Improving Software Maintainability through Automated Refactoring of Code Clones

Simon Baars
simon.mailadres@gmail.com

October 22, 2019, ?? pages

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

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

Abstract

Duplication in source code is often seen as one of the most harmful types of technical debt as it increases the size of the codebase and creates implicit dependencies between fragments of code. To remove such anti-patterns, the codebase should be refactored. Many tools aid in the detection process of such duplication problems but fail to determine whether refactoring a duplicate fragment would improve the maintainability.

We address this shortcoming by combining automated refactoring and maintainability metrics to measure the impact of refactoring code clones. We propose definitions of clones that can be automatically refactored. We then propose a tool to detect such clones, analyze their context and automatically refactor a subset of them. We use a set of established metrics to determine the impact of the applied refactorings on the maintainability of the system. Based on these results, one could decide which clones improve system design and thus should be refactored. We evaluate our approach over a large corpus of open source Java projects.

We analyze the overhead of applying a refactoring in terms of four factors: the size of the clone, the relation between the code fragments in a clone, whether clone fragments create, modify or return data and the amount of external data that cloned fragments use. We find that the size of similar fragments in a clone is the biggest influencing factor: the majority of duplicates with a total volume of 68 or more tokens improve maintainability when refactored. The amount of external data that needs to be passed to the merged location of the duplicate is the other important factor: the refactored method should not have more than 1 parameter.

Contents

Chapter 1

Introduction

Refactoring is the process of restructuring code to improve quality-related attributes of a codebase (maintainability, performance, etc.) without changing the functionality. Many methods have been introduced to help with the process of refactoring [**fowler2018refactoring**, **wake2004refactoring**]. However, most of these methods still require a manual assessment of where and when to apply them. Because of this, refactoring takes up a signification portion of the development process [**lientz1978characteristics**, **mens2004survey**], does not happen at all [**mens2003refactoring**], or leaves unintentional side-effects in the code [**bavota2012does**].

Duplication in source code, also known as "code clones", is often seen as one of the most harmful types of technical debt [**fowler1999refactoring**]. If the clone is altered at one location to correct an erroneous behavior, you cannot be sure that this correction is applied to all the cloned code as well [**ostberg2014automatically**]. Additionally, the size of the codebase increases unnecessarily and so increases the amount of code to be handled when conducting maintenance work, as code clones can contribute up to 25% of the code size [**bruntink2005use**].

Automating the refactoring process of code clones can help to deal with duplication problems. It can also assist in the evaluation whether refactoring a specific clone would improve the codebase, by using established metrics [**heitlager2007practical**] to determine whether the code after applying the refactoring has improved. However, clones detected on the basis of established clone definitions [**roy2007survey**] may not always be suitable for automated refactoring.

In this study, we define clones such that they can be automatically refactored and build a tool to automatically refactor such clones. This allows us to obtain before- and after-refactoring snapshots of software systems. We use software maintainability metrics to measure the impact of these refactorings. We also look into what variability we can allow between code fragments while still improving maintainability when refactoring these clones. Furthermore, 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 clones in a specific context. We then automate the process of applying such refactorings, 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 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 to avoid major delays and unexpected costs as the outcome 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 two aspects:

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

Clone detection tools are configured using a predefined amount of lines or tokens that two duplicate fragments should minimally span for them to be considered clones [**sajnani2016sourcerercc**, **svajlenko2016bigcloneeval**]. Often, such thresholds are intuitively chosen [**li2006cp**, **roy2009mutation**] or based on a quartile distribution of empirical data [**alves2010deriving**]. These methods do not always accurately reflect the effect of refactoring the system, leading to a potentially non-optimal selection of clones that are presented to the developer.

A study by Batova et al. [**bavota2012does**] shows that the process of refactoring often leaves side effects in the code. This study reports that refactoring techniques to remove duplicate code tend to cause faults very frequently. This suggests that more accurate code inspection or testing activities are needed when such refactoring techniques are performed. Because of that, refactoring code clones has been empirically shown 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 align clone type definitions as used in literature [**roy2007survey**] with their corresponding refactoring methods [**fowler2018refactoring**] and address the shortcomings of these definitions regarding automated refactoring. This results in the following research question:

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

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

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

This research question results in a prioritization of refactoring opportunities: *with what refactoring method can what percentage of clones be refactored?* As a result of these first two research questions, we have clone type definitions that can be refactored together with a prioritization of the refactoring methods that can be used. Based on this prioritization, we build a tool that automatically refactors a subset of the detected clones.

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 use a practical model consisting of metrics to measure maintainability [**heitlager2007practical**]. Based on this, we look into what *thresholds* to use to find clones that result in better maintainable code when refactored. This results in our final research question:

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

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 collect statistical 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 on 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 clones can reduce maintainability in source code. Refactoring often includes trade-offs between design alternatives. With some 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 clones to improve the suggestion of clones that should be refactored. This assists with both the identification and refactoring of 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 insight into the impact of refactoring such clones on the maintainability of the software. In this study, we analyze a before- and after-refactoring snapshot of the code to determine the impact of a refactoring. If the maintainability increases after refactoring, this gives a strong indication that the clone should be refactored. 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 assist in the process of applying refactorings to clones. The tool only applies a refactoring if the clone is refactorable and the maintainability of the source code increases as a result of applying the refactoring. Furthermore, the tool applies only refactorings that do not influence the functional correctness of the program. Because of this, potential bugs as a result of manual refactoring can be avoided [**bavota2012does**].

1.3 Scope

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

1.3.1 Java

This research is limited to object-oriented programming languages. We relate our research to the most popular object-oriented programming languages (Python, Java, C#). Our experiments and results have been conducted in the Java programming language because many refactoring practices are language-specific.

1.3.2 Measuring Maintainability

There is a lot of study on 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 a sufficiently accurate indication of maintainability. We use this to quantitatively determine the impact on maintainability when applying refactoring techniques to code clones.

1.3.3 Naming Declarations

Our automated refactoring tool can create new methods, classes, and interfaces when refactoring clones. Each of these declarations needs to have a name. Because the quality of the name will not have an impact on 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 use generated names for these declarations.

1.3.4 Testcode

It is very disputable whether unit tests are subject to the same maintainability metrics that apply to the functional code. Because of that, for this research, unit tests are not taken into scope. The findings of this research may apply to test classes, but we will not argue the validity. Furthermore, any other code than the production code of the system will not be taken into account in our experiments. In Section **??** we describe what classes we perform our experiments on.

1.4 Outline

In Chapter **??** we describe the background of this thesis. In Chapter **??** we list shortcomings with established clone definitions which make them less suitable for automated refactoring. We also propose a set of clone type definitions that strive to solve these shortcomings. 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 setup is explained in Chapter **??**. The results of the experiments are in 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

In this chapter, we define some basic terminology that is 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**] and Jiang et al. [**jiang2007deckard**]. A summary of these concepts can be found in Table **??**. The terminology displayed in this table shows how tokens in code map to clones. When detecting clones of a software system, we end up with a clone class collection. Each clone class denotes a set of duplicate fragments. We call these duplicate fragments clone instances. Each clone instance spans several statements/declarations. We refer to these statements and declarations as "nodes". Each node consists of a set of tokens.

Symbol	Meaning	Definition	Description
T	Token	-	Tokens are the basic lexical building blocks of source code. For this study, this is the smallest relevant entity of a program.
N	Node [**jiang2007deckard**]	Set of tokens.	A statement or declaration node in the AST of a codebase.
I	Clone instance [**bruntink2005use**]	Set of cloned nodes.	A code fragment that appears in multiple locations.
C	Clone class [**bruntink2005use**]	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.

As an example, consider the code fragments in Figure **??**. In this example, two clone classes are displayed. 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.

This example can be represented as a tree structure, as shown in Figure **??**. This example uses the terminology introduced in Table **??**, together with some properties of clone instances and nodes. Each clone instance has a **range**, which denotes the line and column at which a code fragment starts and ends. Each clone instance is found in a certain **file**.

```
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: Example of two clone classes.

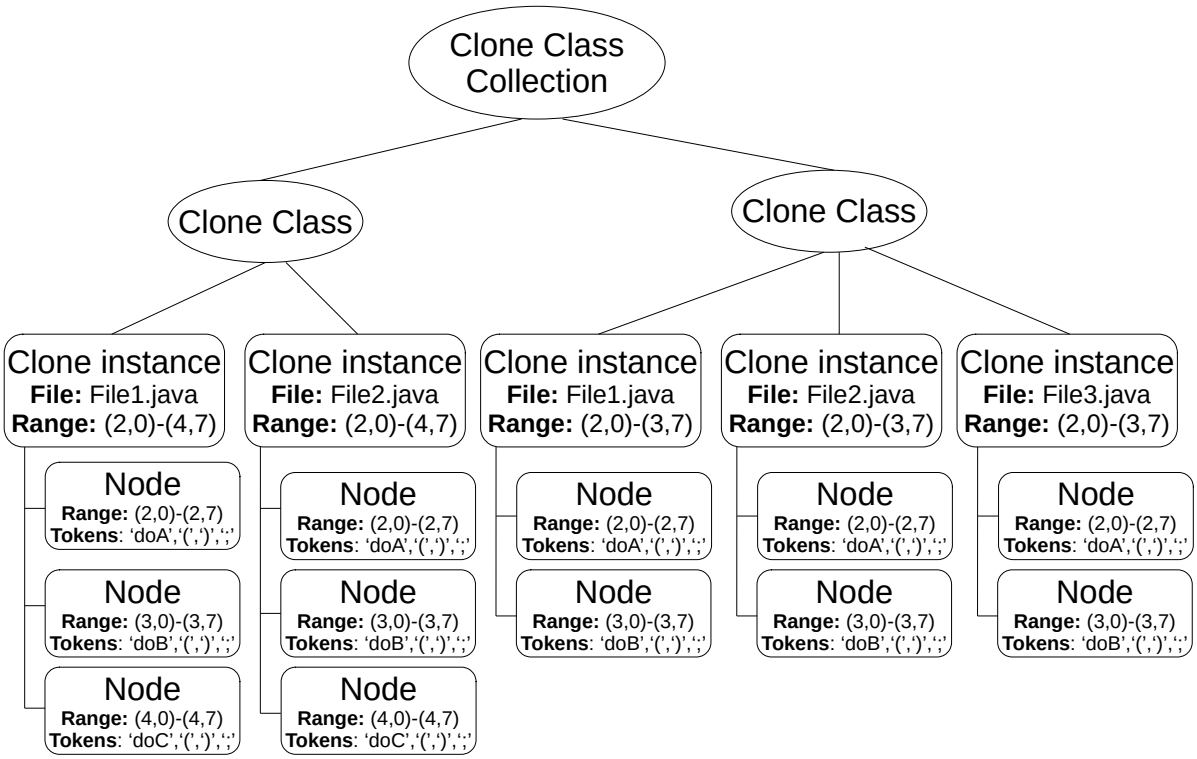

Figure 2.2: Tree representation of Figure ?? using the terminology of Table ??

2.1.1 Clone Classes vs Clone Pairs

In Table **??** we introduced the concept of clone classes. Clone classes consist of any number of clone instances. Some clone detection tools detect "clone pairs" instead, which consider each pair of clone instances separately. There are different arguments in the literature whether clone pairs or clone classes should be considered for detection.

We decided to focus on *clone classes* for our study, because of their advantages for refactoring. 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: the amount of clone references 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**].

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.

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

2.3 Refactoring techniques

In Martin Fowler's popular "Refactoring" book [**fowler2018refactoring**], 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"*. He describes several techniques to deal with duplication, dependent on the context of the clone. In this section, we describe the techniques that we use in this study.

2.3.1 Extract Method

Current clone refactoring research shows that the "Extract Method" refactoring technique can refactor most duplication problems [**fontana2015duplicated**, **tsantalis2015assessing**, **white2016deep**]. "Extract Method" moves matching functionality in method bodies to a common place, namely a new method [**fowler2018refactoring**]. An example of this procedure is displayed in figure **??**.

```
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 can 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 it is 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). An example of this refactoring technique is shown in Figure **??**.

```
1  // Original
2  class Vehicle{
3    void start(){
4      putKeyIn();
5      turnKey();
6      drive();
7    }
8  }
9
10 class Car extends Vehicle{
11   void startEngine(){
12     putKeyIn();
13     turnKey();
14   }
15 }
```

```
1  // Refactored
2  class Vehicle{
3    void start(){
4      startEngine();
5      drive();
6    }
7
8    void startEngine(){
9      putKeyIn();
10     turnKey();
11   }
12 }
13
14 class Car extends Vehicle{
15 }
```

Figure 2.4: Refactoring a clone class using "Pull Up 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 [**fowler2018refactoring**]. If the classes do not yet have a parent/superclass, a parent class can be created and the common functionality can be placed in this newly created class. This makes the relationship between these classes explicit and reduces duplication.

```
// Original
class Truck{
  void startEngine(){
    putKeyIn();
    turnKey();
  }
}

class Car{
  void startEngine(){
    putKeyIn();
    turnKey();
  }
}
```

```
// Refactored
class Vehicle{
  void startEngine(){
    putKeyIn();
    turnKey();
  }
}

class Truck extends Vehicle{
}

class Car extends Vehicle{
}
```

Figure 2.5: Refactoring a clone class using "Create class abstraction".

