Improving Software Maintainability through Automated Refactoring of Code Clones

Simon Baars
simon.mailadres@gmail.com

30 June 2019, ?? pages

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

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

Abstract

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.

Update the abstract

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 [**fowler2018refactoring**, **wake2004refactoring**]. However, most of these methods still require manual assessment of where and when to apply them. Because of this, refactoring takes up a signification portion of the development process [**lientz1978characteristics**, **mens2004survey**], or does not happen at all [**mens2003refactoring**]. For a large part, refactoring requires domain knowledge to do it right. However, there are also refactoring opportunities that are rather trivial and repetitive to execute. In this thesis, we take a look at the challenges and opportunities in automatically refactoring duplicated code, also known as "code clones". The main goal is to improve maintainability of the refactored code.

Duplication in source code is often seen as one of the most harmful types of technical debt. *"Clones are problematic for the maintainability of a program, because if the clone is altered at one location to correct an erroneous behaviour, you cannot be sure that this correction is applied to all the cloned code as well. Additionally, the code base size increases unnecessarily and so increases the amount of code to be handled when conducting maintenance work"* [**ostberg2014automatically**]. Bruntink et al. [**bruntink2005use**] show that code clones can contribute up to 25% of the code size.

In this study, we use refactoring techniques to automatically reduce duplication in software systems. This allows us to obtain before- and after-refactoring snapshots of software systems. We use software maintainability metrics to measure the impact of refactoring clones. This way we can determine better definitions of clones. We also look into what variability we can allow between code fragments to still consider them clones, while still improving maintainability when refactoring these clones. Futhermore, we look into the thresholds that should be used while detecting clones to find clones that should be refactored.

We perform several quantitative experiments on a large corpus of open source software to collect statistical data. With these experiments we map the context of clones: where they reside in the codebase and what the relation is between duplicate fragments. We use the results to find appropriate refactoring opportunities for specific clones. We then automate the process of applying such refactorings, to be able to measure the impact on maintainability when refactoring clones found by certain definitions and thresholds.

1.1 Problem statement

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

1.1.1 The problem

The maintainability of a codebase has a large impact on the time and effort spent on building the desired software system [**bakota2012cost**, **munson1978software**]. The maintainability of software is one of the factors to be kept under control in order to avoid major delays and unexpected costs as a result of a software project [**fowler2018refactoring**]. One factor that has a major impact on the maintainability of a software system is the amount of duplicate code present in a codebase [**heitlager2007practical**, **fowler1999refactoring**].

The process of improving maintainability through the refactoring of duplicate code is time consuming and error-prone. This process mainly consists of these two aspects:

- Find refactoring candidates, either tool-assisted or manually.
- Refactor identified candidates, either tool-assisted or manually.

There are tools that assist in the process of finding duplicate code, but none of these tools identify all duplication problems. Often, these clone detection tools result in false-positives and false negatives [**roy2007survey**]. We define false negatives as clones that can and should be refactored but are not found due to the configuration by which they are detected. We define false-positives as clones that do not improve the system maintainability when refactored. Many false negatives are due to the thresholds that are used. Many clone detection tools [**sajnani2016sourcerercc**, **svajlenko2016bigcloneeval**] only consider clones that have a minimum of 6 lines. Using such an approach is bound to result in many false negatives, but using a lower threshold is shown to increase the number of false-positives.

A study by Batova et al. [**bavota2012does**] shows that the process of refactoring often leaves side effects in the code. This study reports that refactorings involving hierarchies (e.g., pull up method) , tend to induce faults very frequently. *This suggests more accurate code inspection or testing activities when such specific refactorings are performed.* Such specific refactorings often have to be used when dealing with duplication in software [**fowler2018refactoring**, **fontana2015duplicated**]. Because of that, refactoring code clones has been empirically proven to cause bugs or other side effects in code.

1.1.2 Research questions

There is a lot of research on the topic of code clone detection. This research often results in tools that can detect clones by several clone type definitions. However, there is no research yet that looks into how these definitions align with refactoring opportunities. We will align clone type definitions as used in literature [**roy2007survey**] with their corresponding refactoring methods [**fowler2018refactoring**]. For this, we answer the following research question:

> **Research Question 1:**
> How can we define clone types such that they **can** be automatically refactored?

As a result, we expect to formulate clone type definitions that can be refactored. On the basis of this results we can perform analyses on the context of clones by these definitions. The context of a clone (location, relation between clones, etc.) has a big impact on how a clone should be refactored. We will create categories by which we map the context of clones and perform a statistical analysis on this. This results in the following research question:

> **Research Question 2:**
> How can we prioritize refactoring opportunities based on the **context** of clones?

We expect this research question to result in a priorization of refactoring opportunities: *with what refactoring method can what percentage of clones be refactored?* As a result of these first two research questions we expect to have clone type definitions that can be refactored together with a prioritization of the refactoring methods that can be used. On the basis of this we expect to be able to build a script that can automatically refactor the highest prioritized clones. With this script, we expect to be able to answer our final research question:

> **Research Question 3:**
> What are the discriminating factors to decide when a clone **should** be refactored?

Not in all cases will refactoring duplicated code result in a more maintainable codebase. Because of that, we compare the refactored code to the original code and measure the difference in maintainability. To do this, we will use a practical model consisting of metrics to measure maintainability [**heitlager2007practical**]. Based on this, we can look into what *thresholds* result in better maintainable code when refactored. These thresholds consist of the size of clones and the variability we can allow between cloned fragments to still consider them clones.

1.1.3 Research method

We perform an **exploratory** study to look into the opportunities to automatically refactor code clones. To do this, we combine knowledge from literature with our own experience to develop definitions for

refactorable clones. We also develop a tool to detect, analyze and refactor such clones. Using this tool, we perform **quantitative** experiments in which we statistically collect information about duplication in open source software. In these experiments we control several variables to see their impact on the results. During this process we explore concepts and develop understanding, because of which decisions in the study design are based upon the results of the experiments.

1.2 Contributions

Many studies report that code clones negatively affect maintainability [**heitlager2007practical**, **monden2002software**, **juergens2009code**, **chatterji2013effects**]. However, no studies yet show in what cases code clones can reduce maintainability in source code. Refactoring often includes tradeoffs between design alternatives. With some code clones, the refactored alternative is less maintainable than keeping the duplication [**kapser2006cloning**, **aversano2007clones**, **hotta2010duplicate**, **kim2005empirical**, **krinke2007study**, **saha2010evaluating**]. In this study, we analyze the maintainability of refactored code clones, in order to improve the suggestion of code clones that should be refactored. This assists with both the identification and refactoring of code clones.

1.2.1 Identification

There are many tools to detect clones. The goal of most of these tools is to assist developers in reducing duplication in their code, i.e. assisting in the refactoring process. The problem is that these tools have no limited insight on the impact of refactoring such clones on the design of the software. In this study, we can analyze a before- and after-refactoring snapshot of the code to determine the impact. A higher maintainability after refactoring increases the support for the clone being a true-positive. This way the results of our study can support the clone identification process.

1.2.2 Refactoring

The tool that results from this research can aid in the process of applying refactorings to clones. The tool will only apply a refactoring if the clone is refactorable and the maintainability of the source code increases as a result of applying the refactoring. The tool applies only refactorings that do not, in any way, influence the functional correctness of the program. Because of this, potential bugs as a result of refactoring can be avoided [**bavota2012does**].

1.3 Scope

In this study, we perform research efforts to be able to detect refactorable clones. We will apply refactoring techniques to a subset of these clones and analyze the maintainability of the resulting source code.

There is a lot of study on how what metrics to consider to measure maintainability. In this study, we focus solely on the practical maintainability model by Heitlager et al. [**heitlager2007practical**]. We consider the maintainability scores described in that paper as an sufficiently accurate indication of maintainability. This will be used to quantitatively determine the impact on maintainability when applying refactoring techniques to code clones.

Applying refactoring often entails creating new method declarations, class declarations, etc. Each of these declarations needs to have a name. Because the quality of the name is not included in the maintainability metrics we use, finding appropriate names for automatically refactored code fragments is out of the scope for this study. For our automated refactoring efforts, we will use generated names for these declarations.

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

1.4 Outline

In Chapter **??** we describe the background of this thesis. In Chapter **??** we list shortcomings with clone definitions from literature and propose a set of clone type definitions which can be automatically

refactored. In Chapter **??** we propose a tool to detect, analyze and automatically refactor such clones that can be refactored. Using this tool we perform a set of experiments, of which the results are shown in Chapter **??** and discussed in Chapter **??**. Chapter **??** contains the work related to this thesis. Finally, we present our concluding remarks in Chapter **??** 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. [**bruntink2005use**], Roy et al. [**roy2007survey**] and Jiang et al. [**jiang2007deckard**]. We created a summary of these concepts, which can be found in Table **??**.

Symbol	Meaning	Definition	Description	Properties
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 (R):** The range this token spans. **Category (C):** Identifier, Keyword, Literal, Separator, Operator, Comment or Whitespace.
N	Node [**jiang2007deckard**]	Set of tokens.	A statement or declaration node in the AST of a codebase.	**Range (R):** The range this node spans.
I	Clone instance [**bruntink2005use**, **roy2007survey**]	Set of cloned nodes.	A code fragment that appears in multiple locations.	**File (F):** The file in which this clone instance is found.
C	Clone class [**bruntink2005use**, **roy2007survey**]	Set of clone instances.	A set of similar code fragments in different locations. Each of these code fragments is called a "clone instance".	-
S	Clone class collection [**bruntink2005use**]	Set of clone classes.	All clone classes that have been found for a certain software project.	-

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

A **range** denotes the line and column at which a code fragment starts and ends. In this study, we use the symbols given in this table in our formulas.

2.1.1 Example

This section gives an example to make the terminology of Table **??** more concrete. The code fragment in Figure **??** 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
2 doA();
3 doB();
4 doC();
```

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

```
1 // File3.java
2 doA();
3 doB();
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 **??**. This tree structure is displayed in Figure **??**.

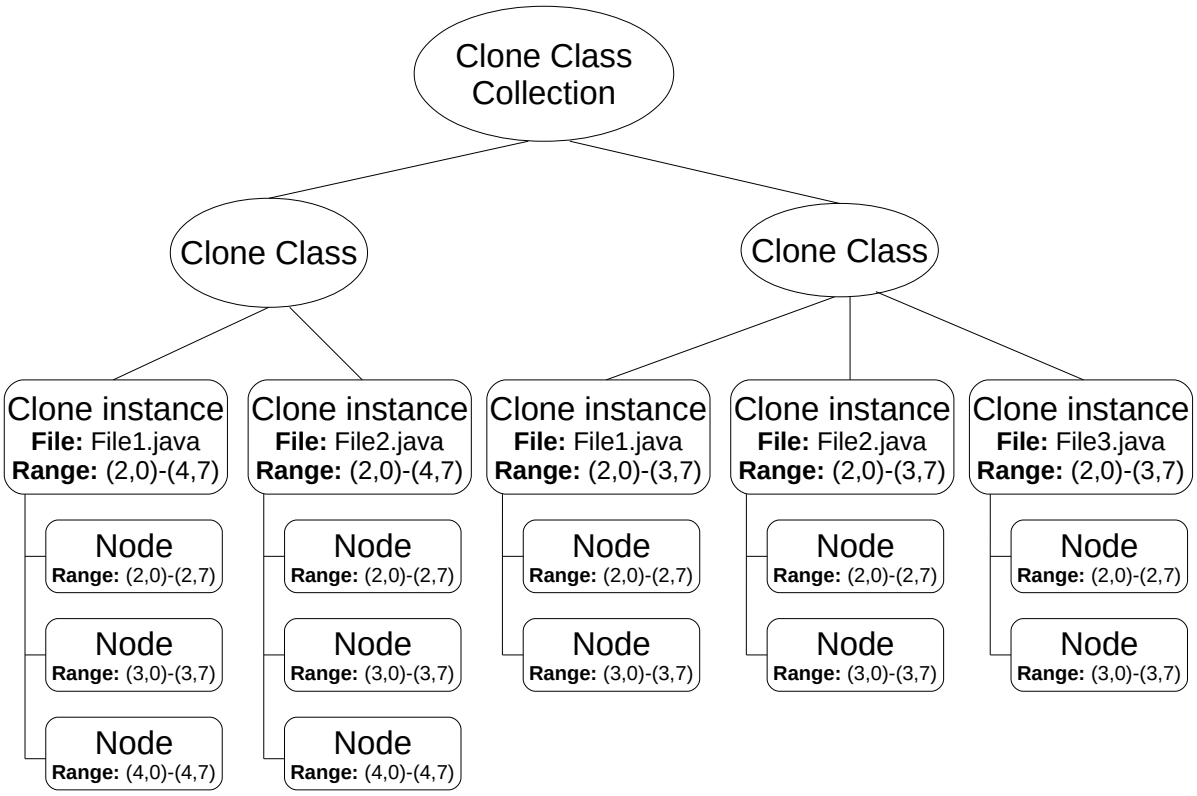

Figure 2.2: Representation of Figure ?? using the terminology introduced in Table ??

2.2 Clone Types

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

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

Type IV: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

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

2.2.1 Type 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 [**kodhai2013method**] looks into the distribution of the different types of clones in several open source systems (see table 6 of his study). It becomes apparent that type 4 clones exist way less in source code than all of the other types of clones. For instance, for the J2sdk-swing system he finds 8115 type 1 clones, 8205 type 2 clones, 11209 type 3 clones and only 30 type 4 clones. Because of that, we can conclude that type 4 clones are relatively less relevant to study.

2.3 Refactoring methods

Martin Fowler recently released the second edition of his Refactoring book [**fowler2018refactoring**]. In this book 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"* [**fowler1999refactoring**, **fowler2018refactoring**]. He describes several methods to deal with duplication, dependent on the context of the clone. In this section we describe the methods which we use in this study.

