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

Simon Baars

simon.mailadres@gmail.com

30 June 2019, 43 pages

UNIVERSITY OF AMSTERDAM

FACULTY OF PHYSICS, MATHEMATICS AND INFORMATICS

MASTER SOFTWARE ENGINEERING

http://www.software-engineering-amsterdam.nl

Abstract

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

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 [1, 2]. 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 [3, 4], or does not happen at all [5]. 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 [1], he exclaims that *"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 [6]. However, most practises used in this thesis will also be applicable with other object-oriented languages, like C#.

1.1 Problem statement

In this section we describe the problem we address with this study and the research questions that we answer in order to contribute to solving the problem.

1.1.1 The problem

1.1.2 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.

1.1.3 Research method

TODO

1.2 Contributions

Rewrite this part

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

Rewrite this part

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

Rewrite this part

In Chapter 2 we describe the background of this thesis. Chapter **??** describes ... Results are shown in Chapter 6 and discussed in Chapter 7. Chapter 3, contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 8 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 Code clone terminology

Many studies present different definitions for code clone concepts. For this study, we mainly use the definitions from Bruntink et al. [7], Roy et al. [8] and Jiang et al. [9]. We created a summary of these concepts, which can be found in table 2.1.

Symbol	Meaning	Description	Properties
S	Clone class collection	All clone classes that have been found for a certain software project.	**Clones:** Set of clone classes.
C	Clone class	A set of similar code fragments in different locations.	**Instances:** Set of clone instances.
I	Clone instance	A code fragment that appears in multiple locations.	**Contents:** Totally ordered set of nodes, ordered by their begin positions. **File:** The file in which this clone instance is found. **Range:** The range this clone instance spans.
N	Node	A statement or declaration in a codebase.	**Range:** The range this node spans.
R	Range	A part of a source code file.	**Tokens:** Set of tokens that this range spans. **Begin line/column:** The line and column at which the range starts. **End line/column:** The line and column at which the range end.
T	Token	Tokens are the basic lexical building blocks of source code. For this study, this is the smallest relevant entity of a program.	**Range:** The range this token spans. **Category:** Identifier, Keyword, Literal, Separator, Operator, Comment or Whitespace

Table 2.1: Clone related terminology and how it maps to the source code.

2.1.1 Example

The code fragment in figure 2.1 shows two clone classes. One of the clone classes consists of two clone instances and has three nodes per instance. The other clone instance has three clone instances and consists of two nodes per instance.

```
1  // File1.java        1  // File2.java        1  // File3.java
2  doA();               2  doA();               2  doA();
3  doB();               3  doB();               3  doB();
4  doC();               4  doC();               4  doD();
```

Figure 2.1: Two clone classes: One clone class with three clone instances and one with two clone instances.

This example can be represented as a tree structure using the concepts from table 2.1. This tree structure is displayed in figure 2.2.
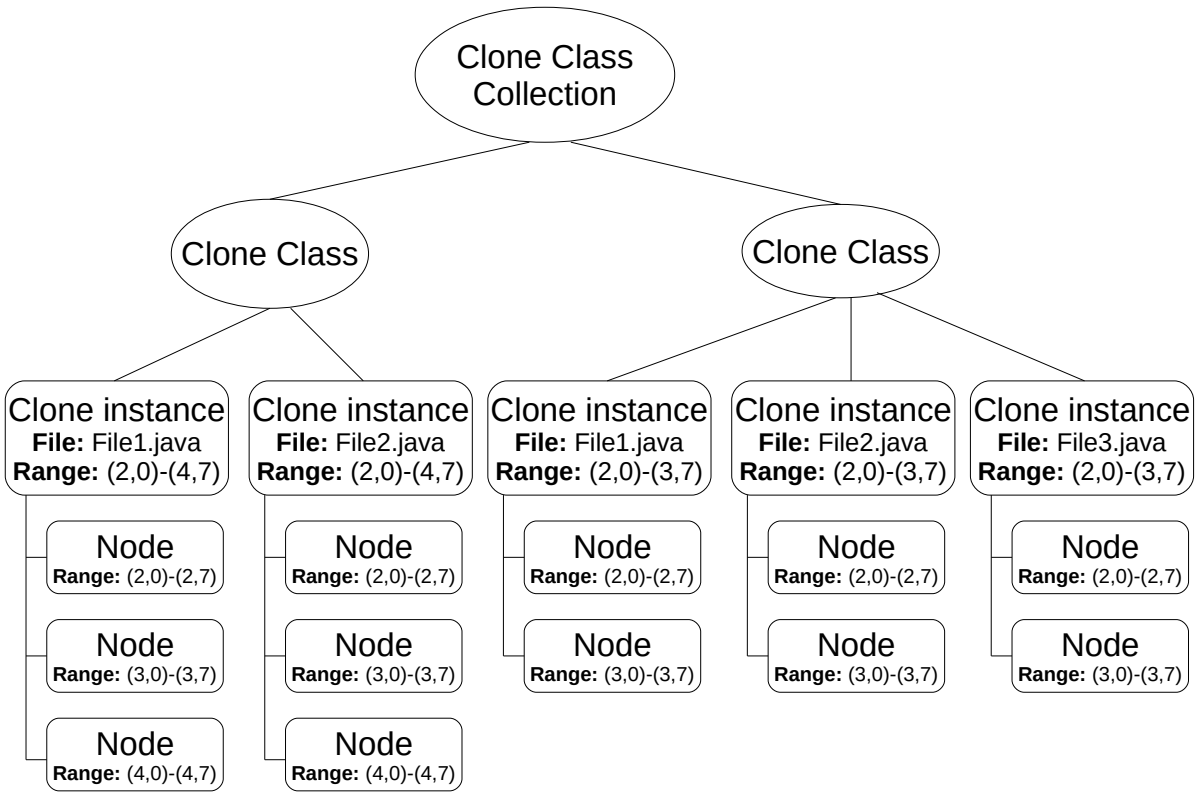
Figure 2.2: Representation of figure 2.1 using the terminology introduced in table 2.1

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 [10, 11]. 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 [8]:

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 [12–14].

2.2.1 Type 4 clones

For this study we have chosen to take type 4 clones out of the scope, because they are both hard to detect and hard to refactor. A study by Kodhai et al [15] 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 Refactoring methods

One of the main drivers for starting this research is the recent release of the second edition of Martin Fowlers' Refactoring book **??**. This book has been updated to the modern state of software engineering. The sentence in which he stresses the importance of unifying duplication in source code remains present in both books **??**.

The main method for dealing with duplication in source code, as described by Martin Fowler, is to extract the duplication to a common place. In most IDE's this refactoring method is called "Extract Method".

2.3.1 Extract Method

Method extraction is one of the most used methods of refactoring duplication in source code. Several studies have already concluded that most duplication in source code is found in the body of methods **??**. Method extraction can move matching functionality between methods to a common place: a new method. An example of this procedure is displayed in figure 2.3.

```
1  // Original
2  public void doStuff(){
3     doA();
4     doB();
5     doC();
6     doA();
7     doB();
8  }
```

```
1   // Refactored
2   public void doStuff(){
3      doAandB();
4      doC();
5      doAandB();
6   }
7
8   public void doAandB(){
9      doA();
10     doB();
11  }
```

Figure 2.3: Refactoring duplication through method extraction.

2.3.2 Move method

In the example of the previous section, both duplicated parts are in the same method. However, a study by Fontana et al. **??** shows that this is most often not the case. Based on the relation between clone instances, the extracted method must be moved to be accessible by all locations of the clone instances. This might require different techniques.

