A Refactoring Technique for Large Groups of Software Clones

Asif S. AlWaqfi

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfilment of the Requirements for

the Degree of Master of Computer Science at

Concordia University

Montréal, Québec, Canada

February 2017

CONCORDIA UNIVERSITY

Division of Graduate Studies

This is to certify that the thesis prepared

By : **Asif S. AlWaqfi**

Entitled : **A Refactoring Technique for Large Groups of Software Clones**

and submitted in partial fulfilment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee :

______________________ Chair
Dr. Tristan Glatard

______________________ Examiner
Dr. Weiyi Shang

______________________ Examiner
Dr. Emad Shihab

______________________ Supervisor
Dr. Nikolaos Tsantalis

Approved by ______________________
Dr. Volker Haarslev
Graduate Program Director

__________ 2017.

____
Dr. Amir Asif
Dean of Faculty
(Engineering and Computer Science)

ABSTRACT

Code duplication, also known as software clones, is a persistent problem in software systems that is usually associated with error-proneness and poor software maintainability. Despite the fact that clone detection is a mature research field, clone refactoring has not been equally investigated. Clone refactoring requires the unification and merging of duplicated code, which is a challenging problem because of the changes that take place on the initial clones after their introduction.

In recent years, more research works attempted to address the challenges around clone refactoring by applying different techniques; however, they suffer from poor accuracy or performance issues, especially for large clone groups containing more than two clone instances. We contribute to this field by proposing an automated approach that a) finds refactorable subgroups (consisting of three clones or more) within the original group of clones, b) finds the statements that to be merged and extracted in a fast yet accurate way, and c) assesses the refactorability of clone subgroups.

We evaluated our approach in comparison to the state-of-the-art, and the results show that we have a high accuracy in matching the clone statements, while maintaining high performance. In a case study, where we carefully examined all clone groups in project JFreeChart 1.0.10, we found that around 49% of the 98 clone subgroups are actually refactorable. Finally, we conducted a large-scale study on over 44k clone groups (13.6k groups containing 3 clones or more) detected by four clone detection tools in nine open source projects to assess the refactorability for clone groups. The outcome of this study revealed the presence of 2,833 refactorable clone subgroups that contain in total 13,398 clone instances.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Nikolaos Tsantalis, for seeing my abilities and accepting me as one of his students. I would like to thank him also for his continues support towards my research and in writing my thesis, as well as for his patience and motivation which inspired and made me thrive to accomplish more in my research.

Apart from my advisor, I would like to thank my thesis examiners, Dr. Weiyi Shang and Dr. Emad Shihab, for taking the time to read my thesis and for their valuable suggestions and comments. I would like to thank other faculty members of the Department of Computer Science and Software Engineering as well, for the necessary guidance.

I thank my fellow lab mates and my friends in Concordia University for all their support towards the completion of this thesis. I express my gratitude to the staff members of our university for their help in providing a clean and safe environment for me to work.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

System development life cycle (SDLC) represents the phases a software has to go through until the user is able to interact with a running system. SDLC starts by collecting requirements from users (Analysis phase), designing the system as a collection of modules or subsystems (Design phase), implementing the design into source code (Implementation phase), and ends by testing and delivering the software product to the user (Testing phase). However, after the initial delivery of the product, the Maintenance phase starts, during which bugs are getting fixed or new requirements are being implemented.

Studies have shown that the Maintenance phase holds the largest percentage of the total software development cost. Grubb et al. [GT03] estimated that 40%-70% of the money spent on a system during its lifetime is spent on maintenance. Mobley [Mob90] and Maggard et al. [MR92] reported that 15%-40% of production cost is spent on maintenance. A study by Wireman [Wir89] estimated the maintenance cost for a group of companies would increase from $200 billions in 1979 to $600 billions in 1989. Coleman et al. [Col+94] reported that, in 1992, 60%-80% of the research and development staff at Hewlett-Packard were involved in maintenance tasks, and that

40%-50% of production cost was spent on maintenance.

The quality of code written during the Implementation phase affects significantly the effort require to complete future maintenance tasks. Dekleva [Dek92] reported in a survey that one of the most severe problems in maintenance is the quality of source code, while Chapin [Cha99] mentioned in a survey that 48% of maintenance problems are related to source code quality, such as poor documentation, high complexity, and poor code structure quality.

The improvement of source code quality can be achieved through *restructuring*. Restructuring is defined by Chikofsky et al. [CC90] as *"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)"*. A special case of *restructuring* in object-oriented software development is *Refactoring*, a term introduced by Opdyke [Opd92]. The object-oriented programming paradigm is based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.

Refactoring improves the quality of software by removing code smells found in the code. Code smells according to Mens et al. [MT04] and Fowler et al. [Fow+99], are *"structures in the code that suggest (sometimes scream for) the possibility of refactoring"* . There is a large variety of code smells, such as duplicated code (or referred to as code clones), and Feature Envy (i.e., when a method uses more features from another class than the one it exists in). The focus of this thesis will be on duplicated code refactoring as a means to enhance software quality and maintainability.

Duplicated code (code clones) increase maintenance effort and cost [LW08], error-proneness when clones are updated inconsistently [Jue+09], and code instability [MRS12]. To address these problems, different techniques were proposed as part of clone management [Kos08]. Clone management [Kos08] can be *preventive* (i.e., avoid the introduction of new clones), *corrective* (i.e., eliminate existing clones), and *compensative* (i.e., limit the negative impact of clones through automatic clone synchronization when a change occurs). This work focuses on the corrective aspect of clone management. It proposes a new approach to handle clone groups containing three or more clone instances, by finding the differences among their statements, mapping properly reordered statements, and finally assessing if the clones in the group can be safely refactored.

1.1 Motivation

The motivation behind this thesis comes from the three main points discussed extensively in the next sub-sections.

1.1.1 Duplicated Code is an extensive and persistent problem

Code clones can comprise a high percentage of systems code base. For instance, Wang and Godfrey estimated 10%-20% of the code in large systems are clones (a copy of other code with or without changes) [WG14], while Ducasse et al. reported the duplicated code in COBOL (COmmon Business Oriented Language, a programming language used to solve business problems [Mic]) systems was about 50% [DRD99] of the written code. Baxter et al. [Bax+98] detected that 12% of the code being duplicated; Baker [Bak95] identified 13%-20% of the code being cloned; Lague et al. [Lag+97] found that clone code is between 6.4%-7.5%; Mayrand et al. [MLM96] estimated the duplicated code in industrial systems ranges within 5% – 20%; and Ducasse et al. [DRD99] reported that clones can comprise 10% –15% of the source code of large systems. A survey done by Arcoverde et al. [AGF11] reported that the most persistent code smell is duplicated code.

To add further evidence, we used a publicly available clone dataset [TMK15] that includes the clones detected by four clone detection tools in 9 open source projects. According to Bellon et al. [Bel+07], the clone detectors that we selected have a precision (P) and recall (R) of: CCFinder (P: 72%, R: 72%), CloneDR (P: 100%, R:9%). Deckard is more scalable than CloneDR [Jia+07], and NiCad has a precision and recall of (P: 96%, R: 100%) [RC09]. For NiCad we used two configurations: Blind, all identifiers are replaced with a single pseudo-variable; while Consistent, the same identifier is replaced with a single pseudo-variable X_{index} . The number of detected clone instances are shown in **Table 1.1**. The numbers in the table represent number of clones detected by each tool, where the last row is the total of clones detected by the tool in all examined projects. These clones are detected in groups, where the minimum size for each group is two (i.e., it contains only 2 clone instances). In the context of this thesis, we focus more on clone groups containing more than two instances, before the majority of previous works has mostly focused on clone pairs (i.e., groups of two clone instances).

Table 1.2 shows the percentage of clone groups containing more than two instances for each examined project. The last row represents the average percentage of groups that consist of three or more clone instances. The average percentage ranges between 28%-35% , showing that there is a significant number of clone groups with multiple instances (i.e., more than two instances). Moreover, these groups contain over 60% of the detected clone instances with a total of 94,650 instances. Table 1.3 shows the percentage of groups detected by each clone detection tool, where groups are categorized based on the number of clone instances they contain. The groups containing 3, 4, and more than 9 clone instances have the highest percentage, while the other categories range from 0.6% to 2.67%.

Table 1.1: Number of clone instances detected by each tool

Project	Clone Detection Tool				
	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	2,798	4,071	1,812	2,853	2,282
Columba	2,046	3,987	1,925	2,260	1,627
EMF	5,479	5,973	955	4,690	3,851
Hibernate	3,733	6,191	3,581	3,041	2,314
JMeter	2,018	525	1,974	2,187	1,703
JEdit	939	1,630	864	1,209	863
JFreeChart	9,546	8,804	6,932	4,793	4,211
JRuby	3,257	4,212	1,628	3,293	2,373
SQuirreL SQL	7,299	8,646	2,204	4,765	3,800
Total	37,115	44,039	21,873	29,091	23,024

Table 1.2: Percentage of groups containing 3 or more clone instances

Project	Clone Detection Tool				
	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	28%	25%	20%	31%	37%
Columba	23%	28%	27%	35%	35%
EMF	34%	28%	40%	38%	36%
Hibernate	30%	25%	28%	33%	32%
JMeter	23%	44%	25%	31%	31%
JEdit	18%	22%	12%	31%	26%
JFreeChart	43%	28%	39%	40%	41%
JRuby	20%	25%	36%	35%	31%
SQuirreL SQL	32%	36%	31%	32%	43%
Average	28%	29%	29%	34%	35%

Table 1.3: Percentage of groups containing x clone instances ($3 \leq x \leq 9$) in the dataset [TMK15]

Tool / Group Size	3	4	5	6	7	8	9	> 9
CCFinder	11.42%	7.69%	2.70%	2.23%	1.10%	1.04%	0.75%	3.47%
CloneDR	12.43%	5.95%	2.43%	1.95%	1.07%	0.88%	0.51%	3.24%
Deckard	15.49%	6.99%	2.08%	1.80%	0.85%	0.84%	0.40%	1.94%
NiCad Blind	15.22%	7.91%	3.09%	2.52%	1.11%	1.17%	0.68%	4.55%
NiCad Consistent	14.84%	7.77%	3.05%	2.65%	1.19%	1.23%	0.71%	4.26%
Average	13.88%	7.26%	2.67%	2.23%	1.06%	1.03%	0.61%	3.49%

1.1.2 Lack of reliable and mature clone refactoring tools

As mentioned before, clone management encompasses three categories of actions; however, researchers in the field mainly focused on two of them, namely preventive and compensative. Many tools and techniques were developed and proposed by researchers such as, CCFinder [KKI02], and NICAD [RC08b] to assist developers in finding clones scattered through project files. Other tools such as CloneTracker [DR08; Ngu+12] act as live recommendation systems to notify users of clones being modified.

The corrective aspect of clone management covers the elimination of clones through refactoring. The process of refactoring requires first to compare the code clones and find differences between them, such as different method calls that need to be parameterized in the common code that will be extracted (i.e., a parameter should be introduced in the extracted common code for each difference found in the clones).

Current clone refactoring tools support mostly trivial differences for parameterization. Eclipse allows only differences in local variable names, and it can be applied on clones within the same Java file only. CeDAR is another Eclipse plug-in proposed and developed by Tairas et al. [TG12], which extends the Eclipse refactoring engine to allow more kinds of differences between the clones. CeDAR was able to refactor 18% of the cases reported by Deckard in comparison to 10.6% that could be refactored by Eclipse. However, CeDAR is limited to *Type-1* (i.e., same code except for differences in whitespace or comments) and *Type-2* (i.e., clones containing differences in identifiers, literals, and types) clones. Meng et al. [Men+15] is the first work that supports more advanced clone refactoring operations, by extracting the common code, creating new types and methods (if necessary), and introducing return objects, but despite these improvements *Type-3* (i.e., clones with statements added, removed, or changed) clones are not supported.

Lastly, a work done by Tsantalis et al. [TMK15] can be seen as the state-of-the-art tool in refactoring pairs of clones (i.e., groups containing two clone instances) that addressed all previous limitations and supports automated refactoring. The only limitation is that it does not support the refactoring of clone groups containing more than two clone instances.

1.1.3 Developers care about clones

In a survey conducted by Yamashita et al. [YM13], about determining the knowledge of developers and their interest in code smell, one of the questions asked to developers was to rank code smells according to their perceived importance **"Are there specific code smells / anti-patterns that you are concerned about? Please list them in order of their perceived importance"**. Duplicated Code was the most popular smell and had the **highest** rank with 19.53 points (Table 4, Yamashita et al. [YM13]) followed by Long Method with 9.78 points, Accidental Complexity with 8.32 points, and Large Class with 7.09 points.

In another survey by Silva et al. [STV16] it was observed that developers are seriously concerned about avoiding code duplication, when working on a given maintenance task. Developers often apply refactorings, especially Extract Method refactorings, to reuse code and avoid duplicating the same functionality.

1.2 Contribution

This work deals with the problem of refactoring (or merging) clone groups containing three or more clone instances. The main challenge is the scalability of the solution, since as the size of a clone group increases, the number of pair-wise clone instance comparisons/combinations grows factorially $\binom{n}{2}$. The contribution of this work can be summarized as:

1. We propose a *multi-clone* statement mapping approach that uses control and data dependence information as well as date type information.

2. We propose an algorithm to find subgroups of clone instances that have a minimum number of differences, within a group of clones. In this way, we reduce the number of combinations to be examined, because we avoid the comparison of clones in different subgroups.

3. We show evidence that large clone groups can become refactorable by splitting them into smaller subgroups.

The rest of the thesis is organized as follows: Chapter 2 covers background knowledge that is needed throughout the thesis. Chapter 3 contains a review of the related literature. Chapter 4 describes the proposed solution for multi-clone analysis and refactoring. Chapter 5 presents a qualitative study on 847 clone groups containing 2307 clone instances in total. Chapter 6 contains statistical information from the analysis of clone groups detected by 4 different clone detection tools in 9 open-source projects. Finally, the conclusions of the thesis and future work are discussed in Chapter 7.

Chapter 2

Background

2.1 Software Maintenance

Software Maintenance is the last phase and an important part of System Development Life Cycle (SDLC), because systems evolve and new requirements are needed as time passes. Lientz et al. [LS81] reported in a survey that around half of the development time is spent on maintenance, and over forty percent of the time is spent on enhancements and adding new features to running systems. Software Maintenance is classified into four types [Tsu+15] based on the goal it carries out:

- **Adaptive**: Ensure system compatibility with the changes in its environment. For instance, sometimes there are improvements to the hardware the system is running on; however, the software needs to be updated to ensure the system will keep running normally and will not fail.
- **Perfective**: Improve the performance and maintainability of a working system.
- **Corrective**: Identify and correct problems in a system, after it has been pushed to a working environment that the users can access and work on.

- **Preventive**: Prevent potential faults occurring in the future. The system is still running perfectly with no bugs, but the developers realize that they have to do some changes to prevent faults in the future if certain conditions are met.

Our work in this thesis is associated with the following maintenance types:

Perfective because duplicated code degrades the quality of the system, which affects future maintenance tasks.

Corrective because duplicated code increases the effort required to fix existing bugs repeated in many different places of the source code.

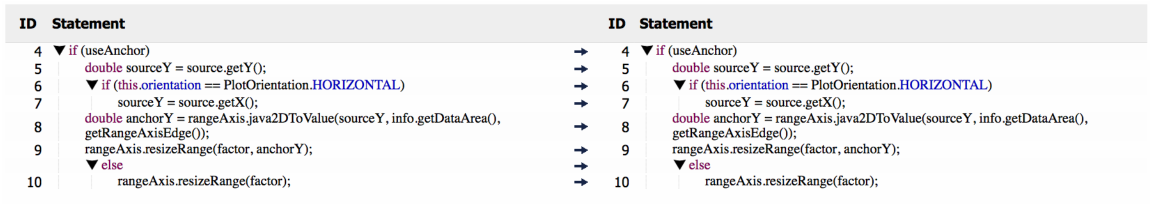
Preventive because maintenance is a continuous process and duplicated code can be a source of inconsistent changes and future bugs.

2.2 Code Clone Types

Copying existing functionality to create new one by making minor modifications can lead to *divergent* clones (i.e., pieces of code that were originally the same, but become syntactically more distant after they undergone some modifications). Some of these changes may include renaming variables, removing or adding statements, changing method calls or instantiated objects. Based on the changes duplicate code is categorized into four types [RCK09].

2.2.1 Type I

The duplicated code, excluding differences in white-spaces, comments, and layout is exactly the same.

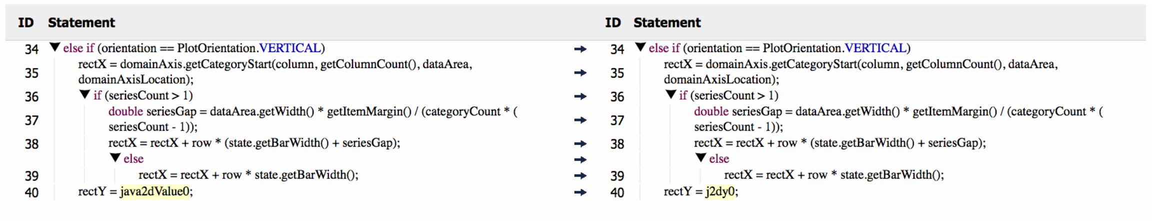


ID	Statement		ID	Statement
4	▼ if (useAnchor)	→	4	▼ if (useAnchor)
5	double sourceY = source.getY();	→	5	double sourceY = source.getY();
6	▼ if (this.orientation == PlotOrientation.HORIZONTAL)	→	6	▼ if (this.orientation == PlotOrientation.HORIZONTAL)
7	sourceY = source.getX();	→	7	sourceY = source.getX();
8	double anchorY = rangeAxis.java2DToValue(sourceY, info.getDataArea(), getRangeAxisEdge());	→	8	double anchorY = rangeAxis.java2DToValue(sourceY, info.getDataArea(), getRangeAxisEdge());
9	rangeAxis.resizeRange(factor, anchorY);	→	9	rangeAxis.resizeRange(factor, anchorY);
	▼ else	→		▼ else
10	rangeAxis.resizeRange(factor);	→	10	rangeAxis.resizeRange(factor);

Figure 2.1: An example of Clone Type I

2.2.2 Type II

Type II is a superset of **Type I**, additionally including differences in the identifiers, literals, and types.

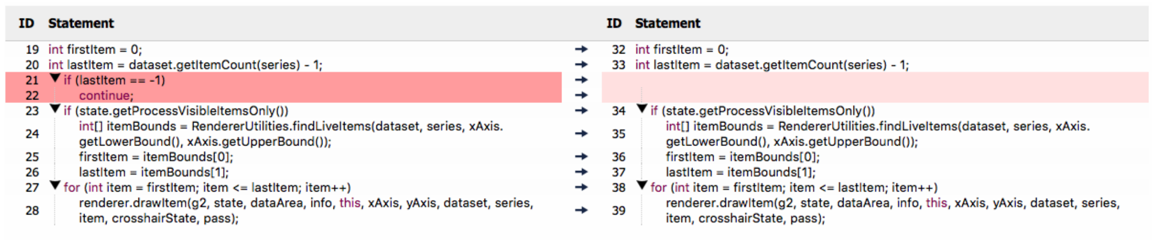


ID	Statement		ID	Statement
34	▼ else if (orientation == PlotOrientation.VERTICAL)	→	34	▼ else if (orientation == PlotOrientation.VERTICAL)
35	rectX = domainAxis.getCategoryStart(column, getColumnCount(), dataArea, domainAxisLocation);	→	35	rectX = domainAxis.getCategoryStart(column, getColumnCount(), dataArea, domainAxisLocation);
36	▼ if (seriesCount > 1)	→	36	▼ if (seriesCount > 1)
37	double seriesGap = dataArea.getWidth() * getItemMargin() / (categoryCount * (seriesCount - 1));	→	37	double seriesGap = dataArea.getWidth() * getItemMargin() / (categoryCount * (seriesCount - 1));
38	rectX = rectX + row * (state.getBarWidth() + seriesGap);	→	38	rectX = rectX + row * (state.getBarWidth() + seriesGap);
	▼ else	→		▼ else
39	rectX = rectX + row * state.getBarWidth();	→	39	rectX = rectX + row * state.getBarWidth();
40	rectY = java2dValue0;	→	40	rectY = j2dy0;

Figure 2.2: An example of Clone Type II

2.2.3 Type III

Type III is a superset of **Type II** with further modifications such as, added, removed, or changed statements. Statements that appear in one clone fragment, but not the others are called gapped/unmapped statements. Figure 2.3 below is an example of a pair of matched clones were statements 21-22 are considered as gaps, because they appear in the left fragment only and could not be mapped with any statement in the second fragment on the right.