2.3.1 Extract Method

The main method for dealing with duplication in source code, as described by Martin Fowler, is to extract the duplication to a common place. This refactoring method is called "Extract Method". Several studies have already concluded that most duplication in source code is found in the body of methods [**lozano2007evaluating**, **white2016deep**, **bergman2004ethnographic**]. Method extraction can move matching functionality in method bodies to a common place, namely a new method. An example of this procedure is displayed in figure [**fig:extractmethod**].

```
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 a clone class 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. [**fontana2015duplicated**] 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" [**fowler2018refactoring**]. 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 [**mohnen2002interfaces**]. 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 [**kodhai2013method**] if all occurrences of the clone reside in the same class. If a method level clone is duplicated among classes in the same inheritance structure, we might need to pull-up a method in the inheritance structure. If instances of a method level clone are not in the same inheritance structure, we might need to either make a static method or create an inheritance structure ourselves. So not only a single instance of a clone has a context, but also the relationship between individual instances in a clone class. This is highly relevant to the way in which the clone has to be refactored.

2.4 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 differentiate between internal and external classes to see what impact they have on the refactoring process.

you don't consider external right?

2.4.1 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.

2.5 (Object-Oriented) Programming Languages

In this section we describe the relevant parts of the programming languages we research and experiment on.

2.5.1 AST

An Abstract Syntax Tree (AST) is a tree consisting of an hierarchical representation of the source code of a program. Detecting code clones on the AST of a program allows for a deeper understanding of the concepts that are being analyzed. Having an AST, we know how concepts in the code are related. This helps to determine in what classes and methods cloned code is found.

Having access to a programs' AST also helps in the process of refactoring. Moving AST nodes rather than textual modifications reduces error margins because the tree structure stays intact.

The AST consists of nodes of the following types (examples are for Java):

- **Declarations**: Method-declaration, class-declaration, etc.
- **Statements**: If-statement, switch-statement, expression-statement, etc.
- **Expressions**: Variables, literals, method calls, variable assignments, etc.
- **Types**: Primitives, reference types, void, etc.
- **Clauses**: Catch-clause, else-clause, etc.

2.5.2 Inheritance

Object-Oriented programming languages use inheritance to model real-world relations between data concepts. Inheritance allows an object to inherit all functionality from another object. For instance, a "Car" object would inherit functionality and data from generalized concepts, such as "Vehicle". Using inheritance it is possible to reduce duplication, as multiple objects can inherit common functionality.

When looking at the inheritance structure of a program, we can get a deeper understanding of a programs' architecture. The use of inheritance to group common functionality and data is called "Abstraction". Duplication in source code can be a result of an inadequate use of abstraction. Because of that, refactoring code clones might require to create such abstractions of common functionality.

2.5.3 Code Quality

Different programming languages have different ways to measure code quality. For object-oriented programming languages this largely comes in the form of code smells (patterns that should be avoided) and design patterns (patterns which may improve code design and comprehensibility when used correctly).

2.5.3.1 Code Smells

Code smells are patterns in code that should be avoided, because they have a negative effect on system design. Duplicate code is, among many others, one example of a code smell. Having many code smells present in source code can significantly increase maintainence costs of the source code or even render a software system unmaintainable. This increases technical debt: a debt present in the system the will have to be repaid at a later point of time. Such technical debt often comes at the expense of developing new features, fixing bugs and other aspects surrounding a programs functional behaviour.

2.5.3.2 Design Patterns

Most code smells have design patterns that can be used to mitigate the smell. For instance, duplicate code can be mitigated by using appropriate abstractions or refactoring duplicate code to a common place. Design patterns differ from refactoring methods in the sense that they are more about an architectural pattern rather than a certain kind of code modification.

2.5.3.3 Maintainability Metrics

Maintainability metrics are features of source code by which an indication of the maintainability of the source code can be measured. Heitlager at al. [**heitlager2007practical**] propose a maintainability model that categorize the results of metrics into different "score" categories. For instance, with duplication, if 0-3% of the project is duplicated, the project will get the highest score on a scale of 1 to 5 (only

integers, no floating point numbers). Such a maintainability model is intuitive for measuring the quality of a software system, especially large software systems with a lot of source code to base the metrics on. However, this maintainability model [**heitlager2007practical**] lacks in measuring fine-grained changes: most small changes will not fall into a different category. Because of that, this model is less suitable for measuring the impact of single refactorings.

2.5.4 Java

We run all our experiments on projects written in Java. Because of that, we get some information that is specific to Java systems. Additionally, we perform automated refactorings on Java systems, which requires the use of Java language concepts. These are explained in this section. Many of these concepts also exist in other programming languages, but often not in all.

2.5.4.1 Interfaces

Java uses the concept of interfaces to loosely couple a specification and its implementation. Interfaces are specifications about what a (group of) objects do. Often, methods don't need an entire object, but just use a few properties of it. Because of that, Java recommends to program to an interface rather than an implementation. This means that we should use abstractions of expected functionality rather than complete implementations of functionality. This can make switching implementations at a later point in time easier.

Often, duplication is the result of an inappropriate use of abstractions. Two methods might describe operations on types following the same contract, but because of a lack of abstraction the programmer may choose to clone such methods. This will result in duplicated methods with minor modifications to match a different implementation. Such duplicates can be mitigated by using proper abstractions, of which one option is to program methods against interfaces (contracts of expected functionality) rather than their full implementations.

2.5.4.2 Packages

Packages (sometimes called "modules" in other programming languages) are hierarchical groupings of related concepts. For instance, all UI logic in an application might go into the "ui" package. The "ui" package can have a subpackage named "keyhandling" for all keyhandling operations. These packages contain package-level declarations, like classes and interfaces. Names of class-level declarations must be unique within the package. However, duplicate declaration names may exists in subpackages and parent packages.

In Java, all declarations used from outside the package a declaration is in, must be *imported*. Importing a declaration from outside the current package goes by the fully qualified identifier (FQI) of the declaration. The FQI of a declaration gives an unique identifier for a symbol in a codebase. For instance, if our "keyhandling" package would contain a "KeyHandler" class, its fully qualified identifier would be "ui.keyhandling.KeyHandler".

2.5.4.3 Visibility

In Java, we can protect declarations from being used in scopes in which they are supposed to be used. If a method should only be used within the class itself, we can give it a "private" visibility. If a method is to be used by its children in its inheritance hierarchy, we can set its visibility to "protected". If a method is supposed to be part of the public contract of an object, we can set it to "public" visibility. Alternatively, there is "package" visibility, for declarations that should only be referenced within the same package. This allows control over what parts of an application should be able to access what declarations.

Chapter 3

Defining refactoring-oriented clone types

In Section **??** we introduced the four clone types as defined in literature. These simple definitions are suitable for analysis of a codebase. Their detection results in simple to understand numbers to argue about a codebase. However, these clone types have a few flaws which makes it hard to argue to what extend two fragments of code can and should be refactored. For each of type 1-3 clones [**roy2007survey**] 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 [**saini2018towards**] outlines different clone detection tools and compares their results for each of type 1-3 clones. Even though they operate on the same type definitions, the tools used in this study yield different results.

3.1 Shortcomings of clone types

Clone types 1-3 (further explained in Section **??**) allow reasoning about the duplication in a software system. Clones by these definitions can relatively easily and efficiently be detected. This has allowed for large scale analyses of duplication [**livieri2007very**]. However, these clone type definitions have shortcomings which makes the clones detected in correspondence with these definitions less valuable for (automated) refactoring purposes.

In this section, we discuss the shortcomings of the different clone type definitions. Because of these shortcomings, clones found by these definitions are often found to require additional judgment whether they should and can be refactored.

3.1.1 Type 1 clones

Type 1 clones are *identical clone fragments except for variations in whitespace and comments* [**roy2007survey**]. 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 implemented as textual equality between code fragments (except for whitespace and comments) by most clone detection tools [**kamiya2002ccfinder**, **semura2017ccfindersw**, **roy2008nicad**, **svajlenko2016bigcloneeval**, **svajlenko2014evaluating**]. 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.

```java
package com.sb.game;

import java.util.List;

public class GameScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void addTen(int x) {
    x = x + 10;
    Notifier.notifyChanged(x);
    return x;
  }
}
```

```java
package com.sb.fruitgame;

import java.awt.List;

public class LoseScene
{
  public void addToList(List l) {
    l.add(getClass().getName());
  }

  public void concatTen(String x) {
    x = x + 10;
    Notifier.notifyChanged(x);
    return x;
  }
}
```

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

In the example in Figure **??**, we see a type 1 clone displaying two clone classes. Defining an automatic way to refactor these clone classes is nearly impossible, as both cloned fragments describe different functional concepts. The first cloned fragment is a method that adds something to a `List`. However, the `List` objects to which something is added are different. Looking at the `import` statement above the class, one fragment uses the `java.util.List` and the other uses the `java.awt.List`. Both happen to have an `add` method, but apart from that their implementation is completely different.

The second cloned fragment shows how equally named variables can have different types and thus yield different functional concepts. The cloned fragment on the left adds a specific amount to an integer. The cloned fragement on the right concatenates a number to a String.

This shows that not all type 1 clones can easily be automatically refactored. In section we describe an alternate approach towards detecting type 1 clones, which results in only clones that can be refactored.

3.1.2 Type 2 clones

Type 2 clones are *structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments* [**roy2007survey**]. 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.

```
1 public boolean redCircles
2           (List<Circle> circles){
3   return circles.stream()
4            .allMatch(Shape::isRed);
5 }
```

```
1 public Apple getEdibleApple
2           (Basket<Apple> basket){
3   return basket.getFruit()
4          .getApple(Fruit::notEaten);
5 }
```

Figure 3.2: Example of a type 2 clone with high variability between fragments.

The example in Figure **??** shows a type 2 clone class. Both methods are, except for their matching structure, completely different in functionality. They operate on different types, call different methods, return different things, etc. Having such a method flagged as a clone does not provide much useful information.

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

3.1.3 Type 3 clones

Type 3 clones are *copied fragments with further modification (with added, removed or changed statements)* [**roy2007survey**]. 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 the basis of a similarity threshold [**roy2008nicad**, **ragkhitwetsagul2019siamese**, **jiang2007deckard**, **semura2017ccfindersw**]. This similarity threshold has been implemented in different ways: textual similarity (for instance using Levenshtein distance) [**lavoie2011automated**], token-level similarity [**sajnani2016sourcerercc**] or statement-level similarity [**kamalpriya2017enhancing**].

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

3.2 Refactoring-oriented clone types

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

3.2.1 Type 1R clones

To solve the issues identified in Section **??**, we introduce an alternative definition: cloned fragments have to be both textually *and* functionally equal. Like type 1 clones, type 1R clones do not consider comments and whitespace. Therefore, T1R clones are a subset of type 1 clones.

We check functional equality of two fragments by validating the equality of the fully qualified identifier (FQI) for referenced types, methods and variables. If an identifier is fully qualified, it means we specify the full location of its declaration (e.g. `com.sb.fruit.Apple` for an `Apple` object).

3.2.1.1 Referenced Types

Many object-oriented programming languages (like Java, Python and C#) require the programmer to import a type (or the class in which it is declared) before it can be used. Based on what is imported, the meaning of the name of a type can differ. For instance, if we import `java.util.List`, we get the

interface which is implemented by all list datastructures in Java. However, importing `java.awt.List`, we get a listbox GUI component for the Java Abstract Window Toolkit (AWT). To be sure we compare between equal types, type 1R clones compare the FQI for all referenced types.

3.2.1.2 Called methods

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

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

3.2.1.3 Variables

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

```
1 public int addFive(int x){
2   return x + 5;
3 }
```

```
1 public String appendFive(String x){
2   return x + 5;
3 }
```

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

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

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

3.2.2 Type 2R clones

Type 2R clones are modelled after type 2 clones, which allow any change in identifiers, literals, types, layout, and comments. For refactoring purposes, this definition is unsuitable; if we allow any change in identifiers, literals, and types, we cannot distinguish between different variables, different types and different method calls anymore. This could render two methods that have an entirely different functionality as clones (as shown in Figure **??** previously).

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, meaning that each node found as cloned for type 1R will also be found as cloned for type 2R. Similar to type 1R, for type 2R we consider the fully qualified identifiers of type-, method- and variable-references. Additionally, type 2R clones allow variability in a controlled set of expressions and identifiers. The identifiers and expressions in which we allow variability must have one of the following properties:

- We allow a difference in identifiers between cloned fragments if it has no impact on the refactoring process.
- We allow a difference in expressions between cloned fragments if the expression can be extracted to a method argument when applying the "Extract Method" refactoring.

Allowing variability between expressions includes a tradeoff. When many expressions vary between cloned the refactored method might require many additional arguments. Because of that, we constrain this variability by a threshold, which is explained in more detail in Section **??**.

3.2.2.1 Allow any variability in some identifiers

When refactored, some identifiers have no detrimental effect on the design if they vary between cloned instances. This section explains several patterns of variability that we can allow between cloned fragements without changing the method by which the fragments can be refactored.

3.2.2.1.1 Declaration names The names of declarations describe the implementation of their body. Examples of declarations are:

- Class declaration
- Method declaration
- Interface declaration (not all languages have a separate declaration for these)
- Enumeration declaration (not all languages have a separate declaration for these)
- Annotation declaration (not all languages support these)
- Etc.

If two declaration bodies and signatures 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. Because of this, type 2R allows any variability in declaration names. However, this will only open up a good refactoring opportunity if the entire body and signature of these declarations are cloned.

3.2.2.1.2 Variable names Cloned code fragments using variables with different names can be refactored without a design tradeoff if the following conditions apply:

- The cloned variables are locally defined or the refactoring method used requires extraction to method argument anyways.
- The cloned variables have the same type.
- The cloned variables are used at the same places in cloned fragments.

Figure **??** shows an example where different variables do not create a tradeoff. Both clone instances in this example use different variables, but in the same places and with the same type. Because these variables are locally defined, the extraced method requires an extra argument for the variable anyways. If the second clone instances would be completely equal, so also use the same variable, the refactoring would be very similar (it would only differ by a single character).

```
1 String a = "a";
2 String b = "b";
3 doA(a);
4 doB(a);
5 doC();
6 doA(b);
7 doB(b);
```

```
1 String a = "a";
2 String b = "b";
3 doAandB(a);
4 doC();
5 doAandB(b);
```

Figure 3.4: Different variables in cloned fragments without a difference in tradeoff on system design when refactored.

The example in Figure **??** shows the same example as Figure **??**, however this time there is a tradeoff. The same variables are used, but they are not used in the same places in cloned fragments. In one fragment we use only one variable, in the other we use both. Because of that, we require an extra argument. Because of this, this variability falls under the threshold described in section , whereas the example in Figure **??** does not.

```
1 String a = "a";
2 String b = "b";
3 doA(a);
4 doB(a);
5 doC();
6 doA(a);
7 doB(b);
```

```
1 String a = "a";
2 String b = "b";
3 doAandB(a, a);
4 doC();
5 doAandB(a, b);
```

Figure 3.5: Different variables in cloned fragments without a difference in tradeoff on system design when refactored.

3.2.2.2 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 [**roy2009comparison**]. However, this does have a big impact on the meaning of the code fragment.

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:

$$\text{T2R Variability} = \frac{\text{Number of different expressions}}{\text{Total number of expressions in clone instance}} * 100 \tag{3.1}$$

In this formula, *number of different expressions* refers to the number of literals, variables and method calls that differ from other clone instances in a clone class. We divide this by the total number of literals, variables and method calls in the clone instance. Based on this threshold, we decide whether a clone should be considered for refactoring. A concrete example of applying this formula to calculate a threshold is given in **??**.

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

```java
// Original
void doABC(){
  doA();
  doB("abc");
  doC();
}

void doDEF(){
  doA();
  doB("def");
  doC();
}
```

```java
// Refactored
void doABC(){
  doThis("abc");
}

void doDEF(){
  doThis(s, "def");
}

void doThis(String letters){
  doA();
  doB(letters);
  doC();
}
```

Figure 3.6: Literal variability refactored.

3.2.2.2.2 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 **??**. 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 **??** can be a type 2R clone (dependent on the thresholds used), but is not a type 2 clone.

```java
// Original
void doABC(){
  doA();
  doB();
  doC();
}

void doADC(){
  doA();
  doD();
  doC();
}
```

```java
// Refactored
void doABC(){
  doThis(this::doB);
}

void doADC(){
  doThis(this::doD);
}

void doThis(Runnable r){
  doA();
  r.run();
  doC();
}
```

Figure 3.7: Method variability refactored.

3.2.3 Type 3R clones

Type 3 clones allow any change in statements, often bounded by a similarity threshold. This means that type 3 clones allow the inclusion of a statement that is not detected by type 1 or 2 clone detection. When looking at how we can refactor a statement that is not included by one clone instance but is in another, we find that we require a conditional block to make up for the difference in statements. See Figure **??** for an example of such a clone.