2.3.2.3 Providing default implementations for common functionality

In Java, C#, Python, and possibly other object-oriented languages, the programmer can create interfaces of which a class can implement any number. Such interfaces can (in Java, C#, and Python) provide default implementations for common functionality [**mohnen2002interfaces**]. This allows making 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. An example of this is shown in Figure **??**, in which two unrelated types (Human and Radio) share a common property (making a sound).

```
// Original
class Human implements MakesSound{
  void broadcastNews(){
    sayNews();
    sayWeather();
  }
}

class Radio implements MakesSound{
  void broadcastNews(){
    sayNews();
    sayWeather();
  }
}

interface MakesSound{
  void sayNews();
  void sayWeather();
}
```

```
// Refactored
class Human implements MakesSound{
}

class Radio implements MakesSound{
}

interface MakesSound{
  default void broadcastNews(){
    sayNews();
    sayWeather();
  }

  void sayNews();
  void sayWeather();
}
```

Figure 2.6: Refactoring a clone class creating a default implementation for common functionality.

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 their 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 (but to an external dependency). Most often these external dependencies are referenced in a file that its build automation system uses to fetch these external libraries.

Regarding refactoring, a software system can often not change the source code of its dependencies. This can be a burden when an external dependency does not use proper abstractions that are required by a software system. In this study, we consider external classes when choosing a refactoring technique, to be sure not to modify them.

2.4.1 Maven

In this study, we perform an analysis of external dependencies, 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*[1] 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 unit tests and to package a project as an executable file.

2.5 (Object-Oriented) Programming Languages

This section explains concepts, terminology, and jargon of (object-oriented) programming languages that we use in this study.

2.5.1 AST

An Abstract Syntax Tree (AST) is a tree consisting of a 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 [**tomassetti2017javaparser**]:

- **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 inadequate use of abstraction. Because of that, refactoring code clones might require to create such abstractions of common functionality.

[1] Apache Maven is a build automation system mainly used for Java: `https://maven.apache.org/`

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 harm 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 the maintenance costs of the source code or even render a software system unmaintainable [**heitlager2007practical**]. This increases technical debt: a debt present in the system the will have to be repaid at a later point of time [**tom2013exploration**]. Such technical debt often comes at the expense of developing new features, fixing bugs and other aspects surrounding the functional behavior of a program [**kruchten2012technical**].

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 et al. [**heitlager2007practical**] propose a maintainability model that categorize the results of metrics into different risk profiles. 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. 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.

2.5.4.1 Interfaces

Java uses the concept of interfaces to loosely couple a specification and its implementation. Interfaces are specifications of 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 programming 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 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 fit a different context. 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 (called "modules" in some 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 keyboard input related operations. These packages contain type declarations, like classes and interfaces. The names of these type declarations must be unique within the package.

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 is the unique identifier of a type declaration in a codebase. For instance, if the "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 not 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. If a method should be accessible to all types in its package, we can give it a "package" visibility. This allows control over what parts of an application should be able to access what declarations.

Chapter 3

Defining refactoring-oriented clone types

Clone type 1-3 (see Section **??**) cannot always be automatically refactored. In Section **??** we list shortcomings of each clone type that invalidate automated refactoring opportunities. In Section **??** we propose new clone type definitions that address these shortcomings.

3.1 Problems with clone types for automated refactoring

In this section, we discuss the shortcomings of clone type 1-3 [**roy2007survey**]. Because of these shortcomings, clones found by these definitions are often found to require additional judgment whether they can and should be (automatically) 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.

```
1  package com.sb.game;
2
3  import java.util.List;
4
5  public class GameScene
6  {
7    public void addToList(List l) {
8      l.add(getClass().getName());
9    }
10
11   public void addTen(int x) {
12     x = x + 10; // add number
13     Notifier.notifyChanged(x);
14     return x;
15   }
16 }
```

```
1  package com.sb.fruitgame;
2
3  import java.awt.List;
4
5  public class LoseScene
6  {
7    public void addToList(List l) {
8      l.add(getClass().getName());
9    }
10
11   public void concatTen(String x) {
12     x = x + 10; // concat string
13     Notifier.notifyChanged(x);
14     return x;
15   }
16 }
```

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

Most clone detection tools [**kamiya2002ccfinder**, **semura2017ccfindersw**, **roy2008nicad**, **svajlenko2016bigclon** **svajlenko2014evaluating**] detect type 1 clones by textually comparing code fragments (except for whitespace and comments). Although textually equal, method, type and variable references can still refer to different declarations. 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.

Figure **??** shows a type 1 clone with two clone classes. Defining an automatic way to refactor these clone classes is nearly impossible, as both cloned fragments describe different functional behavior. 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 perform different functional concepts. The cloned fragment on the left adds a specific amount to an integer. The cloned fragment 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 alternative 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 statically 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 based on a similarity thresh-

old [**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 be automatically refactored. We name these clones type 1R, 2R, and 3R clones. These definitions address the outlined shortcomings of the corresponding literature definitions. The "R" stands for refactoring-oriented.

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, type 1R 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 data structures in Java. However, importing `java.awt.List`, we get a listbox GUI component for the Java Abstract Window Toolkit (AWT). These are entirely different functional concepts. 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 two code fragments call methods with an identical name or signature, they can still call different methods. Because of this, textually identical code fragments can differ functionally.

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 we use variables with the same name but different types in different code fragments, 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 and a String.

For type 1R, cloned variable references are compared by both their name and the FQI of their type.

3.2.2 Type 2R clones

Type 2R clones are modeled 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 does not introduce a trade-off in the refactoring process of the clone.
- We allow a difference in expressions between cloned fragments if the expression can be extracted to a method parameter when applying the "Extract Method" refactoring.

Allowing variability between expressions introduces a design trade-off. When many expressions vary between cloned fragments, the refactored method might require many additional parameters, which is detrimental for system design [**heitlager2007practical**]. 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 fragments without changing the design trade-off introduced by refactoring a certain clone.

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.

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 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 extracted method requires an extra argument for the variable anyways.

```
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 Section **??**.

Literal and variable variability Type 2R allows variability in the value of literals and variables, but not in their types. This is because a difference in literal/variable types may have a big impact on the refactorability of the cloned fragment. When we refactor different literals/variables that have the same type using the "Extract Method" technique, we have to create a parameter for this literal/variable and pass the corresponding literal/variable from cloned locations. However, if two literals have different types, this might not be possible (or will harm 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.

```
1  // Original
2  void doABC(){
3    doA();
4    doB("abc");
5    doC();
6  }
7
8  void doDEF(){
9    doA();
10   doB("def");
11   doC();
12 }
```

```
1  // Refactored
2  void doABC(){
3    doThis("abc");
4  }
5
6  void doDEF(){
7    doThis("def");
8  }
9
10 void doThis(String letters){
11   doA();
12   doB(letters);
13   doC();
14 }
```

Figure 3.6: Literal variability refactored.

Method call variability Most modern programming languages (like Java, Python, and C#) allow passing 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. 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.

```
1  // Original
2  void doABC(){
3    doA();
4    doB();
5    doC();
6  }
7
8  void doADC(){
9    doA();
10   doD();
11   doC();
12 }
```

```
1  // Refactored
2  void doABC(){
3    doThis(this::doB);
4  }
5
6  void doADC(){
7    doThis(this::doD);
8  }
9
10 void doThis(Runnable r){
11   doA();
12   r.run();
13   doC();
14 }
```

Figure 3.7: Method variability refactored.

The method call variability property of type 2R clones implies 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 (which is not allowed by type 2 clones). The example as shown in Figure **??** can be a type 2R clone (dependent on the thresholds used) but is not a type 2 clone.

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 [**ueda2002detection**]. 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 number of nodes that are in the clone instances surrounding the gap:

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

Where I_{before} is the clone instance preceding the gap, I_{after} is the clone instance following the gap and $/I/$ is the number of nodes in the clone instance (the cardinality). As an example, consider the code fragment in Figure **??**.