2.3.2.1 Pull up method

A refactoring technique to move a method up in its inheritance structure is called "Pull up method" **??**. This way, if cloned methods are related in any way through inheritance, they can be called by both classes by placing the method in a class they both have in common. This way its possible to refactor both fully cloned methods (by just pulling up the method) and partially cloned methods (by first performing method extraction and then pulling up the refactored method).

2.3.2.2 Create class abstraction based on implicit relations

Duplication in source code is an implicit relation between fragments of source code. If two classes have many of these implicit relations, then the implementation should be refactored to make this relation explicit. If the classes do not yet have a parent/super class, a parent class can be created and the common functionality can be placed in this newly created class. This makes the relation between these classes explicit and reduces duplication.

2.3.2.3 Providing default implementations for common functionality

Looking specifically at the Java programming language, recent versions of Java introduce a new method of reducing duplication through refactoring. Java has the concept of *interfaces* as an abstract type that is used to specify a behavior that classes must implement. Since Java version 8, Java also supports default implementations to be provided in interfaces **??**. This gives an opportunity to make the relation between classes explicit and reduce duplication. It can be used in instances where creating a new parent class for duplicated classes is undesirable.

2.3.3 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 [15] if all occurrences of the clone reside in the same class. If a method level clone is duplicated among classes in the same inheritance structure, we might need to pull-up a method in the inheritance structure. If instances of a method level clone are not in the same inheritance structure, we might need to either make a static method or create an inheritance structure ourselves. So not only a single instance of a clone has a context, but also the relationship between individual instances in a clone class. This is highly relevant to the way in which the clone has to be refactored.

2.4 Code clone harmfulness

Although duplication in source code is generally seen as a harmful coding practise that should be avoided. 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"* [16]. However, in some instances it is undesirable to remove all duplication, as it can lead to unnecessary complexity.

The harmfulness of clones depends on a lot of factors. A paper by Kapser et al [17] 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 [18] 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.

In this study, one of the main focus points will be to identify what makes a code clone harmful. Analyzing refactoring opportunities of code clones gives an unique insight in the state of the source code in case the duplication is not present. Analyzing these before- and after states of the source code will provide more insights that will help to rank clones based on their harmfulness.

2.5 Internal and external classes

In this study we differentiate between *internal* and *external* classes. Classes are the components of a software system that contain its functionality. When analyzing a software system, *Internal classes* are classes that belong to this software system specifically, and its source code is included in the project.

External classes are classes that a software system uses, but do not belong to this specific software system. Typically, the source code of external classes are not included in the source code of the project. Most often these external dependencies are referenced in a file that its build automation system uses to fetch a projects' dependencies.

Regarding refactoring, a software system can often not change the source code of its dependencies. This can sometimes be a burden, when an external dependency does not use proper abstractions that are required by a software system. Because of that, in this study, we consider external classes to see what impact they have on the refactoring process.

2.5.1 Maven

For this study we perform a shallow analysis on external dependencies, in order to derive more context for internally used concepts. As these external dependencies are most often not included in a software systems' source code, we must first use the projects' build automation system to gather the dependencies. To limit the scope of this process, we decided to focus on only the *Maven* build automation system.

Maven is a build automation tool, mainly used with the Java programming language. Maven has a simple ecosystem to configure and fetch the binaries and source code of all software projects a given codebase is dependent on. Maven can also be used to run tests for the systems and to package the project as an executable file.

Chapter 3

Related work

The Duplicated Code Refactoring Advisor (DCRA) looks into refactoring opportunities for clone pairs [19, 20]. 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. 6.2.2.

A tool named Aries [21, 22] focuses on the detection of refactorable clones. They do this based on the relation between clone instances through inheritance, similar to Fontana et al. [19]. This tool only proposes a refactoring opportunity and does not provide help in the process of applying the refactoring.

We investigated several research efforts that look into code clone refactoring [23–25]. 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 [24] or being limited to clone pairs [22].

Chapter 4

Defining refactoring-oriented clone types

In section 2.2 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 [8] 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 [26] 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.

4.1 Shortcomings of clone types

Clone type 1-3 (further explain in section 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 [27]. 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.

4.1.1 Type 1 clones

Type 1 clones are *identical clone fragments except for variations in whitespace and comments* [8]. 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 [28–32] 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.
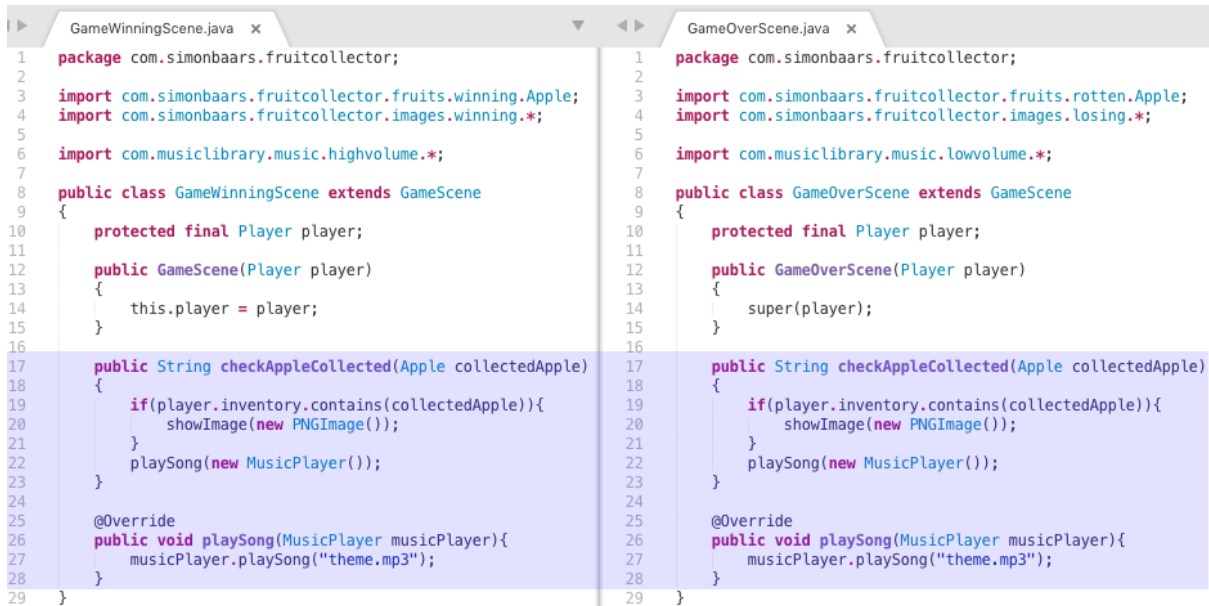
Figure 4.1: Example of a type 1 clone that is functionally different.

In the example in figure 4.1, 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.

4.1.2 Type 2 clones

Type 2 clones are *structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments* [8]. 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 4.2: Example of a type 2 clone.

The example in figure 4.2 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.

4.1.3 Type 3 clones

Type 3 clones are *copied fragments with further modification (with added, removed or changed statements)* [8]. 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 [9, 29, 30, 33]. This similarity threshold has been implemented in different ways: textual similarity (for instance using Levenshtein distance) [34], token-level similarity [12] or statement-level similarity [35].

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).

4.2 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).

4.2.1 Type 1R clones

To solve the issues identified in Sec. 4.1.1, 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).

4.2.1.1 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.

4.2.1.2 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)`.