```
1  // Original
2  void doCwithA(){
3    int a = getA();
4    doC(a);
5  }
6
7  void multiplyA(){
8    int a = getA();
9    a *= 5;
10   doC(a);
11 }
```

```
1  // Refactored
2  void doCwithA(){
3    modifyA(false);
4  }
5
6  void multiplyA(){
7    modifyA(true);
8  }
9
10 void modifyA(boolean multiply){
11   int a = getA();
12   if(multiply) a *= 5;
13   doC(a);
14 }
```

Figure 3.8: Added statement between cloned fragments refactored.

In Figure **??** a single statement is added. This statement is found in between cloned lines. There is a *gap* of non-cloned lines in between two clone classes. The following rules apply to this gap:

- **The difference in statements must bridge a gap between two valid clones**. This entails that, different from type 3 clones, the difference in statements cannot be at the beginning or the end of a cloned block. It is rather somewhere within, as it must bridge two existent clones.
- **The size of the gap between two clones is limited by a threshold.** This threshold is calculated by taking the percentage of the number of statements in the gap over the number of statements that both clones that are being bridged span.
- **The gap may not span a partial block.** To make sure that the T3R clone can be refactored, we do not allow the gap to span a part of a block. This is further explained in Section **??**.

3.2.3.1 Gap threshold

For type 3R, the size of the gap between two clones is bounded by a threshold. This threshold is calculated by the following formula:

$$\text{T3R Gap Size} = \frac{\text{Number of nodes in gap}}{\text{Number of nodes in clones}} * 100 \tag{3.2}$$

In this formula, the "Number of nodes in gap" concerns the amount of non-cloned nodes that are in between both clone instances. The "Number of nodes in clones" is the amount of nodes that are in the clone instances surrounding the gap:

$$\text{Number of nodes in clones} = |I_{before}| + |I_{after}|. \tag{3.3}$$

As an example, consider the code fragment in Figure **??**.

```
1 int a = getA(); // N1-1
2 doC(a); // N1-2
```

```
1 int a = getA(); // N2-1
2 a *= 5; // N2-2
3 doC(a); // N2-3
```

Figure 3.9: Gapped clones.

When applying this formula with the example given in Figure **??**, we get a gap size of 50%:

$$\text{T3R Gap Size} = \frac{|\{N_{2-2}\}|}{|\{N_{2-1}\}| + |\{N_{2-3}\}|} * 100 = \frac{1}{2} * 100 = 50\% \tag{3.4}$$

3.2.3.2 The gap may not span a partial block

Apart from this threshold, the gap in between clones may not span over different blocks. Look at Figure **??** 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.

```
1 int a = getA(); // N1-1
2 while(a<1000) {
3   a *= 5; // N2-2
4 }
5 doC(a); // N1-2
```

```
1 int a = getA();
2 while(a<1000) {
3   a *= 5;
4   doB(a); // nested gap
5 } // gap between blocks
6 doD(a); // part of gap
7 doC(a);
```

Figure 3.10: Statements between clones in different blocks.

The reason for this is that it is not possible to wrap a partially spanned block in a single conditional statement. We could, however, use multiple conditional blocks (one for each block spanned), but due to the detrimental effect on the design of the code (as each conditional block adds a certain complexity), we decided not to allow this for T3R clones.

3.2.4 Clone types summarized

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

- **Type 1R:** Allows no difference between cloned fragments (both functionally and textually), 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 a controlled set of expressions. 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.

Regarding the nodes found as cloned by these clone type definitions, there exists a subset relation between all types:

$$ClonednodesT1R \subseteq ClonednodesT2R \subseteq T3R \tag{3.5}$$

Comparing these clone type definitions to the ones used in literature, type 1R clones are a subset of type 1 clones. However, this subset relation does not exist for type 2R and 3R clones. This is because these definitions allow variability in multi-token method calls, which type 2 and type 3 do not allow. Because of that, the following is true for these clone types:

$$\text{Cloned nodes} = T1R \subseteq T1, \text{ but } T2R \nsubseteq T2 \text{ and } T3R \nsubseteq T3 \tag{3.6}$$

3.2.5 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 the basis of which a fully qualified identifier can be created.

3.3 Suitability of existing Clone Detection Tools for detecting these clones

Is this still up to date?

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

1. **Should find clones in any context.** Some tools only find clones in specific contexts, such as only method level clones. We want to perform an analysis of all clones in projects to get a complete overview.
2. **Finds clone classes in control projects.** We assembled a number of control projects to assess the validity of clone detection tools.
3. **Can analyze resolved symbols.** When detecting the types proposed in section **??**, it is important that we can analyze resolved symbols (for instance a type reference). The rationale for this is further explained in **??**.
4. **Extensive detection configuration.** Detecting our clone definitions, as proposed in section **??**, require to have some understanding about the meaning of tokens in the source code (whether a certain token is a type, variable, etc.). The tool should recognize such structures, in order for us to configure our clone type definitions in the tool.

Table 3.1: Our survey on clone detection tools.

	(1)	(2)	(3)	(4)
Siamese [**ragkhitwetsagul2019siamese**]				✓
NiCAD [**roy2008nicad**, **cordy2011nicad**]	✓	✓		
CPD [**roy2009comparison**]	✓	✓		
CCFinder [**kamiya2002ccfinder**] D-CCFinder [**livieri2007very**]	✓	✓		
CCFinderSW [**semura2017ccfindersw**]	✓			✓
SourcererCC [**sajnani2016sourcerercc**] Oreo [**saini2018oreo**]	✓			✓
BigCloneEval [**svajlenko2016bigcloneeval**]	✓	✓		
Deckard [**jiang2007deckard**]	✓		✓	
Scorpio [**higo2013revisiting**, **kamalpriya2017enhancing**]	✓		✓	✓

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

Chapter 4

CloneRefactor

We bridged a gap between clone detection and refactoring by designing a tool that can detect clones by our refactoring-oriented clone types (chapter **??**). This tool, named CloneRefactor[1], performs a comprehensive context analysis on the detected clones. Based on this this context, CloneRefactor can automatically refactor a subset of the detected clones by applying transformations to the source code of the analyzed software project(s).

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

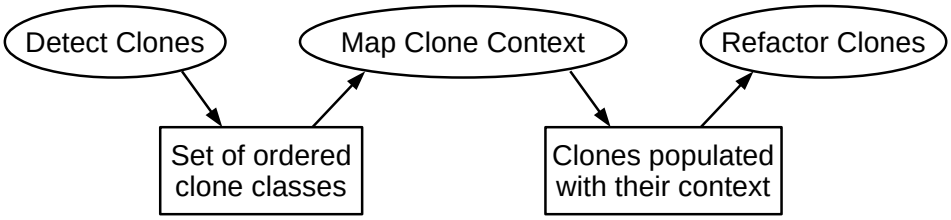

Figure 4.1: CloneRefactor overall process.

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

4.1 JavaParser

An important design decision for CloneRefactor is the usage of a library named JavaParser [**tomassetti2017javaparser**]. JavaParser is a Java library which allows to parse Java source files to an abstract syntax tree (AST[2]). 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. This is visible in our results, as displayed in Section **??**.

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

[2] An AST is a tree-representation of a source code file.

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

4.2 Thresholds

CloneRefactor operates on a set of thresholds to determine the validity of clones that are found. In general, these thresholds are:

- **Minimum Amount of Nodes**: The minimum amount of nodes that should be in each instance of cloned fragements for them to be considered a clone class.
- **Minimum Amount of Tokens**: The minimum amount of tokens that should be in each instance of cloned fragements for them to be considered a clone class.
- **Minimum Amount of Lines**: The minimum amount of lines that should be in each instance of cloned fragements for them to be considered a clone class.

When we analyze type 2R or type 3R clones we use the following additional threshold:

- **T2R Variability**: The percentage of variability we allow in variables, method calls and literals. How this threshold is calculated is explained in Section **??**.

When we analyze type 3 or type 3R clones we use the following additional threshold:

- **T3R Gap Size**: The size of the gap we allow between two valid clones. This is a percentage of the size of the gap against the size of both clones combined. How this threshold is calculated is further explained in Section **??**.

4.3 Clone Detection

To detect clones, CloneRefactor parses the AST aqcuired from JavaParser to an unweighted graph structure. On the 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.

The overall process regarding clone detection is displayed in Figure **??**. 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 **??**. On the basis of this graph, we detect clone classes and verify them using three thresholds (in order of importance):

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

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

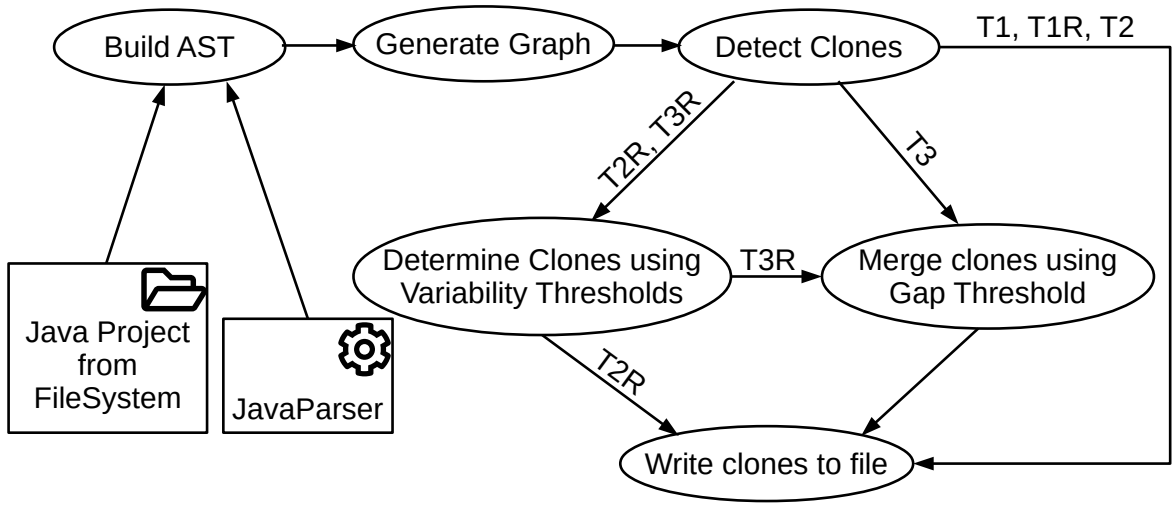

Figure 4.2: CloneRefactor clone detection process.

4.3.1 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 **??**).

4.3.1.1 Filtering the AST

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

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

4.3.1.2 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 **??**.

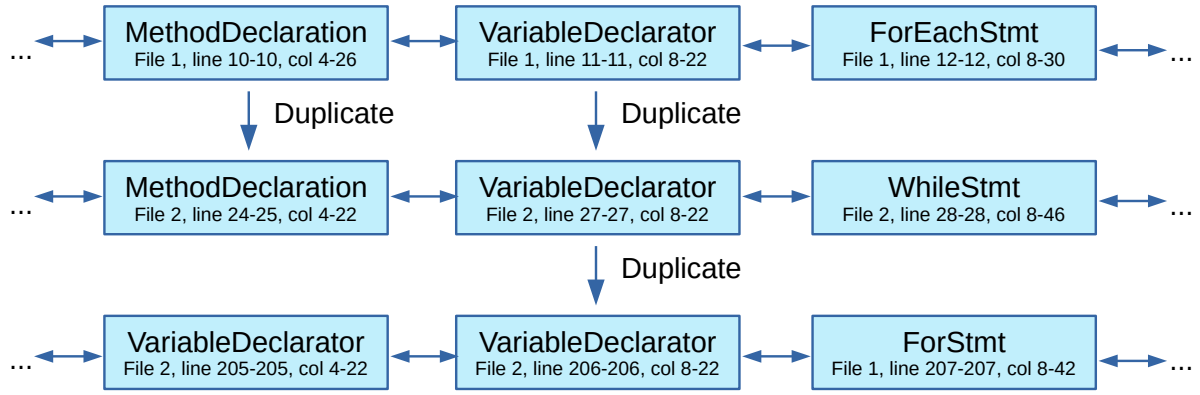

Figure 4.3: Abstract example of a part of a possible clone graph as built by CloneRefactor.

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

4.3.1.3 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 **??**. For **type 1** clones, CloneRefactor filters the tokens of a node to exclude its comments, whitespace and end of line (EOL) characters and then compares these tokens. For **type 2** clones, the tokens are further filtered to omit all identifiers and literals. **Type 3** clones do the same duplication comparison as type 2 clones.

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

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

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

4.3.1.4 Mapping graph nodes to code

The clone graph, as explained in Section **??**, 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 the body of each node. Look at Figure **??** for an example of how source code maps to AST nodes.

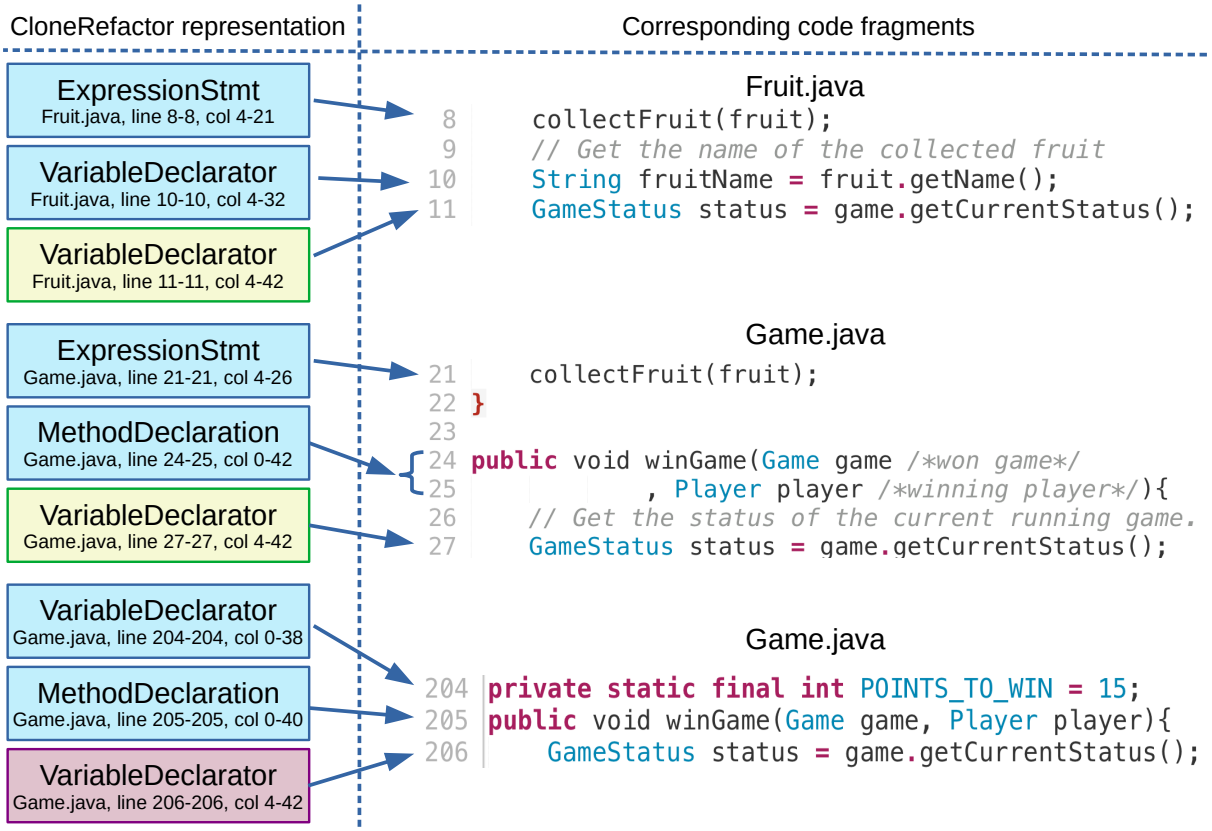

Figure 4.4: CloneRefactor extracts statements and declarations from source code.

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

4.3.2 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 [**fontana2012duplicated**]. 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 [**fontana2012duplicated**].

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 **??** to a clone graph as displayed in Figure **??**.

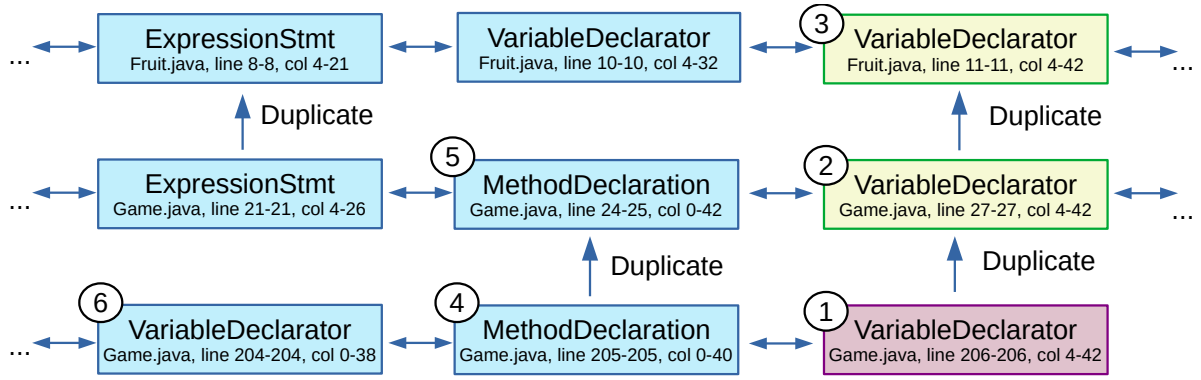

Figure 4.5: Example of a clone graph built by CloneRefactor.

Using the example shown in Figure **??** and **??** we can explain how we detect clones on the 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.

Explain the complexity of the algorithm: O(n)?

4.3.2.1 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 another clone. We define the subset relation between clones as follows:

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

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

$$S_{after} = S_{before} \setminus \{C_{existing} \subseteq C_{new} \mid C_{existing} \in S_{before}\} \tag{4.2}$$

Where S_{before} is the clone class collection containing all clones that are found up in until this point and C_{new} is the clone class that was just found. S_{after} is the clone class collection after

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 class of which the found clone class is a subset:

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

If the newly added clone is a subset of an existing clone, we do not add it to the set of clone classes. This way we avoid redundant clone classes being detected by CloneRefactor.

Think about an example

4.3.3 Validating the type 2R variability threshold

In the definition of type 2R clones (see Section **??**) we described how type 2R clones work on the basis of a variability threshold. This threshold is checked by CloneRefactor to assess the validity of found clone classes. Implementing such a threshold involves some important design decisions and has a lot of complexity. In this section we explain how CloneRefactor detects type 2R clones on the basis of this variability threshold. This process is done as a postprocessing step after clone detection. The type 2R clone detection process is described in more detail in Section **??**. In short, this process detects clones allowing for any variability between expressions. This postprocessing step then determines which (parts of) these clones are valid by the configure variability threshold.

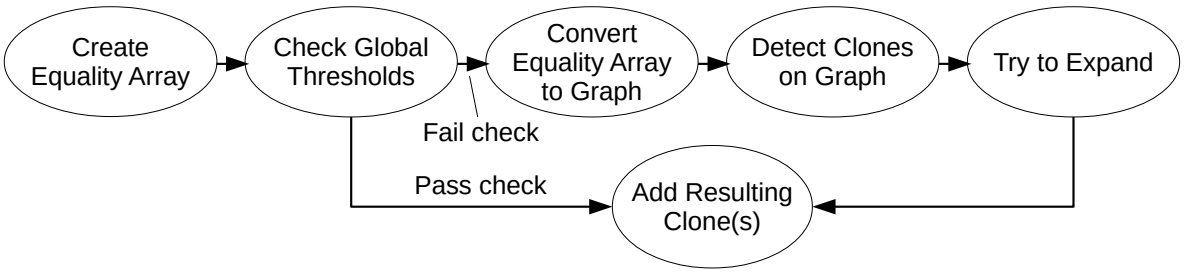

Figure 4.6: Process used to check the variability threshold for T2R clones.

Figure **??** shows the steps that CloneRefactor performs to find clones conforming with the type 2R variability threshold. Each of the following paragraphs will explain a step from this figure.

4.3.3.1 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 **??**.

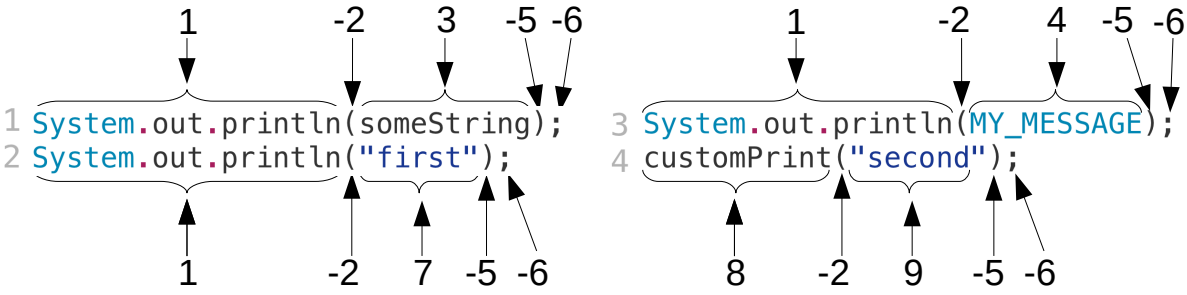

Figure 4.7: The conversion of code to an equality array.

The equality array for each of the lines in this example is shown in Table **??**.

Table 4.1: The equality arrays in our example Figure ??.

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 **??** 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 **??** shows their corresponding equality arrays.

4.3.3.2 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 **??** as follows:

This is not correct set theory. Sets have duplicates and have an order.

$$\text{T2R Variability} = \frac{|\{x > 0 \wedge y > 0 \wedge x \neq y \Rightarrow (x, y) \mid x \in E_1 \cup E_2, y \in E_3 \cup E_4\}|}{|\{x > 0 \wedge y > 0 \Rightarrow (x, y) \mid x \in E_1 \cup E_2, y \in E_3 \cup E_4\}|} * 100 \tag{4.4}$$

Where E_x refers to the equality arrays shown in Table **??** and x is the node number displayed in the same table. When we apply this equation to the example clone classes displayed in Figure **??**, we get the following sum:

$$\frac{|\{(3,4),(1,8),(7,9)\}|}{|\{(1,1),(3,4),(1,8),(7,9)\}|} * 100 = \frac{3}{4} * 100 = 75\% \tag{4.5}$$

Usage of the word adheres is questionable here...

So for the example given in Figure **??** we have a variability of 75%. 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 **??**).

When a found clone class does not adhere to the global thresholds as explained in the previous section, we need to determine whether it contains any valid subclones. Below, we explain a few cases of valid subclones that may exist within an invalid clone class.

