumer<String> printFunction){
    System.out.println("====================");
    printFunction.accept("The weather is nice");
    System.out.println("====================");
}

public void myFancyPrint(String s){
    Arrays.stream(s.toCharArray()).boxed()
        .map(e -> "("+e+")")
        .collect(Collectors.joining);
}
```

Figure 3.5: Method variability refactored.

3.2.2.2 Allow any variability in some identifiers

When refactored, some identifiers have no detrimental effect on the design if they vary between cloned instances. In the previous section, variability in literals, variables and called methods would result in more parameters for the extracted method, thus indicating a detrimental effect on system design. However, not always does a difference between fragments result in such a tradeoff.

This section explains several patterns of variability that we can allow between cloned fragements without changing the method by which the fragments can be refactored.

3.2.2.2.1 Class and method names The names of classes and methods describe the implementation of their body. If two classes/methods are cloned, but their names differ, one of both names should be redundant. When refactoring such clones, we can choose one instance to keep and one to remove. Such a refactoring doesn't affect maintainability in any other way than refactoring a type 1R clone would. The same is true of interface, enumeration and annotation names. Because of this, type 2R allows any variability in class, method, interface, enumeration and annotation names. However, this will only open up a good refactoring opportunity if the entire body is cloned.

3.2.2.2.2 Variable names In section 3.2.2.1.1 we described how variability in used variables often creates a design tradeoff. However, there are cases in which a used variable does not create a design tradeoff. For this, the following conditions need to apply:

- The cloned variables are locally defined.
- The cloned variables have the same type.
- The cloned variables are used at the same places in cloned fragments.

Show examples

3.2.3 Type 3R clones

Type 3 clones allow any change in statements, often bounded by a similarity threshold. This means that type 3 clones allow the inclusion of a statement that is not detected by type 1 or 2 clone detection. 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. See figure 3.6 for an example of such a clone.

```java
// Original
public String wordOfNumbers(int max){
    Random r = new Random();
    StringBuilder word = new StringBuilder();
    for(int i = 0; i<=size; i++){
        word.append(i);
    }
    word.append("that's it");
    return word.toString();
}

public void printWordOfNumbers(int max){
    Random r = new Random();
    StringBuilder word = new StringBuilder();
    for(int i = 0; i<=size; i++){
        word.append(i);
        System.out.println("Appended "+i);
    }
    word.append("that's it");
    System.out.println(word);
}
```

```java
// Refactored
public List<String> wordOfNumbers(int size){
    return wordOfNumbers(n, size, false);
}

public void printWordOfNumbers(int size){
    System.out.println(wordOfNumbers(n, size, true));
}

public StringBuilder wordOfNumbers(int size,
                                   boolean doPrint){
    Random r = new Random();
    StringBuilder word = new StringBuilder();
    for(int i = 0; i<size; i++){
        word.append(i);
        if(doPrint) System.out.println("Appended "+i);
    }
    word.append("that's it");
    return word;
}
```

Figure 3.6: Added statement between cloned methods.

In figure 3.6 a single statement is added. This statement is found in between cloned lines. We have named this difference between the two clones, in this examples containing a single statement, the *gap*. This gap has a threshold for type 3R, which is calculated by the following formula:

Apart from this threshold, the gap in between clones may not span over different blocks. Look at figure 3.7 for an example of this. We cannot refactor both statements into a single conditional block. We could however use two conditional blocks, 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 type 3R clones.

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

$Statements$ = The amount of statements in this 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

Equation 2: Type 3R clone merge opportunities threshold formula

```
1  public String wordOfNumbers(int max){
2      Random r = new Random();
3      StringBuilder word = new StringBuilder();
4      for(int i = 0; i<=size; i++){
5          word.append(i);
6      }
7      word.append("that's it")
8      return word.toString();
9  }
10
11 public void printWordOfNumbers(int max){
12     Random r = new Random();
13     StringBuilder word = new StringBuilder();
14     for(int i = 0; i<=size; i++){
15         word.append(i);
16         System.out.println("Appended "+i);
17     }
18     System.out.println(word);
19     word.append("that's it");
20 }
```

Figure 3.7: Statements between clones in different blocks.

3.2.4 Clone types summarized

The given clone definitions (types 1R, 2R and 3R) are refactoring-oriented in the sense that they were designed after the literature type definitions but with a concrete refactoring opportunity in mind. Summarized, these types can be explained as follows:

- **Type 1R:** Allows no difference between cloned fragments, making it possible to refactor both fragments to a method call that contains the code of both locations.
- **Type 2R:** Allows difference between cloned fragments in controlled features of a codebase. Refactoring opportunities for these controlled features are known, allowing refactoring with a minor tradeoff.
- **Type 3R:** Allows any difference. When refactored, this difference must be wrapped in a conditionally executed block, which entails a major tradeoff.

$$\text{Cloned statements} = T1R \subseteq T2R \subseteq T3R \tag{3.1}$$

$$\text{Cloned lines} = T1R \subseteq T1, \text{ but } T2R \nsubseteq T2 \text{ and } T3R \nsubseteq T3 \tag{3.2}$$

3.3 The challenge of detecting these clones

To detect each type of clone, we need to parse the fully qualified identifier of all types, method calls and variables. This comes with serious challenges, regarding both performance and implementation. Also, to be able to parse all fully qualified identifiers, and trace the declarations of variables, 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 a fully qualified identifier can be created.