4.2.1.3 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 4.3.

```
1  public boolean containsOnlyRedCircles(List<Circle> listOfCircles){
2      return listOfCircles.stream().allMatch(Shape::isRed);
3  }
4
5  public Apple getEdibleAppleFromBasket(FruitBasket<Apple> appleBasket){
6      return appleBasket.getFruitContainer().getAppleByCriterium(Fruit::hasNotYetBeenEaten);
7  }
```

Figure 4.3: Variables with different types but the same name.

The body of both methods in figure 4.3 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.

4.2.2 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 4.2 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.

4.2.2.1 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:

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

$$Diff = \text{Amount that differ from other clone instances in the clone class}$$

$$Total = \text{Total number in the clone instance}$$

$$l = \text{Literals in clone}$$

$$v = \text{Variables in clone}$$

$$m = \text{Method calls in clone}$$

$$t = \text{Tokens in clones}$$

Equation 1: Type 2R variability threshold formula

4.2.2.1.1 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 4.4. 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
1  // Original
2  public void printNice(String s){
3      System.out.println("====================");
4      System.out.println(s);
5      System.out.println("====================");
6  }
7
8  public void printNice2(String s){
9      System.out.println("--------------------");
10     System.out.println(s);
11     System.out.println("--------------------");
12 }
13
14
```

```java
1  // Refactored
2  public void printNice(String s){
3      printNice(s, "====================");
4  }
5
6  public void printNice2(String s){
7      printNice(s, "--------------------");
8  }
9
10 public void printNice(String s, String decoration){
11     System.out.println(decoration);
12     System.out.println(s);
13     System.out.println(decoration);
14 }
```

Figure 4.4: Literal variability refactored.

4.2.2.1.2 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 4.5. 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 4.5 can be a type 2R clone (dependent on the thresholds used), but is not a type 2 clone.

```java
1  // Original
2  public void printNice(){
3      System.out.println("====================");
4      myFancyPrint("The weather is nice");
5      System.out.println("====================");
6  }
7
8  public void printNice2(){
9      System.out.println("====================");
10     System.out.println("The weather is nice");
11     System.out.println("====================");
12 }
13
14 public void myFancyPrint(String s){
15     Arrays.stream(s.toCharArray()).boxed()
16         .map(e -> "("+e+")")
17         .collect(Collectors.joining);
18 }
19
20
```

```java
1  // Refactored
2  public void printWeather(){
3      printNice(this::myFancyPrint);
4  }
5
6  public void printNice2(){
7      printNice(System.out::println);
8  }
9
10 public void printNice(Consumer<String> printFunction){
11     System.out.println("====================");
12     printFunction.accept("The weather is nice");
13     System.out.println("====================");
14 }
15
16 public void myFancyPrint(String s){
17     Arrays.stream(s.toCharArray()).boxed()
18         .map(e -> "("+e+")")
19         .collect(Collectors.joining);
20 }
```

Figure 4.5: Method variability refactored.

4.2.2.2 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 fragments without changing the method by which the fragments can be refactored.

4.2.2.2.1 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.
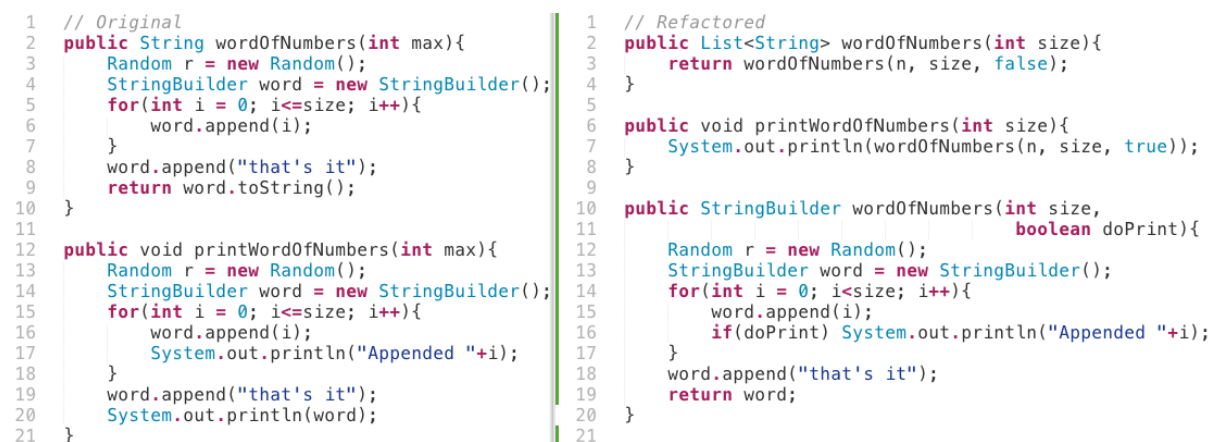
4.2.2.2.2 Variable names In section 4.2.2.1.1 we described how variability in used variables often creates a design tradeoff. However, there are cases in which a used variable does not create a design tradeoff. For this, the following conditions need to apply:

- 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

4.2.3 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 4.6 for an example of such a clone.

```
1   // Original                                        1   // Refactored
2   public String wordOfNumbers(int max){            2   public List<String> wordOfNumbers(int size){
3       Random r = new Random();                      3       return wordOfNumbers(n, size, false);
4       StringBuilder word = new StringBuilder();     4   }
5       for(int i = 0; i<=size; i++){                 5
6           word.append(i);                           6   public void printWordOfNumbers(int size){
7       }                                             7       System.out.println(wordOfNumbers(n, size, true));
8       word.append("that's it");                     8   }
9       return word.toString();                       9
10  }                                                 10  public StringBuilder wordOfNumbers(int size,
11                                                     11                              boolean doPrint){
12  public void printWordOfNumbers(int max){          12      Random r = new Random();
13      Random r = new Random();                       13      StringBuilder word = new StringBuilder();
14      StringBuilder word = new StringBuilder();     14      for(int i = 0; i<size; i++){
15      for(int i = 0; i<=size; i++){                 15          word.append(i);
16          word.append(i);                           16          if(doPrint) System.out.println("Appended "+i);
17          System.out.println("Appended "+i);        17      }
18      }                                             18      word.append("that's it");
19      word.append("that's it");                     19      return word;
20      System.out.println(word);                     20  }
21  }                                                 21
```

Figure 4.6: Added statement between cloned methods.

In figure 4.6 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 4.7 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.

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

$$Statements = \text{The amount of statements in this code fragment}$$

$$C_{gap} = \text{The gap between two clones}$$

$$C_{above} = \text{The clone instance above the gap}$$

$$C_{below} = \text{The clone instance below the gap}$$

Equation 2: Type 3R clone merge opportunities threshold formula