ID	Statement		ID	Statement
19	int firstItem = 0;	→	32	int firstItem = 0;
20	int lastItem = dataset.getItemCount(series) - 1;	→	33	int lastItem = dataset.getItemCount(series) - 1;
21	▼ if (lastItem == -1)	→		
22	continue;	→		
23	▼ if (state.getProcessVisibleItemsOnly())	→	34	▼ if (state.getProcessVisibleItemsOnly())
24	int[] itemBounds = RendererUtilities.findLiveItems(dataset, series, xAxis. getLowerBound(), xAxis.getUpperBound());	→	35	int[] itemBounds = RendererUtilities.findLiveItems(dataset, series, xAxis. getLowerBound(), xAxis.getUpperBound());
25	firstItem = itemBounds[0];	→	36	firstItem = itemBounds[0];
26	lastItem = itemBounds[1];	→	37	lastItem = itemBounds[1];
27	▼ for (int item = firstItem; item <= lastItem; item++)	→	38	▼ for (int item = firstItem; item <= lastItem; item++)
28	renderer.drawItem(g2, state, dataArea, info, this, xAxis, yAxis, dataset, series, item, crosshairState, pass);	→	39	renderer.drawItem(g2, state, dataArea, info, this, xAxis, yAxis, dataset, series, item, crosshairState, pass);

Figure 2.3: Example of Gapped Statements highlighted in red

2.2.4 Type IV

This type is very difficult to be detected by clone detection tools, due to clones having a completely different syntactic implementation, but similar functionality [RCK09; RC09]. Also, it is not necessary that they were copies of each other, as they might be coded by different developers [RC07]. Below is an example of a Type-4 clone were one of the fragments is written using a while-loop, while the other uses a recursive-function:

```
void loopOver(int var){
  while(var > 0){
        System.out.println(var);
        var--;
     }
  }
```

```
void loopOver(int var){
 if(var > 0){
       System.out.println(var);
       loopOver(--var);
    }
}
```

2.3 Clone Detection tools

2.3.1 Clone Detection Techniques

There are many available tools for detecting clones in source code. These tools use different techniques to detect duplicate code [RCK09], for which we will give a general overview in this section along with some indicative tools.

2.3.1.1 Textual technique

In this approach the raw source code is textually compared after some normalization and formatting of the source code before the comparison is executed.
- Tools following this approach are NiCad [RC08a] and Simian [Har16].

2.3.1.2 Lexical technique

This technique is referred to as token-based approach. The source code is converted into a sequence of tokens, which are scanned for duplicate sub-sequences.
- Tools following this approach are CCFinder [KKI02] and CP-Miner [Li+06].

2.3.1.3 Syntactic technique

In contrast to previous techniques, this technique requires the source code to be parsed into an *abstract syntax tree* (AST), and then either a tree or structural matching approach is used to detect clones.

- **Tree matching approach** detects clones by finding similar sub-trees.
  - Tools following this approach are CloneDR [Bax+98] and ccdiml [Pro].
- **Metrics-Based approach** Metrics from code fragments are gathered in feature vectors, and then the similarity of these feature vectors is computed.
  - Tools following this approach are DECKARD [Jia+07] and SMC-Similar Method Classifier [Bal+99].

2.3.1.4 Semantic technique

This technique uses static analysis to generate the Program Dependence Graph (PDG) of functions, and then finds isomorphic sub-graphs.

- A tool following this approach is the tool developed by Jens Krinke [Kri01]

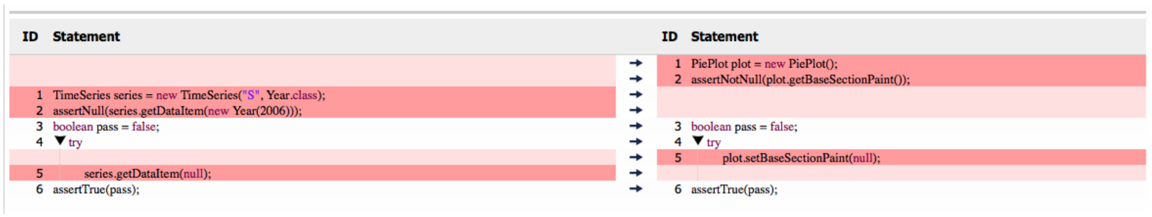
There are many techniques and tools to detect clones; however, their performance and results impose other challenges that need to be addressed before they can be used for the purpose of clone refactoring.

2.3.2 Challenges imposed by clone detection tools

Quality of clones Figure 2.4 shows an example of poor quality clone fragments reported by NiCad in JFreechart project, where only two statements have been mapped while the other statements couldn't be mapped because their abstract syntax tree structure is different. Moreover, some of the groups reported by clone detection tools can be very large, such as 200 clone instances or more in the same group, which puts in question the quality of such a clone group. Finally, the number of statements inside the clone fragments can be problematic, as some tools might report a single statement as a clone.

Incomplete control structure Clone detection tools do not preserve the control structure and report clones with partial control structure. For instance, if the original source code contains If-Else, the clone detection tool might return only the if clause, or the else clause. So, we refer to control statements that all of their nested statements are included in the reported clone fragments as **Complete Control statements**.

Non-Refactorable clones Some clones do not have the aforementioned limitations, but they cannot be refactored, because the fragments belong to different files and the extracted code cannot be pulled up into a common superclass.



ID	Statement		ID	Statement
		→	1	PiePlot plot = new PiePlot();
		→	2	assertNotNull(plot.getBaseSectionPaint());
1	TimeSeries series = new TimeSeries("S", Year.class);	→		
2	assertNull(series.getDataItem(new Year(2006)));	→		
3	boolean pass = false;	→	3	boolean pass = false;
4	▼ try	→	4	▼ try
		→	5	plot.setBaseSectionPaint(null);
5	series.getDataItem(null);	→		
6	assertTrue(pass);	→	6	assertTrue(pass);

Figure 2.4: Example of poor quality clone fragments

2.4 Algorithms

2.4.1 String Similarity

2.4.1.1 Levenshtein Distance

This algorithm was named after Vladimir Levenshtein [Lev66]. Levenshtein Distance (LD) is one of the widely known and used algorithms to measure the similarity between two strings. The distance is measured by computing the minimum number of deletions, insertions, or substitutions required to transform one string into another. The LD distance for two strings A and B is computed as:

$$LD(A, B) = min\{a(i) + b(i) + c(i)\} \tag{2.1}$$

where $a(i)$, $b(i)$, and $c(i)$ are number of replacement, insertion, and deletion operations, respectively. To compute the similarity between strings A, and B we use equation 2.2.

$$similarity(A, B) = 1 - \frac{LD(A, B)}{max\{|A|, |B|\}} \tag{2.2}$$

2.4.2 Vector Similarity

Before we discuss vector similarity algorithms we need to mention that each vector in the computation should have the:

- Same Dimension
- Same Length
- Same Features being compared. Feature X in one vector should be compared to the same feature in the other vector.

2.4.2.1 Cosine Similarity

Cosine Similarity is one of the popular algorithms in text mining and information retrieval [Deh+11], where the similarity between two vectors A and B is computed as:

$$similarity(A, B) = \frac{\sum_{i=1}^{l}(A_i.B_i)}{(\sum_{i=1}^{l}(A_i)^2).(\sum_{i=1}^{l}(B_i)^2)} \tag{2.3}$$

where l is the length of vector, A_i and B_i correspond to the same feature i in vectors A and B. Because the formula of this algorithm is a dot product, if $A_i = B_i = 0$ the similarity will be 0 rather than 1.

2.4.2.2 Hamming Distance

This distance is used often to find the differences in two strings of bits equal in length. The earliest use for it was in 1980, when it was used to measure errors in messages sent over the network [BKR02]. The advantage of using this algorithm is that when two vectors have the same feature equal to 0, it means that the similarity for this feature is 1 rather than 0 as in Cosine similarity. For two feature vectors A and B the distance is:

$$distance(A,B) = \sum_{i=1}^{l} \begin{cases} 0 & if Ai = Bi \\ 1 & if Ai \neq Bi \end{cases} \tag{2.4}$$

where l is the dimension of the vectors. While the similarity for A and B is computed as:

$$similarity = 1 - \frac{distance(A,B)}{|vector|} \tag{2.5}$$

where $|vector|$ is the length of A and B. Equation (2.4) and (2.5) can be re-written as:

$$similarity(A,B) = \frac{\sum_{i=1}^{l} \begin{cases} 1 & if Ai = Bi \\ 0 & if Ai \neq Bi \end{cases}}{|vector|} \tag{2.6}$$

2.4.2.3 Euclidean distance

This metric is the distance or line that connects two points A and B in n-space. Euclidean distance has been used in different fields such as, clustering [PJ09].

$$distance(A,B) = \sqrt{\sum_{i=1}^{n} (Ai - Bi)^2} \tag{2.7}$$

2.4.3 Jaccard Index

In 1901 Paul Jaccard introduced Jaccard Index or what is known as Jaccard similarity coefficient [BJD13]. It is used to measure the similarity between two sets. Assuming

A and B are sets, then Jaccard similarity is computed as:

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.8}$$

2.5 Clustering

Clustering is an analysis done to data in a group with the goal to divide them into smaller groups by putting together the similar data points (Equation 2.9) [RM05]. We will discuss one of the commonly used clustering algorithms, namely **Hierarchical Clustering**.

$$S = \bigcup_{i=1}^{n} C_i \; and \; C_i \cap C_j = \emptyset \; for \; i \neq j \tag{2.9}$$

where clusters $C_1 ... C_n$ are subsets of S.

2.5.1 Hierarchical Clustering

In this type of clustering you don't need to specify or know the number of clusters beforehand, in contrast to other algorithms, such as K-means, in which the number of clusters should be given as input. However, you need to know what level of clustering is the best. There are algorithms such as the Silhouette Coefficient [Rou87] that can help to estimate what level of clustering is the best, which we will discuss later on. There are two approaches in hierarchical clustering based on the starting point as they appear in Figure 2.5, namely **Divisive** and **Agglomerative**.

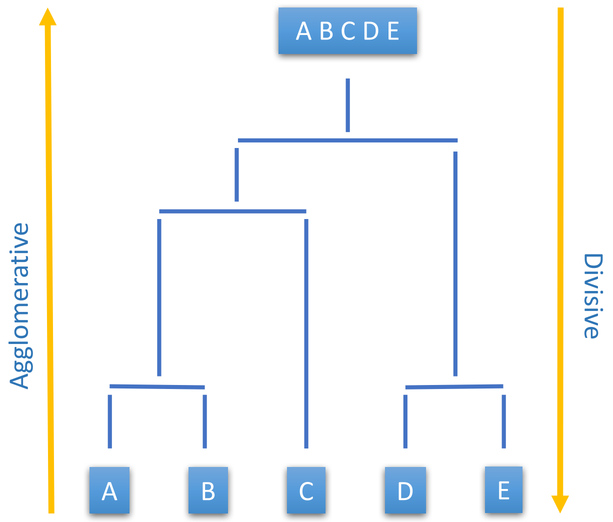


Figure 2.5: Example of Hierarchical Clustering (Dendrogram)

2.5.1.1 Approaches

Divisive (Top - Down) This approach starts by placing all data points in the same cluster **(Top)**, and then it starts splitting the clusters into smaller clusters until each data point is placed in a separate cluster **(Down)**. The partitioning function can be based on for example, the size of the cluster (split largest clusters first), and the average similarity between clusters [DH02]. Algorithm 1 presents the procedure to cluster M points in a divisive manner.

Algorithm 1: Divisive clustering algorithm

```
  Input: M points
  Output: Dendrogram
1 $clusters \leftarrow \emptyset$
2 $clusters \leftarrow clusters \cup M$ points
3 while $|clusters| \neq |M|$ do
4     split clusters using an algorithm
```

Agglomerative (Bottom - Up) This approach starts by putting each data point in its own cluster **(Bottom)**, then merge the data points one pair at a time based on their similarity (or distance) until there is a single cluster that contains all data points **(Up)**. Algorithm 2 presents the procedure to cluster M points in an agglomerative manner.

```
Algorithm 2: Agglomerative clustering algorithm
   Input: M points
   Output: Dendrogram
1  $clusters \leftarrow \emptyset$
2  $clusters \leftarrow clusters \cup M$ clusters for $M$ points
3  while $|clusters| > 1$ do
4      $merge$ nearest clusters in $clusters$
```

2.5.1.2 How to merge two clusters?

There are different algorithms to combine clusters [ZKF05], and some of them are mentioned below. Assuming p_k is a point in cluster C_i, p_r is a point in cluster C_j, and *similarity* is the function used to compute the similarity or distance between two points:

Single Linkage The similarity for (C_i, C_j) is the highest similarity between pairwise points in C_i and C_j. After applying equation 2.10 on all pairs of clusters, the clusters with the maximum similarity are merged.

$$sim_{single}(C_i, C_j) = max_{p_k \in C_i, p_r \in C_j}(similarity(p_k, p_r)) \tag{2.10}$$

Complete Linkage This approach is the opposite of Single-Linkage, as the similarity between two clusters is the lowest similarity (highest distance) between their pairwise points. After applying equation 2.11 on all pairs of clusters, the clusters with minimum similarity (maximum distance) are merged together.

$$sim_{complete}(C_i, C_j) = min_{p_k \in C_i, p_r \in C_j}(similarity(p_k, p_r)) \tag{2.11}$$

Average Linkage This clustering is referred to as UPGMA scheme [JD88]. The similarity between two clusters is computed by finding the average similarity between all points in the two clusters.

$$sim_{average}(C_i, C_j) = (\frac{1}{|C_i||C_j|}) \sum_{p_k \in C_i, p_r \in C_j} similarity(p_k, p_r) \tag{2.12}$$

MinMax Linkage This algorithm was proposed by [Din+01]. The goal is to merge clusters that are less self-similar.

$$sim_{MinMax}(C_p, C_q) = \frac{sim(C_p, C_q)}{sim(C_p, C_p)sim(C_q, C_q)} \tag{2.13}$$

2.5.2 Silhouette Coefficient

Silhouette Coefficient was introduced by Rousseeuw [Rou87]. It is used to measure and estimate the consistency and quality of clusters.

$$SilhouetteScore = \frac{\sum_{i=1}^{n} \sum_{j=1}^{|Ci|} s(j)}{|m|} \tag{2.14}$$

$$s(j) = \frac{bX[j] - aX[j]}{max(bX[j], aX[j])}, (-1 \leq s(j) \leq 1) \tag{2.15}$$

where

n is the number of clusters

Ci is the cluster at iteration i, and $|Ci|$ the cardinality of it

bX is the average dissimilarity between point j in cluster C_i and points outside C_i

aX is the average dissimilarity between point j in cluster C_i and its points

m is the total number of points in all clusters, and $|m|$ the cardinality of it

The closer *s(j)* to 1 the less the dissimilarity within the same cluster and greater to the other clusters ($aX < bX$), so we can say j is **well-clustered**. However, when j is **misclassified** the dissimilarity within the same cluster is greater than the other clusters, and *s(j)* will be close to -1 ($aX > bX$). In the situation where $aX \approx bX$, the value of $s(j)$ will be close to 0, which implies it is not clear if j is placed in the appropriate cluster or not.

2.6 Program Dependence Graph

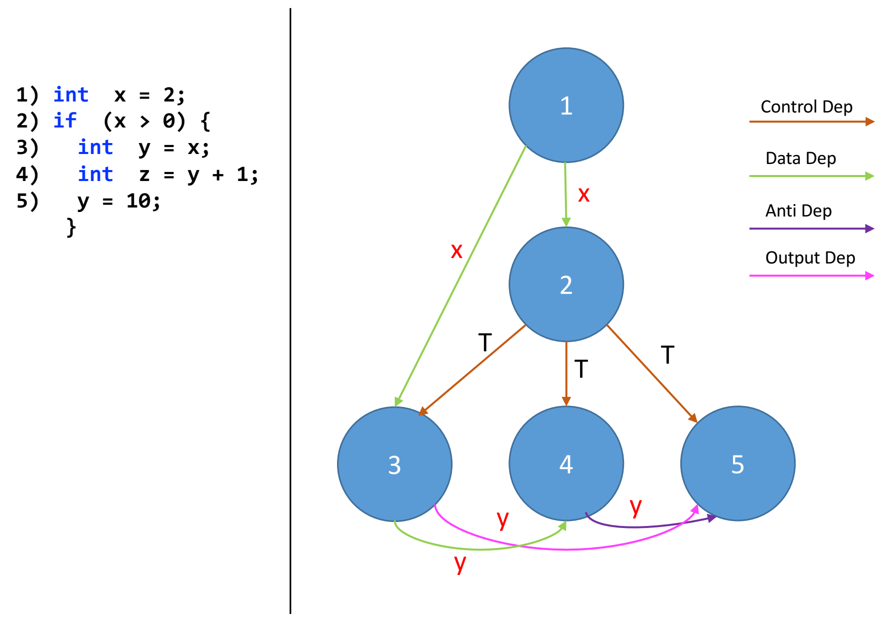
The Program Dependence Graph (PDG) represents the relation between program elements, where elements are statements or predicates. The edges represent relations that connect these elements, which can be either data or control dependencies [FOW84]. Data dependencies are further categorized into three types [WB87]. Assuming S_1 and S_2 are statements, then:

1. **Data-Flow Dependency** $S_1 \xrightarrow[d]{R} S_2$: Assuming that S_2 uses a result R from executing S_1, then we say that S_2 is data dependent on S_1.

2. **Output Dependency** $S_1 \xrightarrow[o]{R} S_2$: Assuming R is the result from executing S_1. If S_2 modifies R then we say S_2 is output dependent on S_1.

3. **Anti Dependency** $S_1 \xrightarrow[a]{R} S_2$: Assuming R is a variable used in S_1 and modified by S_2 then we say that S_2 is anti-dependent on S_1.

4. **Control Dependency** $S_1 \xrightarrow[c]{T or F} S_2$: Assuming S_1 is a control statement (If, For,...). If the execution of S_2 depends on the result of S_1 then we say that S_2 is control dependent on S_1. A control dependency is labeled as either *True* or *False*. For example, the control dependencies to the statements inside the else clause of an If/Else statement are labeled as *False*. All other control dependencies are labeled as *True*.



```
1) int  x = 2;
2) if  (x > 0) {
3)   int  y = x;
4)   int  z = y + 1;
5)   y = 10;
    }
```

Figure 2.6: Program Dependency Graph (PDG)

In the figure above, x and y are the variables affected by each dependency, also T stands for *True* control dependency.