Mainly the requirement to have access to all external libraries is a difficult one to satisfy in diverse projects. Because of this, the proposed clone definitions may be less suitable for large scale clone detection purposes.

3.4 Unifying the types

In this chapter we have proposed refactoring-oriented definitions using the type 1, 2 and 3 clone definitions from literature as a baseline. In literature, these definitions are mainly aimed towards reasoning about duplication in source code. When considering these types for refactoring, the goal becomes slightly different. Because of this, having separate clone type definitions does not have any value. Rather, we need a single clone type definition by which we can detect all clones that can and should be considered for refactoring.

Because of this, the ultimate goal would be not to consider type 1R, 2R and 3R separately, but together. However, this is dependent on good thresholds for the type 2R variability and type 3R gap size. Because of this, we have dedicated section 6.1 to performing measurements to find good thresholds. The ultimate goal is to have a single unified definition of clones that can and should be refactored. Although it will be next to impossible to define such a definition and its corresponding thresholds that does not detect false positives. However, we strive to find at least a near-optimal set of thresholds regarding the type definitions proposed in this chapter.

3.5 Suitability of existing Clone Detection Tools for detecting these clones

We conducted a short survey on (recent) clone detection tools in order to expand them with the proposed types. The results of our survey are displayed in table 3.1. We chose a set of tools that are open source and can analyze at least one 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. We analyzed this behavior using a set of control projects.
2. **TODO: Criterium.** We assembled a number of test 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 refactored. Sometimes, textual equality between code fragments does not imply that these can be refactored (this is described more elaborately in section 3.1.1). 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.

Simon

TODO

Table 3.1: Our survey on clone detection tools.

Clone Detection Tool	**(1)**	**(2)**	**(3)**	**(4)**
Siamese [26]				✓
NiCAD [23, 30]	✓	✓		
CPD [31]	✓	✓		
CCFinder [21] D-CCFinder [20]	✓	✓		
CCFinderSW [22]				✓
SourcererCC [9] Oreo [32]	✓			✓
BigCloneEval [24]	✓	✓		
Deckard [27]	✓		✓	
Scorpio [29, 33]	✓		✓	✓

None of the state-of-the-art tools we identified implement all our criteria, so we decided to implement our own clone detection tool, which is further described in the next chapter.

Chapter 4

CloneRefactor

CloneRefactor is the name of our clone detection and refactoring tool. It features the following novel functions:

- Detection of clone classes rather than clone pairs.
- A novel detection method, aimed at extensibility.
- Detection of refactoring-oriented clone types, in addition to the literature clone types.
- Allows for automated refactoring of a subset of the detected duplication issues.

In this section we describe our approach and rationale for the design decisions regarding this tool.

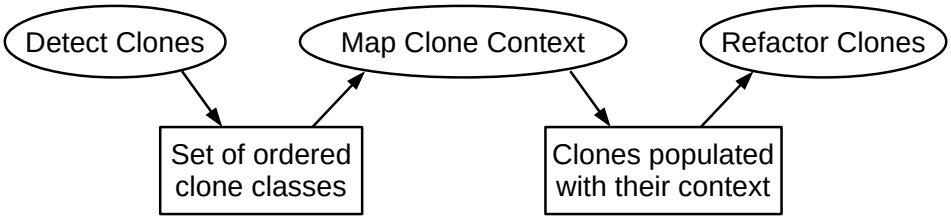


Figure 4.1: CloneRefactor overall process.

Figure 4.1 shows the overall process used by CloneRefactor. First, we detect clones on basis of a Java codebase, given a Java project from disk and a configuration. The clone detection process is further explained in section 4.2. After all clones are found, CloneRefactor maps the context of the clones. On basis of this context, CloneRefactor applies transformations to the source code for clones for which we have configured a refactoring.

4.1 JavaParser

A very important design decision for CloneRefactor is the usage of a library named JavaParser [34]. JavaParser is a Java library which allows to parse Java source files to an abstract syntax tree (AST). JavaParser allows to modify this AST and write the result back to Java source code. This allows us to apply refactorings to the detected problems in the source code.

Integrated in JavaParser is a library named SymbolSolver. This library allows for the resolution of symbols using JavaParser. For instance, we can use it to trace references (methods, variables, types, etc) to their declarations (these referenced identifiers are also called "symbols"). This is very useful for the detection of our refactoring-oriented clone types, as they make use of the fully qualified identifiers of symbols.

In order to be able to trace referenced identifiers SymbolSolver requires access to not only the analyzed Java projects, but also all its dependencies. This requires us to include all dependencies with the project. Along with this, SymbolSolver solves symbols in the JRE System Library (the standard libraries coming with every installation of Java) using the active Java Virtual Machine (JVM). This has a big impact on performance efficiency.

Because of the requirement of symbol resolution, the refactoring-oriented clone types are less suitable for large scale clone analysis.

4.2 Clone Detection

To detect clones, CloneRefactor parses the AST aqcuired from JavaParser to an unweighted graph structure. On basis of this graph structure, clones are detection. Dependent on the type of clones being detected, transformations may be applied. The way in which CloneRefactor was designed does not allow for several clone types to be detected simultaneously, in accordance with our clone type philosophy as described in chapter 3.4.

The overall process regarding clone detection is displayed in figure 4.2. First of all, we use JavaParser to read a project from disk and build an AST, one class file at a time. Each AST is then converted to a directed graph that maps relations between statements, further explained in section 4.2.1. On basis of this graph, we detect clone classes and verify them using three thresholds (in order of importance):

- Amount of tokens.