```java
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);
    }
    System.out.println(word);
    word.append("that's it");
}
```

Figure 4.7: Statements between clones in different blocks.

4.2.4   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.

$$\text{Cloned statements} = T1R \subseteq T2R \subseteq T3R \tag{4.1}$$

$$\text{Cloned lines} = T1R \subseteq T1, \text{ but } T2R \nsubseteq T2 \text{ and } T3R \nsubseteq T3 \tag{4.2}$$

4.3   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.

4.4 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.

4.5 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 4.1. 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.

   Simon TODO

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 4.1.1). 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 4). 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 4.1: Our survey on clone detection tools.

Clone Detection Tool	(1)	(2)	(3)	(4)
Siamese [33]				✓
NiCAD [30, 36]	✓	✓		
CPD [37]	✓	✓		
CCFinder [28] D-CCFinder [27]	✓	✓		
CCFinderSW [29]				✓
SourcererCC [12] Oreo [38]	✓			✓
BigCloneEval [31]	✓	✓		
Deckard [9]	✓		✓	
Scorpio [35, 39]	✓		✓	✓

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 5

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.

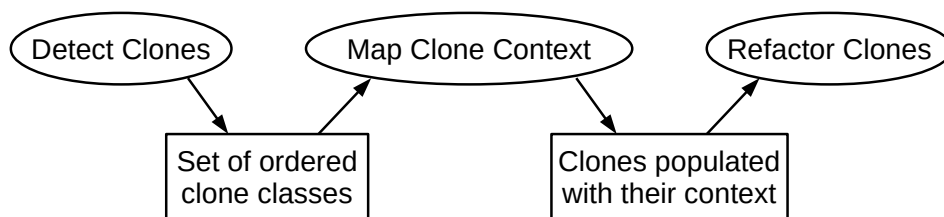In this section we describe our approach and rationale for the design decisions regarding this tool.

Figure 5.1: **CloneRefactor overall process.**

Figure 5.1 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 5.2. 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.

5.1 JavaParser

A very important design decision for CloneRefactor is the usage of a library named JavaParser [40]. 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.

5.2   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 4.4.

The overall process regarding clone detection is displayed in figure 5.2. 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 5.2.1. 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.
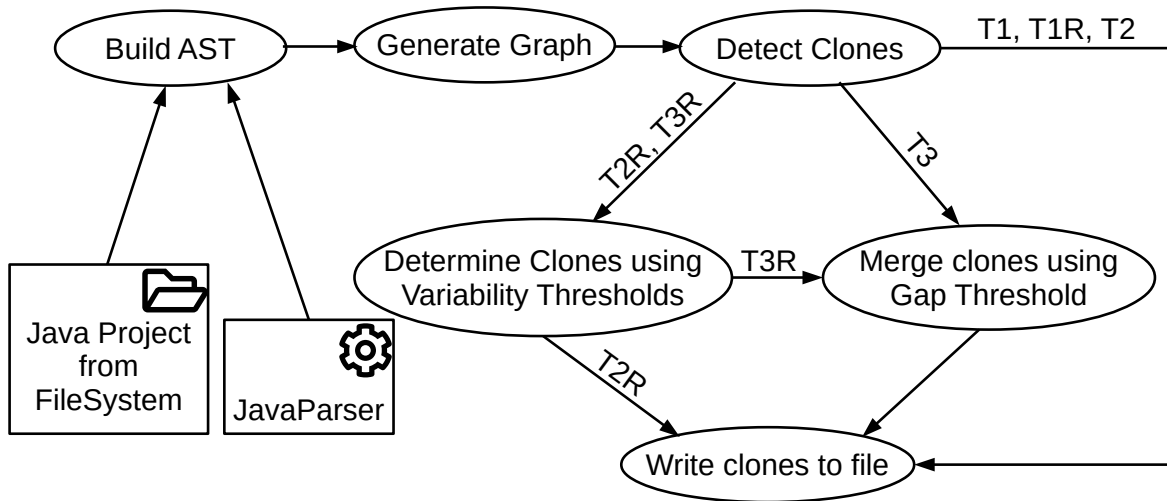

Figure 5.2: CloneRefactor clone detection process.

5.2.1   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).

5.2.1.1   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.

5.2.1.2 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 5.3.
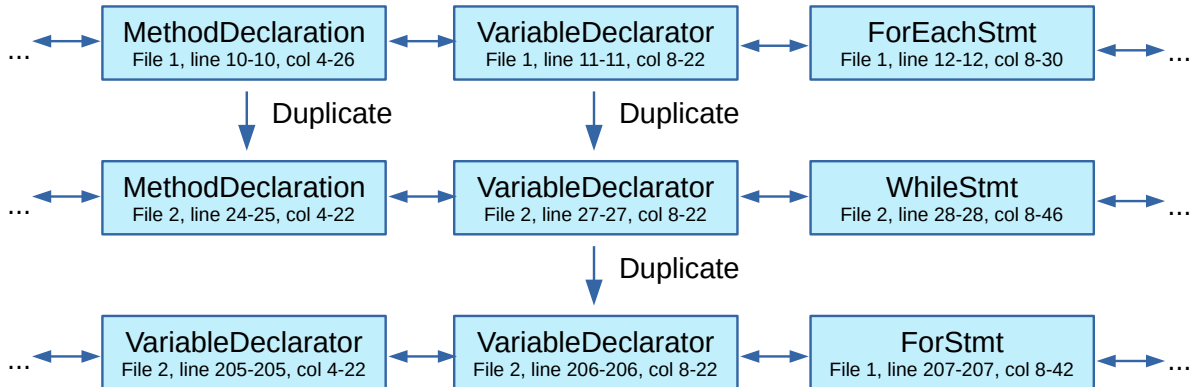

Figure 5.3: Abstract example of a part of a possible clone graph as built by CloneRefactor.

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 5.2.4.

5.2.2 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 4. 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.

5.2.3 Mapping graph nodes to code

The clone graph, as explained in section 5.2.1.2, 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 5.4 for an example of how source code maps to AST nodes.
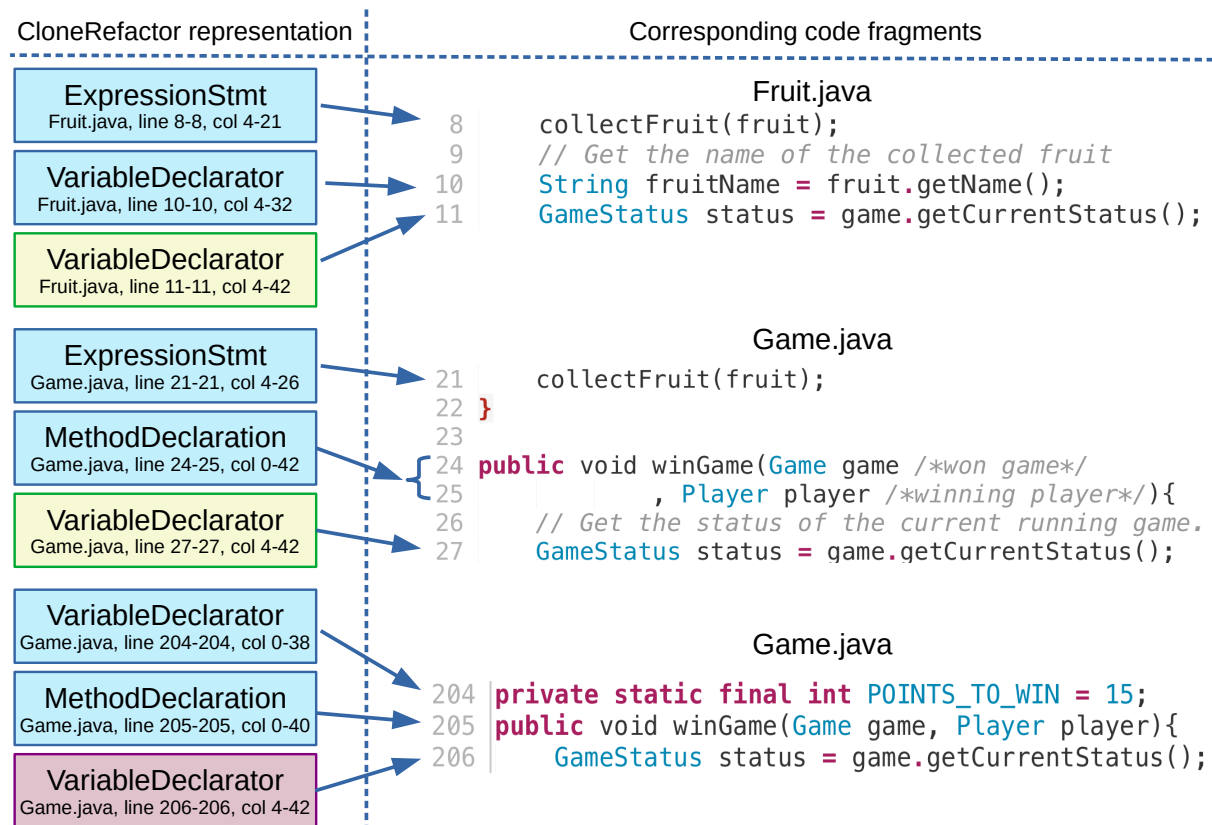
CloneRefactor representation	Corresponding code fragments

ExpressionStmt
Fruit.java, line 8-8, col 4-21

VariableDeclarator
Fruit.java, line 10-10, col 4-32

VariableDeclarator
Fruit.java, line 11-11, col 4-42

ExpressionStmt
Game.java, line 21-21, col 4-26

MethodDeclaration
Game.java, line 24-25, col 0-42

VariableDeclarator
Game.java, line 27-27, col 4-42

VariableDeclarator
Game.java, line 204-204, col 0-38

MethodDeclaration
Game.java, line 205-205, col 0-40

VariableDeclarator
Game.java, line 206-206, col 4-42

Fruit.java
```java
8     collectFruit(fruit);
9     // Get the name of the collected fruit
10    String fruitName = fruit.getName();
11    GameStatus status = game.getCurrentStatus();
```

Game.java
```java
21    collectFruit(fruit);
22  }
23
24  public void winGame(Game game /*won game*/
25          , Player player /*winning player*/){
26      // Get the status of the current running game.
27      GameStatus status = game.getCurrentStatus();
```

Game.java
```java
204 private static final int POINTS_TO_WIN = 15;
205 public void winGame(Game game, Player player){
206     GameStatus status = game.getCurrentStatus();
```

Figure 5.4: 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.

5.2.4 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 [19]. 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 [19].

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 5.4 to a clone graph as displayed in figure 5.5.
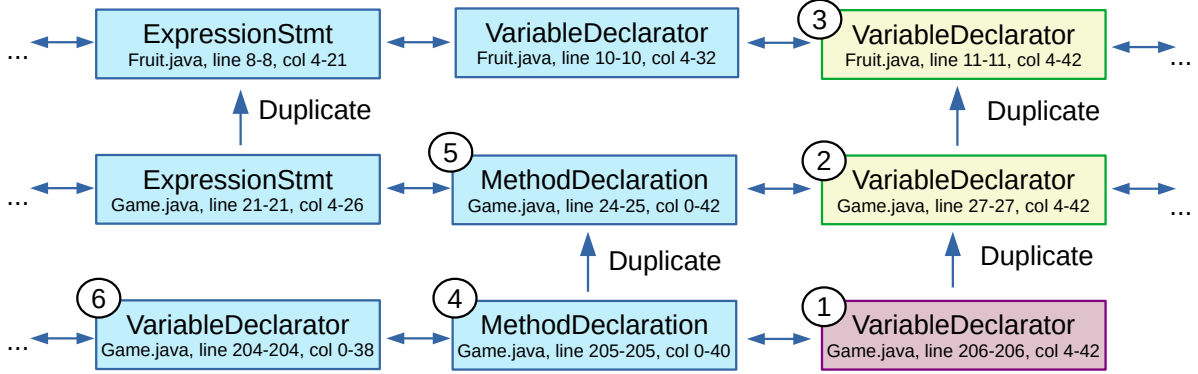

Figure 5.5: Example of a clone graph built by CloneRefactor.

Using the example shown in figure 5.4 and 5.5 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.

5.2.4.1   Conceptual method of removing redundant clone classes

The clone detection method used by CloneRefactor can, for various reasons,result in redundant clone classes. After the insertion of each newly detected clone, we check whether it is redundant and/or any of the existent clones has become redundant by adding this clone. A clone is redundant if it is a subset of an existing clone. We define the subset relation between clones as follows:

$$C_1 \subseteq C_2 \Leftrightarrow \forall (i_1 \in C_1) \; \exists (i_2 \in C_2) \; Fi_1 = Fi_2 \land Ri_1 \subseteq Ri_2 \tag{5.1}$$

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:

$$S = S \setminus \{C_{existing} \subseteq C_{new} \mid C_{existing} \in S\} \tag{5.2}$$

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:

$$\{C_{existing} \subseteq C_{new} \mid C_{existing} \in S\} = \emptyset \Rightarrow S = S \cup C_{new} \tag{5.3}$$

For CloneRefactor we used this conceptual method of removing duplicates, but did not directly implement 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.

5.2.4.2 CloneRefactor method of removing redundant clone classes

Because of the order in which clones are found in the clone graph, removing redundant clone classes can be optimized to not require a search of the entire set of clones. To do this, we mark each node that has been considered for duplication as *visited*. This includes nodes that are interated over and all duplicates of those nodes. When we later find a new clone class of which all nodes are *visited*, we don't add this clone class to the set of clone classes because it is a subset of an existing clone class.

> Removal of existing clones: How does this is possibly redundant thing work?

5.2.5 Validating the type 2R variability threshold

In the definition of type 2R clones (see section 4.2.2) 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.
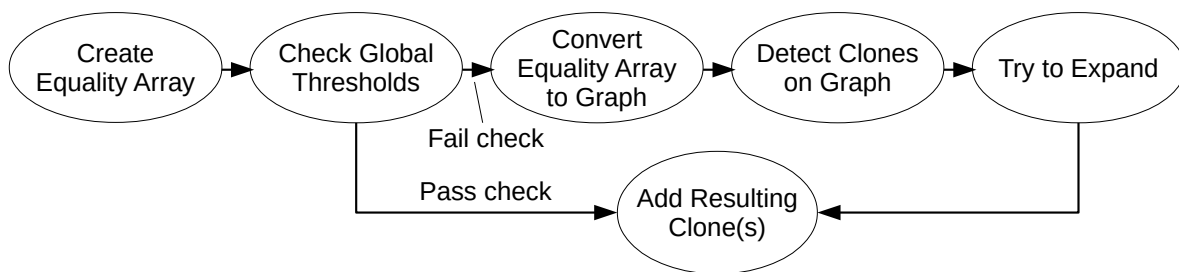
Figure 5.6: Process used to check the variability threshold for T2R clones.