2.7 Abstract Syntax Tree

2.7.1 AST representation

The Abstract Syntax Tree (AST) represents the syntactic structure of the source code inside a file, which is derived from the parse tree or more often referred to as Concrete Syntax Tree (CST). CST contains all information about the source code, including comments, white-spaces, and line-breaks. The AST representation is an abstraction of the CST that keeps only the information necessary to the compiler. Figure 2.7 shows the AST representation for the code below. The content of Figure 2.7 is generated using the Eclipse plug-in AST-View [Foua].

```
public class simpleAST{
 private int x = 0;
 public int getX(){
  return x;
 }
}
```

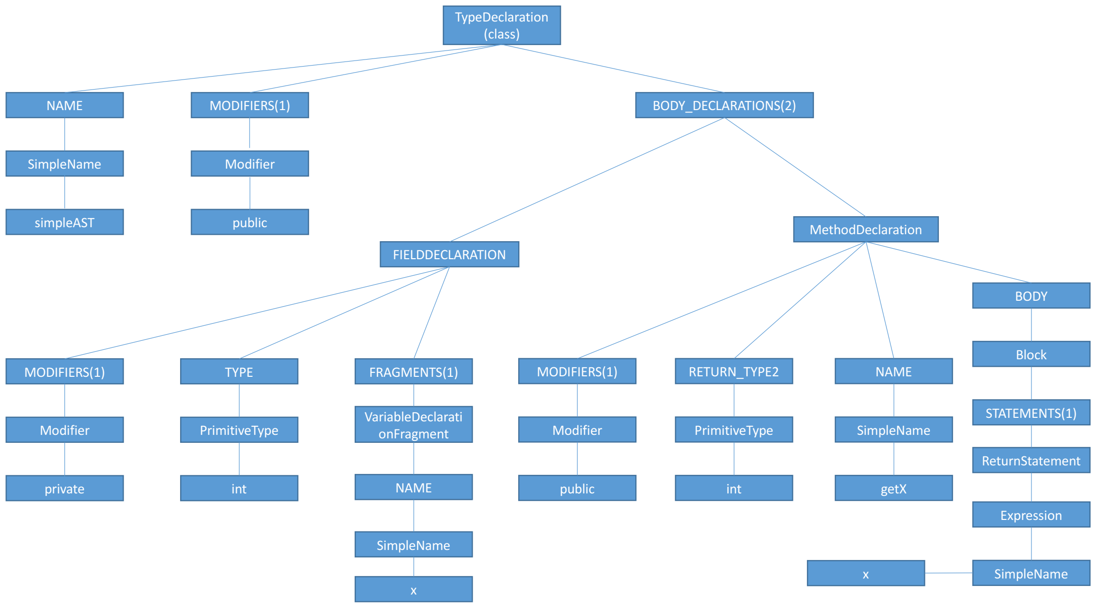


Figure 2.7: Example of an AST

2.7.2 AST Visitor

The ASTVisitor is an abstract class found in *org.eclipse.jdt.core.dom* [Foub] library which is part of the Eclipse IDE. It provides a mechanism to explore or visit every statement and expression in an AST, and to perform an operation on specific types of AST nodes. A class is required to extend ASTVisitor and override the methods `visit`, and `endVisit` to implement custom functionalities.

2.8 Clone Refactoring Preconditions

The goal of this section is to introduce the conditions that duplicate code at pair or group level must comply with to be assessed as refactorable. These conditions are proposed by Tsantalis et al. [TMK15] and we will use them in evaluating the refactorability of clone groups. There are eight preconditions in total, which we will discuss next.

Precondition 1

The parameterization of the differences between the mapped statements should not break any existing control, data, anti, and output dependencies.

In the example shown in Figure 2.8, the behavior of code has changed after applying refactoring because it violated this precondition. In the original code (Before refactoring) in class B, *a.getX()* is assigned to variable x in methods **m1** and **m2**, then *a.foo()* and *a.bar()* are called. Both *a.foo()* and *a.bar()* change the value of field x that exists in class **A**. To refactor **m1** and **m2** the common code needs to be extracted to a new method and parameterize the differences in the statements. The differences are the calls for *a.foo()* and *a.bar()* in **m1** and **m2**, respectively.

However, passing these method calls as arguments will first update field x of class **A**, and then assign it to variable x in the extracted method, thus changing the final result that is printed and the behavior of the program.

Before Refactoring

```
public class A {
   private int x;
   public int getX () { return x; }
   public int foo () { x++; return x; }
   public int bar () { x+=5; return x;}
}
public class B {
   public void test () {
      A a = new A();
      m1(a);
      m2(a);
   }
   public void m1(A a) {
      int x = a. getX ();
      int y = a. foo ();
      System.out.print (x);
   }
   public void m2(A a) {
      int x = a. getX ();
      int y = a.bar ();
      System.out.print (x);
   }
}
```

After Refactoring

```
public class A {
   private int x;
   public int getX () { return x; }
   public int foo () { x++; return x; }
   public int bar () { x+=5; return x; }
}
public class B {
   public void test () {
      A a = new A();
      m1(a);
      m2(a);
   }
   public void m1(A a) {
      ext (a, a. foo ());
   }
   public void m2(A a) {
      ext (a, a. bar ());
   }
   private void ext (A a, int arg ) {
      int x = a. getX ();
      int y = arg ;
      System.out.print (x);
   }
```

Figure 2.8: Change in execution behavior (Figure 5. in Tsantalis et al. [TMK15])

Precondition 2

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.

Precondition 3

The parameterization of fields belonging to differences between the mapped statements is possible only if they are not modified.

In this precondition, fields that are part of the differences are only parameterizable if they are not modified in the extracted code. In other words if the field's value is changed inside the extracted code then this difference cannot be parameterized, because parameters are passed by value in Java, and thus the extracted code would not update the values of the fields after the refactoring.

Precondition 4

The parameterization of method calls belonging to differences between the mapped statements is possible only if they do not return a void type.

Precondition 5

The unmapped statements should be movable before or after the mapped statements without breaking existing control, data, anti, and output dependencies.

Precondition 6

The mapped statements within the clone fragments should return at most one variable of the same type to the original methods from which they are extracted.

This condition implies that after extracting the mapped statements into a new method, the method should return at most one variable of the same type, since in our case the programming language is Java and it does not support passing of parameters by reference. On the contrary, languages supporting parameter passing by reference, such as C and C++, can return multiple variables.

Precondition 7

The mapped statements within the clone fragments should not contain any conditional return statements.

The following example shows a conditional return statement. These statements are used to exit directly the method they exist in. However, if we extract statements (1) and (2) to a new method, they will exit the execution of the extracted method, while the original method *foo()* will continue to execute normally and will not exit. This problem can be solved by returning boolean flag from the extracted method; however, this solution will require to add new statements to the original code *foo()* and to the extracted method.

```
public void  foo(){
   int  x = 0;
   . . .
    (1) if (x > 0)
       (2) return ;
   System.out.println(x);
   . . .
}
```

Precondition 8

The mapped branching statements (break, continue) should be accompanied with the corresponding mapped loop statements.

This precondition implies that if the common statements include a break or continue statement, then the corresponding loop (For, While, Do/While, Enhanced For) statement should be extracted as well.

Chapter 3

Literature Review

3.1 Statement Mapping

Lin et al. [Lin+14] work focused on how to find differences among multiple clones in the same group using *Progressive alignment*, an algorithm used in genetics to align DNA and proteins [HS88]. They developed an Eclipse plugin called **MCIDiff** (Multi-Clone-Instances Differencing) that takes N clones as an input and returns the token differences among them. Their approach starts by parsing each clone into a sequence of tokens and each token is categorized into either Type, Method/Field/-Variable/Literal, Label, Keyword, or Separator (e.g., (,{), and Operator. The six token classifications are divided into three groups based on the attributes attached to them: 1) Name: for Type, Method/Field/Variable/Literal and Label tokens; 2) Symbol: for Keyword, Separator and Operator tokens; lastly, 3) Data type: for Method/Field/Variable/Literal tokens.

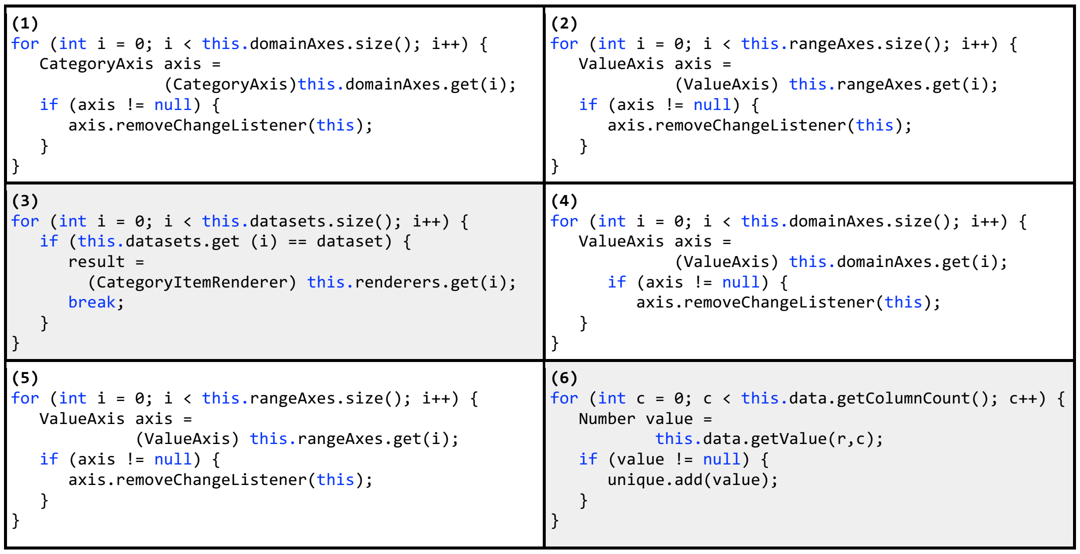
After parsing is complete, in the first step, MCIDiff computes the Longest Common Subsequence (LCS, longest sequence of tokens common between all clones) using *Progressive alignment* for all clone instances starting from the two longest sequences, and then proceeding with the next token sequence and so on. The resulting tokens in the LCS have the same category and attribute. However, there might be tokens that are not common in all sequences, thus they are returned as differential ranges representing the start and end indices for a range of tokens that were not common in all sequences.

Differential ranges are examined again to find common tokens within non-gapped tokens. First, they try to match identical tokens by re-running MCIDiff on non-gapped token sequences within the differential ranges and try to match them with the same criteria as before (same category and attribute). Then, for non-identical unmatched tokens a similarity heuristic is used (i.e., tokens should have the same category, while for attributes a similarity value is calculated). The computed similarity for Keyword/Separator/Operator/Label is equal to 1.0 if they are the same, otherwise it is 0.0; and for Type/Method/Field/Variable/Literal, the Jaccard coefficient [BJD13] is used to find if tokens have a super type in common.

To evaluate the accuracy of MCIDiff, they used 831 clone sets reported by CloneDetective (i.e., a token-based clone detection tool) [Jue+09] in three projects (JavaNewIO, JHotDraw, JFreechart). They filtered the clone sets to keep only those that contain gaps and parameterization (i.e., differences), which resulted in only 638 (77%) clone sets, that consist of 353 sets containing two clones, 235 sets containing three-to-five clones, and 50 sets containing more than 5 clones. The (precision, recall) of MCIDiff for the three projects is respectively (100%, 100%), (98.66%, 95.63%), and (98.82%, 98.39%). While they have a good precision and recall, their work has several limitations that are listed below.

- Refactoring is not supported.
- They detect differences through Progressive alignment, but having a unique sequence might lead to incorrect alignment. For example, in Figure 3.1 we can

see that clones (1), (2), (4), (5) look similar in terms of control statements and the behavior or the operation they execute (i.e., removing an object from a list). However, clones (3) and (6) have different behaviors (unique clones) (i.e., clone (3) is searching for an object, and clone (6) is adding elements to a list).



```
(1)
for (int i = 0; i < this.domainAxes.size(); i++) {
   CategoryAxis axis =
                (CategoryAxis)this.domainAxes.get(i);
   if (axis != null) {
      axis.removeChangeListener(this);
   }
}
```

```
(2)
for (int i = 0; i < this.rangeAxes.size(); i++) {
   ValueAxis axis =
             (ValueAxis) this.rangeAxes.get(i);
   if (axis != null) {
      axis.removeChangeListener(this);
   }
}
```

```
(3)
for (int i = 0; i < this.datasets.size(); i++) {
   if (this.datasets.get (i) == dataset) {
      result =
        (CategoryItemRenderer) this.renderers.get(i);
      break;
   }
}
```

```
(4)
for (int i = 0; i < this.domainAxes.size(); i++) {
   ValueAxis axis =
             (ValueAxis) this.domainAxes.get(i);
      if (axis != null) {
         axis.removeChangeListener(this);
   }
}
```

```
(5)
for (int i = 0; i < this.rangeAxes.size(); i++) {
   ValueAxis axis =
             (ValueAxis) this.rangeAxes.get(i);
   if (axis != null) {
      axis.removeChangeListener(this);
   }
}
```

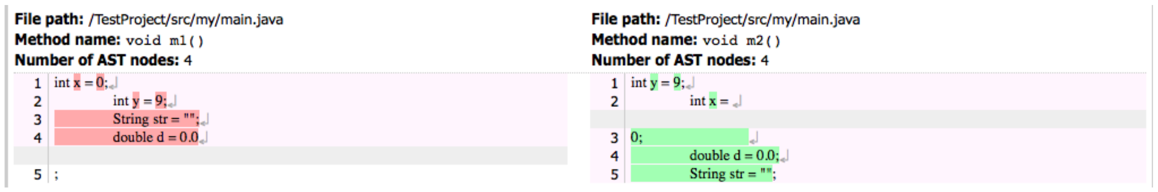
```
(6)
for (int c = 0; c < this.data.getColumnCount(); c++) {
   Number value =
           this.data.getValue(r,c);
   if (value != null) {
      unique.add(value);
   }
}
```

Figure 3.1: Clone group containing unique clones

- Finding similar tokens is almost an exhaustive search. As a result, the computation time increases dramatically as the size of the clone group, the length of token sequences, and the number of differences among the clones increases.

- Examining the token sequences (i.e., clones) in a different order during the alignment process, might produce different results.

- Reordered statements are considered as gaps, because their approach treats the entire clone fragments as sequences of tokens. The example in Figure 3.2 demonstrates the limitation of MCIDiff. As it can be observed from the output of MCIDiff shown in Figure 3.2b the re-ordered statements `String str = "";` and `double d = 0.0;` cannot be properly matched:


File path: /TestProject/src/my/main.java
Method name: `void m1()`
Number of AST nodes: 4

```
int x = 0;
          int y = 9;
          String str = "";
          double d = 0.0

;
```

File path: /TestProject/src/my/main.java
Method name: `void m2()`
Number of AST nodes: 4

```
int y = 9;
          int x =

0;
          double d = 0.0;
          String str = "";
```

(a) Original Code

[int, e*], [y, e*], [=, e*], [9, e*], [;, e*], [int, double], [y, d], [9, 0.0],

[double, e*], [d, e*], [=, e*], [0.0, e*]

(b) Output generated by MCIDiff

Figure 3.2: Example demonstrating the limitation of MCIDiff to match reordered statements.

The results from MCIDiff 3.2b are interpreted as followa:

– Statements `int y = 9;` and `double d = 0.0;` in the first and second clone fragments, respectively are considered as gaps.

– The pairs of tokens (`int`, `double`), (`y`,`d`), (`9`,`0.0`) are considered as differences between the two fragments.

A recent work by Tsantalis et al. [TMK15] assesses the refactorability of clone pairs by examining if the differences between the clones can be parameterized without causing any problems. Their approach consists of three major steps. The first step finds a common nesting structure (isomorphic trees) shared by the pair of clones. The second step maps the statements of the clones in a way that maximizes the number of mapped statements, while minimizes the number of differences between them. The last step examines a list of preconditions to check if the differences found between the clones can be safely parameterized, which we discussed in section 2.8.

In the first step, they look for maximal isomorphic sub-trees in the nesting structure trees (NST) of the clone fragments, and if there are more than one non-overlapping sub-trees, then each one is treated as a separate refactoring opportunity. They follow a combination of Bottom-Up and Top-Down matching approaches when searching for isomorphic sub-trees. They start from the leaf nodes and for each pair of matched leafs (Bottom-Up), they try to find a matching sibling. For each pair of matching siblings, they perform Top-Down matching to check if the resulting common sub-tree is complete (i.e., all child nodes in the sub-trees have been matched with one node from the other sub-tree). If a leaf in the first tree has multiple matches in the second tree, then only the first matching leaf with minimum differences is explored.

After finding the isomorphic trees (common structure) for a pair of fragments, the process of statement mapping follows. Each statement in the first fragment is mapped with a single statement at most in the second fragment. Figure 3.3 shows an example that the isomorphic trees resulting from the first step, do not lead to the best solution, as the number of differences in Figure 3.3a are 24, while if we switch the If/Else-If as in Figure 3.3b the number of differences decreases to only 2. This is solved by comparing the body of **If** in the first fragment with the bodies of **If**, and **Else-If** in the second fragment, and testing whether the resulting mapping maximizes the number of mapped statements and at the same time minimizes the number of differences between the mapped statements. The second step of their approach is essentially a greedy algorithm that makes locally optimal choices at each level of the NST sub-trees with the hope of finding a globally optimal solution.

```
60  if (im.getOutlinePaint() != null &&
        im.getOutlineStroke() != null) {
61    if (orientation == VERTICAL) {
62      Line2D line = new Line2D.Double();
63      double y0 = dataArea.getMinY();
64      double y1 = dataArea.getMaxY();
65      g2.setPaint(im.getOutlinePaint());
66      g2.setStroke(im.getOutlineStroke());
67      if (range.contains(start)) {
68        line.setLine(start2d, y0, start2d, y1);
69        g2.draw(line);
      }
70      if (range.contains(end)) {
71        line.setLine(end2d, y0, end2d, y1);
72        g2.draw(line);
      }
    }
73    else if (orientation == HORIZONTAL) {
74      Line2D line = new Line2D.Double();
75      double x0 = dataArea.getMinX();
76      double x1 = dataArea.getMaxX();
77      g2.setPaint(im.getOutlinePaint());
78      g2.setStroke(im.getOutlineStroke());
79      if (range.contains(start)) {
80        line.setLine(x0, start2d, x1, start2d);
81        g2.draw(line);
      }
82      if (range.contains(end)) {
83        line.setLine(x0, end2d, x1, end2d);
84        g2.draw(line);
      }
    }
  }
```

```
61  if (im.getOutlinePaint() != null &&
        im.getOutlineStroke() != null) {
62    if (orientation == VERTICAL) {
63      Line2D line = new Line2D.Double();
64      double x0 = dataArea.getMinX();
65      double x1 = dataArea.getMaxX();
66      g2.setPaint(im.getOutlinePaint());
67      g2.setStroke(im.getOutlineStroke());
68      if (range.contains(start)) {
69        line.setLine(x0, start2d, x1, start2d);
70        g2.draw(line);
      }
71      if (range.contains(end)) {
72        line.setLine(x0, end2d, x1, end2d);
73        g2.draw(line);
      }
    }
74    else if(orientation == HORIZONTAL) {
75      Line2D line = new Line2D.Double();
76      double y0 = dataArea.getMinY();
77      double y1 = dataArea.getMaxY();
78      g2.setPaint(im.getOutlinePaint());
79      g2.setStroke(im.getOutlineStroke());
80      if (range.contains(start)) {
81        line.setLine(start2d, y0, start2d, y1);
82        g2.draw(line);
      }
83      if (range.contains(end)) {
84        line.setLine(end2d, y0, end2d, y1);
85        g2.draw(line);
      }
    }
  }
```

(a) Non-Optimal Mapping

```
60  if (im.getOutlinePaint() != null &&
        im.getOutlineStroke() != null) {
61    if (orientation == VERTICAL) {
62      Line2D line = new Line2D.Double();
63      double y0 = dataArea.getMinY();
64      double y1 = dataArea.getMaxY();
65      g2.setPaint(im.getOutlinePaint());
66      g2.setStroke(im.getOutlineStroke());
67      if (range.contains(start)) {
68        line.setLine(start2d, y0, start2d, y1);
69        g2.draw(line);
      }
70      if (range.contains(end)) {
71        line.setLine(end2d, y0, end2d, y1);
72        g2.draw(line);
      }
    }
73    else if (orientation == HORIZONTAL) {
74      Line2D line = new Line2D.Double();
75      double x0 = dataArea.getMinX();
76      double x1 = dataArea.getMaxX();
77      g2.setPaint(im.getOutlinePaint());
78      g2.setStroke(im.getOutlineStroke());
79      if (range.contains(start)) {
80        line.setLine(x0, start2d, x1, start2d);
81        g2.draw(line);
      }
82      if (range.contains(end)) {
83        line.setLine(x0, end2d, x1, end2d);
84        g2.draw(line);
      }
    }
  }
```

```
61  if (im.getOutlinePaint() != null &&
        im.getOutlineStroke() != null) {
74    if(orientation == HORIZONTAL) {
75      Line2D line = new Line2D.Double();
76      double y0 = dataArea.getMinY();
77      double y1 = dataArea.getMaxY();
78      g2.setPaint(im.getOutlinePaint());
79      g2.setStroke(im.getOutlineStroke());
80      if (range.contains(start)) {
81        line.setLine(start2d, y0, start2d, y1);
82        g2.draw(line);
      }
83      if (range.contains(end)) {
84        line.setLine(end2d, y0, end2d, y1);
85        g2.draw(line);
      }
    }
62    else if (orientation == VERTICAL) {
63      Line2D line = new Line2D.Double();
64      double x0 = dataArea.getMinX();
65      double x1 = dataArea.getMaxX();
66      g2.setPaint(im.getOutlinePaint());
67      g2.setStroke(im.getOutlineStroke());
68      if (range.contains(start)) {
69        line.setLine(x0, start2d, x1, start2d);
70        g2.draw(line);
      }
71      if (range.contains(end)) {
72        line.setLine(x0, end2d, x1, end2d);
73        g2.draw(line);
      }
    }
  }
```

(b) Optimal Mapping

Figure 3.3: Example showing a non-optimal and an optimal mapping for two clone fragments taken from Tsantalis et al. [TMK15]

The refactorability assessment takes place after the statement mapping step is complete. A list of preconditions is examined to check if any of the eight defined preconditions for clone refactoring (Section 2.8) is violated. In summary, the preconditions check if the mapped statements can be extracted into a method, if the differences found between the clones can be parameterized, and if the unmapped statements can be safely moved before or after the extracted code. This work is considered as the state-of-art in clone refactoring, because it introduces *refactorability analysis*, which is an important feature in clone management to assess if a pair of clones can be safely refactored by preserving the program behavior.