```
1 doA(a, b, c);
2 doA();
3 doB();
4 doC();
```

```
1 doB(d, e, f);
2 doA();
3 doB();
4 doC();
```

Figure 4.8: One node in a type 2R clone has a high variability.

The first line of the cloned fragment shown in Figure **??** has a high variability with its cloned fragement. However, the rest of this method does not have any variability. The global thresholds could indicate a too high variability and thus render this clone invalid. However, in this case, it might still have a valid subclone.

```
1 doA();
2 doB();
3 doC();
```

```
1 doA();
2 doB();
3 doC();
```

```
1 doD();
2 doE();
3 doF();
```

Figure 4.9: One clone instance in a type 2R clone has a high variability.

Another example of high variability between clones, in which a valid subclone can be found, is displayed in Figure **??**. In this case, one clone instance has such a high variability that it shouldn't be refactored. In this case, the clone instance with high variability should be removed.

```
1 doA();
2 doB();
3 doC();
4 doC();
```

```
1 doD();
2 doA();
3 doB();
4 doD();
```

```
1 doE();
2 doE();
3 doA();
4 doB();
```

Figure 4.10: A small subset of nodes has a high variability.

Figure **??** shows an example of a clone class where the valid sequence inside the clone does not align.

If the check for global thresholds fails, we have to seek for valid clones within the clone class. In a single invalid clone class can be zero to many subclones. This requires an extensive search for such clones. This problem is very related to the problem of clone detection, except now it is within the boundaries of a single clone class except for an entire codebase. Because of that, just like the clone detection process, we build a clone graph and detect clones on it. This process is explained over the following sections.

4.3.3.3 Convert Equality Array to Graph

After the global threshold check has failed, we build a clone graph on the basis of the equality arrays. The process used for building this graph is the same as the process described in **??**.

```
1 doA(); //{1, -2}
2 doB(); //{3, -2}
3 doC(); //{4, -2}
4 doC(); //{4, -2}
```

```
1 doD(); //{5, -2}
2 doA(); //{1, -2}
3 doB(); //{3, -2}
4 doD(); //{5, -2}
```

```
1 doE(); //{6, -2}
2 doE(); //{6, -2}
3 doA(); //{1, -2}
4 doB(); //{3, -2}
```

Figure 4.11: Cloned code fragments from Figure ?? together with their equality arrays.

In Figure **??** we display clone classes and their (simplified) equality arrays. We can convert this to a graph, using a similar process as the graph creation process used for clone detection. In this case, duplicate relations represent nodes of which their equality arrays are within the variability threshold. In Figure **??** all relations are between equal equality arrays, but this does not always have to be the case. Large equality arrays, denoting statements that consist of many tokens, may have some variability and still be considered duplicates in this graph.

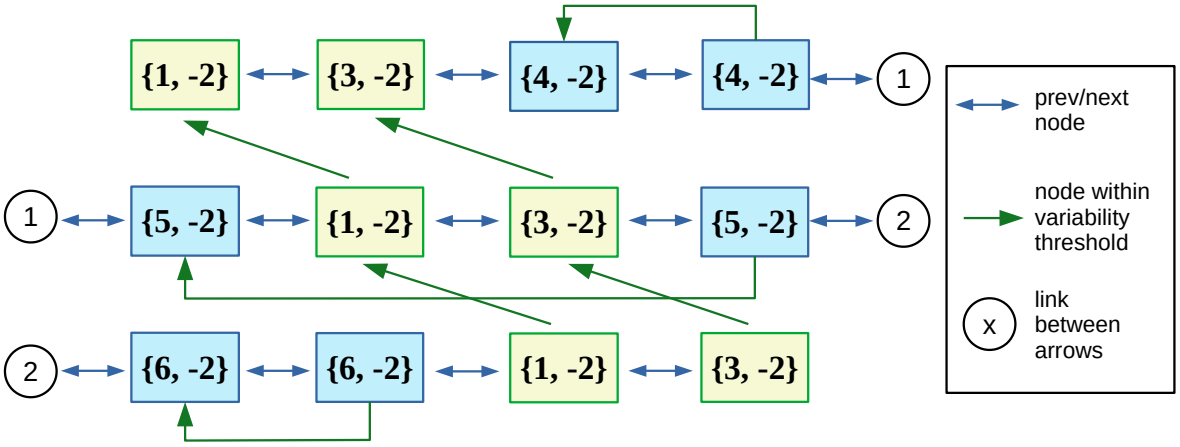

Figure 4.12: Graph representation of the code example displayed in Figure ??.

In this graph, the duplicate relations (green arrows) do not necessarily denote an exact match. The duplicate relations allow for variability in the equality arrays as long as it falls under the variability threshold. However, negative numbers in the equality array may never vary. This is because negative numbers are the tokens/expressions that we do not allow variability in because they cannot be refactored as easily:

$$E_1 \text{ can only be cloned with } E_2 \text{ iff } |E_1| = |E_2| \wedge \forall (x \in E_1, y \in E_2)\ x < 0 \vee y < 0 \Rightarrow x = y \tag{4.6}$$

4.3.3.4 Detect Clones on Graph

Similar to the clone detection process, we detect clones on the basis of the graph described in the previous section. The only difference is that, in this step, we do not remove clones that do not meet the thresholds (as explained in Section **??**). This is because the next step, explained in the next section, could potentially expand the found clones and thus make clones that would currently invalid by the thresholds valid.

Looking back at the example of the previous section, it would result in the following clone class collection:

```
1 doA(); // C1 I3          1 doD(); // C3 I2          1 doE(); // C2 I2
2 doB(); // C1 I3          2 doA(); // C1 I2          2 doE(); // C2 I1
3 doC(); // C4 I2          3 doB(); // C1 I2          3 doA(); // C1 I1
4 doC(); // C4 I1          4 doD(); // C3 I1          4 doB(); // C1 I1
```

Figure 4.13: Clone instances in Figure ?? as determined .

Where C denotes the clone class and I denotes the clone instance. The process by which these clones are found is equal to the clone detection process as described in Section **??**. In this example, we see the four clones found on the graph of Figure **??**. Three of these clone classes have a two clone instances, each consisting of a single nodes. The other clone class has three clone instances, each consisting of two nodes.

4.3.3.5 Try to Expand

The problem with the graph clone detection technique is that only single nodes that are within the variability threshold are considered duplicates of each other. However, single nodes that are not within this threshold could still be part of a clone class if the other cloned nodes are more towards the lower bound of the threshold.

```
1 doA(a); //{1, -2, 7, -3, -4}
2 doB(a); //{5, -2, 7, -3, -4}
3 doD(); //{8, -2, -3, -4}
4 doE(); //{9, -2, -3, -4}
5 doF(); //{10, -2, -3, -4}
6 doG(a); //{11, -2, 7, -3, -4}
```

```
1 doA(a); //{1, -2, 7, -3, -4}
2 doC(a); //{6, -2, 7, -3, -4}
3 doD(); //{8, -2, -3, -4}
4 doE(); //{9, -2, -3, -4}
5 doF(); //{10, -2, -3, -4}
6 doH(b); //{12, -2, 13, -3, -4}
```

Figure 4.14: Type 2R clones with their equality arrays.

As an example, consider the clone fragment in Figure **??**. In this example, we have 2 clone classes. In the "Try To Expand" step, we will check whether we can create a clone class with larger clone instances on the basis of the clones found in the previous step. We start with the largest clone class, measured in clone volume. We measure clone volume as the product of the amount of clone instances in a clone class and the amount of nodes in any of its instances.

In the example in Figure **??** the largest instance has a volume of:

$$3 \text{ nodes} * 2 \text{ clone instances} = 6 \tag{4.7}$$

Starting from this clone, we will try to expand it. We will include the previous node in the clone class and verify its variability threshold (node N_2). In this case, using the formulas shown in Section **??**, we can check whether this clone class would be valid by the variability threshold:

$$\frac{|\{(5,6)\}|}{|\{(5,6),(7,7),(8,8),(9,9),(10,10)\}|} * 100 = \frac{1}{5} * 100 = 20\% \tag{4.8}$$

Dependent on the actual variability threshold, the clone class would get expanded with node N_2. For this example, we assume the variability threshold is set to 15%. In such a case, including this node would not result in a valid clone class. However, we continue trying to expand this clone class until we have reached the first node of the originally cloned fragment (the cloned fragment that did not take the variability threshold into account). Thus, we now try to expand with $\{N_1, N_2\}$. We then get the following formula:

$$\frac{|\{(5,6)\}|}{|\{(1,1),(7,7),(5,6),(7,7),(8,8),(9,9),(10,10)\}|} * 100 = \frac{1}{7} * 100 = 14\% \tag{4.9}$$

This falls under the threshold of 15% so we expand the clone class. Thus, the clone class $\{N_3, N_4, N_5\}$ will become $\{N_1, N_2, N_3, N_4, N_5\}$. We cannot expand further because we have reached the end of the original cloned fragment. When we are done expanding to previous nodes, we will try to expand to next nodes. In this case, we check whether we can include N_6 into the clone class. If N_6 would be included, the variability threshold would be as follows:

$$\frac{|\{(5,6),(11,12),(7,13)\}|}{|\{(1,1),(7,7),(5,6),(7,7),(8,8),(9,9),(10,10),(11,12),(7,13)\}|} * 100 = \frac{3}{9} * 100 = 33\% \tag{4.10}$$

This does not fall within the variability threshold, so we do not include this node in the clone class. After we have checked a single clone class within the bigger cloned fragment for expansion opportunities, we go on with the next clone class (by volume). In this case, the other clone class within this fragment has become a subset of the clone class we just expanded. Because of that, this other clone class is discarded. For the example code fragment in Figure **??** the resulting clone class is $\{N_1, N_2, N_3, N_4, N_5\}$.

4.3.4 Checking for type 3 opportunities

As described in Section **??**, we define type 3R clones as gapped clones. This means that two existing clones may have a gap of non-gapped clones. We check for every found clone class whether it is a type 3R clone with another clone:

$$\forall(c_1 \in S)\ \exists(c_2 \in S)\ x \neq y \wedge isType3R(c_1, c_2) \tag{4.11}$$

Where S is the set of all found clone classes. isType3R(C_1, C_2) is a Boolean-valued function that returns whether two clone classes are type 3R clones of each other. Two clone classes are type 3R clones of each other if each of their instances are within a certain distance of each other:

$$\text{isType3R}(C_1, C_2) = \forall(i_1 \in C_1)\ \exists(i_2 \in C_2)\ Fi_1 = Fi_2 \wedge gap(i_1, i_2) < gap\_threshold \tag{4.12}$$

Where F is the file in which the clone instance is located. $gap\_threshold$ is the defined threshold percentage of the maximum gap size between two clone instances. gap(I_1, I_2) is the function that calculates the gap between two clone instances. This gap is calculated by dividing the amount of statements in the gap by the amount of statements that both clone instances span together:

$$\text{gap}(i_1, i_2) = \frac{|\{(Rn > Ri_1 \wedge Rn < Ri_2) \vee (Rn > Ri_2 \wedge Rn < Ri_1)\ |\ n \in nodes(Fi_1)\}|}{|i_1| + |i_2|} * 100 \tag{4.13}$$

Where F is the file in which the clone instance is located and R is the range of a clone instance or node. A range denotes the line and column at which a code fragment starts and ends. We define the partial order relationship on ranges as follows:

$$r_1 < r_2 \text{ iff } ELr_1 < BLr_2 \vee (ELr_1 = BLr_2 \wedge ECr_1 < BCr_2) \tag{4.14}$$

$$r_1 > r_2 \text{ iff } BLr_1 > ELr_2 \vee (BLr_1 = ELr_2 \wedge BCr_1 > ECr_2) \tag{4.15}$$

Where BL is the begin line of a range, EL is the end line of a range, BC is the begin column of a range and EC is the end column of a range.

4.3.4.1 Example

In this section we show an example of the identification of a type 3R clone.