- Amount of statements.
- Amount of lines.

If the detection was configured to detect either type 2R, 3 or 3R clones we perform some type specific transformations on the resulting set of clones.

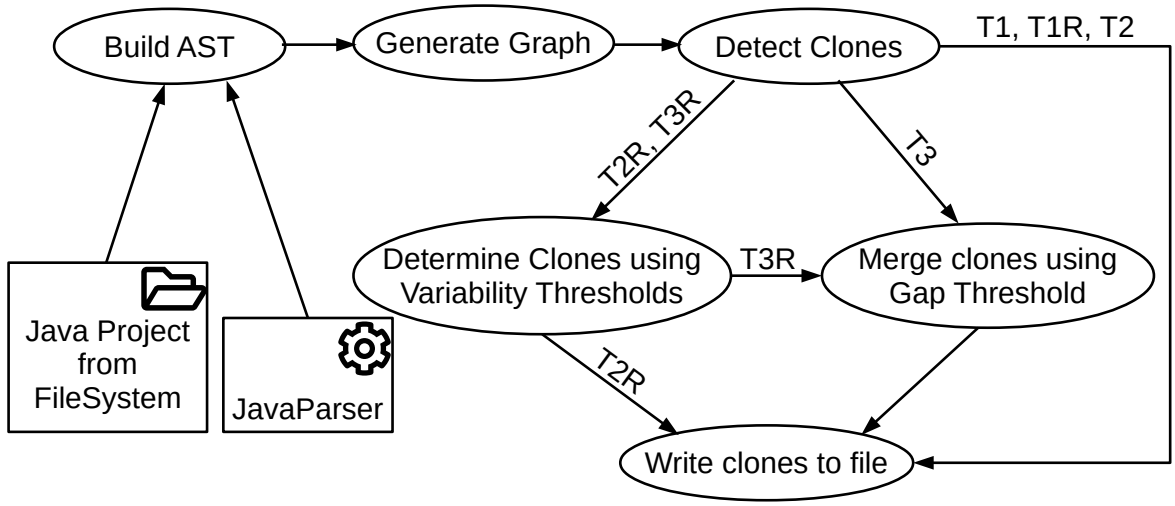


Figure 4.2: CloneRefactor clone detection process.

4.2.1 Generating the clone graph

First of all, we parse the AST obtained from JavaParser into a directed graph structure. We have chosen to base our clone detection around statements as the smallest unit of comparison. This means that a single statement cloned with another single statement is the smallest clone we can find. The rationale for this lies in both simplicity and performance efficiency. This means we won't be able to find when a single expression matches another expression, or even a single token matching another token. This is in most cases not a problem, as expressions are often small and do not span the minimal size to be considered a clone in the first place (more about this in section 6.1).

4.2.1.1 Filtering the AST

As a first step towards building the clone graph, we preprocess the AST to decide which AST nodes should become part of the clone graph. We have decided to consider declarations and statements as the smallest compared entities. The main reasoning for this is because considering smaller AST constructs, like expressions, significantly increases the complexity and CPU usage of our clone detection and refactoring efforts.

Additionally, we exclude package declarations and import statements. These are omitted by most clone detection tools, as clones in import statements hold limited valuable information.

4.2.1.2 Building the clone graph

Building the clone graph consists of walking the AST in-order for each declaration and statement. For each declaration/statement found, we map the following relations:

- The declaration/statement preceding it.
- The declaration/statement following.
- The last **preceding** declaration/statement with which it is cloned.

We do not create a separate graph for each class file, so the statement/declaration preceding or following could be in a different file. While mapping these relations, we maintain a hashed map containing the last occurrence of each unique statement. This map is used to efficiently find out whether a statement is cloned with another. An example of such a graph is displayed in figure 4.3.

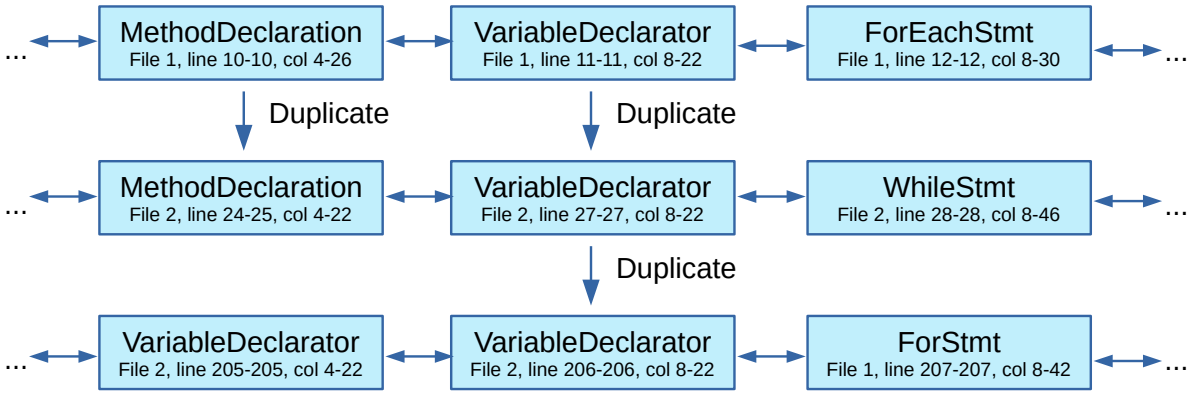


Figure 4.3: Abstract example of a part of a possible clone graph as built by CloneRefactor.