```
1 int a = getA(); // N1
2 doC(a); // N2
```

```
1 int a = getA(); // N3
2 a *= 5; // N4
3 doC(a); // N5
```

Figure 3.9: Gapped clones.

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

$$\text{T3R Gap Size} = \frac{|\{N_4\}|}{|\{N_3, N_5\}|} * 100 = \frac{1}{2} * 100 = 50\% \tag{3.4}$$

Where $\{...\}$ is the notation for sets and $/A/$ is the cardinality of set A.

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. Figure **??** illustrates this. We cannot refactor both statements into a single conditional block. We could 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 automatically refactor the cloned code.
- **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:

$$N_{T1R} \subseteq N_{T2R} \subseteq N_{T3R} \tag{3.5}$$

Where $N_{\text{clone type}}$ is the set of cloned nodes found by the rules of the specified clone type. All cloned clones found for type 1R will also be found for type 2R and type 3R.

Comparing these clone type definitions to the ones used in literature, we have the following subset relations:

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

Where $N_{\text{clone type}}$ is the set of cloned nodes found by the rules of the specified clone type. The reason that type 2R and type 3R are not subsets of type 2 and 3, is that type 2R and type 3R allow variability in method calls which can differ in structure. Type 2 and 3 do not allow such differences.

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. 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, based on which a fully qualified identifier can be created.

3.3 Suitability of existing Clone Detection Tools for detecting these clones

We conduct a short survey on (recent) clone detection tools. We select a set of open-source tools from three modern clone detection tool surveys [**roy2009comparison**, **svajlenko2014evaluating**, **sheneamer2016survey**], to determine which tool is suitable for the detection of our refactoring-oriented clone types. We formulate the following four criteria by which we analyze these tools:

1. **Finds 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 several control projects[1] to assess the validity of clone detection tools.
3. **Analyzes 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, for us to configure our clone type definitions in the tool.

The results of our survey are displayed in Table **??**.

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

Table 3.1: Modern clone detection tools and the criteria they satisfy.

None of the state-of-the-art tools we identified implement all our criteria. Because of that, we propose CloneRefactor (see Chapter **??**).

[1] Control projects for assessing clone detection tools: `https://github.com/SimonBaars/CloneRefactor/tree/master/src/test/resources`

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 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 (CR). First, CloneRefactor detects clones based on a Java codebase, given a Java project from disk and a configuration. The clone detection process is further explained in Section **??**. When all clones are found, CloneRefactor maps the context of the clones. Based on this context, CloneRefactor applies transformations to the source code for clones that are found to be refactorable.

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 parsing Java source files to an abstract syntax tree (AST). JavaParser allows to modify this AST and write the result back to Java source code. This allows us to apply refactorings to the detected problems in the source code.

Integrated into 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 useful for the detection of our refactoring-oriented clone types, as they make use of the fully qualified identifiers of symbols.

To be able to trace referenced identifiers, SymbolSolver requires access to not only the analyzed Java project 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 **??**.

4.2 Thresholds

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

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

- **Minimum Number of Nodes**: The minimum amount of nodes that should be in each instance of cloned fragments for them to be considered clones.
- **Minimum Number of Tokens**: The minimum amount of tokens that should be in each instance of cloned fragments for them to be considered a clones.
- **Minimum Number of Lines**: The minimum amount of lines that should be in each instance of cloned fragments for them to be considered a clones.

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 acquired from JavaParser to a graph structure. Based on this graph structure, clones are detected. Dependent on the type of clones being detected, transformations may be applied. How CloneRefactor was designed does not allow for several clone types to be detected simultaneously. The overall process regarding clone detection is displayed in Figure **??**.

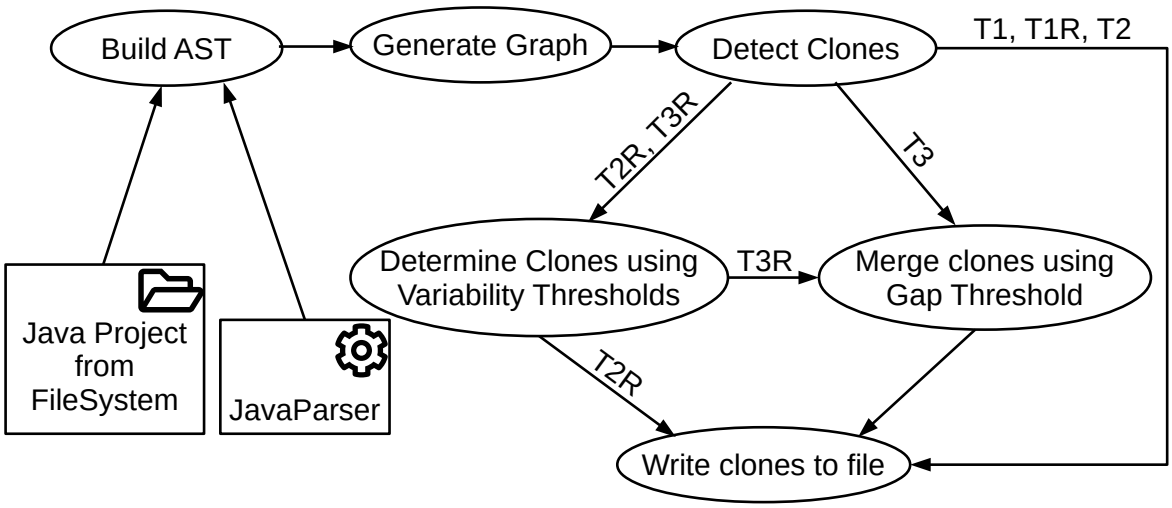

Figure 4.2: CloneRefactor clone detection process.

First of all, CloneRefactor uses 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 **??**. Based on this graph, CloneRefactor detects clone classes and verifies them using the configured thresholds. If the detection was configured to detect either type 2R, 3, or 3R clones, CloneRefactor perform some type-specific transformations on the resulting set of clones.

4.3.1 Generating the clone graph

First of all, CloneRefactor parses 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.

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 exclude package declarations and import statements from this graph. These are omitted by most clone detection tools, as package declarations and import statements are most often generated by the IDE.

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 it.
- 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 **??**.

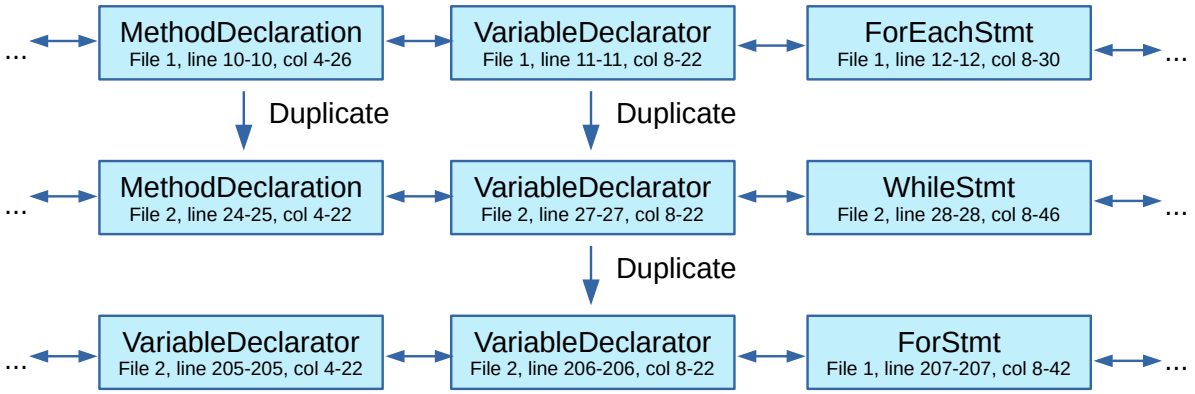

Figure 4.3: Abstract example of a part of a possible clone graph as built by CloneRefactor.

The relations *next* and *previous* in this graph are represented as a bidirectional arrow. The relations representing duplication are directed.

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 more advanced. For *method calls* we determine their fully qualified method signature for comparison with other nodes. For all *referenced types* we use their fully qualified identifier (FQI) for comparison with other nodes. For *variables* we compare their fully qualified type in addition to their name.

Type 2R clones allow any variation in literals, variables, and method calls at this stage in the clone detection process. We compare *variables* by the FQI of their types but not their names. For *method calls*, we compare the FQI of their return type and the type of each of their parameters. Apart from that, we do not compare the names of type declarations. **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 of a software project. 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.

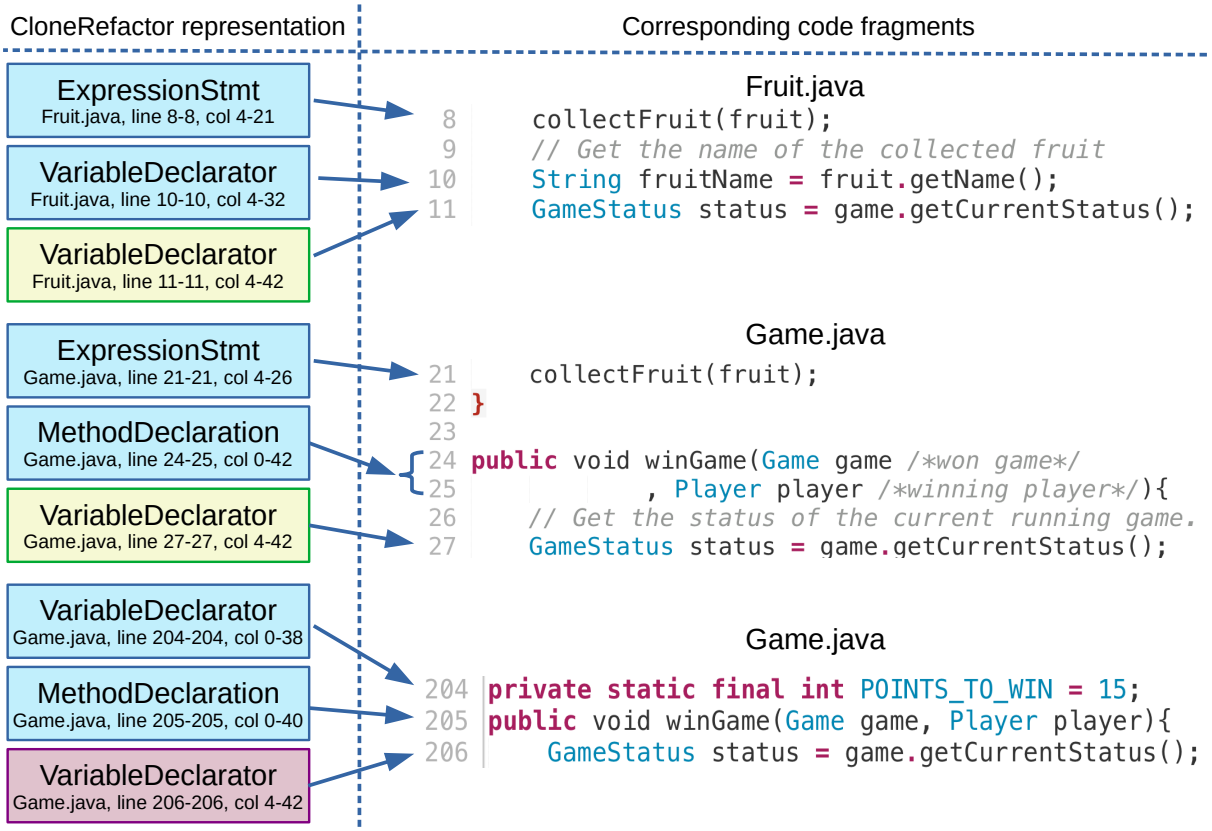

Figure 4.4: CloneRefactor extracts statements and declarations from source code.

On 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 graph, we use it to detect clone classes. We start our clone detection process at the final location encountered while building the graph. As an example, we convert the code example shown in Figure **??** to a clone graph as displayed in Figure **??**.

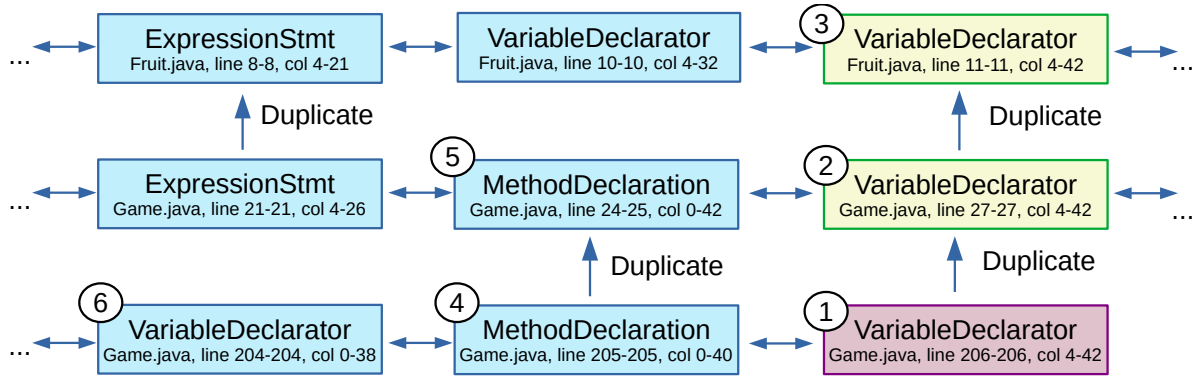

Figure 4.5: Example of a clone graph built by CloneRefactor.

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 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 clone class of three clone instances each with a single node (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 existing clone. If this is the case (which it is for this example), we discard the clone.

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, C_{new} is the clone class that was just found and S_{after} is the clone class collection after removing all subsets of the added clone.

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:

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

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

the newly found clone. 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.

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 based on 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 based on 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 configured variability threshold.

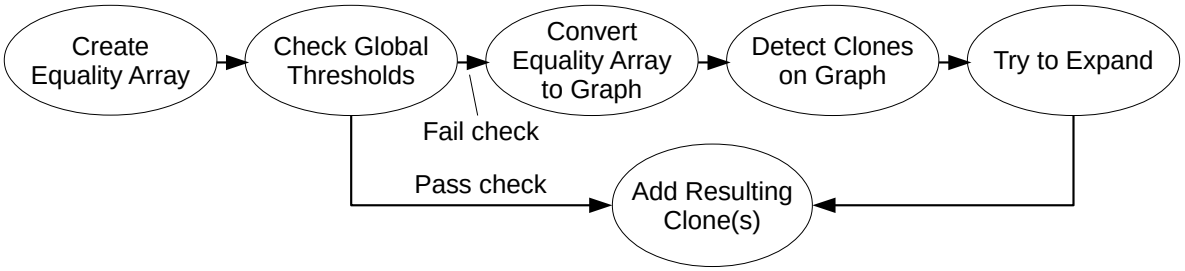

Figure 4.6: Process used to check the variability threshold for T2R clones.

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 expression to a number unique to that expression. Each literal, method call or variable becomes a positive number, whereas each other expression/token becomes a negative number. An example of two code fragments converted to equality arrays is displayed in Figure **??**. The equality array for each of the lines in this example is shown in Table **??**.

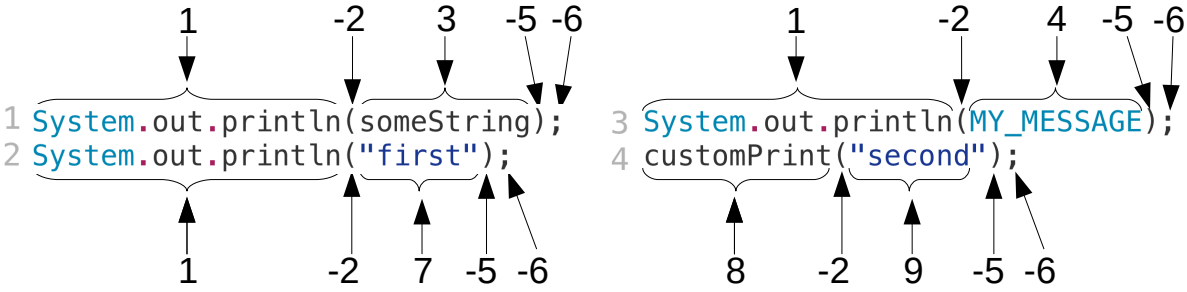

Figure 4.7: The conversion of code to an equality array.

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]

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

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 (equality arrays are represented as total order sets):

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

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:

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

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 satisfies this threshold, it will stay in the set of found clones. However, if it does not, a problem arises. Because a clone does not satisfy 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 **??**). To detect these, CloneRefactor determines 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 fragment. However, the rest of this method does not have any variability.

```
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 instead of 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 based on 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.

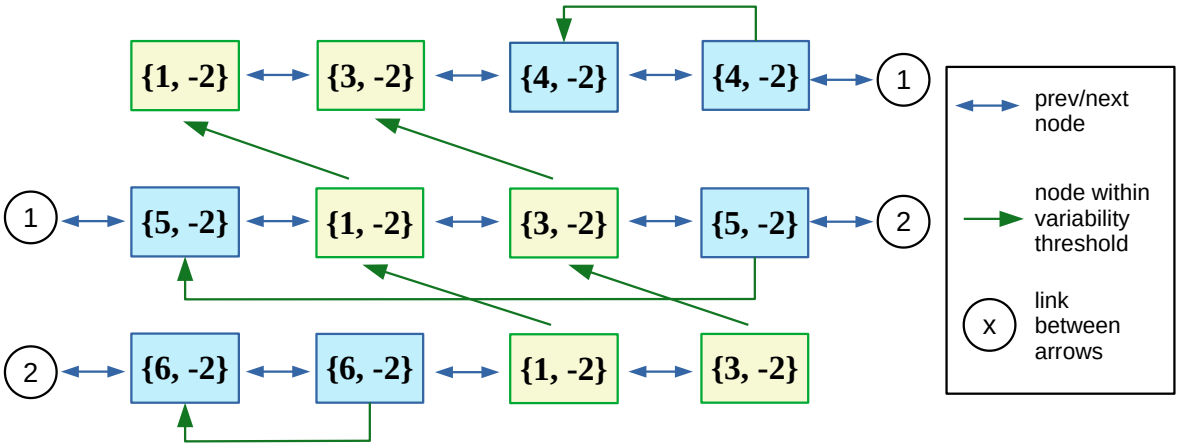

Figure 4.12: Graph representation of the code example displayed in Figure ??.

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.

4.3.3.4 Detect Clones on Graph

Similar to the clone detection process, we detect clones based on 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.