3.2 Clone Refactoring

One of the earliest works on duplicate code refactoring was done by Tairas et al. [TG12], who developed an Eclipse Plug-in named **CeDAR** (Clone Detection, Analysis, and Refactoring). Their contribution falls in the three steps of removing duplicate code. The first step, **Clone Detection** is done by clone detection tools such as CCFinder [KKI02], and NICAD [RC08a]. However, the results from these tools require pre-processing to get correct start-end line numbers for the clone fragments before they can be processed. For example, CCFinder reports clones as ranges of tokens, whereas other tools might report incomplete statements. Therefore, their first contribution is parsing and integrating the output from clone detection tools into the refactoring process.

The output from **Clone Detection** is used as an input for the **Clone Analysis** step. Previous techniques help in finding opportunities for refactoring by examining some properties automatically, such as if the clones belong to classes within the same inheritance hierarchy [Kon01]. The limitation of these techniques are: 1) They will find refactorable and non-refactorable clones, and 2) The decision is left to developers or maintainers to apply the refactoring manually. To overcome the first limitation in finding only refactorable clone groups, CeDAR employs the Eclipse IDE refactoring engine to check if the group of clones meets a set of preconditions for the Extract Method refactoring, i.e., extracting the clone fragments into a common method.

The last step is **Refactoring**, which is also done by the Eclipse IDE refactoring engine. However, the Eclipse refactoring engine was not able to parameterize differences within the clone fragments, with the exception of differences in variable identifiers. CeDAR overcomes this limitation by extending the Eclipse refactoring engine to add support for method call and field access parameterization.

The evaluation of CeDAR was performed by comparing it to the Extract Method refactoring provided by Eclipse. They did an experiment on nine projects (including Apache Ant, JFreeChart, JEdit, and JRuby), and they used the clones reported by Deckard [Jia+07]. However, some of the groups were removed using the clone analysis results from ARIES [HKI08], and SUPREMO [Kon01], as they could not be refactored through extract method. ARIES and SUPREMO are tools used to evaluate and propose how clones can be refactored. The total number of groups they examined in their experiment is 1206. Eclipse was able to refactor 128 (10.6%), and CeDAR 226 (18.7%) of the total groups. The main limitation of their work is that it supports the refactoring of Type-I and simple Type-II clones only. Furthermore, the refactoring support is limited to clones located within the same Java file.

Meng et al. [Men+15] developed a fully automated refactoring tool, called *RASE* that uses the abstract edit script generated by *LASE* [MKM13]. The edit script is computed by comparing different versions of the same method and returning the differences as AST node insert, delete, update and move operations. Figure 3.4 shows an example of edit scripts, where the lines in blue are added statements, in red are deleted statements, and in black are unchanged statements across versions. *RASE* finds next the common changes in all methods (clones) and creates a generalized program transformation, called *abstract edit script*. RASE requires the clones to be adjacent, so that it can generate a single AST node (and all its child sub-trees), or multiple sub-trees under same parent after the **Merge** step is performed.

```
public class CompareEditorInput {
  private ICompareContainer fContainer;
  private boolean fContainerProvided;
  private Splitter fComposite;
  public IActionBars getActionBars (int offset) {
    if (offset == -1)
      return null;
-   if (fContainer == null) {
+   IActionBars actionBars = fContainer.getActionBars();
+   if (actionBars==null&&offset!=0&&!fContainerProvided){
      return Utilities.findActionBars(fComposite, offset);
    }
-   return fContainer.getActionBars();
+   return actionBars;
  }
  public ISLocator getServiceLocator (int offset2) {
-   if (fContainer == null) {
+   ISLocator sLocator = fContainer.getServiceLocator();
+   if(sLocator == null&&offset2!=0&&!fContainerProvided){
      return Utilities.findSite(fComposite, offset2);
    }
-   return fContainer.getServiceLocator();
+   return sLocator;
  }
}
```

Figure 3.4: Example of systematic edit (Figure 1, Page 2 Meng et al. [Men+15])

Their approach consists of three steps: In the **Merge** step, the code in the first version is considered as the root, while each edit is a tree. It works on two trees at a time, and starts with the ones that have the longest path from root, then it merges them based on their *lowest common ancestor*. For instance, if we have two edits in the same code, Edit1 (Root > Nest1 > Nest2 > tree1), Edit2 (Root > Nest1 > Nest3 > Nest4 > tree2), it merges them by selecting Nest1 as the *lowest common ancestor* and adds Nest2 > tree1, and Nest3 > Nest4 > tree2 under Nest1.

In the second step called **Abstract**, RASE abstracts the differences by using wildcards T$, m$, v$, and u$ for differences in type names, method calls, variables names, and expressions, respectively. The goal is to create an abstract template that will be used in the next step. In the last step called **Expand**, RASE tries to extract the maximum amount of common code by including parent and siblings where similar changes were detected. It keeps applying **Abstract** and **Expand** until no common code can be found. After these three steps are done (**Phase I**), **Phase II** starts by trying to apply one or more of the six refactoring operations: extract method, add parameter, parameterize type, form template method, introduce return object, and introduce exit label.

To test RASE they selected 56 method pairs and 30 groups (3-9 methods in each group) that similarly changed. They tried four different scenarios of refactoring in terms of the amount of code to be extracted: (1) refactor as much code as possible (default), (2) smallest amount of code, (3) entire method before edit, and (4) entire method after edit. Out of the 56 method pairs, RASE was able to refactor 30 case for scenarios (1) and (2), while for (3) and (4) only 19 cases. The same scenarios were applied to the groups and 20 were refactorable for scenarios (1) and (2), while for the last two scenarios only 9 cases were refactorable.

The main limitation with RASE comes from its dependence on LASE for the edit script, since LASE does not support clones that have not been updated throughout the history, or have been very recently introduced, and thus have no history of changes. Lastly, RASE cannot handle clones with differences in their control structure, as it requires the presence of a single AST node (and all its child sub-trees), or a set of contiguous sub-trees under the same parent node.

3.3 Program Dependence Graph Mapping

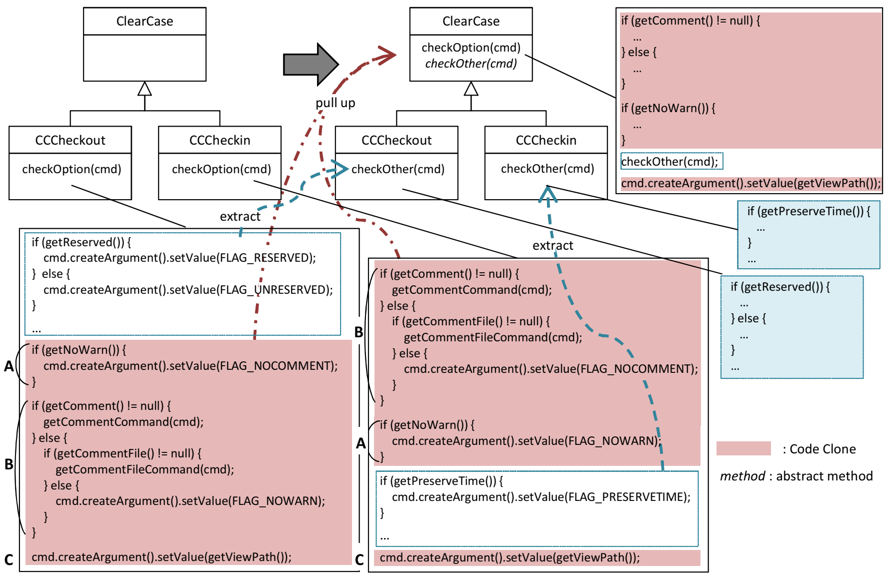
The Program Dependence Graph (PDG, section 2.6), has been used by many works related to clones in order to find duplicate code, and find opportunities for refactoring.

Hotta et al. [HHK12] improved Juillerat et al. [JH07] work by employing PDG to detect reordered statements, and developed a tool called CREIOS (Clone Removal Expediter by Identifying Opportunities with Scorpio). The limitation of Juillerat et al. [JH07] is that they do not handle properly reordered statements.

Hotta et al. approach is divided into three steps: The first step is **Identifying clones**. After the PDG is created, the Scorpio tool [HK11] is used to detect clones allowing for differences in identifiers. In the next step, only the clone pairs that satisfy two criteria are included: 1) Form Template Method refactoring [Bec+99] can be applied, and 2) They have at least a single subgraph in common. In the next step called **Tailoring**, the statements for the clones are separated into common and unique processes. *Common* process are the statements that are shared between the clones, and can be pulled up to the base class. *Unique* process are the statements that will remain in the subclasses, because they are different.

The example in **Figure 3.5** shows an application of the Form Template Method refactoring. The method along with the common processes are pulled up to base class, where the common processes (A, B, C) appear in red boxes and the method to be pulled up is *checkOption(cmd)*. The unique process appears in a white box, and it is extracted into a new method *checkOther(cmd)*. At the same time an abstract method *checkOther(cmd)* is created in the base class and a call to the abstract method is added to *checkOption(cmd)*.

They experimented *Form Template Method* refactoring on 45 cases detected with CREIOS in Apache-Synapse (Synapse is a system for service discovery) and 226 cases in Apache-Ant, but a manual investigation was done only to the cases reported in Apache-Synapse. Out of the 45 cases, 16 of them required some modifications for CREIOS to apply *Form Template Method* refactoring such as changing the visibility of a method. They compared the results from CREIOS and Juillerat et al. for the same candidates in both projects and CREIOS was able to refactor 14 and 3 more cases in Apache-Ant, and Synapse respectively. The limitation of this work exists in 1) the requirements for identifying refactorable candidates, as duplicated methods have to be in different classes, and these classes have to extend the same base class, 2) It works for clone pairs only.



```
if (getReserved()) {
   cmd.createArgument().setValue(FLAG_RESERVED);
}  else {
   cmd.createArgument().setValue(FLAG_UNRESERVED);
}
...
if (getNoWarn()) {
   cmd.createArgument().setValue(FLAG_NOCOMMENT);
}
if (getComment() != null) {
   getCommentCommand(cmd);
} else {
   if (getCommentFile() != null) {
      getCommentFileCommand(cmd);
   } else {
      cmd.createArgument().setValue(FLAG_NOWARN);
   }
}
cmd.createArgument().setValue(getViewPath());
```

```
if (getComment() != null) {
   getCommentCommand(cmd);
} else {
   if (getCommentFile() != null) {
      getCommentFileCommand(cmd);
   } else {
      cmd.createArgument().setValue(FLAG_NOCOMMENT);
   }
}
if (getNoWarn()) {
   cmd.createArgument().setValue(FLAG_NOWARN);
}
if (getPreserveTime()) {
   cmd.createArgument().setValue(FLAG_PRESERVETIME);
}
...
cmd.createArgument().setValue(getViewPath());
```

```
if (getComment() != null) {
   ...
} else {
   ...
}
if (getNoWarn()) {
   ...
}
checkOther(cmd);
cmd.createArgument().setValue(getViewPath());
```

```
if (getPreserveTime()) {
   ...
}
...
```

```
if (getReserved()) {
   ...
} else {
   ...
}
...
```

Figure 3.5: An Example of Form Template Method refactoring taken from Hotta et al. [HHK12]

Bian et al. [Bia+13] is another work that employs PDG in their tool SPAPE to refactor near-miss clones (Type-II and Type-III, section 2.2) in C projects. However, their work focuses only on Type-III clones. Their approach starts by transforming the PDG of the clone fragments in a way that ensures the preservation of structural semantics, and identifying the different statements between the pair of fragments. Then it tries to merge the two clones by introducing control statements and predicates to the AST. Lastly, it replaces the duplicated code fragments with a procedure call.

The transformation of PDG requires to apply a set of rules. **Rule 1** states that to replicate a predicate statement for X, and Y statements, there should be no relation between the definition set of X and the reference set of Y (i.e., no data dependencies). **Rule 2** states that to split a single loop into N loops, there should be no dependencies among the statements inside the loop (same as **Rule 1**). They compared their work with Komondoor et al. [KH03] on 10 projects. The total number of Type-III groups detected by CPBugdetector [Li+06] in the selected projects is 390 and only 283 (72.6%) groups are extractable. The approach by Komondoor and Horwitz was able to detect 220 ($\approx$ 78%), while Bian el al. (SPAPE) was able to detect 252 ($\approx$ 89%) of the extractable groups. A limitation of this work is that they only considered groups containing two clone instances. Moreover, the introduction of new predicates increases the complexity of the source code and might affect code readability.

Program slicing and PDG were used by Komondoor et al. [KH01] to detect clones, which gave them the advantage of detecting reordered and intertwined clones (i.e., clones within the same function as shown in Figure 3.6, where xx and ++ represent the two clones). Their work is divided into three steps, and it starts by finding pairs of clones as the **first step**. Each node X in PDG A is matched with N nodes that have the same syntactic structure in PDG B. For each pair $(X, m) : m \in N$ they try to find an isomorphic subgraph containing them. Finding isomorphic subgraphs starts from the pair of nodes (X, m), and then it uses backward slicing to add matching control predecessors. Whenever it finds a matched control predecessor (p_1, p_2) for a pair matched nodes in the slices, it performs one step of forward slicing to add the successor of (p_1, p_2) to the slices.

```
++ tmpa = UCHAR(*a),
xx tmpb = UCHAR(*b);
++ while (blanks[tmpa])
++   tmpa = UCHAR(*++a);
xx while (blanks[tmpb])
xx   tmpb = UCHAR(*++b);
++ if (tmpa == '-') {
     tmpa = UCHAR(*++a);
     ...
   }
xx else if (tmpb == '-') {
     if (...UCHAR(*++b)...) ...
```

Figure 3.6: An Example of Intertwined Clones taken from Komondoor et al. [KH01]

The **second step** is to filter the subtrees from the first step by removing subtrees that are contained in another subtrees. The **last step** of their approach, is to group clone pairs in a kind of transitive closure (1-2, 1-3, 2-3,. . . , etc). They used CodeSurfer [Gra] to process the code and build the PDGs. They conducted a study by running their tool on three *Unix utilities* (busin, sort, and tail), and four files from the *Graph-Layout* (an in-house program used at IBM). For the Unix utilities they conducted two studies. The first study was on a single file where they did manual investigation and found 4 groups (ideal clone groups), then they ran the tool and found 43 clone groups which contained the four ideal groups. The second study was done on 25 clone groups reported by the tool, and each of the clones contained 30-49 nodes (statements) as an average clone size. Then they performed a manual verification and found that all of the 25 groups are variants of 9 manually identified groups. The second experiment was on the *Graph-Layout* application with the objective to collect quantitative data that can be seen in Figure 3.7. Out of the 30 clone groups in Figure 3.7, 2 groups involved reordered matched statements, 2 groups contained intertwined clones, and 17 groups contained non-contiguous clones, while most of the clones had renamed variables.

	# of lines of source	**# of PDG nodes**	**running time (elapsed)**	**# of clone groups extracted**	**total # of clones extracted**	**file size reduction**	**av. fn size reduction**
file 1	1677	2235	1:02 min	3	6	1.9%	5.0%
file 2	2621	4006	7:49 min	12	24	4.7%	12.4%
file 3	3343	6761	5:15 min	3	7	2.1%	4.4%
file 4	3419	4845	13:00 min	12	40	4.9%	10.3%

Figure 3.7: Quantitative data in IBM application, (Figure 9. Page 13 Komondoor et al. [KH01])

Shepherd et al. [SGP04] is another work that used PDG for detecting refactorable clones by incorporating Aspect Oriented Programming (AOP). They use advices in AspectJ, that is part of a framework they developed called Ophir [SP03]. There are different advices such as: *after*, *before*, and *around*.

Their work only focuses on finding clones that could be refactored using AspectJ advices and that are scattered throughout the software system, and is divided into four steps. The **first step** is to construct PDGs for all methods. The **second step** is to identify an initial clone set by matching the PDGs of the candidate clones and by incorporating AST information when matching statements to improve the accuracy. Control dependencies are used in finding possible matches based on their syntactic structure as in Komondoor et al. [KH01].

The detection of candidate clones starts by comparing all statements that are directly nested (children) under the entry node (method signature), and each pair of AST compatible nodes is added to a working list. After examining the entry node children, a similar process is performed on the elements of the working list. The **third step** is used to filter unwanted candidates by removing those that do not have similar data-dependencies, and removing candidates that have data dependencies to or from nodes outside the clones. The **last step** is to merge similar pairs of clones into the same group. The merging of similar pairs is done by comparing the nodes of the first instances in both pairs and if the AST comparison is successful, then both pairs are placed in the same group.

They evaluated their work that is integrated in Ophir on two projects, namely JHotDraw and Tomcat. Their work achieved an accuracy in mining clones of ($\approx$ 90%). However, the complexity in identifying the initial candidates will increase if *after* and *around* advices are used. Also comparing two subtrees to check for isomorphism is NP-complete, so they had to sacrifice the accuracy for performance by using a lightweight approach when comparing statements.

Software Plagiarism detection is another field that uses PDGs. Liu et al. [Liu+06] developed GPLAG that uses program dependence graphs for detecting plagiarism in software. Previous tools in that field were tricked by developers through five main changes to the copied code: 1) Inserting/remove blanks/comments, 2) renaming identifiers, 3) reordering statements, 4) changing control predicates, such as *for* to *while* loops, and by 5) adding new statements. They formulate their work by answering two questions: How can you decide if two PDGs are similar? and How can you make graph comparison efficient for a large set?

They answered the first question in two parts: In **Part 1**, they validate the original subgraph g and plagiarized subgraph g' against the five concealment operations mentioned before. However, these changes do not affect the graph isomorphism in contrast to some other changes such as, two iteration index variables for independent loops in g, where one of them is removed and the other one is used for both loops. In **Part 2**, to address changes affecting the graph isomorphism, they said if the correspondence γ (γ is a relaxed value that represents the belief that a portion of the

PDG will not be changed) between two subgraphs is $(0 < \gamma \leq 1)$, then they consider them as a match, i.e., a plagiarism is detected.

The search space for plagiarism increases as more PDGs are examined, so some filtration of the PDGs is needed to address the second question. They apply a two step filtration process. With a **Lossless filter**, first they remove PDGs that are smaller than K in both the original and suspected systems, and second if the $|g'| < \gamma|g|$ then those pairs are not tested. The lossless filter is required as a means to have enough proof of plagiarism. The second filtration is a **Lossy Filter**. They use a vertex histogram to represent each PDG, where each element holds the frequency for each type of node, such as jump, label, return. After generating both vertex histograms their similarity is evaluated. The drawback of this filtration is that some false negative cases will be pruned.