```
1 //File1.java
2 doA(); // C1 I1
3 doB(); // C1 I1
4 doC();
5 doC(); // C2 I1
6 doD(); // C2 I1
```

```
1 //File2.java
2 doA(); // C1 I2
3 doB(); // C1 I2
4 doC(); // C2 I2
5 doD(); // C2 I2
```

Figure 4.15: Gapped clones.

In the example of Figure **??** we see two clone classes, which are separated by a single node. Using equation **??** we can calculate the size of the gap as follows:

$$\text{gap}(C_1\ I_1, C_2\ I_1) = \frac{|\{n_4\}|}{|\{n_2, n_3\}| + |\{n_5, n_6\}|} * 100 = \frac{1}{4} = 25\% \tag{4.16}$$

The same calculation is conducted for the other clone instances:

$$\text{gap}(C_1\ I_2, C_2\ I_2) = \frac{|\emptyset|}{|\{n_2, n_3\}| + |\{n_4, n_5\}|} * 100 = \frac{0}{4} = 0\% \tag{4.17}$$

If all resulting percentages are under the threshold, this clone will be added to the collection of found clones and all of its subsets will be removed.

4.4 Context Analysis of Clones

To be able to refactor code clones, CloneRefactor first maps the context of code clones. We define the following aspects of the clone as its context:

1. **Relation:** The relation of clone instances among each other through inheritance.
2. **Location:** Where a clone instance occurs in the code.
3. **Contents:** The statements/declarations of a clone instance.

We define categories for each of these aspects and enable CloneRefactor to determine the categories of clones.

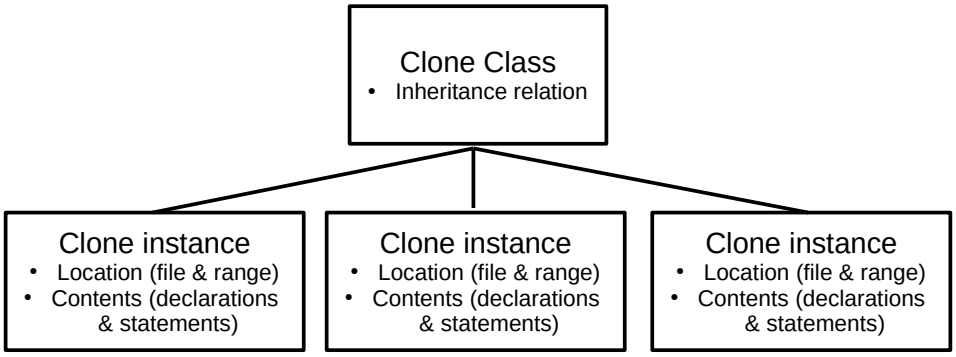

Figure 4.16: Abstract representation of clone classes and clone instances.

Fig. **??** shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is measured for each clone class. The location and contents of clones are measured for each clone instance.

4.4.1 Relation

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

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

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

We added the following categories, to gain more information about clones and reduce the number of unrelated clones:

1. **Same Interface**: All instances of the clone class are in a class or interface that have a common interface anywhere in their inheritance hierarchy.
2. **No Direct Superclass**: All instances of the clone class are in a class that does not have any superclass.
3. **No Indirect Superclass**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.

4. **External Ancestor**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.

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

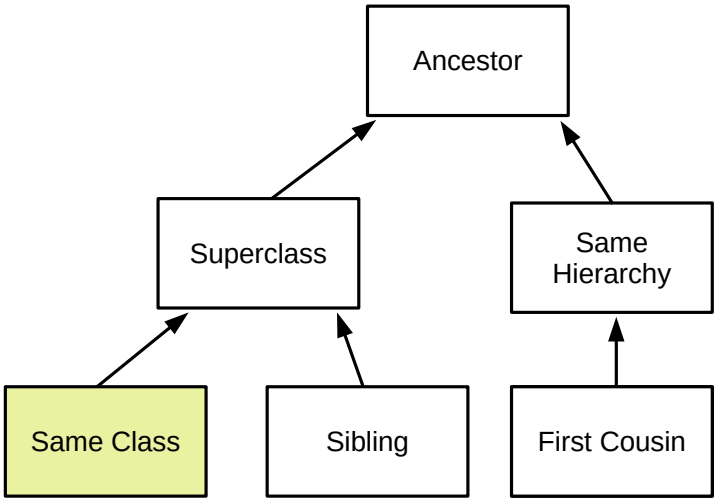

Figure 4.17: Abstract figure displaying relations of clone classes. Arrows represent superclass relations.

We separate these relations into the following categories, because of their related refactoring opportunities:

- **Common Class**: *Same Method*, *Same Class*
- **Common Hierarchy**: *Superclass*, *Sibling Class*, *Ancestor Class*, *First Cousin*, *Same Hierarchy*
- **Common Interface**: *Same Interface*
- **Unrelated**: *No Direct Superclass*, *No Indirect Superclass*, *External Superclass*, *External Ancestor*

Every clone class only has a single relation, which is the first relation from above list that the clone class applies to. For instance: all "Superclass" clones also apply to "Same Hierarchy", but because "Superclass" is earlier in above list they will get the "Superclass" relation. This is because the items earlier in the list denote a more favorable refactoring.

To check a the relation for a given clone class, we compare each node in a clone instance with the respective nodes in each other clone instance within a clone class. So for a clone class with x clone instances and each clone instance has y nodes, we perform:

$$\frac{x * (x + 1)}{2} * y \tag{4.18}$$

comparisons to determine the relation. Each of these checks will result in one of the relations from the list above. Of these relations, the item lowest in the list is going to be the relation of the clone class. So for instance, if a clone class has 15 nodes that denote a *Superclass* relation but 3 nodes are *Unrelated*, the clone class becomes *Unrelated*.

4.4.1.1 Common Class

The *Same method* and *Same class* relations share a common refactoring opportunity. Clones of both these categories, when extracted to a new method, can be placed in the same class. Both of these relations are most favorable for refactoring, as they require a minimal design tradeoff. Furthermore, global variables that are used in the class can be used without having to create method parameters.

Cloned nodes are flagged as *Same method* if these nodes are found in the same method. We define the method as the first method declaration that is encountered when following the parent nodes in the

ast for a given cloned node. Please note that a clone instance may not always be in a method, for which this predicate will fail.

Cloned nodes are flagged as *Same class* if these nodes are found in the same class. We define the class as the first class declaration that is encountered when following the parent nodes in the ast for a given cloned node.

4.4.1.2 Common Hierarchy

Clones that are in a common hierarchy can be refactored by using the "Extract Method" refactoring method followed by "Pull Up Method" until the method reaches a location that is accessible by all clone instances. However, the more often "Pull Up Method" has to be used, the more detrimental the effect is on system design. This is because putting a lot of functionality in classes higher up in an inheritance structure can result in the "God Object" anti-pattern. A god object is an object that knows too much or does too much.

Cloned nodes are flagged as *Superclass* if the classes in which these nodes are found are parent and child class of each other. Cloned nodes are flagged as *Siblings* if the classes in which these nodes are found all share the same parent and this parent class is not external. Cloned nodes are flagged as *First Cousin* if the classes in which these nodes are found all share the same grandparent and this grandparent class is not external.

Cloned nodes are flagged as *Ancestor* if the classes in which these nodes are found are recursively the parent of an antecedent. Cloned nodes are flagged as *Same Hierarchy* if the classes in which these nodes are found are all in the same inheritance hierarchy and not linked by any external classes.

4.4.1.3 Common Interface

Many object-oriented languages know the concept of "interfaces", which are used to specify a behavior that classes must implement. As code clones describe functionality and interfaces originally did not allow for functionality, interfaces did not open up refactoring opportunities for duplicated code. However, many programming languages nowadays support default implementations in interfaces. Since Java 7 and C#8, these programming languages allow for functionality to be defined in interfaces. Many other object-oriented languages like Python allow this by nature, as they do not have a true notion of interfaces.

The greatest downside on system design of putting functionality in interfaces is that interfaces are per definition part of a classes' public contract. That is, all functionality that is shared between classes via an interface cannot be hidden by setting a stricter visibility. Because of that, we favor all "Common Hierarchy" refactoring opportunities over "Common Interface".

To check whether two cloned nodes have common interfaces, we recursively walk all types the class or interface of the cloned nodes implement and extend. The common interface that is closest to the cloned node in terms of depth of recursion will be flagged as the refactoring candidate.

4.4.1.4 Unrelated

Clones are unrelated if they share no common class or interface in their inheritance structure. These clones are least favorable when looking at refactoring, because their refactoring will almost always have a major impact on system design. We formulated four categories of unrelated clones to look into their refactoring opportunities.

Cloned nodes are flagged *No Direct Superclass* if they are in classes that do currently not have a parent. This marks the opportunity for creating a superclass abstraction and placing the extracted method there. Cloned nodes are flagged *No Indirect Superclass* if any of their ancestors does not have a parent. In such a case, it would be possible to create such an abstraction for the ancestor that does not have a parent.

Cloned nodes are flagged *External Superclass* if they are in a class which has an external parent. Cloned nodes are flagged *External Ancestor* if one of their ancestors has an external parent. Both of these relations obstruct the possibility of creating a superclass abstraction. In such a case, an interface abstraction could be created to make their relation explicit.

4.4.2 Location

A paper by Lozano et al. [**lozano2007evaluating**] discusses the harmfulness of cloning. The authors argue that 98% are produced at method-level. However, this claim is based on a small dataset and based

on human copy-paste behavior rather than static code analysis. We decided to measure the locations of clones through static analysis on our dataset. We chose the following categories:

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

We check the location of each clone instance for each of its nodes. If any node reports a different location from the others, we choose the location that is lowest in above list. So for instance, if a clone instance has 15 nodes that denote a *Method Level* location but 3 nodes are *Class Level*, the clone instance becomes *Class Level*.

4.4.2.1 Method/Constructor Level Clones

Method/Constructor Level clones denote clones that span are found in either a method or contructor. A constructor is a special method that is called when an object is instantiated. Current clone refactoring studies only focus on clones at method level [**choi2011extracting**, **yue2018automatic**, **kodhai2013method**, **arcelli2013software**, **lin2014clonepedia**, **mandal2014automatic**, **balazinska2000advanced** **yongting2018detection**, **bouktif2006novel**, **fanqi2014using**, **devi2016study**]. This is because most clones reside at those places [**lozano2007evaluating**, **fontana2015duplicated**] and most of those clones can be refactored with a relatively simple set of refactoring techniques [**kodhai2013method**, **fontana2015duplicated**].

To detect whether a given clone instance is at Method Level, we check for each node in the clone instance whether it has an ancestor node that is a method declaration. A clone instance is considered "Method Level" if:

$$\forall(n \in I)\ Tn = \text{MethodDeclaration} \vee \exists(p \in ancestors(n))\ Tp = \text{MethodDeclaration} \tag{4.19}$$

Where T is the type of a node. *ancestors(n)* denotes all of ancestor nodes of a node. The ancestor nodes are recursively all the parents of a given node in the AST. Apart from this, we check that the MethodDeclaration ancestor of each clone instance is the same, to be sure the clone only spans a *single* method. Figure **??** shows an example of a method level clone and its recursive parents.