We refer to the declarations and statements in this graph as *nodes*. The relations *next* and *previous* in this graph are represented as an twodirectional arrow. The relations representing duplication are directed. This is a restriction we've chosen as it creates an important constraint for the clone detection process. This process is explained in section 4.2.4.

4.2.2 Comparing Statements/Declarations

In the previous section we described a "duplicate" relation between nodes in the clone graph built by CloneRefactor. Whether two nodes in this graph are duplicates of each other is dependent on the clone type. In this section, we will describe for each type how we compare statements and declarations to assess whether they are clones of each other.

CloneRefactor detects six different types of clones: T1, T2, T3, T1R, T2R and T3R. These types are further explained in chapter 3. For **type 1** clones, CloneRefactor filters the tokens of a node to exclude its comments, whitespace and end of line (EOL) characters and then compares these tokens. For **type 2** clones, the tokens are further filtered to omit all identifiers and literals. **Type 3** clones do the same duplication comparison as type 2 clones.

For **type 1R** clones, this comparison is a lot more advanced. For *method calls* we trace their declaration and use its fully qualified method signature for comparison with other nodes. For all *referenced types* we trace their declarations and use assemble their fully qualified identifier for comparison with other nodes. For *variables* we trace their declaration and their types. If the variable type is a primitive we can directly use it for comparison. If it is a referenced type, we have to trace this type first in order to collect their fully qualified identifier for comparison.

Type 2R clones allow any variation in literals, variables and method calls at this stage in the clone detection process. However, for *literals* we do resolve their type in order to verify that they are of the same type. For *variables* we also only verify that their types are the same (but not their names). For *method calls*, we trace their declaration but only compare the fully qualified identifier for its return type and each of its arguments' types. Apart from that, we do not compare the names of method, class, interface and enum declarations.

In this stage, **type 3R** clones have the same compare rules as type 2R clones.

4.2.3 Mapping graph nodes to code

The clone graph, as explained in section 4.2.1.2, contains all declarations and statements in a source code. However, declarations and statements may themselves have child declarations and statements. To avoid redundant duplication checks, we exclude child declarations and statements from each node. Look at figure 4.4 for an example of how source code maps to AST nodes.

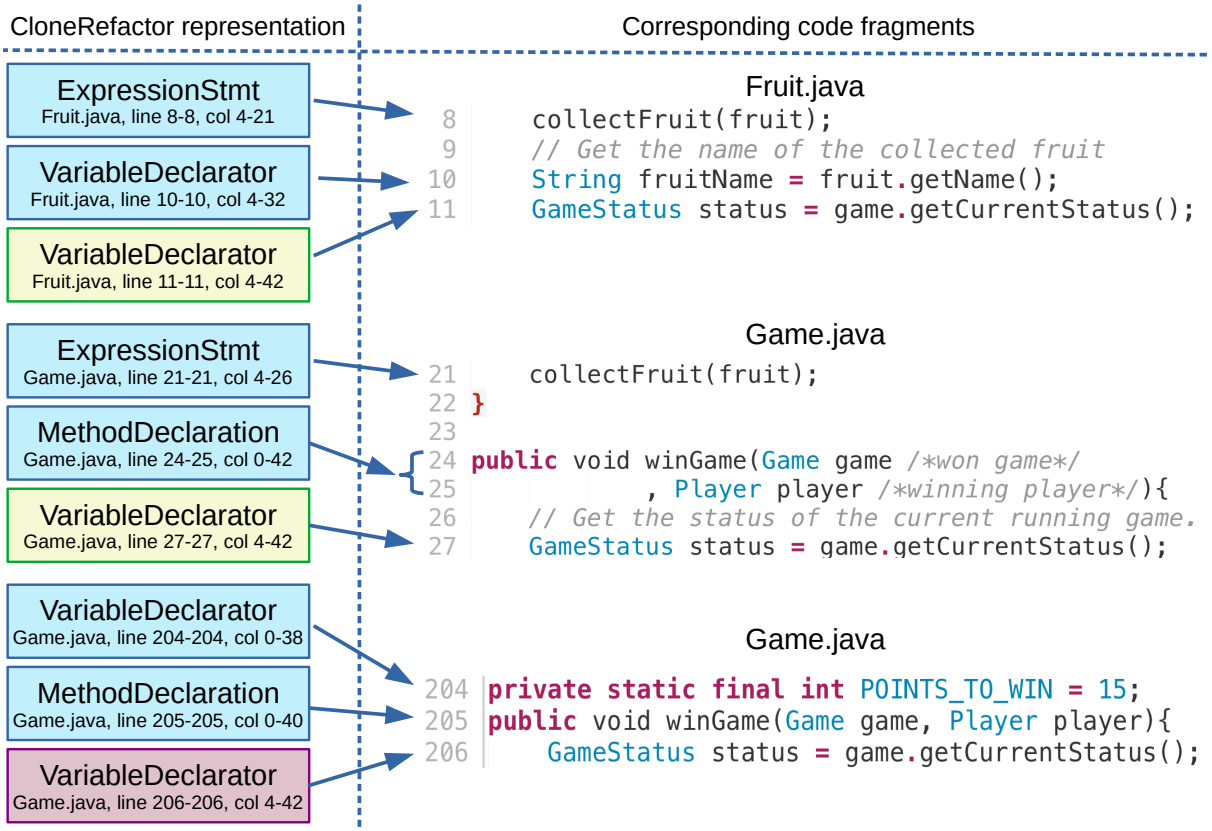


Figure 4.4: CloneRefactor extracts statements and declarations from source code.