They measured the performance of GPLAG in two parts: In the first part, the effectiveness of GPLAG was evaluated against MOSS [SWA03] and JPLAG on the *join* application, which took the authors two hours to manually plagiarize it. MOSS and JPLAG failed to detect plagiarism in *join*, while GPLAG was able to find the plagiarized procedures, which indicates that token-based approaches are not as strong as PDG-based approaches in this problem. In the second part, the efficiency of GPLAG is computed by running it on 3 subjects. They found that the Lossless filter was able to remove 50% of the PDG pairs; however, some non-similar pairs survived, but the lossy filter was able to remove them.

Chapter 4

Approach

In this chapter we will present and discuss our work in clone group refactoring. The workflow of our approach is depicted in Figure 4.1. Our approach requires two inputs. The first input is the *abstract syntax tree* for each file in the project that is generated in the **Project Parsing** step. The second input is the location of the *clones* found in the project that is generated in the **Clone Parsing** step. An intermediate step following the previous two steps is to generate a *Program Dependence Graph* (PDG) for the methods that the clones reside in. Next, the **Information Extraction** step uses the generated PDGs and clone locations to extract additional information that will be used in the subsequent steps of our approach.

For each group of clones, we use **Clustering** to find smaller sub-groups (clusters) within the original clone group that have less differences. We try first to group fragments that share a **Common Structure** (isomorphic control structure trees), then within each of the resulting clusters we try to find sub-groups of clone fragments having less **Differences**. After **Clustering** is done and subgroups are created, the **Clone Matching** process starts. First, we do a **Pairwise Matching** of the clones within the same subgroup in the following sequence of pairs $\{(Clone_1, Clone_2),$

$(Clone_2, Clone_3)$,..., $(Clone_{N-1}, Clone_N)$; where $Clone_{1..N} \in subgroup_M\}$, then the **Statement Alignment** step follows where we align the common statements between fragments within the same subgroup. Lastly, we apply a **Refactorability Assessment** on the subgroups to evaluate if the fragments in each subgroup can be refactored together or if there are any precondition violations (Section 2.8).

Before we move on to an in-depth discussion about our work we have to mention that our work is built on top of **JDeodorant**, a code smell detection and refactoring tool, and the first two steps (**Project Parsing**, **Clone Parsing**), including PDG generation are done by **JDeodorant**.

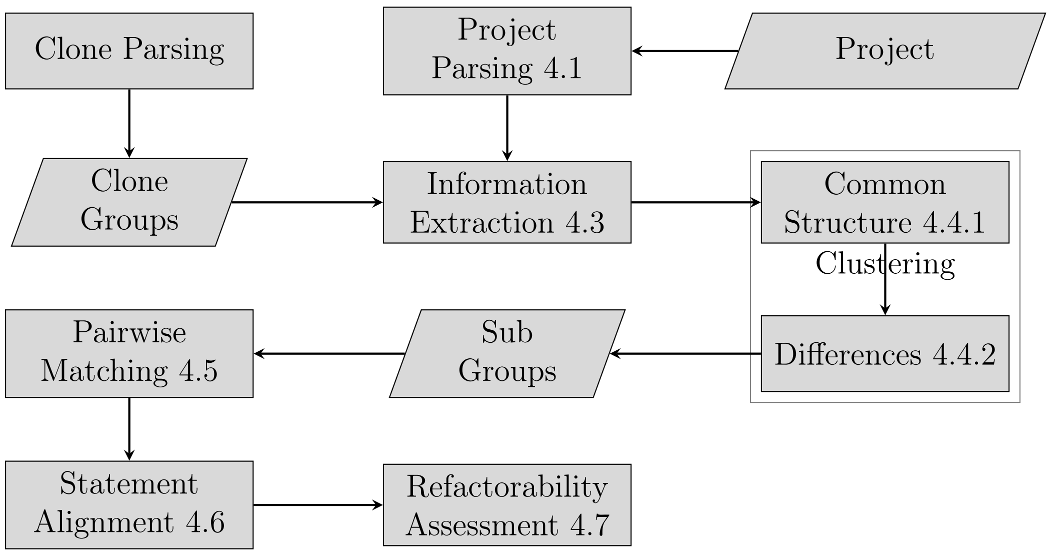


Figure 4.1: Workflow of the Approach

4.1 Project Parsing

Parsing projects is a required step in JDeodorant. The project to be examined needs to be processed to generate an AST for each file inside the project. However, the generated ASTs contain a lot of information that will occupy a huge amount of the system memory, thus the generated ASTs are abstracted to remove some of the unnecessary information, while allowing the ability to recover the original AST when it is needed.

This step is required to generate the *program dependence graph* for the methods that contain the clone instances, but the other projects files are needed as well to generate the data dependencies (Section 2.6) between the clone statements (i.e., data dependence generation may require to analyze other methods being called within the clone instances). Project parsing is executed only once and it is very fast. For instance, the parsing time for Hibernate 3.3.2 that contains 209,000 lines of code is around 70 seconds. The generated abstract syntax trees are then used to build the PDGs, find the **Common Structure**, and perform **Pair Matching**.

4.2 Clone Parsing

Clone detection tools detect and report clones in different formats and file types. Therefore, there is a need to parse these files that contain the reported clones and unify them into a single format. JDeodorant unifies the output from clone detection tools by parsing and creating a new file of type **.xls** (Excel file) that holds the clones and information about them such as: group id, package name, class name, method name, offsets, start-end lines, number of fragments in the group, and other details. JDeodorant so far contains parsers for five clone detection tools, namely CCFinder, CloneDR, NiCad, Deckard, and ConQAT [JDH09].

4.3 Information Extraction

In this step, we extract all required information for the subsequent steps of our approach. Initially, we experimented with a **Metric Approach** (Section 4.3.1) by extracting low-level numerical information for each statement in the PDG, such as the number of method calls, and the number of variable identifiers within the statement. However, as we examined more cases, we observed that this approach would not lead to a generalized solution that avoids the use of similarity thresholds. Therefore, we decided to follow a different approach by extracting higher-level data type information **Data Type Approach** (Section 4.3.2).

4.3.1 Metric Approach

Feature vector is an *n*-dimensional vector of numerical features that represents an object. Feature vectors have been already used for detecting software clones. Deckard [Jia+07] is a clone detection tool that uses AST to detect duplicate code. It transforms the source code into a parse tree (AST tree), which is then transformed into a set of vectors that are clustered based on their similarity, and the resulting clusters represent the detected clone groups.

We follow the same approach as Deckard with the difference that a feature vector is generated for every statement in the detected clones and it is used to map the clone statements. Each entry in the vector represents either the frequency of feature X within a statement (i.e., how many times feature X occurs in a statement) or a boolean value (i.e., feature X exists or not in a statement). The details about the features we extracted can be seen in Table 4.1, which contains all the features that can be extracted from an AST for a given statement, assuming that there is no binding information available from the compiler. Then, by computing the Hamming Distance (Section 2.4.2.2) of the feature vectors corresponding to a pair of statements, we can determine if the statements are similar enough to be matched.

Table 4.1: AST Features in metric approach

#	**Feature**
1	Statement is a control structure (For, If,...) or not
2	Statement creates an instance of a Class (i.e., new Class)
3	Statement creates an instance of an array object (i.e., new Class[n])
4	Number of fields accessed through this reference within statement
5	Number of fields modified through this reference within statement
6	Number of fields accessed within statement that are inherited from a super-class
7	Statement is a super method invocation (i.e., super.method())
8	Number of static methods called within statement
9	Number of boolean literals used within statement
10	Number of char literals used within statement
11	Number of null literals used within statement
12	Number of number literals used within statement
13	Number of string literals used within statement
14	Number of type literals used within statement
15	Number of parameters passed as arguments in method invocations within statement
16	Number of parameters passed as arguments in super method invocations within statement
17	Number of declared variables within statement
18	Number of defined variables within statement
19	Statement is a return statement
20	Statement is a break statement
21	Statement is a continue statement

4.3.1.1 Metric Approach Limitations

To assess the effectiveness of the metric approach, we compared it against the statement matching approach developed by Tsantalis et al. [TMK15], discussed in the Related Work Chapter. More specifically, we counted the percentage of clone pairs for which the statement similarity computed using the metric approach leads to an identical statement mapping as that produced by Tsantalis et al. approach. The metric approach considers two statements mapped if either their feature vector similarity or their textual similarity is equal to 1. The results can be found in Table 4.2. In particular, the quality of the statement mappings produced using the metric approach deteriorates significantly for Type-III clones, since only 43.5% of the examined clone pairs had an identical mapping to Tsantalis et al.

Table 4.2: Results of metric approach at pair level

Clone Type	Identical mapping with Tsantalis et al.
Type I	98.1%
Type II	89%
Type III	43.5%

In the examples that follow, we demonstrate the main limitations of the metric approach.

Example (1): Textually different statements may have perfect similarity according to the metric approach: In the example below, taken from JFreechart, the two statements are matched, because they have exactly the same feature vector **[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0]**. However, their textual similarity is rather low, and they do not use common types.

```
(1) XYDataItem item=getDataItem(count / 2);
(2) ComparableObjectItem item=getDataItem(count / 2);
```

Example (2): Statements that are semantically identical may have different feature vectors: In the example below, taken from Apache-Ant, the feature vector for statement (1) is **[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]**, and for (2) is **[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]**. As a result, the statements are not matched. However, both statements call the same method *addNodeRecursively* passing different argument values. More specifically, the use of a null literal as the last argument in statement (1), and the use of field prefix as the second argument in statement (1), introduce significant differences in the feature vectors of the statements.

```
(1) addNodeRecursively(topChildren.item(i), prefix, null);
(2) addNodeRecursively(nodeChildren.item(i), nodePrefix, nodeObject);
```

The problem with this approach is that it does not consider the data types when mapping statements, and does not tolerate small differences. In the first example, the two statements are considered as mapped because the vector similarity is equal to 1.0, however the two statements use completely different data types and do not share a common type. On the other hand, in the second example, the two statements are exactly the same in terms of their method name, and parameter data types, but due to the differences in the last parameter in both statements *null* and *nodeObject* the similarity is less than 1.0 , and thus they are not mapped. Moreover, other cases such as *int x = getY();* and *int x = this.y;* (i.e., a direct field access is replaced with the corresponding getter method invocation) will not have a vector of similarity of 1.0. However we can see that the right side of the assignment in both statements is of data type int, despite that the first statement is a method call while the second statement is a field access. Therefore, the two statements should be mapped based on the data types being used.

4.3.2 Data Type Approach

In this approach, the information we extract is more related to data types, and names (identifiers, methods, ..., etc). The kind of information we extract depends on the type of the examined statement. For example, if the statement is a method call then we extract different kind of information, compared to a variable declaration statement. We generalized all types of statements into three forms. We have to clarify first that *Expression* in this context refers to a method call, a mathematical or logical operation, an object creation, etc.

Form 1: Type Identifier = Expression In this form we divide the statement based on the location to the assignment operator into left and right. We extract from **Left side** the 1) identifier name, and 2) its type including all its super types. From the **Right side**, if the expression is not a method call, then we extract the type of the expression and all its super types. Below, we provide some examples where the right side is not a method call. If the right side is a method call then we extract different kind of information, which we will explain in the next form.

```
(1) YIntervalSeries s1 = new YIntervalSeries("s1");
(2) result = 29 * result + this.maximumItemCount;
(3) result = 29 * result + (this.allowDuplicateXValues ? 1 : 0);
(4) String result = "@test1@ line testvalue";
```

Form 2: Method call and arguments If the statement or expression is a method/constructor/super-constructor call then we extract three sources of information. **First**, the name of the invoked method. In this case, we extract its name and if it is called through a Class name (i.e., Static method, or super keyword), or an object reference, then we extract it as part of the method name. For instance, in the example below (Case 2) *super.hashCode* is the method name and not just *hashCode*. **Second**, the return type of the method and all its super types. **Third**, the arguments types and all their super types. In other words, we extract the method's return type and arguments types found in the method signature. To explain this, the examples that follow show three cases of method calls followed by an explanation for each case.

```
(1) assertTrue(s1.equals(s2));
(2) int result = super.hashCode();
(3) ComparableObjectItem item = getDataItem(count / 2);
```

Case (1) Method name: assertTrue; Return type: void; Argument types: {Boolean}, i.e., the return type of *s1.equals(s2)*.

Case (2) Method name: super.hashCode; Return Type: int; Argument types: {};

Case (3) Method name: getDataItem; Return Type: ComparableObjectItem; Argument types: {int};

Form 3: Return Expression In this form we extract the expression data type and all its super types.

When extracting types and super types, we might encounter some very common types that need to be excluded. These types are Java library types, such as java.lang.Object, or special interfaces that are inherited or implemented by a very large number of classes. Table 4.3 shows the excluded types and a brief explanation about why we exclude them.

Table 4.3: Excluded types

Type	**Reason**
Object	Super class for every non-primitive type
Serializable	An interface that allows the object to be represented a sequence of bits
Runnable	An interface that contains a single method *run*, used for creating threads
Comparable	An interface that enables sorting of objects
Cloneable	An interface that allows objects to override *clone* method
EventListener	An interfaces that allows the classes implementing it to handle events

4.3.3 Additional information extracted commonly from all statements

There are some common information extracted from all statements regardless of their form:

- Data and Control dependence information: We create a matrix containing the source of each dependence (From/Parent) and the statements that have the incoming dependence (To/Children).
- Fragment location: Class name, Package name, and method name that the clone resides in.

4.4 Clustering

Clone detection tools report multiple clones in the same group, however these clones might not be similar. For instance, token and text based clone detection tools, might even report clones using different control statements, as long as their textual or token similarity is high. Therefore, the objective of the clustering step is to group clone fragments that share a common control structure and have less differences, in order to find subgroups that are actually refactorable. We first try to group fragments that share a common control structure (Section 4.4.1), then for each of the resulting groups we try to find new subgroups containing fragments with less differences (Section 4.4.2).

4.4.1 Common Control Structure

To avoid the cost of combinatorial explosion when trying to find common trees, we follow a transitive approach when examining the clone fragment pairs $(Clone_1, Clone_2), (Clone_2, Clone_3), \ldots, (Clone_{N-1}, Clone_N)$, where N is number of fragments in the group. For instance, assuming that $structure_1$ is the common structure for pair $(Clone_1, Clone_2)$, and $structure_2$ is the common structure for pair $(Clone_2, Clone_3)$, if $structure_1 \equiv structure_2$, then clone instances $(Clone_1, Clone_2, Clone_3)$ can be grouped together.

$structure_1 \equiv structure_2$ iff $|structure_1| = |structure_2|$ and
$\forall x \in structure_1, \exists y \in structure_2 \Rightarrow x = f(y)$ and
$\forall y \in structure_2, \exists x \in structure_1 \Rightarrow y = f(x)$
In other words, there should be a bijection between the nodes of the two common tree structures. If there are more than one common structures in a pair of fragments, then we treat each one of them as a separate opportunity.

Algorithm 3 is the approach we use to group fragments that share a common structure. Clone detection tools, especially, text-based and token-based do not examine if the fragments have the same control structure, thus we need to find first a pair of clone fragments having a common control structure (Lines 2-7). In (Lines

10-17) we start a new group of fragments that share the common structure *firstTree* for fragments F_j and F_k, and in (Lines 19-23), we examine if any of the trees in *secondPair* is equivalent to the *firstTree*.

The algorithm handles two cases in which a common control structure could not be found between the two pairs. In the first case **(Lines 27-29)**, assuming we have two pairs *pair(*F_i, F_j*)* and *pair(*F_j, F_{j+1}*)* and both of them do not have a common control structure, we try next to find if *pair(*F_i, F_j*)* and *pair(*F_j, F_{j+2}*)* share a common structure. In the second case **(Lines 31-34)**, we try to find if *pair(*F_i, F_{j+1}*)* and *pair(*F_{j+1}, F_{j+2}*)* have a common control structure, and if so then we add them to $CommonTrees_{queue}$ as a new starting point for a new subgroup, and then we exit from the current search (*secondPair* = *null*).

Algorithm 3: Clustering using Common Control Structure

```
Input: $F$ contains $n$ clone fragments
Output: List of subgroups of fragments
1  $CommonTrees_{queue} \leftarrow \emptyset$
2  for $(i = 0 \rightarrow (n-1)) \wedge (CommonTrees_{queue} = \emptyset)$ do
3      $tempTree \leftarrow \emptyset$
4      $tempTree \leftarrow findCommonTrees(F_i, F_{i+1})$
5      if $tempTree \neq \emptyset$ then
6          $CommonTrees_{queue} \leftarrow CommonTrees_{queue} \cup tempTree$
7      $i = i + 1$
8  $j = 0, k = 0; j, k \in 1...n$
9  while $CommonTrees_{queue} \neq \emptyset$ do
10     $newGroup \leftarrow \emptyset$
11     $firstTree = CommonTrees_{queue}[0]$
12     $CommonTrees_{queue} \leftarrow CommonTrees_{queue} - firstTree$
13     $F_j = firstFragment(firstTree)$
14     $F_k = secondFragment(firstTree)$
15     $newGroup \leftarrow newGroup \cup F_j \cup F_k$
16     $secondPair = pair(F_k, F_{k+1})$
17     $secondTrees \leftarrow findCommonTrees(F_k, F_{k+1})$
18     while $secondPair \neq null$ do
19         if $firstTree \in secondTrees$ then
20             $newGroup \leftarrow newGroup \cup secondFragment(secondTrees)$
21             $CommonTrees_{queue} \leftarrow$
                 $CommonTrees_{queue} \cup (secondTrees - (firstTree \cap secondTrees))$
22             $firstTree = tree \in secondTrees \mid tree \equiv firstTree$
23             $secondPair = pair(F_{k+1}, F_{k+2})$
24             $secondTrees \leftarrow findCommonTrees(F_{k+1}, F_{k+2})$
25         else
26             $secondTrees \leftarrow findCommonTrees(F_k, F_{k+2})$
27             if $secondTrees \neq \emptyset$ then
28                 $secondPair = pair(F_{k+1}, F_{k+2})$
29             else
30                 $F_j = firstFragment(firstTree)$
31                 $tempTree \leftarrow findCommonTrees(F_j, F_{k+1})$
32                 $CommonTrees_{queue} \leftarrow CommonTrees_{queue} \cup tempTree$
33                 $secondPair = null$
34     add $newGroup$ to $Groups$
```

4.4.2 Differences

The second step of clustering is to group the fragments that are less different. We use the features we extracted and discussed in Section 4.3 to generate a **Distance matrix** (Section 4.4.2.1) that is needed to apply the **Hierarchical clustering** algorithm (Section 4.4.2.2).

4.4.2.1 Distance Matrix

The distance matrix is computed using the features we extracted in the **Data type** approach (Section 4.3.2). We developed a heuristic (**Algorithm 4**) to compute the differences between the fragments which takes two inputs:

For each fragment we create a single:

- Dimension vector (**Info1**) that holds the number of statements that satisfy each feature in Table 4.4.

Table 4.4: Info1 Elements

Element
Number of variable declaration statements
Number of variable assignment statements
Number of method call statements
Number of control statements (e.g., `if`, `for`, `while`)
Number of branching statements (e.g., `return`, `break`)
Total number of statements in the fragment

- **Info2** consists of the elements shown in Table 4.5, where each element is a set.

Table 4.5: Info2 Elements

Element
All types and super types of variable identifiers
All AST statement types (e.g., `if`, `while`, `throw`)
All types and super types of expressions (Form 1, section 4.3.2)
All types of method call arguments (Form 2, section 4.3.2)
All method names (Form 2, section 4.3.2)
All variable identifier names

Algorithm 4 is the heuristic we apply to compute the differences matrix between all fragments in the same subgroup. Score for vector **Info1** is computed using equation 4.1 and is used in (Algorithm 4, Line 9), while the score for **Info2** is calculated using equation 4.2 and is used in (Algorithm 4, Line 10).