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

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

```
1 doD(); // C3 I2
2 doA(); // C1 I2
3 doB(); // C1 I2
4 doD(); // C3 I1
```

```
1 doE(); // C2 I2
2 doE(); // C2 I1
3 doA(); // C1 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 two clone instances, each consisting of a single node. 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 of each other are considered duplicates of each other. However, 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 cloned fragments 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 based on the clones found in the previous step. We start with the largest clone class, measured in clone volume. We define *clone volume* as the combined volume of all clone instances in a clone class.

Starting from this clone, we will try to expand it. We refer to the nodes in Figure **??** as N_x where x is the line number. 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:

$$\frac{|\{(5,6)\}|}{|\{(5,6),(7,7),(8,8),(9,9),(10,10)\}|} * 100 = \frac{1}{5} * 100 = 20\% \tag{4.6}$$

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:

$$\frac{|\{(5,6)\}|}{|\{(1,1),(7,7),(5,6),(7,7),(8,8),(9,9),(10,10)\}|} * 100 = \frac{1}{7} * 100 = 14\% \tag{4.7}$$

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:

$$\frac{|\{(5,6),(11,12),(7,13)\}|}{|\{(1,1),(7,7),(5,6),(7,7),(8,8),(9,9),(10,10),(11,12),(7,13)\}|} * 100 = \frac{3}{9} * 100 = 33\% \tag{4.8}$$

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:

$$\forall(c_1 \in S)\ \exists(c_2 \in S)\ x \neq y \wedge isType3R(c_1, c_2) \tag{4.9}$$

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:

$$\text{isType3R}(C_1, C_2) = \forall(i_1 \in C_1)\ \exists(i_2 \in C_2)\ Fi_1 = Fi_2 \wedge gap(i_1, i_2) < gap\_threshold \tag{4.10}$$

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 nodes that both clone instances span together:

$$\text{gap}(i_1, i_2) = \frac{|\{(Rn > Ri_1 \wedge Rn < Ri_2) \vee (Rn > Ri_2 \wedge Rn < Ri_1) \mid n \in nodes(Fi_1)\}|}{|i_1| + |i_2|} * 100 \tag{4.11}$$

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:

$$r_1 < r_2 \text{ iff } ELr_1 < BLr_2 \vee (ELr_1 = BLr_2 \wedge ECr_1 < BCr_2) \tag{4.12}$$

$$r_1 > r_2 \text{ iff } BLr_1 > ELr_2 \vee (BLr_1 = ELr_2 \wedge BCr_1 > ECr_2) \tag{4.13}$$

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 Type 3R Indentification 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:

$$\text{gap}(C_1\ I_1, C_2\ I_1) = \frac{|\{n_4\}|}{|\{n_2, n_3\}| + |\{n_5, n_6\}|} * 100 = \frac{1}{4} = 25\% \tag{4.14}$$

The same calculation is conducted for the other clone instances:

$$\text{gap}(C_1\ I_2, C_2\ I_2) = \frac{|\emptyset|}{|\{n_2, n_3\}| + |\{n_4, n_5\}|} * 100 = \frac{0}{4} = 0\% \tag{4.15}$$

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.

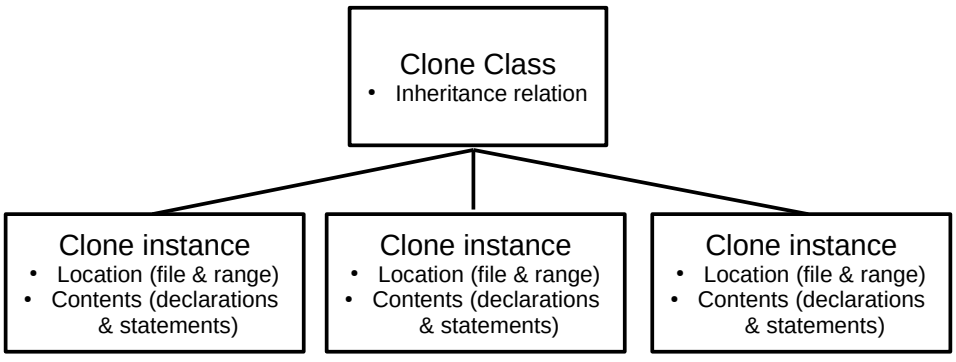

Figure 4.16: Abstract representation of clone classes and clone instances.

Fig. **??** shows an abstract representation of clone classes and clone instances. The relation of clones through inheritance is determined for each clone class. The location and contents of clones are determined 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 are 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 Direct Interface**: All instances of the clone class are in a class or interface implement the same interface.
2. **Same Indirect Interface**: All instances of the clone class are in a class or interface that have a common interface anywhere in their inheritance hierarchy.
3. **No Direct Superclass**: All instances of the clone class are in a class that does not have any superclass.
4. **No Indirect Superclass**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.
5. **External Ancestor**: All instances of the clone class are in a class that does not have any external classes in its inheritance hierarchy.

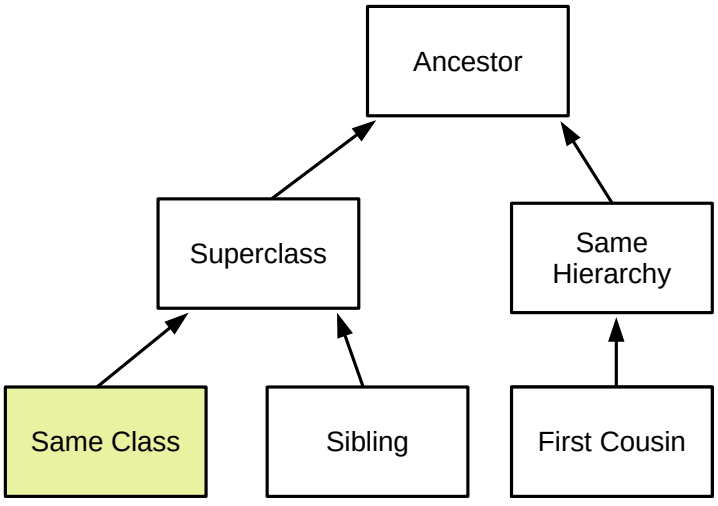

Figure 4.17: Abstract figure displaying relations of clone classes. Arrows represent superclass relations.

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 Direct Interface*, *Same Indirect 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 the above list that the clone class applies to. For instance: all "Superclass" clones also apply to "Same Hierarchy", but because "Superclass" is earlier in the above list they will get the "Superclass" relation. This is because the items earlier in the above list denote a more favorable refactoring.

When CloneRefactor applies automated refactorings, it uses this inheritance relation to see where it must place the refactored code. We explain this for each relation category over the following sections.

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.

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 [**fowler2018refactoring**].

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 stricter visibility. Because of that, we favor all "Common Hierarchy" refactoring opportunities over "Common Interface".

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 for 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 classes with a *No Direct Superclass* relation mark the opportunity for creating a superclass abstraction and placing the extracted method there. For clone classes with a *No Indirect Superclass* relation, CloneRefactor creates such an abstraction for the ancestor that does not have a parent. Clone classes with a *External Superclass* or *External Ancestor* relation obstruct the possibility of creating a superclass abstraction. In such a case, CloneRefactor creates an interface abstraction 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/Enumeration Level:** A clone that is (a part of) an interface or 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 the 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 are found in either a method or constructor. A constructor is a special method that is called when an object is instantiated. Most modern 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**].

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 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 an inheritance hierarchy. Or, we might have to perform a refactoring on more of an architectural level, if a large set of methods is cloned.

4.4.3 Contents

Finally, we looked at what nodes individual clone instances span. We selected a set of categories based on 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. Sometimes, the declaration should be moved to a location that is accessible by all usages.

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.

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.

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 indicates 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.

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 the above stated concepts. For instance, a combination of constructors and methods or a combination of fields and methods is cloned. Such clones indicate, like the "Several Methods", the requirement of performing a more architectural-level refactoring. These are often more complicated to refactor, especially when aiming to automate this process.

4.5 Automated Refactoring

After clone detection and context analysis, CloneRefactor applies refactorings to a subset of type 1R clones. This section describes what clone classes are refactored and which techniques are used.

4.5.1 Method extraction opportunities

The most used technique to refactor clones is "Extract Method" [**fowler2018refactoring**] (creating a new method based on 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 extent this refactoring method can be used to automatically refactor clones.

We defined a set of categories that can obstruct a refactoring opportunity using the "Extract Method" technique. These categories are party derived from the clone refactoring preconditions defined by Tsantalis et al. [**tsantalis2015assessing**] (see Section **??**). These categories are as follows:

- **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 block.
- **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` statement, but not all paths through the cloned fragment 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 flag the clone class "Complex Control Flow".

4.5.1.2 Spans Part Of A Block

When applying an "Extract Method" refactoring, CloneRefactor creates 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 4.18: 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. In Section **??** we describe the "Partial Method" category 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 the 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. However, there are possibilities where a clone consists of something else than a statement as a top-level node. Upon manual inspection, we noticed that most clones falling into this category are clones that span part of a lambda or anonymous class.

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. 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.

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 to refactor clones.

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