```
1 public class Class1 { // p3
2   public void doA() { // p2
3     if(isA()) // p1
4       doB(); // n1
5   }
6 }
```

```
1 \begin{javacode}
2 public class Class2 { // p3
3   public void doB() { // p2
4     if(isB()) // p1
5       doB(); // n2
6   }
7 }
```

Figure 4.18: Example of a method level clone and its parents.

4.4.2.2 Class/Interface/Enumeration Level Clones

Class/Interface/Enumeration Level clone instances are found inside the body of one of these declarations and optionally include the declaration itself. It can also be a clone instance that exceeds the boundaries of a single method. These clone instances can contain a fields, (abstract) methods, inner classes, enumeration fields, etc. These types of clones require various refactoring techniques to refactor. For instance, we might have to move fields in a inheritance hierarchy. Or, we might have to perform a refactoring on more of an architectural level, if a large set of methods is cloned.

We detect these clones in the same way we detect method level clones: we walk up the AST until we find the first node that indicates one of these categories.

4.4.3 Contents

Finally, we looked at what nodes individual clone instances span. We selected a set of catagories on the basis of empirical evaluation of a set of clones in our dataset. We selected the following categories to be relevant for refactoring:

1. **Full Method/Constructor/Class/Interface/Enumeration:** A clone that spans a full class, method, constructor, interface or enumeration, including its declaration.
2. **Partial Method/Constructor:** A clone that spans (a part of) the body of a method/constructor. The declaration itself is not included.
3. **Several Methods:** A clone that spans over two or more methods, either fully or partially, but does not span anything but methods (so not fields or anything in between).
4. **Only Fields:** A clone that spans only global variables.
5. **Other:** Anything that does not match with above-stated categories.

4.4.3.1 Full Method/Constructor/Class/Interface/Enumeration

These categories denote that a full declaration, including its body, is cloned with another declaration. These categories often denote redundancy and are often easy to refactor: one of both declarations is redundant and should be removed. All usages of the removed declaration should be redirected to the clone instance that was not removed. We check for clones in this category by checking that the first node in a clone instance is a declaration and the last node in the clone instance is the last node in the body of that declaration.

4.4.3.2 Partial Method/Constructor

These categories describe clone instances which are found in the body of a method or constructor. These clones can often be refactored by extracting a new method out of the cloned code. We detect such clones by checking that each node in the clone instance has a ancestor node that is a MethodDeclaration, but none of the nodes is a method declaration. We describe the procedure by with we recursively seek the parent nodes of a node in Section **??**.

4.4.3.3 Several Methods

Several methods cloned in a single class is a strong indication of implicit dependencies between two classes. This increases the chance that these classes are missing some form of abstraction, or their abstraction is used inadequately. We detect such clones by checking that each cloned node has a parent node that is a MethodDeclaration. This is the same process as described in equation **??**. Additionally, we check that the MethodDeclaration ancestor of the first node in the clone instance differs from the last, to be sure that the clone instance spans over at least two methods.

4.4.3.4 Only Fields

This category denotes that the clone spans over only global variables, fields that are declared outside of a method. This is to indicate data redundancy: pieces of data have an implicit dependency. In such cases, these fields may have to be encapsulated in a new object. Or, the fields should be somewhere in the inheritance structure where all objects containing the clone can access them. We check for this category by validating that all nodes in the clone instance are a VariableDeclarator. Because VariableDeclarators in methods would already fall into the "Partial Method" category, this includes only globally defined fields.

4.4.3.5 Other

The "Other" category denotes all configurations of clone contents that do not fall into above categories. Often, these are combinations of above states concepts. For instance, a combination of constructors and methods is clones, or a combination of fields and methods. Such clones indicate, like the "Several Methods", the requirement of performing a more architectural-level refactoring. These are often more complicated, especially when aiming to automate this process.

4.5 Automated Refactoring

After clone detection and context analysis, CloneRefactor applies refactorings to a subset of the identified clones. This section describes what clone classes are included in the refactoring and the refactoring methods CloneRefactor supports.

4.5.1 Method extraction opportunities

The most used technique to refactor clones is "Extract Method" (creating a new method on the basis of the contents of clones). However, method extraction cannot be applied in all cases. In some instances, more conditions may apply to be able to conduct a refactoring, if beneficial at all. CloneRefactor maps to what extend this refactoring method can be used to refactor the found clones automatically.

Because of this, we defined a set of categories of things that can obstruct a refactoring opportunity using the "Extract Method" technique. We defined the following categories:

- **Complex Control Flow**: This clone contains `break`, `continue` or `return` statements, obstructing the possibility of method extraction.
- **Spans Part Of A Block**: This clone spans a part of a statement.
- **Is Not A Partial Method**: If the clone does not fall in the "Partial method" category of Section **??**, the "extract method" refactoring technique cannot be applied.
- **Top-level Node Is Not A Statement**: The topmost node of a clone instance is not a statement.
- **Overlaps**: There is overlap within the clone class.
- **Can Be Extracted**: This clone is a fragment of code that can directly be extracted to a new method. Then, based on the relation between the clone instances, further refactoring techniques can be used to refactor the extracted methods (for instance "pull up method" for clones in sibling classes).

This does not mean that clones outside the "Can be extracted" category cannot be refactored using method extraction. However, they may require additional transformations, other refactoring techniques, etc. Because of that, we took those categories out of the scope for our automated refactoring efforts. We further explain opportunities to refactor such clones outside the "Can be extracted" category in Section **??**.

4.5.1.1 Complex Control Flow

`break`, `continue` and `return` statements in clone classes can obstruct the possibility of refactoring. This is the case if:

- The clone class has at least one `return` statements, but not all paths through the clone class return.
- There are `break` and/or `continue` statements in the clone class, but the corresponding loop/switch is not included.

If there is a `break` and/or `continue` statement in the cloned fragment, we walk up the AST to find the loop- or switch-statement it corresponds with. If this statement is included in the clone class, then the `break` and/or `continue` will be no problem for refactoring. However, if it isn't, then we mark it "Complex Control Flow".

4.5.1.2 Spans Part Of A Block

When applying an "Abstract Method" refactoring, we author a new method declaration. The body of this method declaration consists of a set of statements. However, if a single statement is only partially cloned, this might give issues with refactoring. An example of this is shown in Figure **??**