Algorithm 4: Distance Matrix

```
   Input: $F$ contains $m$ fragments for a SubGroup
   Output: $m \times m$ differences matrix
1  $result \leftarrow$ create $m \times m$ empty matrix
2  for $i = 0 \rightarrow (m-1)$ do
3      $vector_{\alpha_1}$ = extract Info1 for $F_i$
4      $list_{\beta_1}$ = extract Info2 for $F_i$
5      for $j = (i+1) \rightarrow m$ do
6          $score1, score2, score = 0$
7          $vector_{\alpha_2}$ = extract Info1 for $F_j$
8          $list_{\beta_2}$ = extract Info2 for $F_j$
9          $score_\alpha$ = score from calculating the differences between $vector_{\alpha_1}$ and
             $vector_{\alpha_2}$ using equation 4.1
10         $score_\beta$ = score from calculating the differences between $list_{\beta_1}$ and $list_{\beta_2}$
             using equation 4.2
11         $score = score_\alpha + score_\beta$
12         add $score$ to $result$
13         $j = j + 1$
14     $i = i + 1$
```

$$score = \sum_{i=1}^{n}(|A_i - B_i|) \tag{4.1}$$

were A, B correspond to **Info1** for the two fragments, respectively, and n is the length of **Info1**.

$$score = \sum_{i=1}^{n}(|Union(A_i, B_i) - Intersection(A_i, B_i)|) \tag{4.2}$$

were A, B correspond to **Info2** for the two fragments, respectively, and n is the size of **Info2**.

4.4.2.2 Hierarchical Clustering

We apply the Hierarchical clustering algorithm that we described in Section 2.5.1, using the distance matrix that we computed earlier to find subgroups within the clusters that we obtained in the previous clustering step based on the common control structure of the clone fragments. We select the first and the highest Silhouette Coefficient (Section 2.5.2) value to determine the best quality clustering as can be seen in Algorithm 5.

Algorithm 5: Clustering using Differences

```
   Input: SubGroup $S$ contains $F$ fragments
   Result: $m$ clusters
1  $distancematrix$ compute the distance matrix for $S$
2  $clustering \leftarrow$ each fragment in $F$ is assigned to a separate cluster
3  $clusteringScoreMap \leftarrow <\emptyset, 0.0>$ stores pairs of a clustering and its
     Silhouette Coefficient score
4  $clustering_{new} \leftarrow clustering$
5  while $|clustering_{new}| > 1$ do
6      $clustering_{new} \leftarrow$ result of merging the two closest clusters in $clustering_{new}$
7      $score =$ compute Silhouette Coefficient for $clustering_{new}$
8      $clusteringScoreMap \leftarrow clusteringScoreMap \cup <clustering_{new}, score>$
9  $\rightarrow$ return the first $clustering$ in $clusteringScoreMap$ with the highest $score$
```

4.5 Pairwise Matching

The next step in our approach, after extracting the required information (Section 4.3), and clustering the clone fragments into subgroups (Section 4.4), is the pairwise statement matching. Figure 4.2 represents the flow we follow in pairwise statement matching.

4.5.1 Generating Similarity Matrices

Before we explain the algorithms behind statement mapping, we have to generate first an $m \times n$ matrix, where m and n are the number of statements in the first and second clone fragment, respectively.

4.5.1.1 Feature Vector Similarity

The first matrix is computed from the information we extract in section 4.3.2. For each statement, we create a vector that consists of the elements shown in Table 4.6. We encode the information we extract into a feature vector to avoid the overhead from using conditional logic while looking for a match, and to have a more generalized representation.

Table 4.6: Features of the Feature Vector

Feature	**Description**
Statement Type	Single value from Table 4.7
AST Statement Type	`if`, `for`, `while` ...
Identifier Type	hashCode of the identifier type
Identifier name	hashCode of the identifier name
Expression Type	hashCode of the expression type
Number of Arguments	integer (Form 2, section 4.3.2)

Table 4.7: Statement Type

Value	**Feature**	**Format**
1	Declaration	`Type variableName = . . .`
2	Assignment	`variableName = . . .`
3	Method call	`method(. . . )` or `expr.method(. . . )`
4	Control	`if`, `for`, `while`, `do-while`, `switch`
5	Branching	`return`, `break`, `continue`, and `throw`

We developed a heuristic (Algorithm 6), where the Hamming Distance (Section 2.6) is used to compute the similarity for elements in Table 4.6, and the Jaccard Index (Section 2.8) is used in (Line 19) to find the similarity between methods arguments.

Algorithm 6: Vector Similarity Heuristic

```
Data: Fragment1 has $m$ statements, Fragment2 has $n$ statements
Output: $m$ $x$ $n$ matrix
$\alpha \leftarrow$ vector for statement $x$ in the first fragment
$\beta \leftarrow$ vector for statement $y$ in the second fragment
$score = 0$
$denominator = |\alpha|$
for $i = 0$ to $|\alpha|$ do
    if $\alpha_i \equiv \beta_i$ then
        if $\alpha_i > 0$ then
            $score = score + 1$
        else
            $denominator = denominator - 1$
    $i = i + 1$
if $\alpha_{i_{args}} > 0 \wedge \beta_{i_{args}} > 0$ then
    $denominator = denominator + 1$
    $arguments_\alpha = \alpha_{i_{args}}$
    $arguments_\beta = \beta_{i_{args}}$
    if $|arguments_\alpha| = |arguments_\beta|$ then
        $validArgs = True$
        for $j$=0 to $|arguments_\alpha|$ do
            if $(arguments_{\alpha_j} \cap arguments_{\beta_j}) = \emptyset$ then
                $validArgs = False$
                break
            $j = j + 1$
        if $validArgs = True$ then
            $score = score + 1$
$similarity = (score/denominator)$
```

4.5.1.2 String Similarity

Levenshtein Similarity (Section 2.2) is used to generate the string similarity matrix, however the text is preprocessed before applying the algorithm. The preprocessing consists of two steps, which are:

Transformation : In this step, we transform operations `+=`, `-=`, `/=`, `*=` to the expanded equivalent form, for example, `x += 9` will become `x = x + 9`.

Formatting : in this step we remove spaces, tabs, new line characters, and "this.".

4.5.1.3 Thresholds

In this work we decided to avoid the use of thresholds at any step when mapping clone statements, in order to have a generic solution that does not rely on threshold tuning. Therefore, instead of using thresholds, we apply multiple rounds of statement mapping and in each round we relax the constraints used to determine if two statements match, as long as the data and control dependencies are preserved (Figure 4.2).

4.5.2 Statement mapping algorithms

After the two similarity matrices are generated, we round all numbers that are equal or greater than 0.99 (≥ 0.99) to 1.0, and any value less than 0.99 is set to 0.0. So, the elements of the matrices become either 1.0 when the statements have identical similarity, or 0.0 when the statements are not similar.

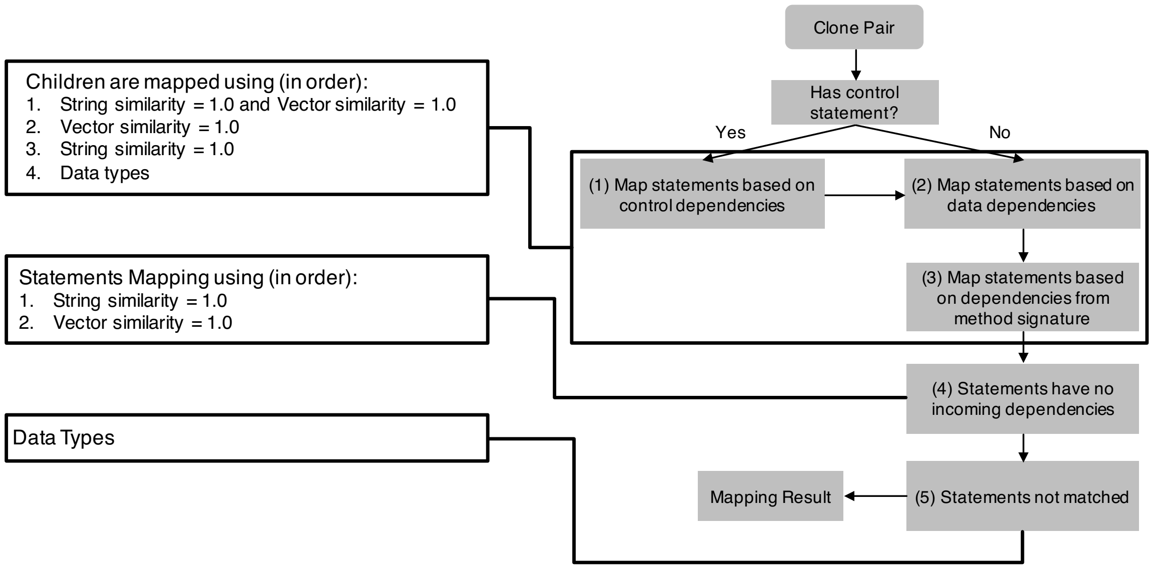

Clone Pair
Has control statement?
Yes
No
(1) Map statements based on control dependencies
(2) Map statements based on data dependencies
(3) Map statements based on dependencies from method signature
(4) Statements have no incoming dependencies
(5) Statements not matched
Mapping Result
Children are mapped using (in order):
1. String similarity = 1.0 and Vector similarity = 1.0
2. Vector similarity = 1.0
3. String similarity = 1.0
4. Data types
Statements Mapping using (in order):
1. String similarity = 1.0
2. Vector similarity = 1.0
Data Types

Figure 4.2: Workflow of the Approach

Algorithm 7 is the starting point in the statement mapping process. If the clone pair has a common control structure, then we apply Algorithm 7. This step is skipped for the clone pairs that do not have a common structure.

Algorithm 7: Initial Step in Pair matching

```
Data: $Fragment_1$, $Fragment_2$
Output: Statement mapping
1 if commonStructure($Fragment_1$, $Fragment_2$) then
2     Add the mapped control statements in the common structure to the set of
      mapped statements;
3     Proceed to Algorithm 8;
4 else
5     Proceed to Algorithm 9;
```

Algorithms 8 and 9 try to find the best matches for control parent statements and their nested children. In Algorithm 8, the parent statements are control statements that have been mapped when the common control structure was found (Section 4.4.1). The next step is to match the statements directly nest under the parent statements. On the other hand, in Algorithm 9 we have to look for the best match for parent statements P (statements that have outgoing dependencies) and their children C (statements that have incoming dependencies from their parents), and to solve this problem we use a **Scoring System**.

To explain the **Scoring System**, let us assume $P1 \rightarrow C1$ where *P1* is a parent statement and *C1* is the set of its children in the first fragment. Also let us assume that the candidate matches in the second fragment are $\{P2_1 \rightarrow C2_1, P2_2 \rightarrow C2_2, \ldots, P2_n \rightarrow C2_n\}$. The candidate matches are selected based on the parent statements ($P1$, $\{P2_1, P2_2, \ldots, P2_n\}$) and the similarity metrics that were generated earlier. For every combination ($P1$, $P2 \in \{P2_1, P2_2, \ldots, P2_n\}$) we first try to match their children ($C1$, $C2 \in \{C2_1, C2_2, \ldots, C2_n\}$), then we compute the score for every pair $\{(P1 \rightarrow C1, P2_1 \rightarrow C2_1), (P1 \rightarrow C1, P2_2 \rightarrow C2_2), (P1 \rightarrow C1, P2_n \rightarrow C2_n)\}$.

The score for a pair $(P1 \rightarrow C1, P2_n \rightarrow C2_n)$ is computed using equation 4.3.

$$score(P1, P2_n) = \sum_{i=1,j=1}^{|C1|,|C2_n|} (vectorSimilarty(C1_i, C2_{nj}) + stringSimilarty(C1_i, C2_{nj})) \quad (4.3)$$

where $C2_{nj}$ is the match found for $C1_i$

When the child statements are mapped, they are added to the set of **mapped statements**, and are not further considered. Thus, if a statement is matched by applying Algorithm 8, we do not attempt to match it again in the following algorithms.

In Algorithm 8: **(Lines 8-11)** are executed to match statements that are inside the `else` clause, when we have `if-else` control statements. In **Steps 1-3**, the matched statements are required to have same AST structure, while in **Step 4** the marching is more relaxed, and we apply **Algorithm 11**. In **Algorithm 11**, statements will be considered as matched if they satisfy one of the conditions below:

- The two statements have an assignment operator (Declaration or initialization statement), and both variables share a common type or super type **(Lines 3-5)**.

- The two statements are method calls and both have to have the same number of arguments that share a data type or super type in respect to the order of parameters in both statements, and the same name as we described in Section 4.3.2 **(Lines 7-8)**.

- The two statements are return statements and the expressions they are returning have the same type or super type **(Lines 10-11)**.

```
Algorithm 8: Pair matching using Control Dependencies
Data: Common structure of length $k$
Output: Mapped child statements ($C1$, $C2$)
1  for $i$ from 0 to $k$ do
2      $C1_T \leftarrow$ statements that have true incoming control dependence from $P1_i$
3      $C2_T \leftarrow$ statements that have true incoming control dependence from $P2_i$
4      Step 1: match($C1_T$, $C2_T$) iff string similarity = vector similarity = 1.0
5      Step 2: match($C1_T$, $C2_T$) iff vector similarity = 1.0
6      Step 3: match($C1_T$, $C2_T$) iff string similarity = 1.0
7      Step 4: match($C1_T$, $C2_T$) iff there is type matching
8      $C1_F \leftarrow$ statements that have false incoming control dependence from $P1_i$
9      $C2_F \leftarrow$ statements that have false incoming control dependence from $P2_i$
10     if $(|C1_F| > 0) \wedge (|C2_F| > 0)$ then
11         repeat Step 1-4 for pair ($C1_F$, $C2_F$)
12 $\rightarrow$ Proceed to Algorithm 9
```

```
Algorithm 9: Pair matching based on Data Dependencies
Data: $P1_m \rightarrow C1_m$ in $Fragment_1$, $P2_n \rightarrow C2_n$ in $Fragment_2$
Output: Best match for $P1 \rightarrow C1$ in $Fragmen_t2$
1  for $i = 0$ to $n$ do
2      Step 1: match($C1$, $C2_i$) iff string similarity = vector similarity = 1.0
3      Step 2: match($C1$, $C2_i$) iff vector similarity = 1.0
4      Step 3: match($C1$, $C2_i$) iff string similarity = 1.0
5      Step 4: match($C1$, $C2_i$) iff thre is type matching
6      $score$ = use equation 4.3 to compute the score for $(P1, P2_i)$
7  $\rightarrow$ return the first $P2_k \rightarrow C2_k$ with the highest score
```

So far, the algorithms we used in statement mapping rely on incoming dependencies, but this does not apply always as some statements might not have any kind of incoming dependencies, such as a method call where it does not take any arguments, and it is not nested under control statement. **Algorithm 10** is used to handle this problem. In the first step (Lines 3-4), we look for a statement that has a string similarity equal to 1.0, then in the second step (Lines 5-6), for any statement could not be matched using string similarity we use vector similarity equal to 1.0.

Algorithm 10: Pair matching for statements having no incoming dependencies

```
Data: $S_1$ statements in $Fragment_1$ and $S_2$ statements in $Fragment_2$
Output: Matched statements
1 $noDeps1 \leftarrow$ find all statements in $S_1$ having no incoming data or control
    dependencies
2 $noDeps2 \leftarrow$ find all statements in $S_2$ having no incoming data or control
    dependencies
3 foreach statement in noDeps1 do
4     look for statement in $noDeps2$ that has a string similarity equal to 1.0
5 foreach statement in noDeps1 do
6     look for statement in $noDeps2$ that has a vector similarity equal to 1.0
```

Finally, Algorithm 11, is more lenient than the previous algorithms we used.

- In case the statements are **assignments**, then we check the left hand side of the assignment to see if both statements share a common data type or super type (Lines 7-8).
- In case the statements are **method calls**, then we check if the method names are the same and they have the same number of arguments (Lines 11-12) without validating if the arguments share a data type or super type.
- In case the statements are **return**, then we check if the returned expressions share a common data type or super type (Line 15-16).

Algorithm 11: Matching of previously unmatched statements

```
Data: $P1 \rightarrow C1$ in $Fragment_1$ and $P2 \rightarrow C2$ in $Fragment_2$
Output: Matched statemens ($C1$, $C2$)
1  for i from 0 to $|C1|$ do
2      for j from 0 to $|C2|$ do
3          if $C1_i$ and $C2_j$ are assignment statements then
4              if $variableTypes(C1_i) \cap variableTypes(C2_j) \neq \emptyset$ then
5                  match is found ($C1_i$, $C2_j$)
6          else
7              if $C1_i$ and $C2_j$ are method calls $\wedge$ same method name $\wedge$ same
                 number of arguments then
8                  match is found ($C1_i$, $C2_j$)
9              else
10                 if $C1_i$ and $C2_j$ are return statements $\wedge$ expressions returned
                     shares a type then
11                     match is found ($C1_i$, $C2_j$)
```

There are few conditions we apply before adding the statements to the list of mapped statements:

- For algorithms 8, 9, and 10 we are more strict as the statements need to have same AST structure.
- For all algorithms we ensure that the statements have at least the same AST statement type (`if`, `for`, `while`, etc.).

4.6 Statement Alignment

Since statement mapping is performed between pairs of clone fragments, we need an additional step to align the statement mapping results obtained from the previous step and find all statements that are common across all fragments in the same clone subgroup. Algorithm 12 describes the approach we use to find common statements across all fragments within the same subgroup.

Algorithm 12: Statement alignment within a clone subgroup

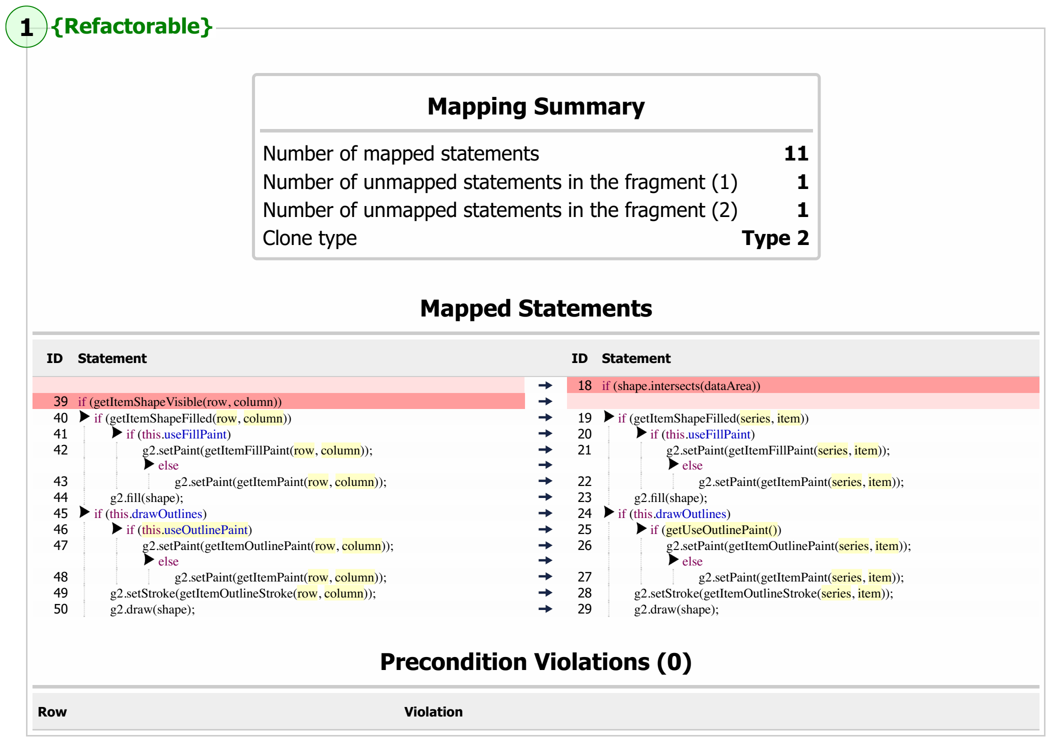
```
Data: Fragments $F$ in $subGroup_m$
Output: Statements matched across all fragments in $subGroup_m$
1 $N = |F|$
2 $statements_{common} \leftarrow pairMatching(F_0, F_1)$
3 for $i = 1$ to $(N - 1)$ do
4     $statements_{common} \leftarrow statements_{common} \cap pairMatching(F_i, F_{i+1})$
5     $i = i + 1$
```

4.7 Refactorability Assessment

The refactorability assessment is performed based on Tsantalis et al. [TMK15] work that is part of the JDeodorant tool. There are two sets of preconditions examined by JDeodorant: The first set determines if the clones can be merged by parameterizing the differences using regular parameters (Section 2.8). The second set determines if the clones can be merges by parameterizing the differences using Lambda expressions (introduced in Java 8).