Figure 5.6 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 5.2.2. Each of the following paragraphs will explain a step from figure 5.6.

5.2.5.1 Create equality array

To determine the difference in literals, method calls and variables, we convert the code to an equality array. This equality array converts each (group of) token(s) to a number unique to that (group of) tokens. Each literal, method call or variable becomes a positive number, whereas each other token becomes a negative number. An example of two code fragments converted to equality arrays is displayed in figure 5.7.
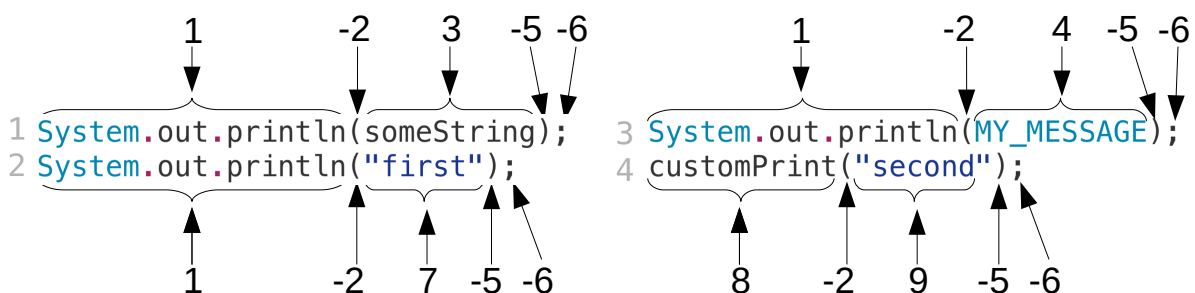
Figure 5.7: The conversion of code to an equality array.

The equality array for each of the lines in this example is shown in table 5.1.

Table 5.1: The equality arrays in our example figure 5.7.

Node (n)	Equality Array (E)
1	[1, -2, 3, -5, -6]
2	[1, -2, 7, -5, -6]
3	[1, -2, 4, -5, -6]
4	[8, -2, 9, -5, -6]

In the example of figure 5.7 the fragment on the left (consisting of n_1 and n_2) and the fragment on the right (consisting of n_3 and n_4) are two clone instances of a clone class. Table 5.1 shows their corresponding equality arrays.

5.2.5.2 Checking global thresholds

Using the equality arrays explained in the previous section, we can determine the variability threshold of any clone class. We calculate the variability of the example given in table 5.1 as follows:

$$\frac{|\{x > 0 \land y > 0 \land x \neq y \mid x \in E_1 \cup E_2, y \in E_3 \cup E_4\}|}{(|E_1| + |E_2|) + (|E_3| + |E_4|)} * 100 \tag{5.4}$$

The symbols in this equation refer to the symbols of the column headers of table 5.1. When we apply this equation to the example clone classes displayed in figure 5.7, we get the following sum:

$$\frac{|(3, 4), (1, 8), (7, 9)|}{(5 + 5) + (5 + 5)} * 100 = \frac{3}{20} * 100 = 15\% \tag{5.5}$$

So for the example given in figure 5.7 we have a variability of 15%. In CloneRefactor, the maximum variability percentage is a threshold that is entered in a configuration file. If a clone adheres to this threshold, it will stay in the set of found clones. However, if it does not adhere to the thresholds, a problem arises. Because a clone does not adhere to the thresholds, it does not yet mean it has to be discarded. This is because an invalid clone class can still contain valid clones that are a subset of the invalid clone (our definition of subsets of clones is given in section 5.2.4.1).

5.2.5.3 Convert Equality Array to Graph

`TODO`

Table 5.2: The equality arrays in our example figure 5.7.

Node (n)	Equality Array (E)
1	[1, -2, 3, -5, -6]
2	[1, -2, 3, -5, -6]
3	[8, -2, 4, -5, -6]
4	[1, -2, 7, -5, -6]

5.2.5.4 Detect Clones on Graph

`TODO`

$$E_1 \text{ is cloned with } E_2 \text{ iff } |E_1| = |E_2| \land \forall(x \in E_1)\exists(y \in E_2)x < 0 \lor y < 0 \Rightarrow x = y \tag{5.6}$$

5.2.5.5 Try to Expand

`TODO`

5.2.6 Checking for type 3 opportunities

`TODO`

Chapter 6

Results

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

6.1 Thresholds

6.2 Context Analysis of Clones

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 6.2.3; 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 [8]. 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

In section ?? we introduced our experiments regarding relations between clones. Table 6.2 contains our results regarding the relations between clone instances.

Table 6.2: Clone relations

Relation	#	%
Unrelated	12,368	34.88
Same Class	11,483	32.38
Same Method	5,056	14.26
Sibling	4,182	11.79
External Superclass	1,066	3.01
Superclass	558	1.57
First Cousin	489	1.38
Common Hierarchy	206	0.58
Ancestor	54	0.15

The most notable difference when comparing it to the results of Fontana et al. [20] is that in our results most of the clones are unrelated (34.44%), while for them it was only 15.70%. This is likely 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 third of all clone classes have all instances in the same class, which is generally easy to refactor. On the third place come the clones that are in the same method, which are similarly easy to refactor.

6.2.3 Clone instance location

Measuring the clone locations categories defined in section **??** yields the results displayed in Table 6.4.

Table 6.3: Clone instance locations

Location	#	%
Method Level	83,813	82.62
Class Level	12,534	12.35
Constructor Level	4,391	4.33
Interface Level	567	0.56
Enum Level	145	0.14

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

Measuring the clone contents categories defined in section **??** yields the results displayed in Table 6.4.

Table 6.4: Clone instance contents

Contents	#	%
Partial Method	79,945	78.80
Several Methods	7,323	7.22
Partial Constructor	4,385	4.32
Full Method	3,868	3.81
Only Fields	3,039	3.00
Includes Constructor	1,522	1.50
Includes Field	616	0.61
Includes Class Declaration	376	0.37
Other Categories	376	0.37

Unsurprisingly, most clones span a part of a method. The most used refactoring technique for clones that span part of a method is "Extract Method". Because of that, we focus our research efforts on refactoring such clones.

6.4   Merging duplicate code through refactoring

In section **??** we define catagories regarding the extent to which clones can be refactored through method extraction. Our results are displayed in Table 6.5.

Table 6.5: Refactorability through method extraction

	#	%
Can be extracted	14,664	41.35
Spans part of a block	10,801	30.46
Is not a partial method	8,074	22.77
Complex control flow	1,922	5.42

From Table 6.5, we can see that 41% 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.

6.5   Refactoring clones

TODO

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

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 [8] 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.

8.1 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.

8.2 Future work

This study presents a foundation for research in a largely unexplored field of studies: analyzing maintainability through automated refactoring. However, we scratched just the tip of the iceberg regarding all research opportunities in this field. In this section we describe possible extensions to this research, as well as other research opportunities in this field of studies.

8.2.1 Automated Refactoring for more metrics