In line 24-25 of the code fragment, we see a `MethodDeclaration`. The node corresponding with this MethodDeclaration denotes all tokens found on these two lines, line 24 and 25. Although the statements following this method declaration (those that are part of its body) officially belong to the method declaration, they are not included in its graph node. Because of that, in this example, the `MethodDeclaration` on line 24-25 will be considered a clone of the `MethodDeclaration` on line 205 even though their bodies might differ. Even the range (the line and column that this node spans) does not include its child statements and declarations.

4.2.4 Detecting Clones

After building the clone clone graph, we use it to detect clones. We decided to focus on the detection of clone classes rather than clone pairs because clone pairs do not provide a general overview of all entities containing the clones, with all their related issues and characteristics [35]. 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 [35].

As stated in the previous section, nodes in the graph link to *preceding* cloned statement. This implies that the first node that is cloned does not have any clone relation, as there are no clones preceding it (only following it). Because of this, we start our clone detection process at the final location encountered while building the graph. For this node, we collect all nodes it is cloned with. Even though the final node only links to the preceding node it is cloned with, we can collect all clones. This is because the

preceding clone also has a preceding clone (if applicable) and we can follow this trail to collect all clones of a single node. As an example, we convert the code example shown in figure 4.4 to a clone graph as displayed in figure .

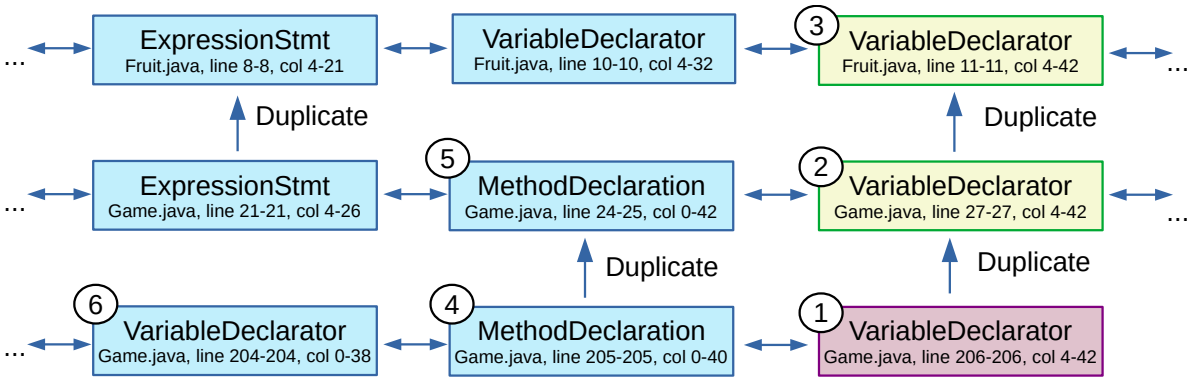


Figure 4.5: Example of a clone graph built by CloneRefactor.

Using the example shown in figure 4.4 and we can explain how we detect clones on basis of this graph. Suppose we are finding clones for two files and the final node of the second file is a variable declarator. This final node is represented in the example figure by the purple box (1). We then follow all "duplicate" relations until we have found all clones of this node (2 and 3). We now have a single statement clone class of three clone instances (1, 2 and 3).

Next, we move to the previous line (4). Here again, we collect all duplicates of this node (4 and 5). For each of these duplicates, we check whether the node following it is already in the clone class we collected in the previous iteration. In this case, (2) follows (5) and (1) follows (4). This means that node (3) does not form a 'chain' with other cloned statements. Because of this, the clone class of (1, 2 and 3) comes to an end. It will be checked against the thresholds, and if adhering to the thresholds, considered a clone.

We then go further to the previous node (6). In this case, this node does not have any clones. This means we check the (2 and 5, 1 and 4) clone class against the thresholds, and if it adheres, consider it a clone. Dependent on the thresholds, this example can result in a total of two clone classes.

Eventually, following only the "previous node" relations, we can get from (6) to (2). When we are at that point, we will find only one cloned node for (2), namely (3). However, after we check this clone against the thresholds, we check whether it is a subset of any exisiting clone. If this is the case (which it is for this example), we discard the clone.

4.2.4.1 Conceptual method of removing redundant clone classes

Using the method described in the this section, we find redundant clones. We remove these clones if they are a subset of an existant clone. We define the subset relation between clones as displayed as follows:

$$C_1 \subseteq C_2 \Leftrightarrow \forall(i_1 \in C_1)\ \exists(i_2 \in C_2)\ Fi_1 = Fi_2 \wedge Ri_1 \subseteq Ri_2 \tag{4.1}$$

Where C refers to a clone class (a set of clone instances), i refers to a clone instance, F is the file in which a clone instance is located and R is the range of tokens that a clone instance spans. For each clone added, we remove all existing clones that are a subset of the newly added clone:

$$S = S \setminus (C_{existing} \subseteq C_{new} \mid C_{existing} \in S) \tag{4.2}$$

Where S is the set of all clone classes that are found up in until this point. Likewise, we should not add the new clone to our list of clones if its a subset of an existing clone. Because of that, we check for each clone added whether there exists a clone of which the newly added clone is a subset:

$$(C_{existing} \subseteq C_{new} \mid C_{existing} \in S) = \emptyset \Rightarrow S = S \cup C_{new} \tag{4.3}$$

For CloneRefactor, we used this conceptual method of removing duplicates, but did not directly implemented these equations. The main reason for this is the runtime overhead, as this process has to iterate over each existing clone class and each of their clone instances. For large projects with a lot of duplication this process has a significant impact on performance.