We extended only the first set of preconditions to be examined for a group of clone fragments (the original preconditions were examined only for a pair of clone fragments). A pair or a group of clones is safe to be refactored, if none of the preconditions listed in Section 2.8 is violated, otherwise it is considered as non-refactorable.

The examples below demonstrate two cases where in Example 4.3 the pair of fragments can be refactored safely since there are no precondition violations. Also, in this example we note that the control statements (highlighted in red) were excluded from the mapping process, because they do not have the same AST structure. However, in Example 4.4 the pair cannot be refactored, since there is a violation for Precondition 6 [TMK15], which states that *"The mapped statements within the clone fragments should return at most one variable of the same type to the original methods from which they are extracted"*.



1 {Refactorable}

Mapping Summary

Number of mapped statements	**11**
Number of unmapped statements in the fragment (1)	**1**
Number of unmapped statements in the fragment (2)	**1**
Clone type	**Type 2**

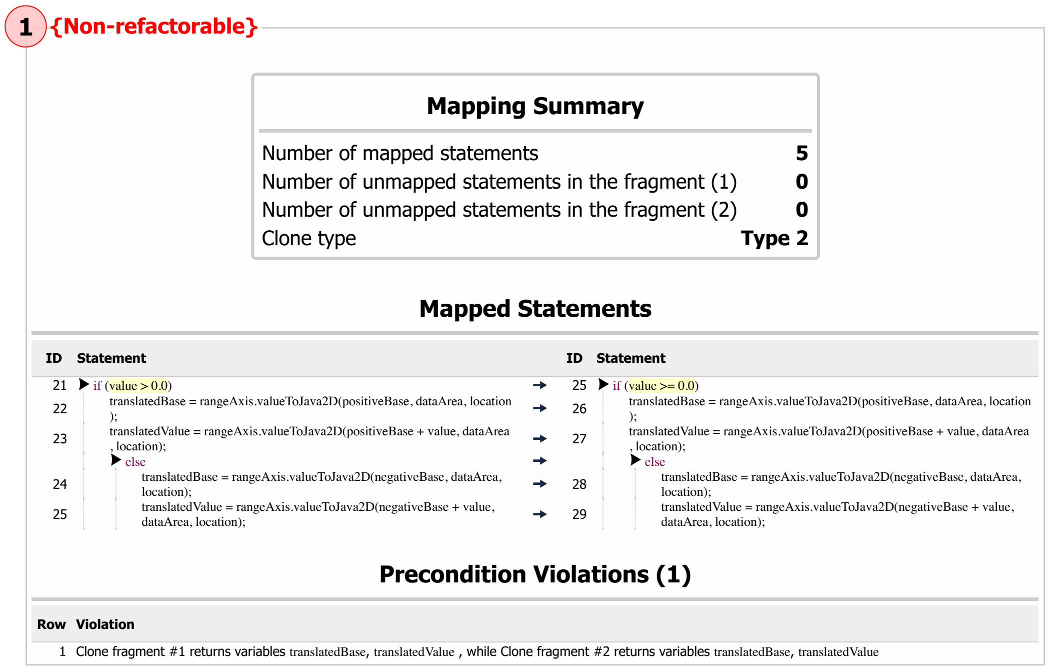
Mapped Statements

ID	Statement		ID	Statement
		➔	18	if (shape.intersects(dataArea))
39	if (getItemShapeVisible(row, column))	➔		
40	▶ if (getItemShapeFilled(row, column))	➔	19	▶ if (getItemShapeFilled(series, item))
41	▶ if (this.useFillPaint)	➔	20	▶ if (this.useFillPaint)
42	g2.setPaint(getItemFillPaint(row, column));	➔	21	g2.setPaint(getItemFillPaint(series, item));
	▶ else	➔		▶ else
43	g2.setPaint(getItemPaint(row, column));	➔	22	g2.setPaint(getItemPaint(series, item));
44	g2.fill(shape);	➔	23	g2.fill(shape);
45	▶ if (this.drawOutlines)	➔	24	▶ if (this.drawOutlines)
46	▶ if (this.useOutlinePaint)	➔	25	▶ if (getUseOutlinePaint())
47	g2.setPaint(getItemOutlinePaint(row, column));	➔	26	g2.setPaint(getItemOutlinePaint(series, item));
	▶ else	➔		▶ else
48	g2.setPaint(getItemPaint(row, column));	➔	27	g2.setPaint(getItemPaint(series, item));
49	g2.setStroke(getItemOutlineStroke(row, column));	➔	28	g2.setStroke(getItemOutlineStroke(series, item));
50	g2.draw(shape);	➔	29	g2.draw(shape);

Precondition Violations (0)

Row	Violation

Figure 4.3: Example of a refactorable clone pair



1 {Non-refactorable}

Mapping Summary

Number of mapped statements	**5**
Number of unmapped statements in the fragment (1)	**0**
Number of unmapped statements in the fragment (2)	**0**
Clone type	**Type 2**

Mapped Statements

ID	Statement		ID	Statement
21	▶ if (value > 0.0)	➔	25	▶ if (value >= 0.0)
22	translatedBase = rangeAxis.valueToJava2D(positiveBase, dataArea, location);	➔	26	translatedBase = rangeAxis.valueToJava2D(positiveBase, dataArea, location);
23	translatedValue = rangeAxis.valueToJava2D(positiveBase + value, dataArea , location);	➔	27	translatedValue = rangeAxis.valueToJava2D(positiveBase + value, dataArea , location);
	▶ else	➔		▶ else
24	translatedBase = rangeAxis.valueToJava2D(negativeBase, dataArea, location);	➔	28	translatedBase = rangeAxis.valueToJava2D(negativeBase, dataArea, location);
25	translatedValue = rangeAxis.valueToJava2D(negativeBase + value, dataArea, location);	➔	29	translatedValue = rangeAxis.valueToJava2D(negativeBase + value, dataArea, location);

Precondition Violations (1)

Row	Violation
1	Clone fragment #1 returns variables translatedBase, translatedValue , while Clone fragment #2 returns variables translatedBase, translatedValue

Figure 4.4: Example of a non-refactorable clone pair

Chapter 5

Qualitative Study

This chapter evaluates our statement mapping approach. The evaluation covers three parts, a) the accuracy of our statement mapping process, b) the performance of our algorithms in terms of execution time, and c) the refactorability for the resulting clone groups. The accuracy and performance of our approach are measured in comparison to Tsantalis et al. [TMK15]. This study includes the clone groups studied in [Tsaa], which were detected by Deckard [Jia+07] in the JFreechart project [JFr].

JFreechart was selected for this study because : 1) It contains a large set of clone pairs; 2) It has a test suite with high code coverage, increasing the probability of having clone pairs covered by a test; 3) The test suite has no failing tests, allowing us to examine if our clone refactoring implementation would cause tests failures by changing the behavior of the program. As for the clone detection tool, we selected Deckard because of the location diversity of the refactorable clone groups, which requires to apply different refactoring strategies.

5.1 Results

The clone dataset [Tsaa] contains 847 groups consisting of 2306 clone instances. **Table 5.1** gives more insight about the clone dataset by listing the number of clone groups per group size. 92.2% of the groups contain less than 5 clone instances.

Table 5.1: Examined clone dataset

Group size	Number of groups
2	591
3	92
4	98
5	21
>5	45

Table 5.2 provides an overview of the results we got by comparing the mapped statements by our work and Tsantalis et al. [TMK15]. The results are categorized by clone type. The third column shows the percentage of clone pairs whose statements were mapped identically by the two approaches. The fourth column shows the percentage of statements that were mapped identically by the two approaches.

Table 5.2: Overall results

Clone type	Number of clone pairs	Identical mappings at clone pair level (%)	Identical mappings at statement level (%)
Type I	337	97.6	99.2
Type II	858	82.3	86.0
Type III	27	55.5	85.2

To enable a more fair comparison of the two approaches, we filtered out some clone pairs falling into two main categories: 1) clone pairs with symmetrical control structures, since the approach developed by Tsantalis et al. uses a greedy search algorithm allowing to obtain a better quality statement mapping in the case of symmetrical `if/else` and `if-else-if` structures (i.e., the code inside the `if` clause of the first clone appears inside the `else-if` or `else` clause of the second clone), and 2) clone pairs labeled with a different clone type by each approach (e.g., first approach produces a statement mapping that can be characterized as Type-II clone, while the second approach produces a statement mapping that can be characterized as Type-III clone).

The results of this filtered clone dataset can be seen in **Table 5.3**. The second column in Table 5.3 shows the total number of pairs we included in our experiment categorized based on their clone type.

Table 5.3: Results included in the comparison

Clone type	**Number of clone pairs**	**Identical mappings at clone pair level (%)**	**Identical mappings at statement level (%)**
Type I	326	100	100
Type II	732	94.0	98.0
Type III	24	62.5	93.0

5.2 Discussion

To evaluate our statement mapping process (Section 4.5) we will investigate two questions listed below, by comparing our work with Tsantalis et al. work.

RQ1: How accurate is our statement mapping algorithm? (Section 5.2.1)

RQ2: How fast is our statement mapping algorithm? (Section 5.2.2)

RQ3: How effective is our approach in terms of cluster-level refactorability? (Section 5.2.3)

RQ4: How effective is our approach in terms of group-level refactorability and performance? (Section 5.2.4)

5.2.1 Accuracy Evaluation

We measure the accuracy of our work by counting the number of cases that we have an identical statement mapping to [TMK15]. As discussed in the Related Work Chapter, the approach by [TMK15] is the current state-of-the-art in clone statement mapping and refactorability analysis.

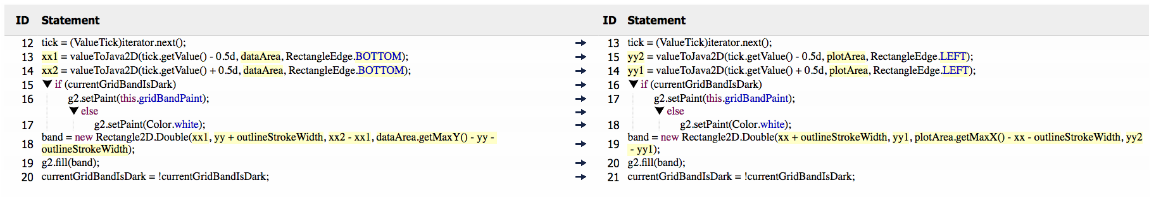
In Table 5.3, we can see that our approach has 100% identical statement mapping with Tsantalis et al. at clone-pair and statement level for **Type-I** clones. On the other hand, we have the same statement mapping for 688 clone pairs out of 732 Type-II pairs (94%), and 15 clone pairs out of 23 Type-III pairs (65.2%). Table 5.4 shows the number of clone pairs with non-identical mapping categorized into **Different mapping**, **More mapped statements**, and **Less mapped statements** if our approach has a different mapping for the same statements, more mapped statements, or less mapped statements, respectively. The last two columns, show the number of clone pairs with non-identical mapping and at the same time more mapped statements, and less mapped statements, respectively.

Table 5.4: Cases with non-identical statement mapping

Clone Type	**Number of clone pairs**	**Different mapping**	**More mapped statements**	**Less mapped statements**	**Different mapping & more mapped statements**	**Different mapping & less mapped statements**
Type II	44	24	15	1	4	0
Type III	8	3	4	1	0	0

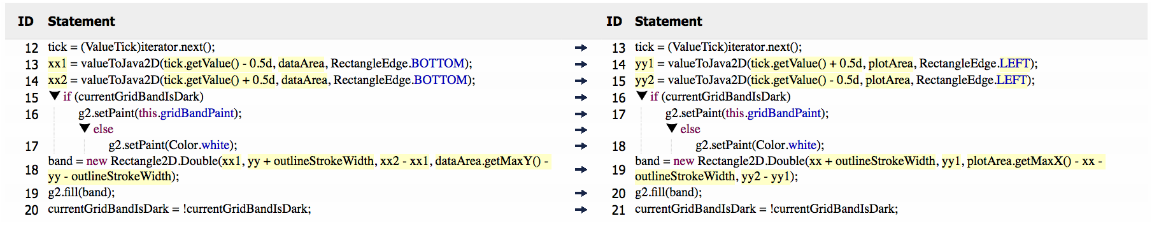
To explain these differences in statement mapping, we will split the discussion into three parts according to the columns in Table 5.4 (Different mapping, More mapped statements, Less mapped statements) and present a representative example from the clone dataset we used in the experiment.

Different statement mapping: Our approach maps the statements in the order they appear within the same block or under the same matched control statement, while the approach by Tsantalis et al. looks for a best match that has the minimum number of differences between the mapped statements. Figure 5.1 shows an example of a different mapping in Type-II clones. The difference appears in the mapping of statements 13 and 14 in the first fragment (clone on the left), where in Figure 5.1a they are mapped to statements 15 and 14, respectively, while in Figure 5.1b they are mapped to statements 14 and 15, respectively (clone on the right). As we can see, Tsantalis et al. approach reversed the order of two statements to minimize the number of differences, while our approach preserved the order of statements in the two clone fragments.



ID	Statement		ID	Statement
12	tick = (ValueTick)iterator.next();	→	13	tick = (ValueTick)iterator.next();
13	xx1 = valueToJava2D(tick.getValue() - 0.5d, dataArea, RectangleEdge.BOTTOM);	→	15	yy2 = valueToJava2D(tick.getValue() - 0.5d, plotArea, RectangleEdge.LEFT);
14	xx2 = valueToJava2D(tick.getValue() + 0.5d, dataArea, RectangleEdge.BOTTOM);	→	14	yy1 = valueToJava2D(tick.getValue() + 0.5d, plotArea, RectangleEdge.LEFT);
15	▼ if (currentGridBandIsDark)	→	16	▼ if (currentGridBandIsDark)
16	g2.setPaint(this.gridBandPaint);	→	17	g2.setPaint(this.gridBandPaint);
	▼ else	→		▼ else
17	g2.setPaint(Color.white);	→	18	g2.setPaint(Color.white);
18	band = new Rectangle2D.Double(xx1, yy + outlineStrokeWidth, xx2 - xx1, dataArea.getMaxY() - yy - outlineStrokeWidth);	→	19	band = new Rectangle2D.Double(xx + outlineStrokeWidth, yy1, plotArea.getMaxX() - xx - outlineStrokeWidth, yy2 - yy1);
19	g2.fill(band);	→	20	g2.fill(band);
20	currentGridBandIsDark = !currentGridBandIsDark;	→	21	currentGridBandIsDark = !currentGridBandIsDark;

(a) Tsantalis et al. statement mapping

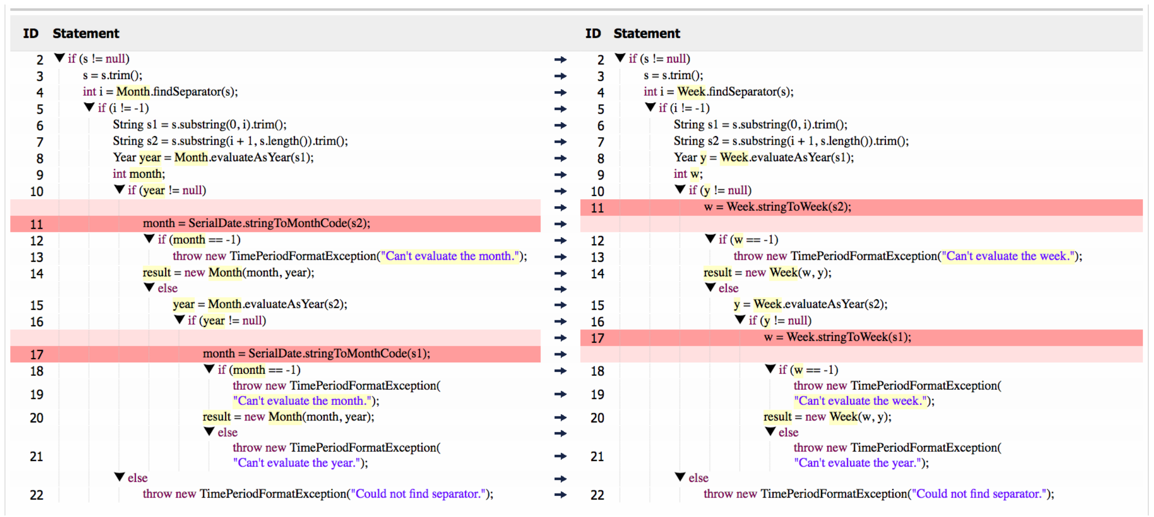


ID	Statement		ID	Statement
12	tick = (ValueTick)iterator.next();	→	13	tick = (ValueTick)iterator.next();
13	xx1 = valueToJava2D(tick.getValue() - 0.5d, dataArea, RectangleEdge.BOTTOM);	→	14	yy1 = valueToJava2D(tick.getValue() + 0.5d, plotArea, RectangleEdge.LEFT);
14	xx2 = valueToJava2D(tick.getValue() + 0.5d, dataArea, RectangleEdge.BOTTOM);	→	15	yy2 = valueToJava2D(tick.getValue() - 0.5d, plotArea, RectangleEdge.LEFT);
15	▼ if (currentGridBandIsDark)	→	16	▼ if (currentGridBandIsDark)
16	g2.setPaint(this.gridBandPaint);	→	17	g2.setPaint(this.gridBandPaint);
	▼ else	→		▼ else
17	g2.setPaint(Color.white);	→	18	g2.setPaint(Color.white);
18	band = new Rectangle2D.Double(xx1, yy + outlineStrokeWidth, xx2 - xx1, dataArea.getMaxY() - yy - outlineStrokeWidth);	→	19	band = new Rectangle2D.Double(xx + outlineStrokeWidth, yy1, plotArea.getMaxX() - xx - outlineStrokeWidth, yy2 - yy1);
19	g2.fill(band);	→	20	g2.fill(band);
20	currentGridBandIsDark = !currentGridBandIsDark;	→	21	currentGridBandIsDark = !currentGridBandIsDark;