In this study we presented evidence regarding the value of applying automated refactoring to analyze the before- and after state of source code and refactored source code. Analyzing these states we were able to analyze the improvement in maintainability after applying certain refactorings. This allows us to better assess thresholds by which maintainability issues in source code are identified. We also get better insights in the costs and values of applying certain refactoring efforts.

For this study we chose to focus only on the automated refactoring of duplication in source code. However, software maintainability depends on more factors and can be measured by more metrics. These factors also have opportunities to automate their refactorings. We think that several similarly sized studies can be conducted to automate the refactoring of other maintainability metrics.

A study by Heitlager et al. [41] presents several metrics by which the maintainability of source code can be assessed. They propose thresholds that indicate issues with these metrics. Many of these metrics have automated refactoring opportunities. In this section we will focus on several of these metrics to outline their opportunities for automated refactoring.

8.2.1.1 Long parameter list

When multiple parameters are used together in a method, there is an implicit dependency between these parameters: the dependency of being required by that method. If a lot of data hangs around really tight together, they should be made into their own object [42, 43]. Guidelines describe to limit the number of parameters per method to at most 4 [43].

If a method has many parameters, we can group strongly related parameters into an object. This can be done automatically, but two things must be considered:

- How do we determine whether parameters are strongly related?
- At what other places is this data used in unison and should thus use the new abstraction?

To determine whether parameters are strongly related we must look into at what places they are used in the codebase. We must then define some threshold that denotes the percentage of usages of these variables in which they are used together. If this threshold exceeds a certain amount, we can group them into an object. We must then trace all places in which they are used together and replace the variables by the newly created abstractions.

8.2.1.2 Method complexity

Method complexity refers to the complexity of the logic in a method body. There are several methods to compute method complexity. The most used complexity metric is (MCCabe) Cyclomatic Complexity [43], which refers to the amount of independent paths that can be taken though the source code. Another complexity metric that has recently become fairly popular is Cognitive Complexity [44], which attempts to measure the human perceived complexity. Both indicate an aspect of source code maintainability.

Dealing with method complexity can largely be done by method extraction. We extract a part of a complex method to a new method. This way we split the complexity of the original method into separately testable methods. Also, the methods become easier to read.

Refactoring complex methods can largely be done automatically. Also, many of the results of this study can be reused. To assess an automated refactoring opportunity for complex methods, we should assess which parts of methods can be split to end up with parts of similar complexity. For this, our research can be used to assess whether a given piece of code can be extracted to a new method (see section 6.4).

We recommend to extend our tool, CloneRefactor, to allow for such capabilities. CloneRefactor already contains a component that calculates the Cyclomatic Complexity of a given method. Using our automated refactored model, identified problems can relatively easily be refactored.

8.2.1.3 Method size

Method size has a strong relation, in terms of refactoring, with Cyclomatic Complexity. Although method size is often related to Cyclomatic Complexity, a study by Landman et al. [45] shows that Cyclomatic Complexity is not redundant to method size. However, they do share a similar method of refactoring. The automated refactoring opportunities described in the previous section also apply to method size.

8.2.1.4 Combining the metrics

Combining the automated refactoring models for each of these metrics can result in a model that ultimately provides significant improvement in the ease of writing well-maintainable source code. In this study, we presented a few tradeoffs that are the result of refactoring code clones. For instance, refactoring code clones with a lot of variability can result in long parameter lists. However, if we can combine this with automated refactoring of long parameter lists, this tradeoff can be mitigated. This way, it is possible to work towards an model of automated refactoring that reduces manual refactoring efforts significantly.

When programming, there are often trade-offs between technical debt and velocity. When a deadline comes near, often software quality in sacrificed to gain velocity [46–48]. Apart from that, a low programmer aptitude can result in low quality code **??**. This is because often forming appropriate abstractions requires time, effort and critical thinking. By introducing these abstractions automatically, this negative impact can (partly) be mitigated.

8.2.2 Identifying more types/definitions for clones

In this study, we proposed a set of clone type definitions for which a refactoring opportunity is known. We based these clone type definitions on the clone type definitions that are commonly used in literature. The design of these clone type definitions entails some decisions that have a large impact on the clones we considered for refactoring in this study.

8.2.2.1 Type 2R clones

With type 2R clones we allow variability in some identifiers and literals such that the code can and should still be refactored. For type 2R clones we chose a set of expressions in which we allow variability and proposed a recommended refactoring strategy. We think however that type 2R could still use a lot of improvement to find more duplication patterns that can be refactored.

One method we think can be used to find more refactoring opportunities is to allow variability in expressions that have/return the same type. If expressions have/return the same type, they can be extracted to a parameter and the corresponding expression can be passed as a parameter. An example of this is displayed in figure 8.1. The only thing to watch out for is method that have side effects. Because methods may be executed in another point during execution, this might affect the functionality of the code.