4.2.4.2 Method of removing redundant clone classes

Write this section

4.2.5 Validating the type 2R variability threshold

In the definition of type 2R clones (see section 3.2.2) we described how type 2R clones work on basis of a variability threshold. This threshold, shown more formally in formula **??**, is checked by CloneRefactor. Implementing such a threshold involves some important design decisions, and has a lot of underwater complexity. In this section we explain how CloneRefactor detects type 2R clones on basis of this variability threshold.

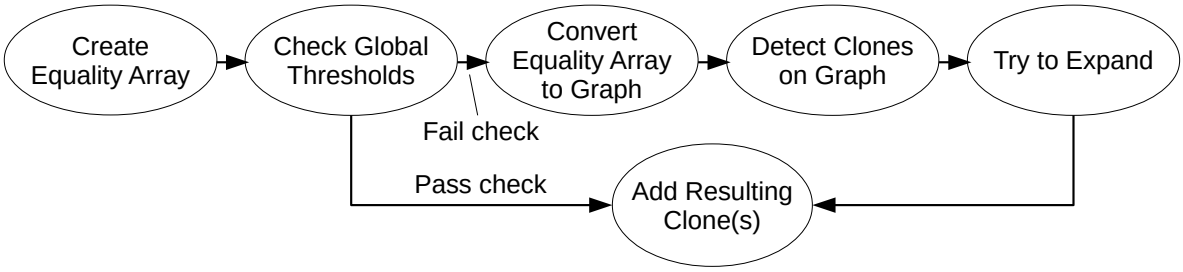


Figure 4.6: Process used to check the variability threshold for T2R clones.

Figure 4.6 shows the steps that CloneRefactor performs to find clones conforming with the type 2R variability threshold. This process is done as a postprocessing step after clone detection. The type 2R clone determination process is described in more detail in section 4.2.2. Each of the following paragraphs will explain a step from figure 4.6.

4.2.6 Checking for type 3 opportunities

Chapter 5

Experimental setup

In this chapter we describe the setup we use for our experiments. Our most prominent contribution is the proposal of a tool called CloneRefactor. This tool allows us to map clones with all clone definitions as described in chapter 3.

All results of our experiments, as displayed in chapter 6, are measured over a corpus of Java projects. In this chapter we will explain how we prepared this corpus.

5.1 The corpus

For our measurements we use a large corpus of open source projects [36]. 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.

As indicated in chapter 3.3 CloneRefactor requires all libraries of software projects we test. As these are not included in the used corpus [36], we decided to filter the corpus to only include Maven projects. Maven is a build automation tool used primarily for Java, and works on basis of an `pom.xml` file to describe the projects' dependencies. As no `pom.xml` files are included in the corpus, we cloned the latest version of each project in the corpus. We then removed each project that has no `pom.xml` file. As a final step, we collected all dependencies for each project by using the `mvn dependency:copy-dependencies -DoutputDirectory=lib` Maven command, and removed each project for which not all dependencies were available (due to non-Maven dependencies being used or unsatisfiable dependencies being referenced in the `pom.xml` file).

Some general data regarding this corpus is displayed in Table 5.1.

Table 5.1: General results for GitHub Java projects corpus [36].

Amount of projects	1,361
Amount of lines (excluding whitespace, comments and newlines.)	1,414,996
Amount of statements/declarations	1,212,189
Amount of tokens (excluding whitespace, comments and newlines.)	11,643,194

Chapter 6

Results

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

6.1 Thresholds

6.2 Relation, Location and Content Analysis of Clones

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 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 6.1: Abstract representation of clone classes and clone instances.

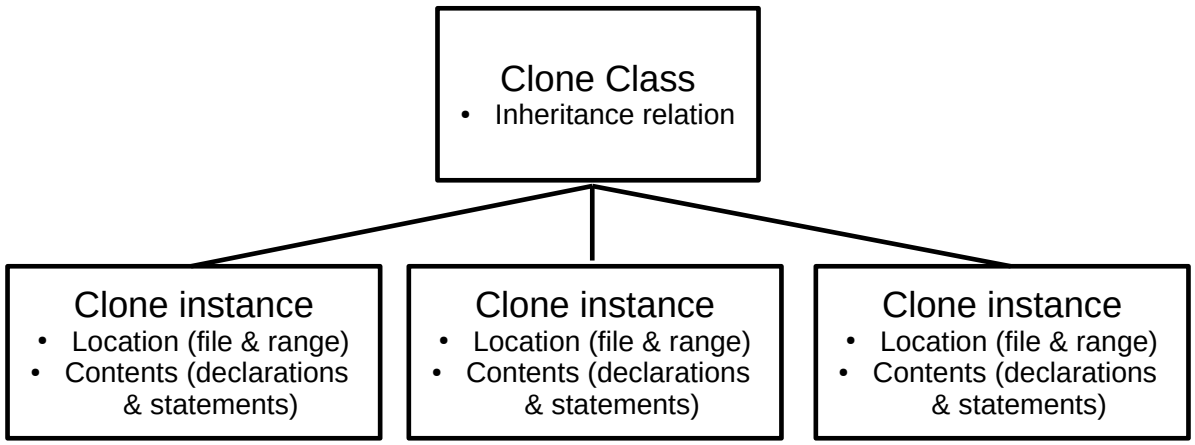


Figure 6.1 shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is measured on clone class level: it involves all child clone instances. The location and contents of clones is measured on clone instance level. A clone's location involves the 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.