```java
// Original
public void sayHello(){
  System.out.println("hello!");
  System.out.println("hello!");
  System.out.println("hello!");
  System.out.println("hello!");
  System.out.println("hello!");
}
```

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

Figure 4.19: Refactoring a clone class that has overlap between clone instances.

4.5.2.1 Relation

For each of the identified clone relations (see Section **??**) we define a location and a visibility for the extracted method.

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.

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

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 the extracted method into a default implementation using the `default` keyword.

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" or "common interface". 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 [**mazinanian2016jdeodor 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, CloneRefactor analyzes the method content to determine whether it needs to add parameters, a return value or a `throws` clause to the method declaration.

Method parameters CloneRefactor determines 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, it adds a parameter to the extracted method. It then passes the appropriate data to each method call.

Return value The extracted 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 change the type of the returned extracted method to the type of the returned variable. 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.

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, CloneRefactor analyzes the extracted method for `throw` statements, which denote an exception being thrown. It also analyzes all called methods to find out whether they throw exceptions. For each of these exceptions, it checks whether they are caught in the extracted method. Any exception that is not a `RuntimeException` and that is not 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) CloneRefactor collects 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 determine 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 *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

For each clone class that is refactored, CloneRefactor determines how this influences a set of maintainability metrics. 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 as 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 the small-grained changes introduced by CloneRefactor). 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. Heitlager et al. [**heitlager2007practical**] measure volume in lines of code. We use the amount of tokens, to have a stronger reflection of the impact of the refactoring by measuring a more fine-grained system property.
- **Complexity**: Heitlager et al. use MCCabe complexity [**mccabe1976complexity**] to calculate their complexity metric. The MCCabe complexity is a quantitative measure of the number of linearly independent paths through a method.
- **Number of Parameters**: The number 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 increase/decrease of these metrics (delta) by considering the transformations that are done by the refactoring. We calculate these metrics as follows:

- **Δ Duplication**: The duplication can only decrease as a result of the refactoring. We calculate the reduction of duplication as the combined size of all removed clone instances.
- **Δ Volume**: The volume can both decrease and increase as a result of the refactoring. The reduction of the volume is equal to the reduction of duplication. The volume is increased by the number of tokens in each method call to the extracted method plus 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 the volume of that declaration instead (because it contains the extracted method). Optionally, each method call may include an assignment, declaration or return (based on the return category). The added volume minus the reduced volume results in the delta volume.
- **Δ 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 complexity minus the reduced complexity results in the delta complexity.
- **Δ Number of Parameters**: The number of parameters can only increase as a result of the refactoring. This metric increases by the number of parameters in the extracted method.

Chapter 5

Experimental Setup

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 how 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 assembled by Allamanis et al. [**githubCorpus2013**][1]. This corpus contains a set of Java projects from GitHub, selected by the number of forks. The projects and files in this corpus were 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.

Because CloneRefactor requires all dependencies for the projects it analyses, we created a set of scripts[2] to filter the corpus for all projects for which we can obtain all dependencies using Maven. In this section, we explain the actions performed by the scripts and the rationale behind those actions.

5.1.1 Filtering Maven projects

As explained in Section **??**, CloneRefactor requires all the libraries a software project is dependent on to perform its clone analysis. As these are not included in the corpus by Allamanis [**githubCorpus2013**], we created a script to gather these dependencies. As Java projects manage their dependencies in many different ways, 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. Our script clones the latest version of each project in the corpus. It then removes each project that has no `pom.xml` file (which indicates that those projects do not use the Maven build automation system).

We then continued to filter all test code and generated files. As many projects store these files in many different places, we decided to further filter the corpus to only include projects with a conventional structure. When using 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. We instruct CloneRefactor only to use the sources in the `src/main/java` folder. This decreases the chance of generated files and test code being considered for clone detection.

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 fails if the project requires external dependencies that are no longer available. Our scripts remove such projects for which it cannot obtain all dependencies.

[1] The corpus can be downloaded at: `http://groups.inf.ed.ac.uk/cup/javaGithub/`

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

5.1.3 Building an AST of all Java files

To verify the syntactical 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. For instance, some projects had code that was still in development on their *master* branch, which did not compile. We remove them because CloneRefactor cannot parse them into an AST.

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 clones 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[3]. 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 files.

5.2 Tool Validation

We have validated the correctness of CloneRefactor through unit tests and empirical validation. First, we created a set of 57 control projects[4] to verify the correctness in many (edge) cases. These projects test each identified relation, location, contents and refactorability category (see Section **??** and **??**), to see whether they are correctly identified. Next, we run the tool over the corpus and manually verify samples of the acquired results. This way, we check the correctness of the identified clones, their context, and their proposed refactoring.

We also test the correctness of the resulting code after refactoring. For this, we use a project named JFreeChart[5]. JFreeChart has high test coverage and working tests, which allows us to test the correctness of the program after running CloneRefactor. We run CloneRefactor manually over the JFreeChart project, run its test cases and check the correctness of the proposed refactorings.

5.3 Minimum clone size

In this study, we want to find out what thresholds will improve maintainability if clones by those thresholds are refactored. For example, when clones are very small, refactoring them might not improve maintainability since 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 almost never improve maintainability when refactored.

The reason for this is that when applying the "Extract Method" refactoring technique, we create a new method. An empty method in Java consists of 7 tokens. We also need calls to the extracted method, which is 4 tokens per call if no parameters/arguments are required. Empirically, we found all clones below 10 tokens in size, are very unlikely to affect maintainability positively. To avoid such clones polluting our data, we do not consider clones smaller than 10 tokens.

5.4 Calculating a maintainability score

In this study, we use four metrics to determine maintainability (see Section **??**). For each of these metrics, we collect their increase/decrease (delta) after each applied refactoring. We aggregate these delta metric

[3]The full list of projects is in the following file in our GitHub repository: `https://github.com/SimonBaars/GitHub-Java-Corpus-Scripts/blob/master/filtered_projects.txt`

[4]Control projects for testing CloneRefactor: `https://github.com/SimonBaars/CloneRefactor/tree/master/src/test/resources`

[5]JFreeChart is available on GitHub: `https://github.com/jfree/jfreechart`

values 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.
- Normalizing each obtained metric delta over all deltas obtained for that metric in our dataset results in equally weighted scores.

We derived these assumptions from supporting evidence shown by Heitlager et al [**heitlager2007practical**] and Alves et al. [**alves2010deriving**]. Using the resulting aggregated maintainability score, we can argue for each refactoring whether it increases or decreases the maintainability of the system.

We normalize each obtained metric delta using the "Standard score", which is calculated as follows:

$$N_{metric} = \frac{\Delta X - \mu}{\sigma} \tag{5.1}$$

Where ΔX is a metric delta, μ is the mean of all deltas for this metric and σ is the standard deviation of all deltas for this metric. This method works well for normalization of our data because it is resilient against outliers. We divide by the standard deviation so outliers do not influence the resulting scores much.

We then calculate the maintainability score for a specific refactoring as follows:

$$\text{Maintainability Score} = N_{duplication} + N_{complexity} + N_{volume} + N_{parameters} \tag{5.2}$$

Chapter 6

Results

In this chapter, we present the results of our experiments[1]. 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 clone types that were introduced in Chapter **??**.

6.1.1 Found nodes

In this section, we display our results regarding the differences in the number of cloned nodes (see Section **??** for our terminology) 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 fragments to be considered clones. For type 3 and 3R clones, we allow a gap size of 20% (see Section **??**).

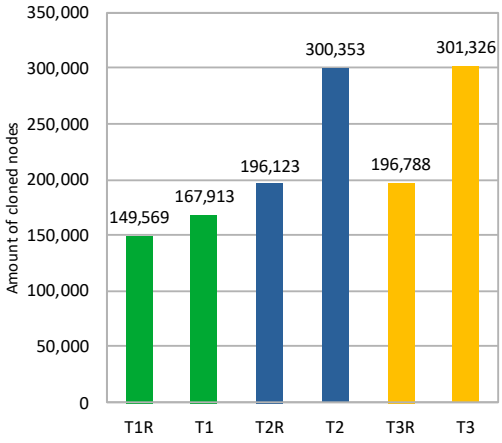

Figure 6.1: Number of cloned nodes.

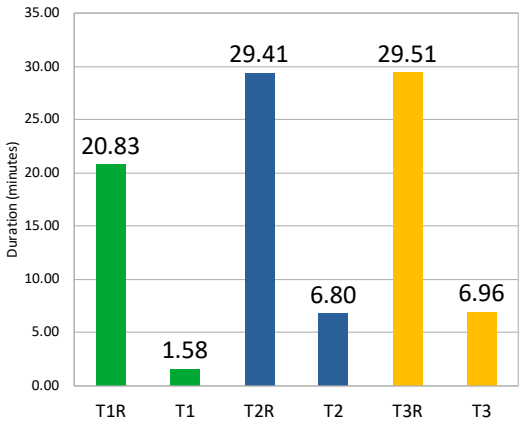

Figure 6.2: Clone type performance.

In Figure **??** we see that the difference between T1R and T1 is relatively small (11% more cloned nodes found for T1). The difference between T2R and T2 is bigger (35% more cloned nodes for T2). The number of cloned nodes of T3R and T3 are only a little bit higher than T2R and T2.

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