```
1 int a = getA();
2 while(a<1000) {
3   a *= 5;
4   doC();
5 }
```

```
1 int a = getA();
2 while(a<1000) {
3   a *= 5;
4   doB(a);
5 }
```

Figure 4.19: Partial block.

In this example, the while loop is partially cloned. This obstructs us from extracting the cloned code to a new method.

4.5.1.3 Is Not A Partial Method

The "Extract Method" refactoring can only refactor code in the body of a method. We mapped the contents of clones in Section **??**. Here we defined the "Partial Method" catagory as *a clone that spans (a part of) the body of a method. The declaration itself is not included.* If the clone does not fall within this category, "Extract Method" is likely not to be correct technique for refactoring such clones.

4.5.1.4 Top-level Node Is Not A Statement

A method body can only consist of statements. We cannot put declarations in a method. However, there are possibilities where a clone consists of something else than a statement as top level node. The most prominent example is when a clause is cloned (see Section **??** for examples of such concepts). For instance, we cannot extract only the catch-clause of a try-statement to a new method. In such a case, we should either extract the complete try-statement, or only the contents of the catch clause.

4.5.1.5 Overlaps

We do not consider clone classes for refactoring that overlap in the clone class itself. That is because such clones may 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 **??**. In this example one clone instance is found between line **??** and **??** and the other between **??** and **??**. 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 4.20: Refactoring a clone class that has overlap between clone instances.

4.5.1.6 Can be extracted

All clones that do not adhere to any of above-stated categories, are considered suitable for automated method extraction.

4.5.2 Applying refactorings

All clones that are marked as "Can be Extracted" will then be refactored. The refactoring techniques that we use depend on the relation and contents of the clone. In this section, we explain the transformations we apply to the program AST in order to refactor clones.

First, we create a new method and give it an uniquely generated name. We replace all nodes from the clone instances of the clone to method calls of the newly created method. We place a copy the nodes that were removed from a single clone instance in the new method.

4.5.2.1 Relation

For each of the identified clone relations (Section **??**) we define a method of automatically refactoring clones in these categories. The relation determines the location at which we place the newly created method.

4.5.2.1.1 Common Class For clone instances that are in the same class, we place the method in the body of that class. We give the method a `private` visibility, which indicates that the method can only be used within its class.

4.5.2.1.2 Common Hierarchy We place the extracted method in the class which all clone instances share as a superclass. We give the method a `protected` visibility, which indicates that the method can only be used within its hierarchy.

4.5.2.1.3 Common Interface We place the extracted method in the interface all clone instances have in common. We give the method a `public` visibility (because interfaces do not allow for `protected` visibility) and turn it into a default implementation using the `default` keyword.

4.5.2.1.4 Unrelated If all clone instances in a clone class are in different classes that do not yet have a superclass, or their ancestors have no superclass, we create a superclass abstraction for them. This way we make the implicit dependency between both clone fragments explicit, while provisioning a common place for common functionality. We give the method a `protected` visibility, which indicates that the method can only be used within its hierarchy.

If the clone instances have an external superclass or ancestor, we create an interface abstraction instead to place the extracted method in. We give the method a `public` visibility (because interfaces do not allow for `protected` visibility) and turn it into a default implementation using the `default` keyword.

Because for unrelated clone instances we create new abstractions, we could change the relation of other clones. When we introduce a new superclass abstraction, other clones might turn from "unrelated" into "common hierarchy". Because of this, creating a single abstraction could favor multiple refactoring opportunities.

For unrelated clones, we always create a new abstraction instead of introducing a "utility class". Most current code clone refactoring support tools use utility classes to refactor unrelated code clones [**mazinanian2016jdeodorant**, **yoshida2005refactoring**, **gode2010clone**]. For this study we refrain from doing this, because the introduction of many static methods can make a codebase hard to comprehend [**xing2003pseudo**]. Additionally, we can use the created abstractions for further refactorings.

4.5.2.2 Populate method

After placing the newly abstracted method at an appropriate location, we analyze the method content to determine whether we need to add parameters, a return value or a `throws` clause to the method declaration.

4.5.2.2.1 Method parameters We determine which variables and methods that are used in the method body are not accessible anymore at the location where the extracted method is placed. For each of these variables and called methods we add a parameter to the extracted method. We then pass the appropriate data to each method call.

4.5.2.2.2 Return value The exracted method has a return value if one of these three conditions is met:

- **There is a return statement in the extracted fragment**: If the extracted code fragment ends in a return statement, we set the type of the returned variable as the type of the extracted method. We turn all method calls of the extracted method into return statements followed by the method call.
- **A method parameter is re-assigned in the extracted fragment**: If a method parameter is re-assigned, we must return its changed value. We add a return statement to the end of the body of the extracted method returning the variable that was changed. We set the return type of the extracted method to the type of the variable that was assigned. We turn all method calls of the extracted method into variable assignment statements in which we assign the changed variable.
- **A variable is declared in the extracted fragment**: If a variable is declared in the extracted fragment, but used outside it, we return the declared variable. We add a return statement to the end of the body of the extracted method that returns the declared variable. If the variable declaration statement is the final statement in the method body, we remove the variable declaration and directly return the value it assigns to the declared variable. We set the return type of the extracted method to the type of the variable that was declared. We turn all method calls of the extracted method into variable declaration statements in which we declare the variable returned by the extracted method.

4.5.2.2.3 Thrown exception If a method in Java throws an exception that is not a `RuntimeException`, it must be denoted in the method declaration. Because of this, we analyze the extracted method for `throw` statements, which denote an exception being thrown. We also analyze all called methods to find out whether they throw exceptions. For each of these exception, we check whether they are caught in the extracted method. Any exception that is that is not a `RuntimeException` and that is not being caught is added to the thrown exceptions of the extracted method.

4.5.3 Collecting data

To find out whether clones should be refactored (RQ3) we collect data about the applied refactorings. We use this data to find out whether a specific refactoring succeeds in improving the maintainability of the system. For each refactored clone class, we map the characteristics of the extracted method to the impact of the refactoring on the maintainability of the system/

4.5.3.1 Characteristics of the extracted method

We define the following characteristics that have an impact on the refactoring of the clone class:

- **Nodes**: The number of nodes that *each clone instance of the clone class* spans. This concerns the size of a single clone instance before it was refactored.
- **Tokens**: The number of tokens that the *each clone instance of the clone class* spans. This concerns the size of a single clone instance before it was refactored.
- **Relation**: The relation category (Section **??**) that this clone class belongs to.
- **Returns**: The category of what the extracted method returns (Section **??**).
- **Parameters**: The number of parameters the extracted method has.

We hypothesize that these characteristics are the main factors influencing the impact on the maintainability of the system as a result of refactoring the clone.

4.5.3.2 Impact on maintainability metrics

We measure the impact on maintainability metrics of the refactored source code for each clone class that is refactored. These metrics are derived from Heitlager et al. [**heitlager2007practical**]. This paper defines a set of metrics to measure the maintainability of a system. For each of these metrics, risk profiles are proposed to determine the maintainability impact on the system of a whole.

In this case we are dealing with single refactorings and we want to measure the maintainability impact of such a small change. Because of that, we measure only a subset of the metrics [**heitlager2007practical**] and focus on the absolute metric changes (instead of the risk profiles). The subset of metrics we decided to focus on are all metrics that are measured on method level (as the other metrics show a lesser impact on the maintainability for these small changes). These metrics are

- **Duplication**: In Heitlager et al. [**heitlager2007practical**] this metric is measured by taking the amount of duplicated lines. We decided to use the amount of duplicated tokens part of a clone class instead, to have a stronger reflection of the impact of the refactoring by measuring a more fine-grained system property.
- **Volume**: The more code a system has, the more code has to be maintained. The paper [**heitlager2007practical**] measures volume as lines of code. As with duplication, we use the amount of tokens instead.
- **Complexity**: Heitlager et al. use MCCabe complexity [**mccabe1976complexity**] to calculate their complexity metric. The MCCabe complexity is a quantative measure of the number of linearly independent paths through a method.
- **Method Interface Size**: The amount of parameters that a method has. If a method has many parameters, the code may become harder to understand and it is an indication that data is not encapsulated adequately in objects [**fowler2018refactoring**].

We can determine the change of the values of these metrics by considering the transformations that are done by the refactoring. We calculate these metrics as follows:

- **Duplication**: The duplication can only decrease as a result of the refactoring. We calculate the reduction of duplication as the product of the number of tokens in the body of the extracted and the number of clone instances in the clone class.
- **Volume**: The volume can both decrease and increase as a result of the refactoring. The reduction of volume is equal to the reduction of duplication. The volume is increased by the amount of tokens in each method call to the extracted method, plus optionally the return, assignment or declaration of the extracted method. We also add the size in tokens of the extracted method itself to the volume. If the extracted method is moved to a newly created class or interface, we use volume of that declaration instead (because it in turn contains the extracted method). The added volume minus the reduced volume results in the change in volume.
- **Complexity**: The complexity can both decrease and increase as a result of the refactoring. To calculate the reduction of complexity, we take the sum of all branching statements in all clone instances. The increase in complexity is the complexity of the extracted method (the number of branches plus one). The added volume complexity the reduced complexity results in the change in complexity.
- **Method Interface Size**: The duplication can only increase as a result of the refactoring. The method interface size increases by the amount of parameters in the extracted method.

4.5.4 Saving refactored code

CloneRefactor features several "Refactoring Strategies", that indicate how and where refactored code will be saved. These are as follows:

- **Do not refactor**: CloneRefactor does not apply refactorings at all, it only reports found clones.
- **Simulate**: CloneRefactor only applies the refactorings to its internal AST structure and calculates the results of applying these refactorings. It does not modify/copy the codebase on disk.
- **Replace**: CloneRefactor replaces the source code with the refactored source code.
- **Copy Only Changes**: CloneRefactor writes only the changed files to a new location.
- **Copy All**: CloneRefactor copies the entire project to a new location and applies the refactorings there. This makes sure the original project stays unmodified, while also having a snapshot of the refactored project.
- **Git Replace**: CloneRefactor creates a branch on the git VCS, or initializes a git repository if it is not available. For every clone class refactored, CloneRefactor creates a commit in the git repository.
- **Git Copy**: CloneRefactor copies the entire project to a new location. CloneRefactor then performs the same procedure as for "Git Replace", except now at the new location so the original project remains unmodified.

These "Refactoring Strategies" can be used to get more data regarding the transformations applied by CloneRefactor. In our experiments, we only make use of the "Simulate" "Refactoring Strategy".

Chapter 5

Experimental Setup

In this chapter we explain the setup of our experiments. All our experiments are quantitative experiments, measured over a corpus using CloneRefactor. In this chapter, we first explain the corpus we use. We then explain the way in which we calculate the impact of refactorings on the software maintainability.

5.1 The corpus

For our experiments we use a large corpus of open source projects [**githubCorpus2013**]. This corpus contains a set of Java projects from GitHub, selected by number of forks. The projects in this corpus were then de-duplicated manually. This results in a variety of Java projects that reflect the quality of average open source Java systems and are useful to perform measurements on.

For our purposes, we have further filtered this corpus to get a controlled set of projects for our experiments. In this section, we explain the steps we took to prepare the corpus[1] and the rationale behind those steps.

5.1.1 Filtering Maven projects

As explained in Section **??**, CloneRefactor requires all the libraries a software project is dependent on in order to perform its clone analysis. As these are not included in the used corpus [**githubCorpus2013**], we decided to limit our scope to a single build automation system. We chose Maven, as it is one of the most used build automation systems for Java projects. Maven uses a file, named `pom.xml`, to describe a projects' dependencies. As no `pom.xml` files are included in the corpus, we cloned the latest version of each project in the corpus. We then removed each project that has no `pom.xml` file (which indicates that those do not use the Maven build automation system).

Because we cloned the projects (and do not use the sources provided in the corpus), they still included generated source code. Because we do not want to analyze generated Java files, we decided to further filter the corpus to only include projects with a conventional structure. For Maven, projects are recommended to put their production code in the folder structure `src/main/java`. We omitted Maven projects that do not adhere to this convention.

5.1.2 Gathering Dependencies

As a next step, we collect all dependencies of the downloaded software projects. Maven uses the following command to automate this process: `mvn dependency:copy-dependencies -DoutputDirectory=lib`. This process can fail if external dependencies are no longer available. We removed such projects for which we could not obtain all dependencies.

5.1.3 Building an AST of all Java files

To verify the correctness of all Java files in the projects, we used JavaParser [**tomassetti2017javaparser**] to create an AST of every Java file in the `src/main/java` folder of the software projects. A very small set of projects had syntactical errors, which we removed from the corpus.

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

5.1.4 Inspecting outliers

We ran CloneRefactor over every project in the corpus and inspected the output for each project. Some projects gave unusual high numbers of projects of a certain size, which we manually inspected. Often these projects contained generated source files. We removed these projects from the corpus.

5.1.5 Resulting corpus

This procedure results in 2,267 Java projects including all their dependencies. The projects vary in size and quality. The total size of all projects is 14.210.357 lines (11,315,484 when excluding whitespace) over a total of 99,586 Java files. This is an average of 6,268 lines over an average of 44 files per project, 141 lines on average per file. The largest project in the corpus is *VisAD* with 502,052 lines over 1,527.

5.2 Minimum clone size

In this study, we want to find out what thresholds to use to improve maintainability if clones by those thresholds are refactored. However, when clones are very small, they may never be able to improve maintainability. The detrimental effect of the added volume of the newly created method exceeds the positive effect of removing duplication. Because of that, we perform all our experiments with a minimum clone size of 10 tokens, because smaller clones cannot improve maintainability when refactored.

5.3 Calculating a maintainability score

In this study, we use four metrics to determine maintainability (see Section **??**). For our experiments, we aggregate the scores obtained by these metrics to draw a conclusion about the maintainability increase or decrease after applying a refactoring. We base our aggregation on the following assumptions:

- All metrics are equal in terms of weight towards system maintenance effort.
- Higher values for the metrics imply lower maintainability.
- The obtained increase of the metric divided by the average distance from zero over our corpus weights the metric equally against the other metrics.

We derived these assumptions partly from Heitlager et al **[heitlager2007practical]** and our own expert intuition. The validity of these metrics influences the validity of the conclusions we can draw from our data. Because of that, in our experiments we focus mainly on significant differences in maintainability to draw a conclusion.

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

$$A = \frac{\sum_{x \in M} abs(x)}{|M|} \tag{5.1}$$

Where M is the multiset of all metric increase/decrease values for all refactorings applied to a certain metric over our corpus. $/M/$ is the cardinality of this multiset. We then calculate the maintainability for a specific refactoring as follows:

$$A_M = \frac{dup}{A_{dup}} + \frac{com}{A_{com}} + \frac{par}{A_{par}} + \frac{vol}{A_{vol}} \tag{5.2}$$

Where *dup* is the decrease in duplication, *com* is the decrease in complexity, *par* is the decrease is method parameters and *vol* is the decrease in system volume after the refactoring is applied. When comparing the maintainability of a set of refactorings we divide the sum of the maintainability score by the amount of clone instances of all refactored clones in the set.

Chapter 6

Results

In this chapter, we present the results of our experiments. We perform exploratory experiments to map the differences between clone types, contexts and refactoring opportunities.

6.1 Clone types

In this section, we look into the differences between the different clone types that were introduced in Chapter **??**.

6.1.1 Found nodes

In this section, we display our results regarding the differences in found nodes between clone type 1-3 [**roy2007survey**] and type 1R-3R. Figure **??** shows the number of cloned nodes found over the entire corpus for the different clone types (the word "Type" is abbreviated as "T"). For type 2R clones, we set the variability threshold (see Section **??**) to 10%, meaning that no more than 10% of expressions may differ between cloned fragments. For type 3 and 3R clones, we allow a gap of 20% the clones surrounding it.

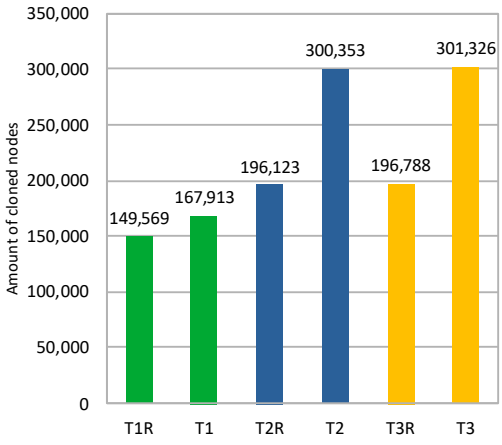

Figure 6.1: Number of cloned nodes.

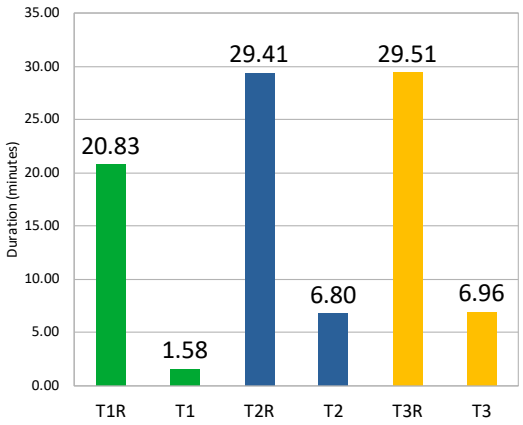

Figure 6.2: Clone type performance.

6.1.2 Performance

To map the viability of refactoring-oriented clone types for large scale clone detection, we measured the duration of finding clones by the different clone types. Figure **??** shows the duration of detecting all clones in the corpus using CloneRefactor for different clone types. This figure shows the average duration of running CloneRefactor 10 times for a certain clone type. We ran these performance tests on the compute nodes of the DAS4 supercomputer [**bal2016medium**], which do not have many background processes severely affecting the results. The durations are of course dependent on our implementation of the clone types.

6.1.3 Type 2R Variability Threshold

To determine the influence of the type 2R variability threshold on the amount of found nodes, we ran CloneRefactor for all variability percentages (0% meaning no variance in expressions, 100% meaning any variance in expression). Figure **??** shows the results.

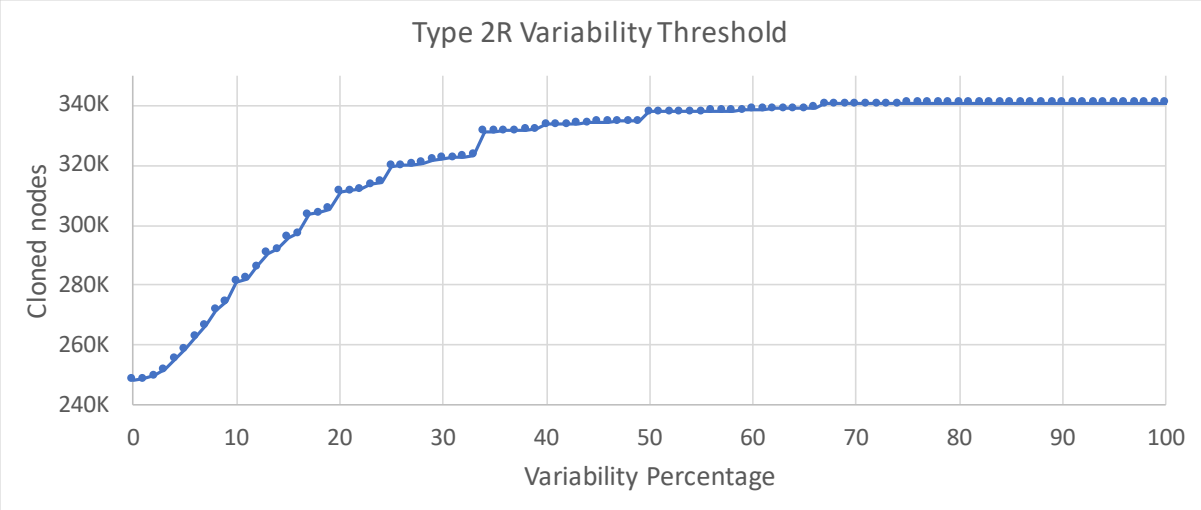

Figure 6.3: Cloned nodes found for different Type 2R thresholds.

6.1.4 Type 3R Gap Size Thresholds

To determine the influence of the type 3R gap size threshold on the amount of found nodes, we ran CloneRefactor for all gap size percentages between 0% and 200%. Figure **??** shows the results.

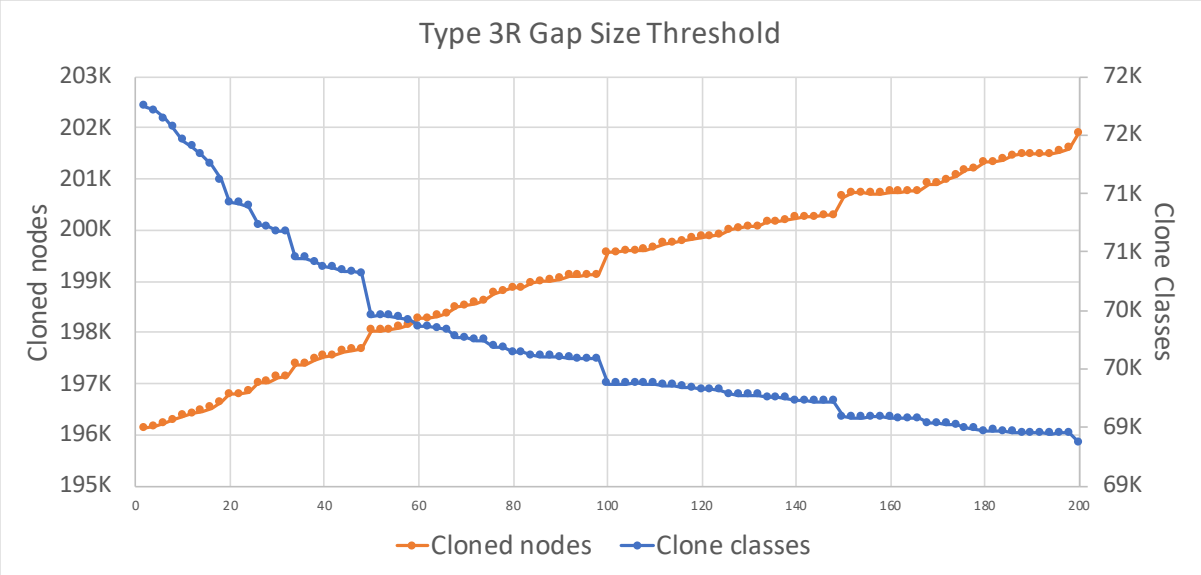

Figure 6.4: Cloned nodes and clone classes found for different Type 3R thresholds.

6.2 Clone context

To determine the refactoring method(s) that can be used to refactor most clones, we perform statistical analysis on the context of clones (see Section **??**).

6.2.1 Relation

Table **??** displays the number of clone classes found for the entire corpus for different relations (see Section **??**).

Category	***Relation***	***Clone Classes***	***%***	***Total***	***%***
Common Class	Same Class	22,893	26.8%	31,848	37.2%
	Same Method	8,955	10.5%		
Common Hierarchy	Sibling	15,588	18.2%	20,342	23.8%
	Superclass	2,616	3.1%		
	First Cousin	1,219	1.4%		
	Common Hierarchy	720	0.8%		
	Ancestor	199	0.2%		
Unrelated	No Direct Superclass	10,677	12.5%	20,314	23.7%
	External Superclass	4,525	5.3%		
	External Ancestor	3,347	3.9%		
	No Indirect Superclass	1,765	2.1%		
Common Interface	Same Direct Interface	7,522	8.8%	13,074	15.3%
	Same Indirect Interface	5,552	6.5%		

Table 6.1: Number of clone classes per clone relation

6.2.2 Location

Table **??** displays the number of clone classes found for the entire corpus for different locations (see Section **??**).

Category	Clone instances	%
Method Level	232,545	78.43%
Class Level	50,402	17.00%
Constructor Level	10,039	3.39%
Interface Level	2,693	0.91%
Enum Level	788	0.27%

Table 6.2: Amount of clone instances with a certain location

6.2.3 Contents

Table **??** displays the number of clone classes found for the entire corpus for different contents (see Section **??**).

Category	*Contents*	*Clone instances*	*Total*		
Partial	Partial Method	219,540	74.05%	229,521	77.42%
	Partial Constructor	9,981	3.37%		
Full	Full Method	12,990	4.38%	13,173	4,44%
	Full Interface	64	0.02%		
	Full Constructor	58	0.02%		
	Full Class	37	0.01%		
	Full Enum	24	0.01%		
Other	Several Methods	22,749	7.67%	53,773	18.14%
	Only Fields	17,700	5.97%		
	Other	13,324	4.49%		

Table 6.3: Number of clone instances for clone contents categories

6.3 Extract Method

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

Category	*All*	*% (All)*	*Method Body*	*% (Method Body)*
Can Be Extracted	24,157	28.2%	26,109	50.4%
Is Not A Partial Method	21,625	25.3%	0	0.0%
Top-level AST-Node is not a Statement	19,887	23.2%	4,607	8.9%
Spans Part of a Block	12,964	15.2%	13,460	26.1%
Multiple Return Values	5,622	6.6%	6,131	11.9%
Complex Control Flow	1,106	1.3%	1,216	2.4%
Overlap In Clone Class	147	0.2%	147	0.3%
Not In Class Or Interface	70	0.1%	93	0.2%

Table 6.4: Number of clones that can be extracted using the "Extract Method" refactoring technique

6.4 Refactoring

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

6.4.1 Clone Token Volume

Figure **??** shows the obtained results when plotting the clone size (in tokens) vs the maintainability increase/decrease. On the secondary axis the amount of refactorings that have refactored a clone with

the specified amount of tokens is displayed (e.g. the amount of data points the data is based on). As the amount of data points decreases, the datapoints gain less statistical significance.

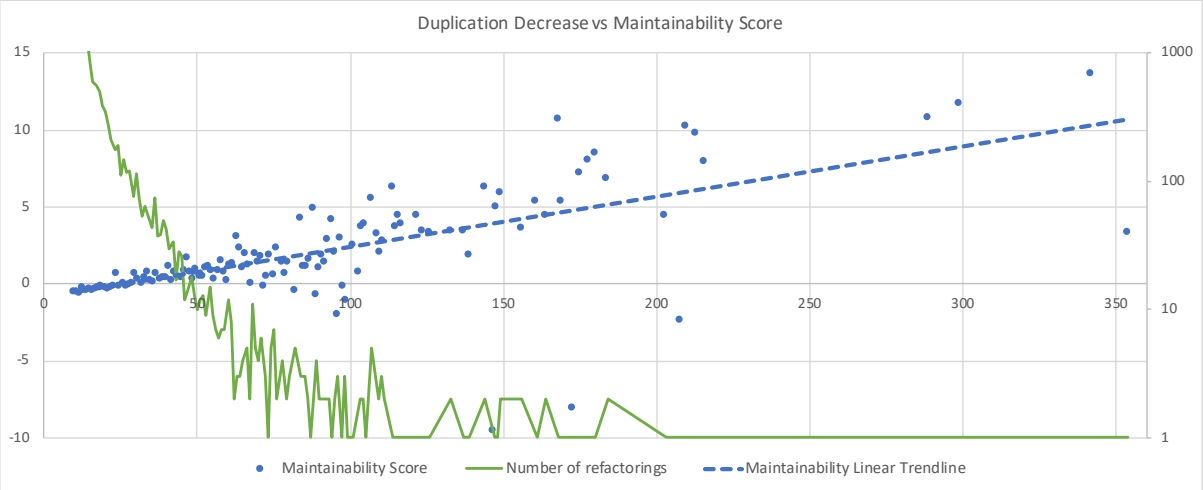

Figure 6.5: A graph that shows how the size in tokens of the refactored clone affects maintainability. Maintainability on the primary axis and amount of refactorings on the secondary axis.

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

6.4.2 Relation

Table **??** shows our data regarding how different relations influence maintainability. We have marked rows based on less than 100 refactorings red, as their result does not have statistical significance.

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

Table 6.5: Influence on maintainability of refactoring clones by certain relations.

6.4.3 Return Value

Table **??** shows how the return value of the extracted method influences the maintainability of the resulting system.

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

Table 6.6: Maintainability scores for different return values

6.4.4 Parameters

Fig. **??** shows how an increase in parameters lowers the maintainability of the refactored code. On the primary x-axis, the maintainability is displayed. The secondary x-axis shows the number of refactorings. The y-axis shows the number of parameters.

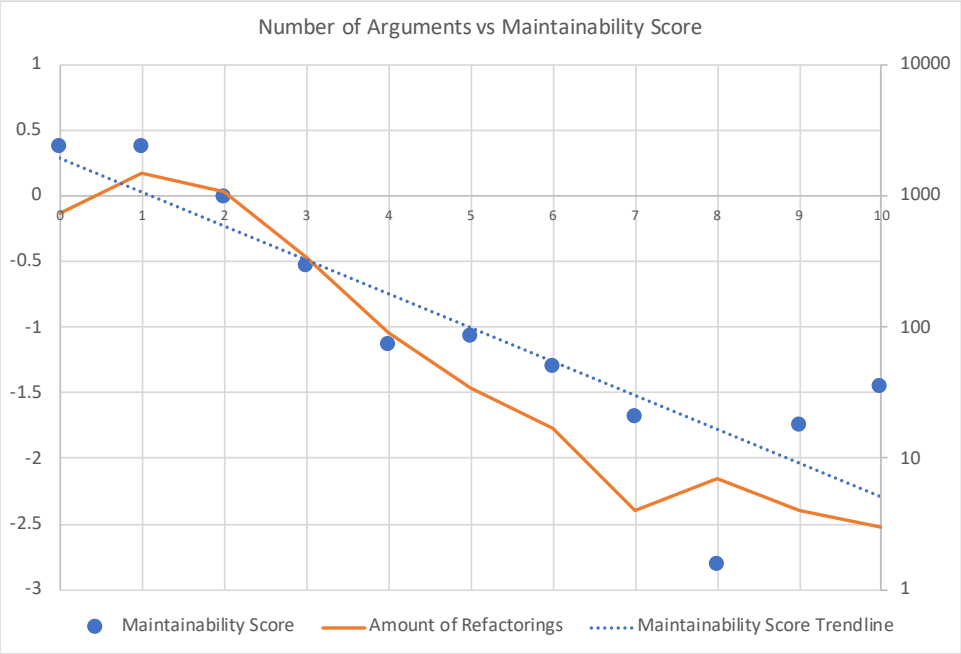

Figure 6.6: Influence of number of method parameters on system maintainability.

Chapter 7

Discussion

In this chapter, we discuss the results of our research and experiments.

7.1 Clone Type Definitions

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. In this section we discuss our clone type definitions as proposed in Section **??**.

7.1.1 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 **??**. The only thing to watch out for is method that has 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 7.1: Refactoring different expressions that have the same return type.

7.2 Clone Context Analysis

In Section **??** we introduced the categories we defined for mapping the context of clones. In this section, we discuss this together with the related experiments.

7.2.1 Refactorability

In Section **??** 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.

7.2.1.1 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 **??** 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 7.2: Refactoring a method that is obstructed by a complex control flow.

7.2.1.2 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 **??** 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.