```java
// Original
public void doStuff(){
   int numbers = 456;
   doA(getTitle());
   doB(123);
   doC();
   doA("456");
   doB(numbers);
}

public String getTitle(){
   return "123";
}
```

```java
// Refactored
public void doStuff(){
   int numbers = 456;
   doAandB(getTitle(), 123);
   doC();
   doAandB("456", numbers);
}

public void doAandB(String var1,
    int var2){
   doA(var1);
   doB(var2);
}

public String getTitle(){
   return "123";
}
```

Figure 8.1: Refactoring different expressions that have the same return type.

8.2.2.2 Type 4 clones

In section 2.2 we introduced the clone types that are used in literature. Of these clone types, we considered type 1, 2 and 3 for refactoring. We chose not to consider type 4 clones because they appear far less in source code and are more difficult to detect and refactor. However, that does not mean that these clones should not be refactored.

We think that there are relevant research opportunities in refactoring type 4 clones. Type 4 clones are functionally equal pieces of code that are implemented through different implementations. Although functionally equal, type 4 clones might still differ in other aspects. For instance, for type 4 clones, one alternative might be easier to maintain. Also, they can differ (significantly) in performance.

There are interesting research opportunities in automatically choosing the better alternative among type 4 clone instances.

8.2.2.3 Proposing new clone type definitions

We think that current definitions of clone types still lack in the identification of all duplication issues that a developer should invest his/her refactoring efforts in. Current clone detection techniques still result in many false positives and false negatives. By proposing good definitions for code clones that mark the characteristics of harmful anti-patterns, we can create more accurate suggestions to developers.

8.2.3 Refactoring the harder to refactor clones

In section 6.4 we introduced to what extent clones can be refactored through method extraction. Because we strived to get results fast, we excluded categories that could not not be directly refactored through method extraction. However, with a few transformations or further considerations it might be possible to make these clones refactorable. In this section we will highlight a few of these categories which we believe to be refactorable through method extraction with a bit more effort.

8.2.3.1 Partial block

We did not consider clones for refactoring that span a part of a block. Although it is indeed not possible to refactor such clones, there are possibilities to make such clones refactorable. For instance, if the programming language supports lambda expressions, we can move the difference of statements in the block in a lambda expression. Figure 8.2 shows an example of such a refactoring opportunity.

```java
// Original
public void doStuff(){
    for(int i = 0; i<5; i++) { //Only
        the declaration of this for
        loop is cloned, but the loop
        body is not.
        System.out.println("hello!");
    }
    for(int i = 0; i<5; i++) {
        CoreController.activateCore(i);
    }
}
```

```java
// Refactored
public void doStuff(){
    doFiveTimes(() -> System.out.
        println("hello!"));
    doFiveTimes(() -> CoreController.
        activateCore(i));
}

public void doFiveTimes(Runnable
    runnable){
    for(int i = 0; i<5; i++) { //Only
        the declaration of this for
        loop is cloned, but the loop
        body is not.
        runnable.run();
    }
}
```

Figure 8.2: Refactoring a method that is obstructed by a complex control flow.

8.2.3.2   Overlap within a clone class

We do not consider clone classes for refactoring that overlap in the clone class itself. That is because such clones might require other techniques to refactor them. We consider analyzing such refactoring methods future work. An example refactoring of clone instances that overlap within their clone class is displayed in figure 8.3. In this example one clone instance is found between line 3 and 6 and the other between 4 and 7. To refactor such clones, other techniques may be required, like introducing a new for-loop to reduce the duplication.

```
1  // Original
2  public void sayHello(){
3      System.out.println("hello!");
4      System.out.println("hello!");
5      System.out.println("hello!");
6      System.out.println("hello!");
7      System.out.println("hello!");
8  }
```

```
1  // Refactored
2  public void sayHello(){
3      for(int i = 0; i<5; i++){
4          System.out.println("hello!");
5      }
6  }
```

Figure 8.3: Refactoring a clone class that has overlap between clone instances.

8.2.3.3   Complex control flow

Break, continue and return statements can obstruct the possibility of performing method extraction. However, with some extra transformations, method extraction will still be possible in such cases. Figure 8.4 shows such a transformation. We can wrap the newly extracted method in a conditional to indicate whether the "control flow modifying statement" should be executed. In other cases, other methods might apply to refactor such clones.