[1]The underlying datasheets for all our results can be found on GitHub: `https://github.com/SimonBaars/Master-Thesis-Simon-Baars/tree/master/results`

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 allow for higher accuracy performance measurements as they have limited background processes affecting the results.

6.1.3 Type 2R Variability Threshold

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

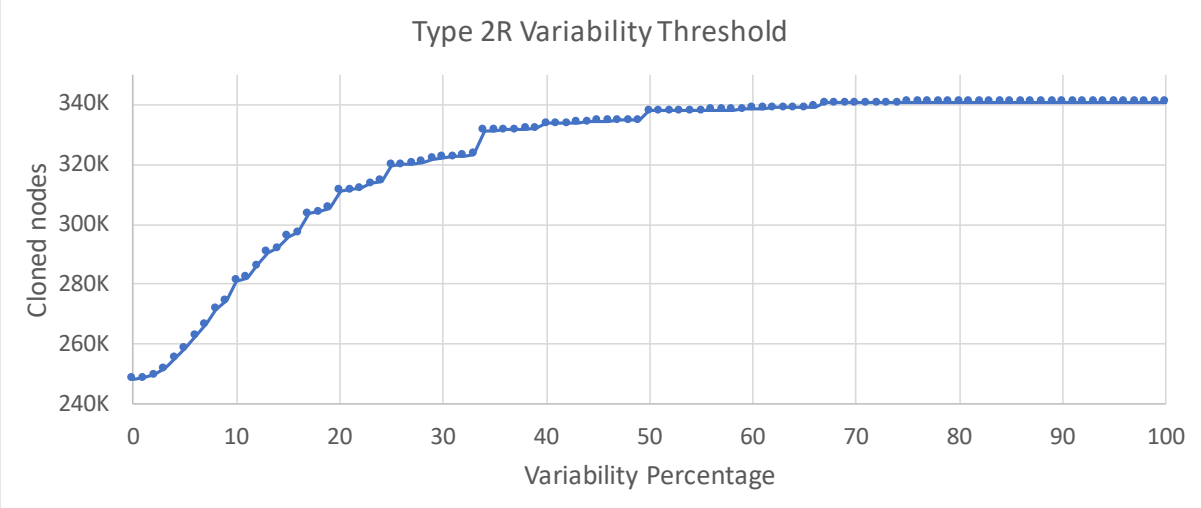

Figure 6.3: Cloned nodes found for different Type 2R thresholds.

In this graph, we see that the amount of clones increases when we allow more variability between cloned fragments. The graph is logarithmic: as the variability increases, the increase in nodes decreases.

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.

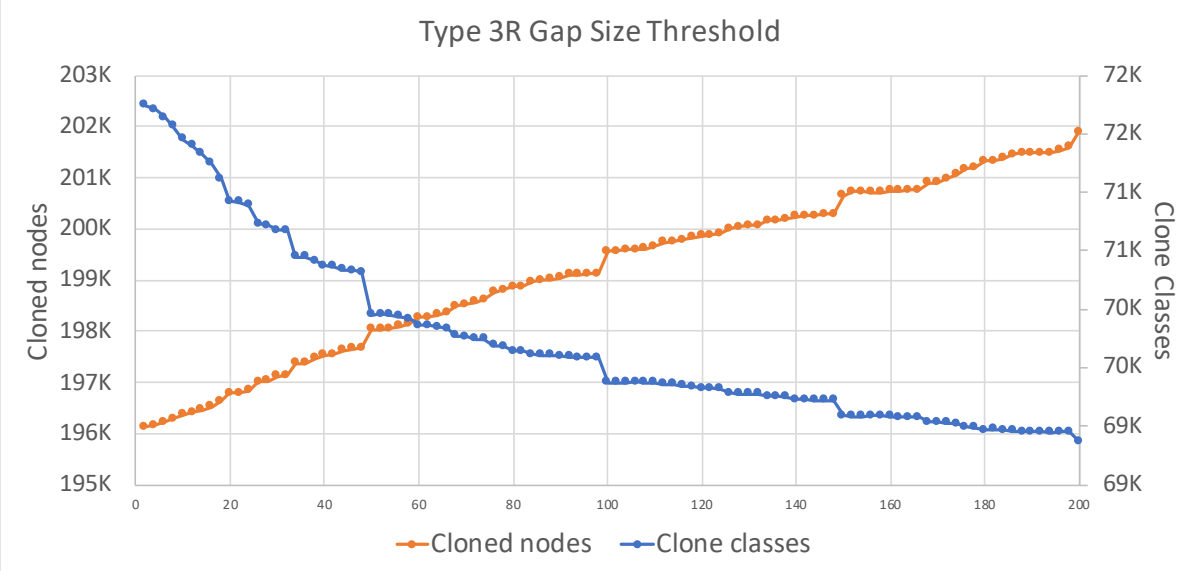

Figure 6.4: Cloned nodes and clone classes found for different Type 3R thresholds.

In this graph, we see a decrease in clone classes and an increase in cloned nodes when we increase the type 3R gap size threshold. In the graph, we see that both lines are mostly linear with minor jumps at certain percentages. The jumps are at 33%, 50%, 100%, 150% and 200%. The jump at 50% is the largest. This makes sense regarding the nature of the threshold. At 50%, all clone instances of 2 nodes with a gap of one node will merge into type 3R clones.

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.

Our results show that most clones (37%) are in a common class. 24% of clones are in a common hierarchy. Another 24% of clones are unrelated. 15% of clones are in an interface.

6.2.2 Location

Table **??** displays the number of clone classes found for the entire corpus for different location categories (see Section **??**). We can see from these results that nearly 80% of clones are found at method level. 17% of clones are found at class level, meaning they exceed the boundaries of a single method (or do not span methods at all). Constructors account for approximatelyTop- 3% of clones. In interfaces, only 1% of clones is found.

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 per location category.

6.2.3 Contents

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

Category	*Contents*	*Clone instances*	*Total*		
Partial	Method Body	219,540	74.05%	229,521	77.42%
	Constructor Body	9,981	3.37%		
Other	Several Methods	22,749	7.67%	53,773	18.14%
	Only Fields	17,700	5.97%		
	Other	13,324	4.49%		
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%		

Table 6.3: Number of clone instances for clone contents categories

From these results, we see that 74% of clones span part of a method body (77% if we include constructors). 8% of clones span several methods. 6% of clones span only global variables. Only 4% of clones span a full declaration (method, class, constructor, etc.).

6.3 Refactorability

Table **??** shows to what extent clone classes can be refactored by using the "Extract Method" refactoring technique. From these results we see that approximately 28% of clones in our corpus can be automatically refactored. The main reason clones cannot be automatically refactored is because they are not flagged as "Partial Method" in Table **??**. Another reason is that top level AST-nodes are not a statement.

6.4 Thresholds

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. Using this data, we find supporting evidence regarding which thresholds are most likely to find clones that should be refactored to improve maintainability.

Category	*All*	*% (All)*
Can be Extracted	24,157	28.2%
Is not in a Method Body	21,625	25.3%
Top-level AST-Node is not a Statement	19,887	23.2%
Spans Part of a Block	12,964	15.2%
Multiple Return Values	5,622	6.6%
Complex Control Flow	1,106	1.3%
Overlap in Clone Class	147	0.2%
Not in Class or Interface	70	0.1%

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

6.4.1 Clone Token Volume

Figure **??** shows the obtained results when plotting the clone volume against the maintainability increase/decrease. We define the *token volume* as the combined number of tokens in all clone instances in a refactored clone class. The maintainability score, as shown in Figure **??**, is the average delta maintainability score of all clone classes with the specified token volume that are refactored. For larger token volumes we have fewer refactorings that refactor such clones, because of which we represent the x-axis as a logarithmic scale. The trendline intersects the "zero" line (maintainability does not increase nor decrease) at a token volume of 63.

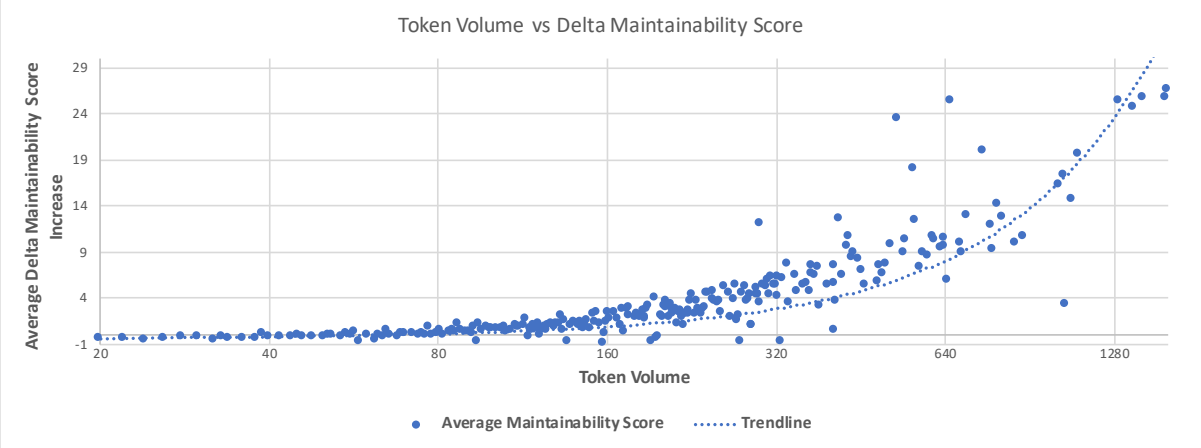

Figure 6.5: The effect of token volume on the delta maintainability score of the refactored clones

The token volume of the clone is equal to the decrease of duplicated tokens, which is part of the maintainability score. Because of that, what we see in Figure **??** could largely be caused by this metric. To determine the effect of clone token volume on each metric, we also plot graphs in which each metric is considered singularly.

In Figure **??** the average delta volume is shown when clones with the specified token volume are refactored. The trendline intersects the "zero" line (average delta volume does not increase nor decrease) at 60 tokens. A higher average delta volume implies lower maintainability.

In Figure **??** we show the average delta number of parameters when clones with the specified token volume are refactored. This metric can only increase by the refactoring because CloneRefactor only applies "Extract Method" refactorings that create a method with a certain number of parameters (according to the data needed by the extracted method).

In Figure **??** we show the average delta complexity when clones with the specified token volume are refactored. We did not add a trendline, as there does not seem to be a clear trend. The maximum delta complexity is 1 because the extracted method itself has a complexity of 1. For each branching point in the cloned fragment, the complexity of the system will decrease when such a clone is refactored.

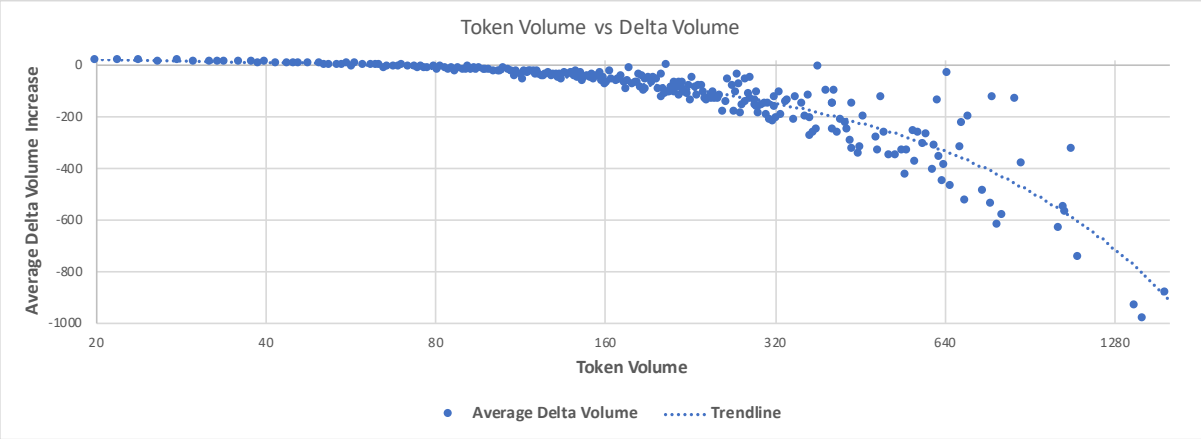

Figure 6.6: The effect of token volume on the delta volume of the refactored clones

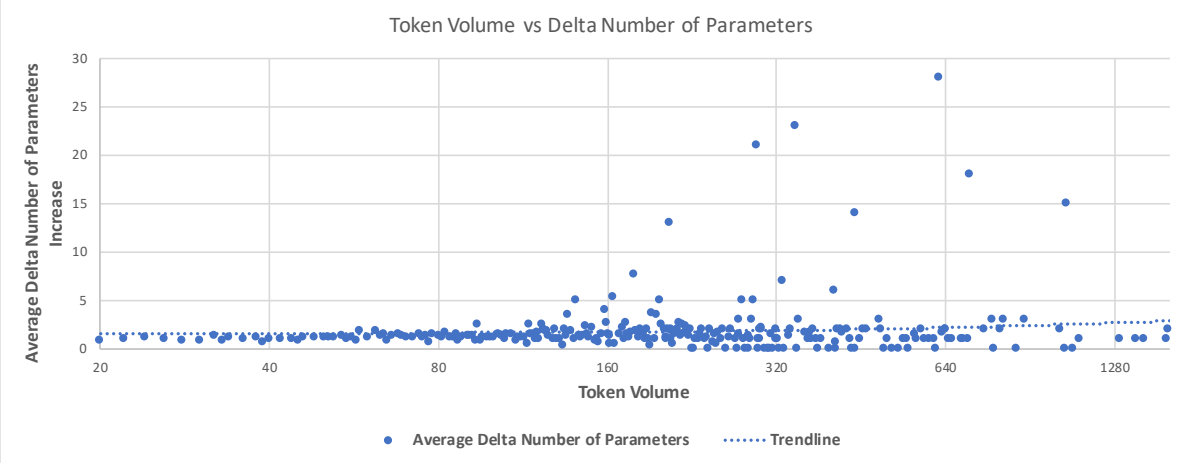

Figure 6.7: The effect of token volume on the number of parameters of the refactored clones

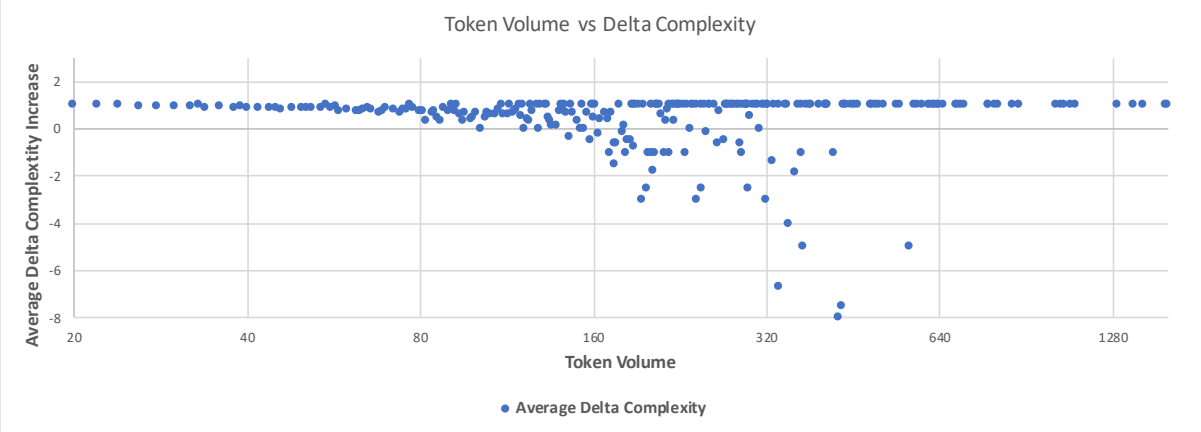

Figure 6.8: The effect of token volume on the complexity of the refactored clones

6.4.2 Relation

Table **??** shows our data regarding how different relations influence maintainability. We mark rows based on less than 100 refactorings red, as their result is less likely to have statistical significance. The **Relation** column refers to the relations introduced in Section **??**. Relations are mutually exclusive so all percentages add up to 100%. We use the following keywords as headers of the columns of our tables:

- **Complexity**: The average delta complexity for all refactored clone classes in the specified category.
- **Parameters**: The average delta number of parameters for all refactored clone classes in the specified category.
- **Volume**: The average delta number of tokens for all refactored clone classes in the specified category.
- **Duplication**: The average delta number of duplicate tokens for all refactored clone classes in the specified category.
- #: The number of times the specified category is refactored (and thus the number of data points we have for this category).
- **Score**: The average delta maintainability score (see Section **??** for the formula by which this score is calculated) for all refactored clone classes in the specified category.

In this context, the *delta* denotes the change in this metric between the before- and after-refactoring snapshot of the system, for each separate clone class that was refactored.

Relation	*Duplication*	*Complexity*	*Parameters*	*Volume*	*#*	*Score*
Common Hierarchy	**-66.33**	**0.73**	**1.20**	**-8.85**	**2,202**	**0.23**
Superclass	-64.48	0.79	0.94	-7.22	229	0.42
Sibling	-70.07	0.69	1.28	-10.97	1,722	0.23
Same Hierarchy	-44.18	0.95	0.89	1.54	87	0.10
First Cousin	-42.69	0.89	0.93	4.86	144	0.02
Ancestor	-32.75	1.00	0.75	11.00	20	-0.03
Common Interface	**-47.06**	**0.83**	**1.04**	**4.50**	**1,044**	**-0.02**
Same Indirect Interface	-37.08	0.93	0.82	9.96	487	-0.01
Same Direct Interface	-55.79	0.75	1.24	-0.28	557	-0.02
Common Class	**-52.42**	**0.87**	**1.13**	**1.47**	**7,239**	**-0.02**
Same Class	-51.85	0.86	1.03	3.36	4,874	0.04
Same Method	-53.60	0.90	1.32	-2.44	2,365	-0.15
Unrelated	**-45.86**	**0.88**	**1.08**	**9.56**	**2,198**	**-0.15**
No Direct Superclass	-52.24	0.84	1.12	6.04	811	-0.06
External Superclass	-47.09	0.87	1.13	8.77	697	-0.17
External Ancestor	-35.73	0.93	0.95	14.58	586	-0.21
No Indirect Superclass	-44.89	0.84	1.18	14.08	104	-0.30
Grand Total	**-53.26**	**0.84**	**1.12**	**1.33**	**12,683**	**0.00**

Table 6.5: Average metric values for refactorings of clone classes with the specified relations

The displayed relations are ordered by their scores. Overall, the scores do not deviate much (-0.15 to 0.23), indicating that the relation between clones has a minor impact on maintainability. Overall, we see that common hierarchy clones have the highest maintainability, whereas unrelated clones have the lowest maintainability.

6.4.3 Return Category

Table **??** shows how the return category of the extracted method influences the maintainability of the resulting system. The **Return Category** column refers to the categories introduced in Section **??**.

Return Category	*Complexity*	*Parameters*	*Size*	*Duplication*	*#*	*Score*
Return	0.85	1.02	-3.84	-55.00	1,571	0.19
Declare	0.94	0.74	11.11	-49.19	5,177	0.15
Assign	0.79	1.07	0.43	-56.29	14	0.12
Void	0.76	1.49	-5.85	-56.35	5,921	-0.18
Grand Total	**0.84**	**1.12**	**1.33**	**-53.26**	**12,683**	**0.00**

Table 6.6: Average metric values for refactorings of clone classes with the specified return category

The displayed return categories are ordered by their scores. Overall, the scores do not deviate much (-0.18 to 0.19), indicating that the return category has a minor impact on maintainability. When the result of the call to the extracted method is directly returned, the maintainability score is the highest. When no value is returned, the maintainability score is the lowest. The main reason that no return value ends up lowest, is that it is linked to a higher number of parameters required for the extracted method.

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. The table containing all underlying data and separate metrics is displayed in Appendix **??**.

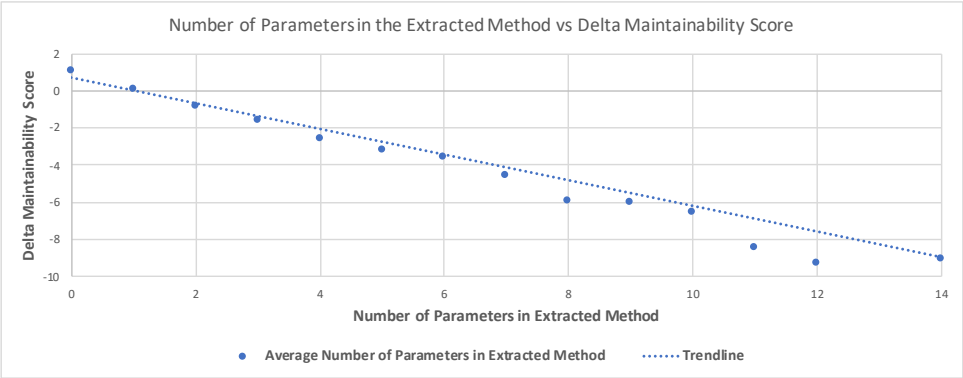

Figure 6.9: Influence of number of method parameters on system maintainability.

Chapter 7

Discussion

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

7.1 Type 2R Clones

In this study, we propose a set of clone type definitions that can be automatically refactored. In this section, we discuss our type 2R definition as proposed in Section **??**.

With type 2R clones we allow variability in some expressions such that the code can 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 that type 2R could be improved 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 methods that have side effects. Because methods may be executed in another point during execution, this might affect the functionality of the code. Because of that, before applying such a refactoring it should be verified that the called method has to side effects.