We analyzed the context of clones in a large corpus of open source projects. For these experiments, we used our CloneRefactor tool. These experiments follow the structure of the context: The relation between clone instances is explained, measured and discussed in chapter 6.2.2; the location of clone instances is

explained, measured and discussed in chapter 6.2.4; the content of clone instances are explained, measured and discussed in chapter 6.3.

6.2.1 Clone detection results

Currently, we have implemented two clone detection algorithms into CloneRefactor. The first one finds clones by comparing tokens (excluding whitespace, comments and newlines), equal to the definition of type 1 clones in literature [6]. The second algorithm implements our type 1R, as explained in chapter **??**. The differences between the clones found for these algorithms is displayed in Table 6.1.

Table 6.1: CloneRefactor clone detection results for the two different algorithms.

	Type 1	**Type 1R**
Amount of lines cloned	200,362	129,519
Amount of statements/ declarations cloned	182,466	118,980
Amount of tokens cloned	1,582,845	973,596

Looking at Table 6.1, it becomes apparent that the type 1R algorithm finds significantly less clones than the type 1 algorithm. This indicates that about a third of the clones have textual equality, but are not actually equal when considering the types of expressions. This makes these clones less suitable for automated refactoring.

6.2.2 Relations Between Clone Instances

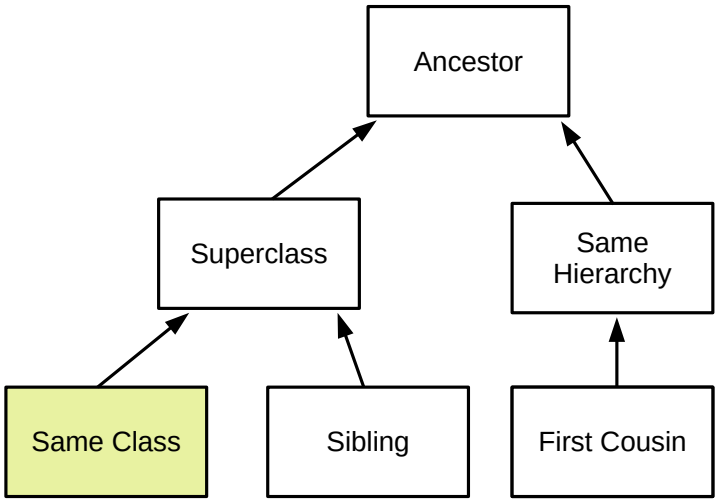
When merging code clones in object-oriented languages, it is very important to consider the relation between clone instances. This relation has a big impact on how a clone should be merged, in order to improve the software design in the process. In this chapter, we display measurements we conducted on the corpus introduced in chapter 5.1. These measurements are based on an experiment by Fontana et al. [37], which we will briefly introduce in chapter 6.2.2.1. We use a vastly different setup, which is explained in chapter 6.2.2.2. We then show our results in chapter 6.2.3.

6.2.2.1 Categorizing Clone Instance Relations

Fontana et al. [37] 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. A few of these categories are shown in Figure 6.2. 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 6.2: Abstract figure displaying some relations of clone classes. Arrows represent superclass relations.



Please note that none 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.

6.2.2.2 Our setup

We use a similar setup to that used by Fontana et al. (Table 3 of Fontana et al. [37]). Fontana et al. measure clones using their own tool (DCRA). As explained in chapter 3.5, 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 chapter **??**.
- 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 chapter **??**.
- 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 5.1, which contains a diverse set of projects (diverse both in volume and code quality).

6.2.3 Our results

Table 6.2 contains our results regarding the relations between clone instances. In this table, "T1" stands for the type 1 algorithm from literature and "1R" stands for our type 1R definition as explained in chapter **??**.

Table 6.2: Clone relations

Relation	# T1	% T1	# 1R	% 1R
Unrelated	6,134	35.48	4,762	38.14
Same Class	4,772	27.60	3,131	25.07
Sibling	2,680	15.50	1,949	15.61
Same Method	2,247	13.00	1,685	13.49
External Superclass	794	4.59	558	4.47
First Cousin	269	1.56	197	1.58
Superclass	237	1.37	118	0.94
Common Hierarchy	123	0.71	73	0.58
Ancestor	35	0.20	14	0.11

The most notable difference when comparing it to the results of Fontana et al. [37] is that in our results most of the clones are unrelated (38.14% with type 1R), 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 type 1R clones.

6.2.4 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. [13] 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 6.3. 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 chapter 6.2.3.

Table 6.3: Clone instance locations

Location	**# T1**	**% T1**	**# 1R**	**% 1R**
Method Level	32,861	66.02	19,075	58.23
Class Level	15,069	30.27	12,207	37.27
Constructor Level	1,391	2.79	1,080	3.30
Interface Level	282	0.57	247	0.75
Enum Level	171	0.34	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 type 1R, it seems that there are relatively less method level clones and more class level clones for type 1R. This is probably due to that the main reason for variability between type 1 and type 1R is variable references, which occur more at method level than class level.

6.3 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 6.4.

Table 6.4: 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 type 1R) span over several methods, which either requires more advanced refactoring techniques or indicates a non-harmful clone.