```java
// Original
public boolean doStuff(){
  if(doA());
    return false;
  doB();
  doC();
  if(doA());
    return false;
  doB();
  return true;
}
```

```java
// Refactored
public boolean doStuff(){
  if(!doAandB())
    return false;
  doC();
  return doAandB();
}

public boolean doAandB(){
  if(doA())
    return false;
  doB();
  return true;
}
```

Figure 7.3: Refactoring a method that is obstructed by a complex control flow.

7.3 CloneRefactor

In this section we discuss our decisions for the design of our CloneRefactor tool.

7.3.1 Clone Detection

For CloneRefactor we chose to design a novel method of detecting clones, rather than using an existing clone detection technique or tool. Our rationale is as follows:

- We perform comprehensive analysis on the source code which requires us to use an AST-based clone detection method.
- We perform dependency graph analysis, which requires us to resolve symbols in the source code.
- None of the existing clone detection methods implement all criteria required to build such a system.

By building a graph that maps relations between nodes in the AST, we can find clones in an efficient manner, allowing to perform a comprehensive analysis of large systems. This method has worked well for our purposes.

The main limitation we encountered is the memory required to build the clone graph. As we load the entire graph into memory before starting the clone detection procedure, this can cause issues on systems with low available memory. For a system consisting of 1.000.000 nodes, the clone graph requires about 4GB of RAM. For our corpus, there were no larger systems, so this was not a big issue. However, for industry projects our tool might require optimization.

7.3.2 Context Analysis

In this study we identified categories for three properties of clones: relation, location and contents. We chose a set of relations that indicate different refactoring opportunities. However, as our CloneRefactor tool only analyses Java source code, we we biased towards categories that are often found in Java source code. For other languages, other categories might be valuable to analyze to find suitable categories for that programming language.

7.3.3 Refactoring

7.3.3.1 Metrics

For this study we chose to focus on a set of four metrics to measure maintainability: method size, duplication, method parameters and cyclomatic complexity. These metrics give an indication of the impact of the refactoring, but do not give a complete overview. There are many more metrics that could be considered to measure the maintainability impact on the system. An example of such a metric is "coupling", which focuses on the amount of incoming calls into a method or class and what modules these calls come from. This metric is also influenced by the transformations we applied and might deliver valuable insights in the quality of the refactoring.

In general, considering other metrics can result in a more reliable measure of the increase or decrease of maintainability after applying a specific refactoring.

7.4 Experimental setup

In this section we discuss the setup of our experiments. We discuss alternative setups that could lead to other perspectives on the problem addressed.

7.4.1 Java Corpus

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. Furthermore, it took an extensive filtering process to remove all files not suitable for application in this research (like generated files).

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.

7.4.2 Survey

To limit the scope of this research, we chose not to include human subjects in this study. However, we think that the results of this study could be strengthened by performing a survey on software engineering practisioners. We have determined a set of refactoring opportunities only based on literature input and quantitive analysis. It would be valuable to have a control group rate the refactorings performed by CloneRefactor, as an extra form of input regarding the quality of the code transformation that CloneRefactor performs.

7.5 Results

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

7.5.1 Clone Types

In this section, we discuss the differences between clone types 1-3 and 1R-3R. In Figure **??** we see that the difference between T1R and T1 is relatively small (11% more cloned nodes found for T1). Often textually equal code is also functionally equal. The difference between T2R and T2 is bigger (34.7% more cloned nodes for T2). The main reason for this is that T2R does not allow any variability in types, wheareas T2 allows any variability in types. T3R and T3 only a little bit higher than T2R and T2.

Regarding performance (Figure **??**, there is a notable difference between the refactoring-oriented clone types and the literature clone types. Type 1R-3R take about 6 times longer to detect than type 1-3. The main reason for this is type resolution: finding the fully qualified identifiers of type-, variable- and method-references.

7.5.2 Clone Context

Regarding clone context, our results indicate that most clones (37%) are in a common class. This is favorable for refactoring because the extracted method does not have to be moved after extraction. 24% of clones are in a common hierarchy. These refactorings are also often favorable. Another 24% of clones are unrelated, which is often unfavorable because it often requires a more comprehensive refactoring. 15% of clones are in an interface.

Regarding clone contents, 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods, which often require refactorings on a more architectural level. 6% of clones span only global variables, requiring an abstraction to encapsulate these data declarations. Only 4% of clones span a full declaration (method, class, constructor, etc.).

7.5.3 Extract Method

28% of clones can be refactored using the "Extract Method" refactoring technique (50% if we limit our searching scope to method bodies). About 25% of clones do not span part of a method, because of which they cannot be refactored. Many clones (23%) do not have a statement as top-level AST-Node. Upon manual inspection, we noticed that the main reason for this is clones in anonymous functions or anonymous classes. About 15% of clones span only part of an AST-Node.

7.5.4 Refactoring

In Fig. **??** we see an increase in maintainability for refactoring larger clone classes. The tipping point, between a better maintainable refactoring and a worse maintainable refactoring, seems to lie around 28 tokens. There are fewer large clones than small clones, resulting in a very limited statistical significance on our corpus when considering clones larger than 100 tokens.

In Table **??** we see the results regarding refactorings that are applied to clones with diverse relations. We see that most refactored clones are in a common class, over 54%. This is significantly more than the percentage of clones in the common class relation as reported in Table **??**. Meanwhile, the number of refactored unrelated clones is smaller than the number reported in Table **??** (24% -> 18%). The main reason for this is that refactoring unrelated clones can change the relation of other clones in the same system. If we create a superclass abstraction to refactor an unrelated clone, other clones in those classes that were previously unrelated might become related.

The maintainability scores displayed in Table **??** show that the most favorable clones to refactor are clones with a Sibling relation. The most unfavorable is to refactor clones to interfaces. However, the differences in maintainability in this table are generally small; there is no indication that relations have a major impact on the maintainability of clones.

Regarding the return type of refactored clones, we see in Table **??** that this has no major impact on maintainability. A method call to the extracted method that is directly returned and no return type extracted methods are slightly more favorable than the others. We think the main reason that the "Return" category is on top is that when a variable is declared at the end of the cloned fragment, CloneRefactor directly returns its value and removes the declaration. This decreases the volume slightly.

A higher number of parameters directly influences the corresponding metric. Because of this, we see in Fig. **??** that more parameters negatively influence maintainability. Not only the number of parameters metric is negatively influenced, but more method parameters also increase volume for the extracted method and each of the calls to it. Because of that, we see that trend of the graph in Fig. **??** decreases relatively rapidly.

Chapter 8

Related work

In this chapter we present the work related to this study.

8.1 Refactoring suggestion tools

We are novel in the introduction of a fully automated refactoring tool. However, several tools have already been proposed for refactoring suggestion. In this section we discuss these tools.

8.1.1 SUPREMO

A thesis by Golomingi [**koni2001scenario**] is the first to explore mapping the relation between clone instances to refactoring methods. The author analyses the refactoring methods described by Martin Fowler [**fowler1999refactoring**]. Mapping these to relations between clones, this results in Table **??**.

	Ancestor	Common Hierarchy	First Cousin	Same Method	Sibling	Single Class	Superclass	Unrelated
Extract Method	✓	✓	✓	✓	✓	✓		
Insert Method Call				✓		✓		
Insert Super Call							✓	
Parameterization	✓	✓	✓	✓	✓	✓	✓	
Pull Up Method	✓	✓	✓		✓		✓	
Form Template Method	✓	✓	✓	✓	✓	✓	✓	
Push Down Method							✓	
Extract Superclass		✓	✓		✓			

Table 8.1: Mapping clone relations to refactoring techniques [koni2001scenario]

The author then proposes a tool named "SUPREMO". This tool determines the relations between clone instances, computes the impact of the clones and proposes refactorings. Their tool features visualizations to show how clones are related in the inheritance structure. This tool is written for the Smalltalk programming language. The authors verify their approach by presenting several cases in which their tool analyses source code and outputs a refactoring suggestion.

8.1.2 Duplicated Code Refactoring Advisor (DCRA)

Fontana et al. [**fontana2012duplicated**, **fontana2015duplicated**] combine the research by Golomingi [**koni2001scena**
with clone types [**roy2007survey**]. They use a large corpus [**tempero2010qualitas**] on which they perform statistical analysis of clone relations together with clone types. Table **??** displays the result of this analysis. We added percentages and ordering to this table for easier comparison with the results of our studies (see **??**).

	Type 1	Percentage	Type 3	Percentage
Same Class	5,645	32.1%	51,308	28.7%
Same External Superclass	4,384	25.0%	66,391	37.1%
Unrelated Class	2,758	15.7%	35,035	19.6%
Sibling Class	2,721	15.5%	13,868	7.8%
Common Hierarchy Class	970	5.5%	3,152	1.8%
Same Method	569	3.2%	4,901	2.7%
First Cousin Class	416	2.4%	2,980	1.7%
Superclass	91	0.5%	981	0.5%
Ancestor Class	13	0.1%	281	0.2%

Table 8.2: Clone relation analysis by Fontana et al. [fontana2012duplicated] for type 1 and 3 clones measured over the Qualitas Corpus [tempero2010qualitas].

Fontana et al. [**fontana2012duplicated**, **fontana2015duplicated**] propose a tool to suggest refactorings for the clones found by the NiCAD tool [**roy2008nicad**, **cordy2011nicad**]. This tool is named Duplicated Code Refactoring Advisor (DCRA). The DCRA suggests refactorings, based on Table **??**, to clones found in Java projects.

8.1.3 Aries

A tool named Aries [**higo2004aries**, **higo2008metric**] focuses on the detection of refactorable clones. They do this based on the relation between clone instances through inheritance, similar to Fontana et al. [**fontana2012duplicated**]. 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 [**alwaqfi2017refactoring**, **chen2018clone**, **koni2001scenario**]. 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 [**chen2018clone**] or being limited to clone pairs [**higo2008metric**].

Chapter 9

Conclusion

Rewrite

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 [**roy2007survey**] and proposed solutions for problems that these types have with regards to automated refactoring. We propose that fully qualified identifiers of method call signatures and type references should be considered instead of their plain text representation, to ensure refactorability. Furthermore, we propose that one should define thresholds for variability in variables, literals and method calls, in order to limit the number of parameters that the merged unit shall have.

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

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

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

9.1 Threats to validity

Rewrite

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.

Possible stuff: future work might be a threat to validity

9.2 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.

Implementati specific for CloneRefactor: limited validation

9.2.1 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. [**heitlager2007practical**] 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.

9.2.1.1 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* [**fowler1999refactoring**, **visser2016building**]. Guidelines describe to limit the number of parameters per method to at most 4 [**visser2016building**].

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.

9.2.1.2 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 [**visser2016building**], 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 [**campbell2017cognitive**], 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 **??**).

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.

9.2.1.3 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. [**landman2016empirical**] 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.

9.2.1.4 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 [**costello1984software**, **austin2001effects**, **shah2014global**]. Apart from that, a low programmer aptitude can result in low quality code [**cheney1984effects**]. 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.

9.2.2 Type 4 clones

In Section **??** 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 in performance.

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

9.2.2.1 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.

9.2.3 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. [**allamanis2015suggesting**] 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 the 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. [**alon2018code2seq**] 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.).

9.2.4 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.

9.2.5 Automatically refactoring code that is duplicated with a library

One of the clone detection tools we analyzed in this study is Sisyphus [**eremondi2017sisyphus**]. 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.

Appendix A

Non-crucial information

Need to think about what to put here. I have a lot of stuff I'd like to put here, however my datasets are rather large. For one, I'd like to put my list of used systems here. However, it has 2267 items. I would like to put the refactorings here. However, it has 12684 items. I'd like to share more CloneRefactor output and stuff. However, it is rather much.