```
1  // Original
2  public void doStuff(){
3    int numbers = 456;
4    doA(getTitle());
5    doB(123);
6    doC();
7    doA("456");
8    doB(numbers);
9  }
10
11 public String getTitle(){
12   return "123";
13 }
```

```
1  // Refactored
2  public void doStuff(){
3    int numbers = 456;
4    doAandB(getTitle(), 123);
5    doC();
6    doAandB("456", numbers);
7  }
8
9  public void doAandB(String var1,
       int var2){
10   doA(var1);
11   doB(var2);
12 }
13
14 public String getTitle(){
15   return "123";
16 }
```

Figure 7.1: Refactoring different expressions that have the same return type.

7.2 Automated Refactoring

In this section, we discuss our implementation decisions regarding the automated refactoring of source code using CloneRefactor.

7.2.1 Refactorability

In Section **??** we introduced categories to determine the refactorability of clones through method extraction. We excluded categories that cannot 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 more effort.

7.2.2 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 [**tsantalis2017clone**]. Figure **??** shows an example of such a refactoring opportunity.

```
1 // Original
2 public void doStuff(){
3   for(int i = 0; i<5; i++) { //Only
      the declaration of this for
     loop is cloned, but the loop
     body is not.
4     System.out.println("hello!");
5   }
6   for(int i = 0; i<5; i++) {
7     CoreController.activateCore(i);
8   }
9 }
```