6.4 Merging duplicate code through refactoring

The most used technique to merge 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 6.5. 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 merge 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 6.4, the "extract method" refactoring technique cannot be applied.

Table 6.5: 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 6.5, 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.

Chapter 7

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 8

Related work

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

Chapter 9

Conclusion

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 [6] and proposed solutions for problems that these types have with regards to automated refactoring. We propose that fully qualified identifiers 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 merged 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.

9.1 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 manual assessment, seemed optimal. However, by using these, we definitely miss some harmful clones.

9.2 Future work

Acknowledgements

Thanks to you, for reading this :)

Bibliography

[1] W. C. Wake, *Refactoring workbook*. Addison-Wesley Professional, 2004.

[2] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance", *Communications of the ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[3] T. Mens and T. Tourwé, "A survey of software refactoring", *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[4] T. Mens, A. Van Deursen, *et al.*, "Refactoring: Emerging trends and open problems", in *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[5] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano, "Extracting code clones for refactoring using combinations of clone metrics", in *Proceedings of the 5th International Workshop on Software Clones*, ACM, 2011, pp. 7–13.

[6] C. K. Roy and J. R. Cordy, "A survey on software clone detection research", *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[7] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software", *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees", in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, IEEE, 1998, pp. 368–377.

[9] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code", in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, IEEE, 2016, pp. 1157–1168.

[10] E Kodhai, S Kanmani, A Kamatchi, R Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics", in *2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, IEEE, 2010, pp. 241–243.

[11] B. van Bladel and S. Demeyer, "A novel approach for detecting type-iv clones in test code", in *2019 IEEE 13th International Workshop on Software Clones (IWSC)*, IEEE, 2019, pp. 8–12.

[12] E Kodhai and S Kanmani, "Method-level code clone modification using refactoring techniques for clone maintenance", *Advanced Computing*, vol. 4, no. 2, p. 7, 2013.

[13] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment", in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, IEEE, 2007, pp. 18–18.

[14] and L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in oopl", in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, 2004, pp. 83–92. DOI: 10.1109/ISESE.2004.1334896.

[15] J. Ostberg and S. Wagner, "On automatically collectable metrics for software maintainability evaluation", in *2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, 2014, pp. 32–37. DOI: 10.1109/IWSM.Mensura.2014.19.

[16] C. Kapser and M. W. Godfrey, "'cloning considered harmful" considered harmful", in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, Citeseer, 2006, pp. 19–28.

[17] S. Jarzabek and Y. Xue, "Are clones harmful for maintenance?", in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10, Cape Town, South Africa: ACM, 2010, pp. 73–74, ISBN: 978-1-60558-980-0. DOI: 10.1145/1808901.1808911. [Online]. Available: http://doi.acm.org/10.1145/1808901.1808911.

[18] Y. Higo, S. Kusumoto, and K. 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*, vol. 20, no. 6, pp. 435–461, 2008.

[19] V. Saini, F. Farmahinifarahani, Y. Lu, D. Yang, P. Martins, H. Sajnani, P. Baldi, and C. Lopes, "Towards automating precision studies of clone detectors", *arXiv preprint arXiv:1812.05195*, 2018.

[20] S. Livieri, Y. Higo, M. Matushita, and K. 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)*, IEEE, 2007, pp. 106–115.

[21] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code", *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[22] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "Ccfindersw: Clone detection tool with flexible multilingual tokenization", in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2017, pp. 654–659.

[23] C. K. Roy and J. 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*, IEEE, 2008, pp. 172–181.

[24] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench", in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2016, pp. 596–600.

[25] ——, "Evaluating modern clone detection tools", in *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 321–330.

[26] C. Ragkhitwetsagul and J. Krinke, "Siamese: Scalable and incremental code clone search via multiple code representations", *Empirical Software Engineering*, pp. 1–49, 2019.

[27] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones", in *Proceedings of the 29th international conference on Software Engineering*, IEEE Computer Society, 2007, pp. 96–105.

[28] T. Lavoie and E. Merlo, "Automated type-3 clone oracle using levenshtein metric", in *Proceedings of the 5th international workshop on software clones*, ACM, 2011, pp. 34–40.

[29] C. Kamalpriya and P. Singh, "Enhancing program dependency graph based clone detection using approximate subgraph matching", in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, IEEE, 2017, pp. 1–7.

[30] J. R. Cordy and C. K. Roy, "The nicad clone detector", in *2011 IEEE 19th International Conference on Program Comprehension*, IEEE, 2011, pp. 219–220.

[31] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[32] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. 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*, ACM, 2018, pp. 354–365.

[33] Y. Higo, H. Murakami, and S. Kusumoto, "Revisiting capability of pdg-based clone detection", Citeseer, Tech. Rep., 2013.

[34] N. Smith, D. van Bruggen, and F. Tomassetti, *Javaparser*, May 2018.

[35] F. A. Fontana, M. Zanoni, and F. 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.

[36] M. Allamanis and C. Sutton, "Mining Source Code Repositories at Massive Scale using Language Modeling", in *The 10th Working Conference on Mining Software Repositories*, IEEE, 2013, pp. 207–216.

[37] F. A. Fontana and M. Zanoni, "A duplicated code refactoring advisor", in *International Conference on Agile Software Development*, Springer, 2015, pp. 3–14.

Appendix A

Non-crucial information