```
1  // Original
2  public boolean doStuff(){
3      if(doA());
4          return false;
5      doB();
6      doC();
7      if(doA());
8          return false;
9      doB();
10     return true;
11 }
```

```
1  // Refactored
2  public boolean doStuff(){
3      if(!doAandB())
4          return false;
5      doC();
6      return doAandB();
7  }
8
9  public boolean doAandB(){
10     if(doA())
11         return false;
12     doB();
13     return true;
14 }
```

Figure 8.4: Refactoring a method that is obstructed by a complex control flow.

8.2.4   Naming of refactored methods/classes/etc.

In this study, we took naming of refactored entities out of the scope. We applied automated refactoring to duplication, and gave all generated methods, classes and interfaces generated names. For our purposes that was not much of an issue, because of the validation methods used. We validated our approach using the SIG Maintainability model, which does not take naming quality into account. Because of that, the results of our experiments are in no way dependent on the names we give the generated methods, classes and interfaces.

When using our work with the purpose of refactoring assistance, these names will have to be manually provided. However, recent studies allow assistance in this process by generating a name that matches the body of a declaration. If this can be done in a reliable way, we can apply refactorings without any manual steps required.

A study by Allamanis et al. [49] proposes a machine learning model that can suggest accurate method and class names. This study shows promising results towards generating method and class names on basis of their body and context. However, the source code of this study in not available, making it harder to apply their findings to generate class and method names for any software project.

In a recent study by Alon et al. [50] they propose code2seq. Code2seq is a machine learning model that guesses the name of a method given a method body. This model has been trained on a large set of method bodies and names. The model already shows promising results. The source code of code2seq is publicly available, making it possible to embed this model in any application. Although this model is still far from perfect, combining it with our research could already greatly reduce the manual refactoring effort required. The main deficiency of this model lies in that its limited to the method body and does currently not consider anything else in a method body. This method is flawed, because the meaning of method bodies depends heavily on their context. For instance, the body of the method `get()` in `ArrayList` does not make any sense in relation to its name without considering its context (class, inheritance hierarchy, etc.).

8.2.5   Looking into other languages/paradigms

In this study we describe duplicate code refactoring opportunities for object-oriented languages. We built a tool to refactor code clones in Java and used it to run our experiments.

Applying our experiments with other programming languages than just Java might result in valuable results. Refactoring opportunities are greatly dependent on coding conventions, which differ per language. Other languages might result in different results, which might result in different insights regarding the resolution of duplication problems found. We prioritized refactoring efforts based on Java, which might differ from the prioritization that can result from running our experiments with other programming languages.

In this study we focused only on the object-oriented paradigm because of the shared concepts. However, the problems that come with duplication in source code also appears in different programming languages. Because of that, more insights could be obtained when looking into automated refactoring opportunities for other paradigms. For instance, it might be valuable to look into the opportunities to reduce duplication in the functional domain. Dealing with code clones in the functional paradigm is pretty much an unexplored field and might hold valuable results.

8.2.6   Running our experiments with a different set of software projects

In our experiments we chose to use a GitHub based software corpus. This corpus contains a diversity of projects of many different sizes. We noticed that there is a lot of difference in the quality of these software systems: some systems had a lot of duplication and some did not have any duplication.

We think that there is value in running our experiments with different corpuses. We would, for instance, be interested in the results for industrial projects instead of open source software systems. We are curious to see whether the distributions are comparable, or whether they show large differences.

Furthermore, we think that there is value in running our experiments with a set of larger sized and more professionally developed open source systems. We noticed that in our corpus there were a lot of projects that have only few commits and contributors. We think it would be valuable to for instance perform our experiments with a set of larger open source systems, like the systems in the Apache ecosystem.

8.2.7 Automatically refactoring code that is duplicated with a library

One of the clone detection tools we analyzed in this study is Sisyphus **??**. This clone detection tool searches a codebase for methods that are part of the Java standard libraries. There are definite automated refactoring opportunities here, automatically switching to the library implementation if available.

This doesn't have to be limited to just the Java standard library. Although using a new library for a single cloned library, we can also search the libraries the project are already uses. CloneRefactor is an excellent tool to expand for finding such clones with external libraries and refactoring the source code to use the library implementation if available. This is because CloneRefactor has scripts to gather all dependent libraries for Maven projects. It loads these libraries in order to be able to resolve symbols. If this would be expanded not only to resolve symbols but also to consider the clones between the analyzed project and its dependencies, we could find such refactoring opportunities.

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance. I would first like to thank my supervisor, Dr. A.M. Oprescu , whose expertise was invaluable in the academic writing and mathematical aspects of this thesis.

Next, I would like to thank my peer student Sander Meester for always being there when I was stuck. Whenever I needed to test some ideas you would always provide realistic insights. The long conversations about the problems I have been dealing with in my thesis would not have been solved in the same way if it wasn't for you. Your proofreading efforts helped take my writing to the next level.

Finally, I would like to acknowledge my colleagues from my internship at SIG for their wonderful collaboration. You supported me greatly and were always willing to help me.

Bibliography

[1] M. Fowler, *Refactoring: improving the design of existing code*, Second. Addison-Wesley Professional, 2018.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] M. Bruntink, A. Van Deursen, R. Van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code", *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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*, Oct. 2014, pp. 32–37. DOI: 10.1109/IWSM.Mensura.2014.19.

[17] 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.

[18] S. Jarzabek and Y. Xue, "Are clones harmful for maintenance?", in *Proceedings of the 4th International Workshop on Software Clones*, ser. IWSC '10, New York, NY, USA: 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.

[19] 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.

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

[21] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K Words, "Aries: Refactoring support environment based on code clone analysis.", in *IASTED Conf. on Software Engineering and Applications*, 2004, pp. 222–229.

[22] 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.

[23] A. Alwaqfi, "A refactoring technique for large groups of software clones", PhD thesis, Concordia University, 2017.

[24] Z. Chen, Y.-W. Kwon, and M. Song, "Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution", *Journal of Software: Evolution and Process*, vol. 30, no. 10, e1951, 2018.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

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

[41] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability", in *6th international conference on the quality of information and communications technology (QUATIC 2007)*, IEEE, 2007, pp. 30–39.

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

[43] J. Visser, S. Rigal, G. Wijnholds, P. van Eck, and R. van der Leek, *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.", 2016.

[44] G. A. Campbell, "Cognitive complexity-a new way of measuring understandability", *Technical Report. SonarSource SA, Switzerland*, 2017.

[45] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions", *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 589–618, 2016.

[46] S. H. Costello, "Software engineering under deadline pressure", *ACM SIGSOFT Software Engineering Notes*, vol. 9, no. 5, pp. 15–19, 1984.

[47] R. D. Austin, "The effects of time pressure on quality in software development: An agency model", *Information systems research*, vol. 12, no. 2, pp. 195–207, 2001.

[48] H. Shah, M. J. Harrold, and S. Sinha, "Global software testing under deadline pressure: Vendor-side experiences", *Information and Software Technology*, vol. 56, no. 1, pp. 6–19, 2014.

[49] M. Allamanis, E. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names", Aug. 2015, pp. 38–49. DOI: 10.1145/2786805.2786849.

[50] U. Alon, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code", *arXiv preprint arXiv:1808.01400*, 2018.

Appendix A

Non-crucial information