```
1 // Refactored
2 public void doStuff(){
3   doFiveTimes(() -> System.out.
      println("hello!"));
4   doFiveTimes(() -> CoreController.
      activateCore(i));
5 }
6
7 public void doFiveTimes(Runnable
     runnable){
8   for(int i = 0; i<5; i++) { //Only
       the declaration of this for
      loop is cloned, but the loop
      body is not.
9     runnable.run();
10  }
11 }
```

Figure 7.2: Refactoring a method that is obstructed by a complex control flow.

7.2.3 Complex control flow

`break`, `continue` and `return` statements can obstruct the possibility of performing method extraction. However, with some extra transformations, method extraction will still be possible in such cases. Figure **??** 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.

7.2.4 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 indicate 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 number 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 into 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.

7.3 Results

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

7.3.1 Clone Types

In this section, we discuss the differences between clone types 1-3 and 1R-3R.

7.3.1.1 Cloned Nodes

We see that the difference between T1R and T1 in terms of found clones is small (11% more nodes found as cloned for T1). This implies that most often textually equal code is also functionally equal. The difference between T2R and T2 is bigger (35% more nodes found as cloned). Upon manual inspection, we found that the main reason for this is that T2R does not allow any variability in types, whereas T2 allows any variability in types.

7.3.1.2 Performance

Regarding performance (Figure **??**), there is a notable difference between the refactoring-oriented clone types and the literature clone types. Type 1R-3R take on average approximately 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.

7.3.1.3 Type 2R Variability

In Figure **??** we show the increase of cloned nodes for a higher variability between clone instances. This graph is logarithmic: as the variability increases, the increase in nodes decreases. This implies that semantical equality increases the chances that tokens are equal.

7.3.1.4 Type 3R Gap Size

In Figure **??** we show the increase of cloned nodes for higher type 3R gap sizes. The line denoting the number of clone classes seems a bit exponential whereas the line denoting cloned nodes is mostly linear. This makes sense regarding the nature of the threshold. Type 3R clones are formed by merging two gapped clone classes into a type 3R clone class. As we get into higher gap size percentages, fewer clone classes merge. However, at these higher gap size percentages, the gaps contain more nodes. Because of this, the increase of the number of cloned nodes is more linear.

7.3.2 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.).

7.3.3 Extract Method

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

7.3.4 Refactoring

7.3.4.1 Token Volume

In Fig. **??** we see an increase in the **maintainability score** for refactoring larger clone classes. The tipping point, between a better maintainable refactoring and a worse maintainable refactoring, lies at a token volume of 63 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.

When looking at the relation between maintainability and the **volume** metric, we see that on average the delta volume increases if the average clone volume is less than 60 tokens, but decreases above 60 tokens. When token volume increases, the **number of parameters** of the extracted method rises very slowly. Overall, on average, most extracted methods have only one parameter. When the token volume of clones becomes larger than 160 tokens, we also see that there are more outliers in terms of the number of parameters. For the **complexity** we see that most clones do not have any complexity, because for most refactorings the complexity increases by one. This complexity is the added complexity of the extracted method and implies that the removed clones did not have any complexity. There is also no clear trend regarding the complexity metric against the token volume.

7.3.4.2 Relation

In Table **??** we see the results regarding refactorings that are applied to clones with the specified relation category. We see that 57% of the refactored clones are in a common class. 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% -> 17%). 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 become related.

The maintainability scores displayed in Table **??** show that the most favorable clones to refactor are clones with a *superclass* relation. The most unfavorable is to refactor *unrelated* clones. The main reason for this difference in maintainability is that on average unrelated clones have a higher volume.

The differences in maintainability of relation are, compared to the differences in maintainability of different token volumes, very small. *Common hierarchy* (which has the highest score) has a maintainability score of 0.23 whereas *unrelated* clones (which has the lowest score) have a maintainability score of -0.15. This is a very small difference, which implies that according to our data relations have a minor impact on the maintainability of clones.

7.3.4.3 Return Category

Regarding the return category of refactored clones, we see in Table **??** that this has no major impact on maintainability: the differences between the maintainability scores of the categories are very small. The worst maintainable return category is when methods do not return anything (void), mainly because it is related to a higher number of parameters in the extracted method.

7.3.4.4 Number of Parameters

We see in Figure **??** that the average extracted method that requires more than one parameter decreases maintainability. This is because a higher number of parameters does not only increase the "number of parameters" metric, but also the volume for the extracted method and each of the calls to it. Because of that, we see that the trend of the graph in Fig. **??** decreases relatively rapidly and that the number of parameters in the extracted method has a major influence on maintainability.

Chapter 8

Related work

In this chapter, we present studies in the field of code clone refactoring. The sections are ordered chronologically, because many of these studies use findings of preceding studies.

8.1 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 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 on system design and proposes refactorings. It also features visualizations to show how clones are related in the inheritance structure. SUPREMO 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. In our work, we use these same relations and refactoring techniques.

8.2 Aries

A study by Higo et al. [**higo2004aries**, **higo2008metric**] looks into to what extent clones can be refactored. The authors look into what parts of a clone can be extracted to a new method. They define that partially cloned blocks obstruct method extraction. They propose to make the clone smaller to find a part of the clone that can be refactored. This is similar to our "partial block" category of Section **??** and our discussion in Section **??**.

Higo et al. also look into the amount of variability that can be allowed between clone fragments. Figure **??** shows an example of merging two code fragments from this study. In this figure, the clone spans a part of a different block so it cannot be refactored. They propose to omit both "else if" lines from the merged fragment so that it can be refactored (in our study we would have flagged these clones as

"partial block" and thus unsuitable for refactoring). They also allow variability in literals and variables and replace them with method parameters.

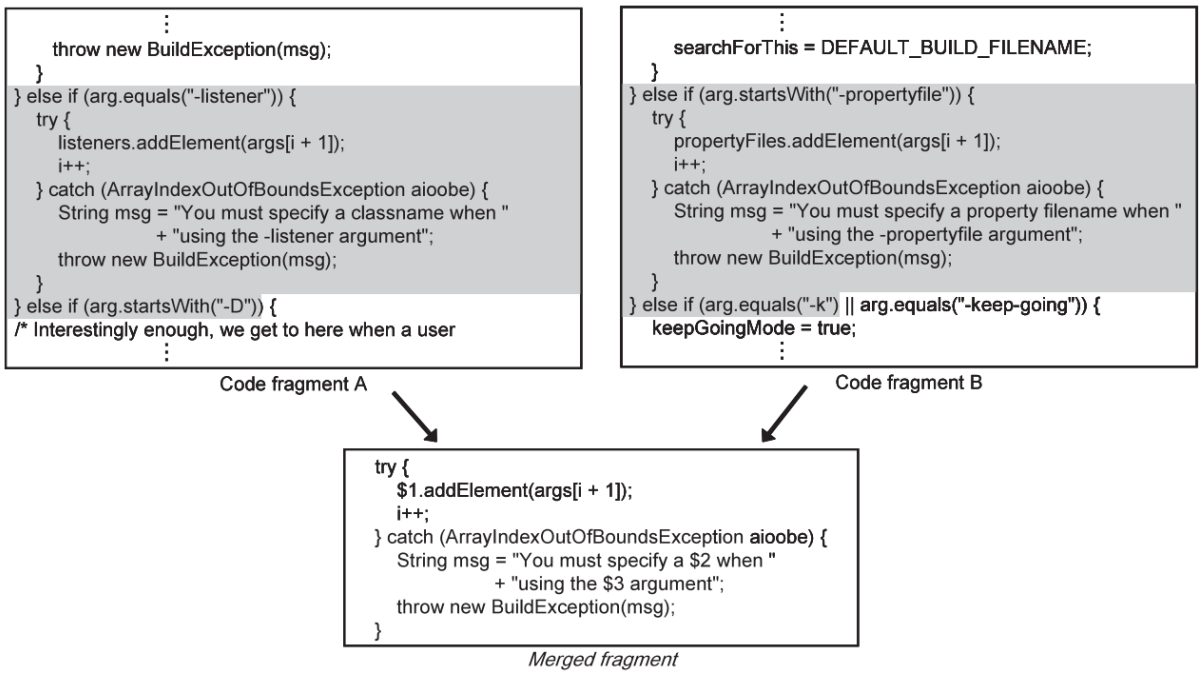

Figure 8.1: Example of merging two code fragments from Higo et al. [higo2008metric].

Higo et al. propose a tool named Aries [**higo2004aries**, **higo2008metric**] that focuses on the detection of refactorable clones. They focus entirely on whether a clone **can** be refactored and not whether it **should** be refactored. This tool only proposes refactoring opportunities and does not provide help in the process of applying the refactoring, which our work does.

8.3 Duplicated Code Refactoring Advisor (DCRA)

Fontana et al. [**fontana2012duplicated**, **fontana2015duplicated**] combine the research by Golomingi [**koni2001scena** with clone types 1 and 3 [**roy2007survey**]. They use a large corpus [**tempero2010qualitas**] on which they perform statistical analyses 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 work (see **??**). We also added the percentages of our work to this table.

	Type 1	Percentage	Type 3	Percentage	Type 1R (Our Work)
Same Class	5,645	32.1%	51,308	28.7%	26.8%
Same External Superclass	4,384	25.0%	66,391	37.1%	20.6%
Unrelated Class	2,758	15.7%	35,035	19.6%	18.4%
Sibling Class	2,721	15.5%	13,868	7.8%	18.2%
Common Hierarchy Class	970	5.5%	3,152	1.8%	0.8%
Same Method	569	3.2%	4,901	2.7%	10.5%
First Cousin Class	416	2.4%	2,980	1.7%	1.4%
Superclass	91	0.5%	981	0.5%	3.1%
Ancestor Class	13	0.1%	281	0.2%	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].

Some of our results differ quite a lot from their results. We think this is mostly accounted due to two

differences in their setup:

- The used clone type definitions differ (type 1 vs type 1R).
- Fontana et al. use a corpus consisting of large higher-quality open-source software systems [**tempero2010qualitas**] where we use a more varied corpus [**githubCorpus2013**].
- Fontana et al. use clone pairs while we use clone classes.

Fontana et al. [**fontana2012duplicated**, **fontana2015duplicated**] propose DCRA, a tool to suggest refactorings for the clones found by the NiCAD tool [**roy2008nicad**, **cordy2011nicad**]. DCRA suggests refactorings to clones found in Java projects based on the mapping between relation and refactoring techniques shown in Table **??**.

8.4 JDeodorant

The most used clone refactoring tool is JDeodorant [**mazinanian2016jdeodorant**]. This tool is based on a set of studies, that look into several aspects of the clone refactoring process. The first of these studies is "Refactoring Clones: An Optimization Problem" by Krishnan et al. [**krishnan2013refactoring**]. The authors describe clones that are refactored by extracting their functionality to a single method. They describe that variability between identifiers and literals have to become parameters in the extracted method. They identify the optimization problem where the variability between cloned fragments should be limited, to avoid limiting the reusability of methods because of too many method parameters. They propose an algorithm that calculates the variance between code fragments on basis of the difference in AST nodes.

In the second study, "Unification and Refactoring of Clones" by Krishnan et al. [**krishnan2014unification**], the authors address the problem of the modifications made to copied fragments and their impact on the refactoring process. They identify a list of expressions in which they have found variability, which is displayed in Table **??**. The authors map the variability in this table to refactoring opportunities. They list a set of four preconditions for the refactoring process:

1. The parameterization of differences between the matched statements should not break existing data-, anti-, and output-dependences.
2. The unmatched statements should be movable before or after the matched statements without breaking existing data-, anti-, and output-dependences.
3. The duplicated code fragments should return at most one variable of the same type.
4. Matched branching (`break`, `continue`) statements should be accompanied with corresponding matched loop statements.

In our study, we address precondition 1 and 2 and propose a solution. To address precondition 1, we only refactor variability in such a way that existing data-, anti-, and output-dependences are not affected (see Section **??**). For precondition 2, we propose using conditional or lambda expressions to wrap the difference in statements (see Section **??**). For precondition 3 and 4, we flag these clones as "not refactorable" (see Section **??**).

Difference Type	***Example***	
Variable Identifier	`int x = y;`	`int x = z;`
Literal Value	`String s = "s1";`	`String s = "s2";`
Method Name	`expr.foo(arg);`	`expr.bar(arg);`
Argument Number	`foo(arg0, arg1);`	`foo(arg0);`
Caller Expression	`expr.foo(arg);`	`foo(arg);`
Array Dimension	`int x = a[i];`	`int x = a[i][j];`
AST Compatible	`int x = foo();`	`int x = 5;`
Operator	`int x = y + z;`	`int x = y - z;`
Variable Type	`int x = 5;`	`double x = 5;`

Table 8.3: Expression variability table from Krishnan et al. [krishnan2014unification]

A study by Tsantalis et al. [**tsantalis2015assessing**] looks into the refactorability of code clones.

This results in a set of eight preconditions to determine whether a cloned fragment is refactorable. Four of these preconditions are similar to the Krishnan et al. [**krishnan2014unification**] study. The following preconditions have been added:

5. Matched variables having different subclass types should call only methods that are declared in the common superclass or are being overridden in the respective subclasses.
6. The parameterization of fields belonging to differences between the mapped statements is possible only if they are not modified.
7. The parameterization of method calls belonging to differences between the mapped statements is possible only if they do not return a void type.
8. The mapped statements within the clone fragments should not contain any conditional return statements.

In our study, we simplify precondition 5 by not allowing variability in types, because of which we satisfy this precondition. We address precondition 6 by allowing modified statements as return values (and turning the calls to the extracted method into variable assignments). If more than 1 variable is assigned or other return values would be present, we flag the cloned fragments as "multiple return values" (see Section **??**). We address precondition 7 by using lambda expressions, which allow methods without return value to be used as parameters (using the `Runnable` interface). We address precondition 8 by analyzing all branches in the code. If all branches return, the fragment can still be refactored even though there is a conditional return statement.

After these studies that determine what preconditions apply to be able to refactor modified clones, the authors propose a tool named JDeodorant that helps in the process of refactoring such clones [**mazinanian2016jdeodora**
This tool takes the output of many popular clone detection tools [**kamiya2002ccfinder**, **deissenboeck2010flexible**, **jiang2007deckard**, **cordy2011nicad**] and proposes refactoring opportunities. The tool is integrated in the Eclipse IDE, making it easier to directly apply the refactoring. The tool is based on previously discussed preconditions and does not allow refactorings if the preconditions are not met.

A study by Tsantalis et al. [**tsantalis2017clone**] looks into how lambda expressions can be used to help in the process of refactoring clones. They do this in order to mitigate some of the earlier established preconditions. Working with lambda expressions makes it possible to refactor statement gaps and expression differences. They find that lambda expressions are very suitable to refactor clones with variability. The authors do not discuss the impact of such refactorings on system maintainability, their main focus is to positively influence the refactorability of clone classes.

8.5 Refactoring Technique for Large Groups of Software Clones

In a thesis by AlWaqfi [**alwaqfi2017refactoring**] the author proposes a method for assessing refactorability of clone classes, whereas most of the previous tools only focus on clone pairs. The author surveys a set of clone detection tools [**kamiya2002ccfinder**, **baxter1999clonedr**, **jiang2007deckard**, **cordy2011nicad**] and uses a publicly available dataset [**tsantalis2015assessing**] to measure the amount of clones found by these tools. The resulting data shows that there is a lot of variability in the amount of detected clones between different clone detection tools. The author measures of how many clone instances there are in clone classes. According to these experiments, about 68% of clone classes consist of 2 clone instances, 13% have 3 clone instances, 7% have 4 clone instances and about 12% is even larger.

The authors perform a lot of statistical comparisons between clone detection tools. The part most relevant to our study is their measurements regarding refactorability. They find that about 29-54% of clones are refactorable in the set of projects they inspect (Table 6.13 of [**alwaqfi2017refactoring**]). To determine whether a clone is refactorable, the author uses the set of eight preconditions proposed by Tsantalis et al. [**tsantalis2015assessing**].

8.6 Pattern-based clone Refactoring Inspection (PRI)

A study by Chen et al. [**chen2018clone**] addresses the problem that developers cannot easily determine which clones can be refactored or how they should be maintained scattered throughout a large code base in evolving systems. They propose PRI, a technique for managing refactorings for clones with a Sibling relation. PRI is a plugin for the Eclipse IDE that shows which refactoring techniques can be used to refactor clones. Because it is integrated in the Eclipse IDE, the refactorings can be executed largely automatically, as this IDE has a large set of tools for performing such transformations.

Another feature of the PRI tool is to find incompletely refactored clones. To do this, PRI traces clones through revisions of a software project. When a clone class is incompletely refactored, PRI informs the user that there are clone instances for which an extracted method has already been created.

8.7 Impact of Clone Refactoring on External Quality Attributes

A study by Vashisht et al. [**vashisht2018impact**] determines the impact of clone refactoring on several quality attributes. They calculate the metrics of the system before refactoring, detect clones using CCFinder [**kamiya2002ccfinder**], use JDeodorant to refactor a subset of the detected clones [**mazinanian2016jdeodorant**] and measure the metrics again after refactoring to assess the impact of the refactoring process on quality attributes. They conclude that clone refactoring affects some quality attributes positively and some negatively.

In this study, the authors only compare before- and after-snapshots of refactoring the entire system. This makes it hard to determine why some metrics are negatively influenced. In our study, we consider each separate refactoring, which yields more precise results.

8.8 Clone Recommendation System for Extract Method Refactoring

Yoshida et al. [**yoshida2019proactive**] propose a plugin for the Eclipse IDE that recommends clone refactoring candidates on basis of keystroke tracking. Whenever the user performs the "Extract Method" refactoring, the proposed system scans the project for clones that can now use the extracted method. It then shows the developer all places that should be refactored aswell now that a new method has been created. The approach was validated by requesting human participants to apply refactorings to open source systems. On average, it took participants 1,043 seconds to refactor 2.75 clones. They report that this is 35% faster than when working with a popular clone detection tool (GemX).

8.9 Effect of clone refactoring on the size of unit test cases

Badri et al. [**badri2019measuring**] measure the impact on clone refactoring on metrics that affect the testability of software. The metrics they measure are size, cyclomatic complexity and coupling. To collect this data, they use two systems that have been de-duplicated manually. They conclude that there is a strong, positive correlation between clone refactoring and the reduction of the size of unit tests. They also show that code quality attributes that are related to testability are significantly improved.

Chapter 9

Conclusion

In this work, we defined automatically refactorable clones and created a tool to detect and refactor them. We measured statistical data with this tool over a large corpus of open-source Java software systems to get more information about the context of clones and how refactoring them influences system maintainability.

To determine which refactoring techniques are most suitable to refactor most clones, we analyzed their context. We measured the **relation** through inheritance of clone instances in a clone class. In our corpus, we found that 37% of clones are found in the same class. 24% of clones are in the same inheritance hierarchy. Another 24% of clones are unrelated. The final 15% of clones have the same interface. We also looked at the **location** of clones: 78% of clones are found at method-level of which 77% is found in the body of a method or constructor. Because of this, the "Extract Method" refactoring technique is most suitable to refactor most clones.

We built a tool that can automatically apply refactorings to 28% of clones in our corpus using the "Extract Method" refactoring technique. The main reason our tool could not refactor all clones is that many clones span certain statements that obstruct method extraction, for instance when code outside the method body is part of a clone.

We measured the effect of clone refactoring on the volume, cyclomatic complexity, number of parameters and duplication metrics of the codebase. We measured these metrics before- and after each clone class that was refactored to determine the impact of each refactoring on system maintainability. We found that the most prominent factor influencing maintainability is the size of the clone. For our dataset, at a token volume of 63 tokens per clone instance the average clone refactoring improved system maintainability. We define the *token volume* as the combined number of tokens in all clone instances in a refactored clone class.

Another factor with a major impact on maintainability is the number of parameters that the extracted method requires to get all required data. Most refactored clones for which extracted methods with more than one parameter were created decreased maintainability. We found that the inheritance relation of the clone and the return value of the extracted method has only a minor impact on system maintainability.

9.1 Threats to validity

9.1.1 CloneRefactor

CloneRefactor can refactor all clones that are flagged "can be extracted". This is done through a series of AST transformations as described in Section **??**. Currently, these refactoring methods do not yet allow for variability (type 2R and 3R), as allowing variability quickly increases the complexity of the refactoring problem and the number of edge cases to consider [**tsantalis2015assessing**]. Because of this, CloneRefactor does not yet realistically emulate human refactoring efforts, as humans would take variability between code fragments into account. This influences the results, as the numbers obtained might be higher if it would more realistically emulate human behavior.

9.1.2 Corpus

We used a corpus of 2,267 open source Java projects from GitHub for our experiments. This corpus consists of only Maven projects and we only take the "production sources" folder of these projects take

into account. We inspected the results of CloneRefactor for outliers and excluded projects that have generated sources in their "production sources" folder. This increases the chance that we are not taking into account test sources and generated files, but it does not eliminate this chance. Because of that, there is a threat to validity in the source code in the corpus, as it potentially includes sources that do not reflect human-maintained production code.

We think that there is value in running our experiments with different corpora. 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 a 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.

9.1.3 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 practitioners. We have determined maintainability scores only based on literature input and statistical 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 transformations that CloneRefactor performs.

9.1.4 Aggregated Maintainability Score

In Section **??** we proposed an aggregation to measure a maintainability score on small-grained changes. The assumptions stated in this section influence the validity of the maintainability scores shown in our results. We noticed that the maintainability scores displayed in Table **??** and Table **??** were mostly influenced by the differences in the "number of parameters" metric. This is because, for the scores, we normalized the metrics over all values obtained by these metrics. As our data regarding "number of parameters" has a very low standard deviation, small changes in this metric will influence the maintainability score a lot.

9.2 Future work

9.2.1 Automated Refactoring for more metrics

In this study, we apply automated refactoring to analyze the before- and after state of refactored source code. This allows us to better determine which thresholds identify problems that should be refactored.

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

To refactor a method with many parameters, one should 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 the places in which they are used together and replace the variables by the newly created abstractions. If long parameter lists can be automatically refactored,

automated clone refactoring can be improved by automatically grouping data if the extracted method requires many parameters.

9.2.1.2 Method complexity/size

Refactoring complex/large methods can largely be done automatically by applying "Extract Method" to large methods. 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 **??**).

9.2.2 Type 4 clones

Type 4 clones are defined as *two or more code fragments that perform the same computation but implemented through different syntactic variants.* [**roy2007survey**]. We think that there are relevant research opportunities in refactoring type 4 clones. Although type 4 clone instances perform the same computation, they may still differ in other aspects. For instance, one alternative might be easier to maintain or have a better performance. There are interesting research opportunities in automatically choosing the better alternative among type 4 clone instances.

9.2.3 Naming of refactored methods/classes/etc.

In this study, 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 we used a maintainability model [**heitlager2007practical**] that does not take naming quality into account. Because of that, the results of our experiments are not dependent on the names we give the generated declarations.

When using our work for 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 reliably, it is possible to apply automated 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 based on their body and context. However, the source code of this study is not available, making it harder to apply their findings.

In a recent study by Alon et al. [**alon2018code2seq**] the authors 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 it only guesses the name on basis of the body of the method and not its context.

9.2.4 Looking into other languages

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 yield 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 our refactoring efforts based on Java, which might differ from the prioritization that can result from running our experiments with other programming languages.

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. Thank you for helping me concisely deliver the message I try to deliver.

Next, we 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 valuable insights. Your proofreading efforts helped me a lot to write this thesis.

I would like to thank my supervisor, Xander Schrijen, for all his input during our weekly meetings. We had long discussions about all my ideas, which helped to put my ideas in perspective.

I would like to thank Danny van Bruggen for all his help in my use of JavaParser. I am amazed by his willingness to help others to tackle their problems. JavaParser is at the core of the CloneRefactor tool and without Danny, I would never have come as far in the development of this tool.

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

Number of Parameters Table

Table **??** shows average scores and metrics for all refactored clones with the specified number of parameters in the extracted method. The **Token volume** column refers to the average token volume of all refactored clones with the specified number of parameters. We define the *token volume* as the combined number of tokens in all clone instances in a refactored clone class. Further explanation of the columns used in this table can be found in Section **??**

Parameters	***Volume***	***Duplication***	***Complexity***	***Token volume***	*#*	***Score***
0	0.44	-52.35	0.89	52.35	3,231	1.06
1	1.70	-49.57	0.82	49.57	5,793	0.09
2	1.85	-54.81	0.84	54.81	2,973	-0.83
3	1.34	-70.15	0.84	70.15	500	-1.64
4	5.51	-74.11	0.80	74.11	108	-2.60
5	-0.40	-106.49	0.77	106.49	35	-3.17
6	-14.00	-144.31	0.69	144.31	13	-3.57
7	-5.50	-152.50	0.75	152.50	4	-4.59
8	13.86	-132.14	0.71	132.14	7	-5.97
9	-18.50	-186.00	0.75	186.00	4	-6.02
10	-36.00	-218.67	1.00	218.67	3	-6.56
11	9.00	-178.00	1.00	178.00	2	-8.49
12	7.00	-192.00	1.00	192.00	2	-9.29
13	2.00	-206.00	1.00	206.00	1	-10.05
14	-68.50	-318.00	1.00	318.00	2	-9.05
15	-572.00	-1,053.00	1.00	1,053.00	1	3.38
18	-202.00	-708.00	1.00	708.00	1	-7.57
21	96.00	-294.00	1.00	294.00	1	-18.17
23	25.00	-346.00	1.00	346.00	1	-18.66
28	-138.00	-624.00	1.00	624.00	1	-18.75
Grand Total	**1.33**	**-53.26**	**0.84**	**53.26**	**12,683**	**0.00**

Table A.1: Average metric values for refactorings of clone classes with the specified number of parameters.