(b) Our statement Mapping

Figure 5.1: Example of different mapping

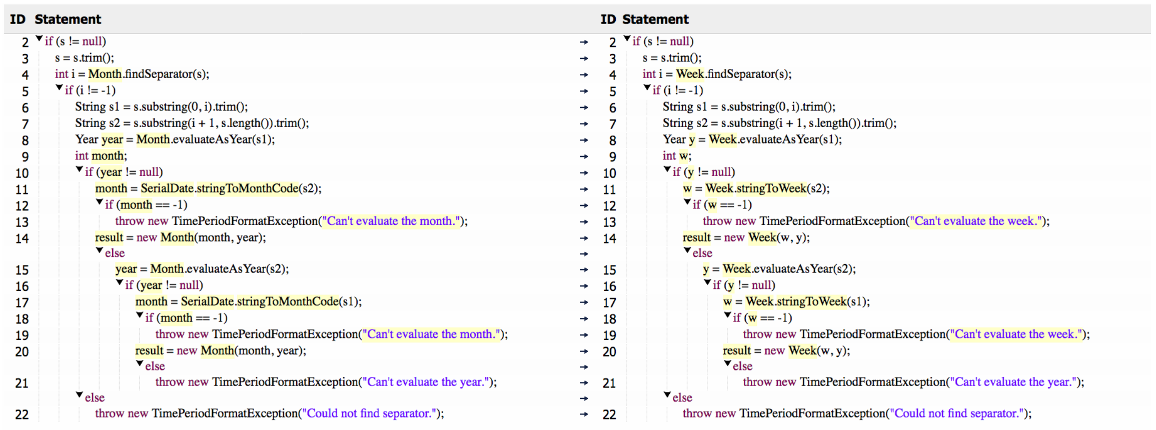
More mapped statements: Our approach is more lenient in statement mapping, because we allow statements to be mapped if they share at least one data type or super type, while Tsantalis et al. require statements to have the same data types or super types in their entire AST structures. For instance, Tsantalis et al. (Figure 5.2a) does not allow the mapping of statements (11, 17) in the first clone (on the left) to statements (11, 17) in the second clone (on the right), because the types `SerialDate` and `Week` are not compatible. While in our work (Figure 5.2b) we allow the statements to be mapped because the variables `month` and `w`, in the first and second clones, have the same data type (i.e., int), and the return type of methods `stringToMonthCode` and `stringToWeek` are the same (i.e., int).



```
ID  Statement
2   ▼ if (s != null)
3       s = s.trim();
4       int i = Month.findSeparator(s);
5       ▼ if (i != -1)
6           String s1 = s.substring(0, i).trim();
7           String s2 = s.substring(i + 1, s.length()).trim();
8           Year year = Month.evaluateAsYear(s1);
9           int month;
10          ▼ if (year != null)

11              month = SerialDate.stringToMonthCode(s2);
12              ▼ if (month == -1)
13                  throw new TimePeriodFormatException("Can't evaluate the month.");
14              result = new Month(month, year);
                ▼ else
15                  year = Month.evaluateAsYear(s2);
16                  ▼ if (year != null)

17                      month = SerialDate.stringToMonthCode(s1);
18                      ▼ if (month == -1)
19                          throw new TimePeriodFormatException(
                            "Can't evaluate the month.");
20                      result = new Month(month, year);
                        ▼ else
21                          throw new TimePeriodFormatException(
                            "Can't evaluate the year.");
            ▼ else
22              throw new TimePeriodFormatException("Could not find separator.");
```

```
ID  Statement
2   ▼ if (s != null)
3       s = s.trim();
4       int i = Week.findSeparator(s);
5       ▼ if (i != -1)
6           String s1 = s.substring(0, i).trim();
7           String s2 = s.substring(i + 1, s.length()).trim();
8           Year y = Week.evaluateAsYear(s1);
9           int w;
10          ▼ if (y != null)
11              w = Week.stringToWeek(s2);

12              ▼ if (w == -1)
13                  throw new TimePeriodFormatException("Can't evaluate the week.");
14              result = new Week(w, y);
                ▼ else
15                  y = Week.evaluateAsYear(s2);
16                  ▼ if (y != null)
17                      w = Week.stringToWeek(s1);

18                      ▼ if (w == -1)
19                          throw new TimePeriodFormatException(
                            "Can't evaluate the week.");
20                      result = new Week(w, y);
                        ▼ else
21                          throw new TimePeriodFormatException(
                            "Can't evaluate the year.");
            ▼ else
22              throw new TimePeriodFormatException("Could not find separator.");
```

(a) Tsantalis et al. statement mapping



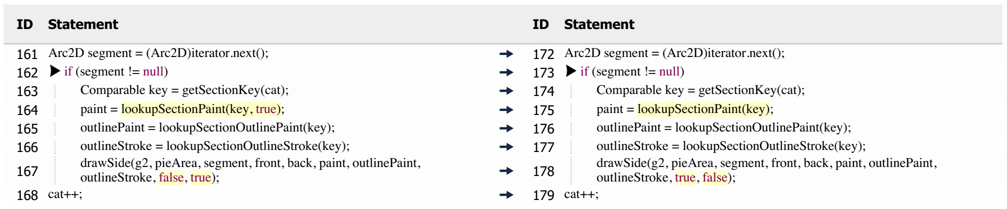
```
ID  Statement
2   ▼if (s != null)
3     s = s.trim();
4     int i = Month.findSeparator(s);
5     ▼if (i != -1)
6       String s1 = s.substring(0, i).trim();
7       String s2 = s.substring(i + 1, s.length()).trim();
8       Year year = Month.evaluateAsYear(s1);
9       int month;
10      ▼if (year != null)
11        month = SerialDate.stringToMonthCode(s2);
12        ▼if (month == -1)
13          throw new TimePeriodFormatException("Can't evaluate the month.");
14        result = new Month(month, year);
          ▼else
15          year = Month.evaluateAsYear(s2);
16          ▼if (year != null)
17            month = SerialDate.stringToMonthCode(s1);
18            ▼if (month == -1)
19              throw new TimePeriodFormatException("Can't evaluate the month.");
20            result = new Month(month, year);
              ▼else
21              throw new TimePeriodFormatException("Can't evaluate the year.");
        ▼else
22        throw new TimePeriodFormatException("Could not find separator.");
```

```
ID  Statement
2   ▼if (s != null)
3     s = s.trim();
4     int i = Week.findSeparator(s);
5     ▼if (i != -1)
6       String s1 = s.substring(0, i).trim();
7       String s2 = s.substring(i + 1, s.length()).trim();
8       Year y = Week.evaluateAsYear(s1);
9       int w;
10      ▼if (y != null)
11        w = Week.stringToWeek(s2);
12        ▼if (w == -1)
13          throw new TimePeriodFormatException("Can't evaluate the week.");
14        result = new Week(w, y);
          ▼else
15          y = Week.evaluateAsYear(s2);
16          ▼if (y != null)
17            w = Week.stringToWeek(s1);
18            ▼if (w == -1)
19              throw new TimePeriodFormatException("Can't evaluate the week.");
20            result = new Week(w, y);
              ▼else
21              throw new TimePeriodFormatException("Can't evaluate the year.");
        ▼else
22        throw new TimePeriodFormatException("Could not find separator.");
```

(b) Our statement mapping

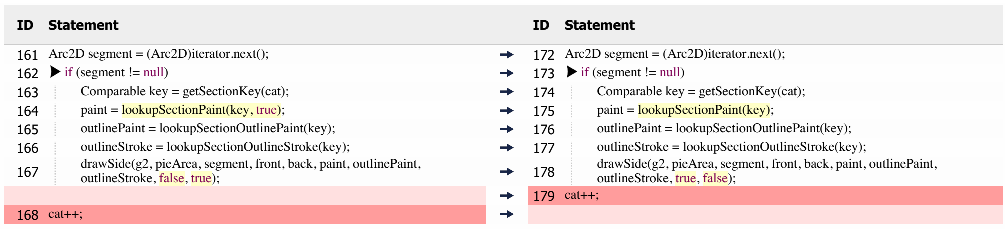
Figure 5.2: Example of a statement mapping with more mapped statements

Less mapped statements: Our approach is more lenient by allowing some statements to be mapped if they share at least one data type or super type. However, this does not apply when we have, for instance a method call, or when the statement does not follow any of the forms we discussed in section 4.3.2. In Figure 5.3, below, our approach did not match the last statements (168 in first clone and 179 in the second clone), because they do not follow any of the forms we discussed in section 4.3.2.



ID	Statement		ID	Statement
161	Arc2D segment = (Arc2D)iterator.next();	→	172	Arc2D segment = (Arc2D)iterator.next();
162	▶ if (segment != null)	→	173	▶ if (segment != null)
163	Comparable key = getSectionKey(cat);	→	174	Comparable key = getSectionKey(cat);
164	paint = lookupSectionPaint(key, true);	→	175	paint = lookupSectionPaint(key);
165	outlinePaint = lookupSectionOutlinePaint(key);	→	176	outlinePaint = lookupSectionOutlinePaint(key);
166	outlineStroke = lookupSectionOutlineStroke(key);	→	177	outlineStroke = lookupSectionOutlineStroke(key);
167	drawSide(g2, pieArea, segment, front, back, paint, outlinePaint, outlineStroke, false, true);	→	178	drawSide(g2, pieArea, segment, front, back, paint, outlinePaint, outlineStroke, true, false);
168	cat++;	→	179	cat++;

(a) Tsantalis et al. statement mapping



ID	Statement		ID	Statement
161	Arc2D segment = (Arc2D)iterator.next();	→	172	Arc2D segment = (Arc2D)iterator.next();
162	▶ if (segment != null)	→	173	▶ if (segment != null)
163	Comparable key = getSectionKey(cat);	→	174	Comparable key = getSectionKey(cat);
164	paint = lookupSectionPaint(key, true);	→	175	paint = lookupSectionPaint(key);
165	outlinePaint = lookupSectionOutlinePaint(key);	→	176	outlinePaint = lookupSectionOutlinePaint(key);
166	outlineStroke = lookupSectionOutlineStroke(key);	→	177	outlineStroke = lookupSectionOutlineStroke(key);
167	drawSide(g2, pieArea, segment, front, back, paint, outlinePaint, outlineStroke, false, true);	→	178	drawSide(g2, pieArea, segment, front, back, paint, outlinePaint, outlineStroke, true, false);
		→	179	cat++;
168	cat++;	→		

(b) Our statement mapping

Figure 5.3: Example of a statement mapping with less mapped statements

5.2.2 Performance Evaluation

After we evaluated the quality of our statement mapping approach, we need to evaluate its performance in terms of execution time. The computed execution time in our approach includes the time to a) extract the required information from the clones, b) find common control structures among the clone fragments, and c) find a mapping between the statements of the clone fragments. In this section, we will compare the performance of our approach and Tsantalis et al. approach.

5.2.2.1 Mean or Median comparison?

First, we need to select the correct measure to compare the performance of the two approaches. Median and Mean are two common measures used to compare groups of data points. To select the correct measure we apply two tests, discussed below, on the execution times reported by both approaches for the clone pairs in Table 5.3, in order to identify whether the time points are normally distributed or not. If the points are normally distributed then we will use the Mean, otherwise we will use the

Median in the comparison.

Skewness [Whe95] This measure describes the symmetry of the data points around the Mean. When the skewness is equal to 0, then the data points are normally distributed and symmetrical; However, when the skewness is greater than 0, or less than 0, it means that the data points are not normally distributed and are skewed to the right or left, respectively.

Kurtosis [Wes14] This measure describes if the shape of the data is the same as the Gaussian distribution (kurtosis $= 0$), or if it has a tail. Distributions that have higher kurtosis (kurtosis > 0) will have thicker and peaked tails, while distributions that have lower kurtosis will have thinner and flatter tails.

Wilcoxon Ranked Test [Wil45] This is a nonparametric test that does not assume the normality in the distribution, and the two score sets to be tested are from the same data set [Sta]. This test examines if there is a statistically significant difference between the two score sets. If the *p-value* of the test is less than 0.05 it implies that there is a statistically significant difference, i.e., the results are not by chance and can be generalized on other data sets.

Cliff's delta [GK05] This is a nonparametric test that is used to calculate the effect size of the differences in the data. Effect size means how regular are the large values in the first distribution in comparison to another distribution [WSG16]. The thresholds for Cliff's delta d (regularity of large values) are: Negligible $|d| < 0.147$; Small $|d| < 0.33$; Medium $|d| < 0.474$; and Large [Mar; Rom+06].

5.2.2.2 The Comparison

The skewness and kurtosis values for our work and Tsantalis et al. are (0.9, 6.3) and (6.6, 82.5), respectively. These results indicate that the data points are not normally distributed as both skewness and kurtosis diverge from 0. Figure 5.4 (excluding outliers) shows the plot for the execution time of both approaches, which shows that the median for both works is almost the same with 69.1(ms) for our approach and 72.3(ms) for Tsantalis et al. approach. The difference in Figure 5.4 appears in the distribution range where in our work the time ranges from 5.1(ms) to around 160(ms), while for Tsantalis et al. it ranges from 6.2(ms) to around 210(ms).

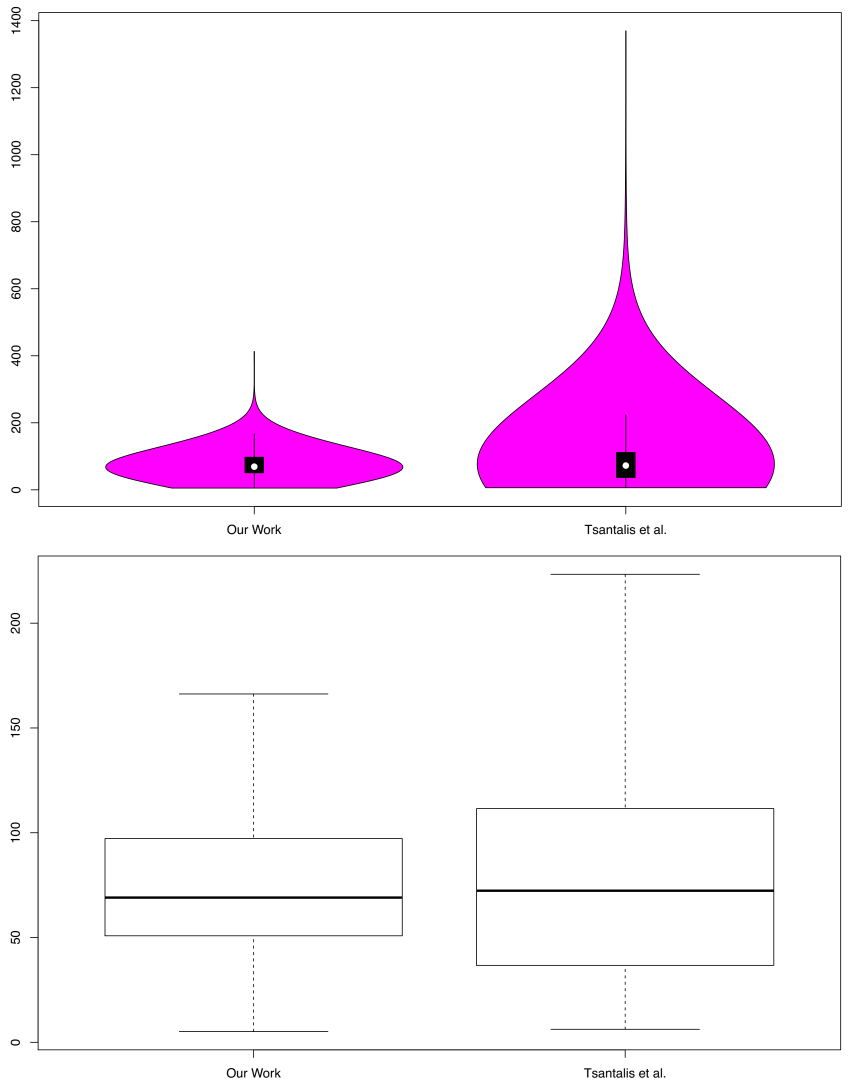


Figure 5.4: Execution time distribution: Our approach vs. Tsantalis et al. in millisecond

Table 5.5a shows the skewness and kurtosis per clone type. Table 5.5b shows the median, minimum, and maximum execution time per-clone type.

Table 5.5: Execution time for each clone type

(a) Statistical tests

Clone Type	**Our Work**		**Tsantalis et al.**	
	Skewness	**Kurtosis**	**Skewness**	**Kurtosis**
Type I	0.57	3.62	0.92	7.1
Type II	1.04	6.83	6.53	72.43
Type III	0.013	2.1	0.56	2.37

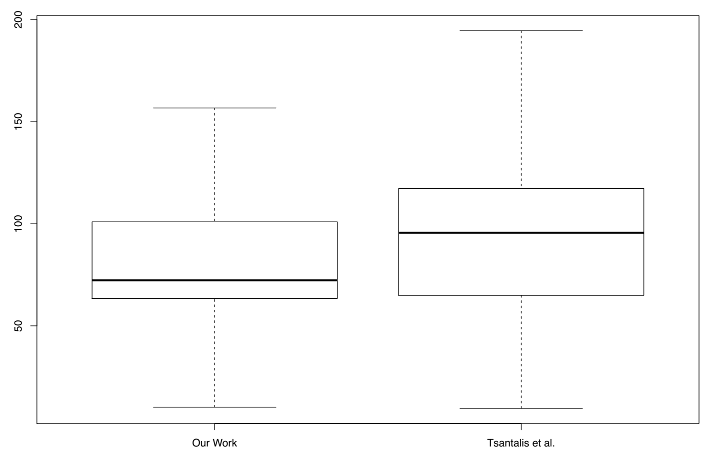
(b) Execution time in (ms)

Clone Type	**Our Work**			**Tsantalis et al.**		
	Median	**Min**	**Max**	**Median**	**Min**	**Max**
Type I	72.3	10.2	234.3	95.7	9.6	371.1
Type II	67.31	5.1	412.7	60.0	6.2	1369.6
Type III	57.8	17.3	99.2	76.9	14.8	221.3

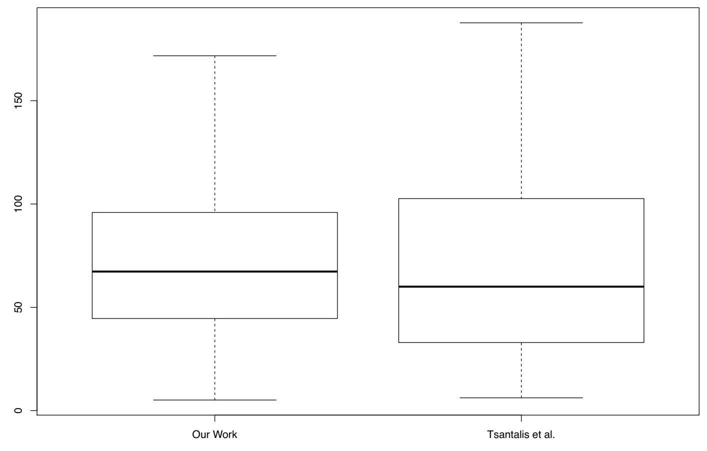
Figure 5.5 shows the box-plots (excluding outliers) for each clone type. For **Clone Type I**, the median time for our work is faster by 23.4(ms) and the execution time is distributed in a smaller range compared to Tsantalis et al. For **Clone Type II**, the median time in our work is slower by 7.3(ms), but the overall execution time distribution shows that our work is slightly faster than Tsantalis et al., since the execution time for our work ranges from 5.1(ms) to 170(ms), while for Tsantalis et al. it ranges from 5.7(ms) to 195(ms). Lastly, for **Clone Type III**, the median time and the overall execution time distribution of our work is faster than Tsantalis et al., since the median time for our work is lower by 19.1(ms) and the execution time

ranges from 17.3(ms) to 100(ms), while for Tsantalis et al. the execution time ranges from 14.8(ms) to 220(ms).

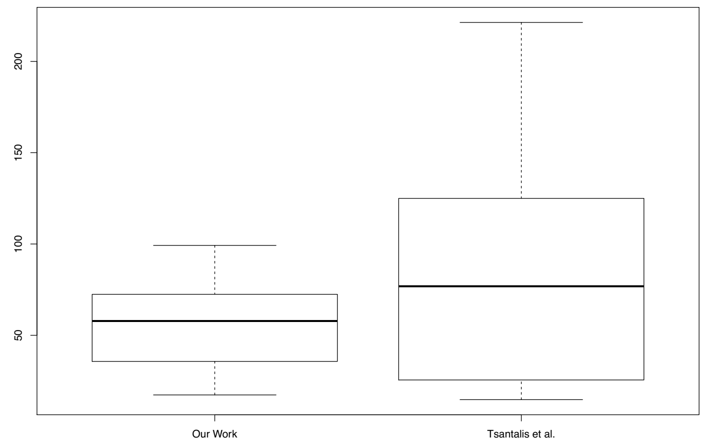
The Wilcoxon test p-values for Clone Type I, Clone Type II, and Clone Type III are $7.42 * 10^{-12}$, $1.48 * 10^{-8}$, and 0.011, respectively. The p-values indicate that there is a statistically significant difference in the performance of our work in comparison to Tsantalis et al. Further more, we compute Cliff's delta to find the effect size of the differences, and they are: small for Clone Type I, negligible for Clone Type II, and small for Clone Type III.



(a) Clone Type I



(b) Clone Type II



(c) Clone Type III

Figure 5.5: Execution time distribution for each clone type in (ms)

5.2.3 Clustering Evaluation

The goal of clustering step (Section 4.4) is to improve the refactorability of the original group of clones by grouping the clone fragments into smaller clusters that share a common structure and have less differences. To evaluate this step, we examined the clusters resulting from the two clustering steps based on **Common Structure** (Section 4.4.1) and **Differences** (Section 4.4.2). We found 45 and 48 refactorable clusters that contain more than two clone instances, respectively.

We performed a further examination on the refactorable clusters resulting from both clustering methods, by comparing if the clustering based on **Differences** affected the clusters resulting from the clustering based on **Common Structure**. The outcome of the comparison can be seen in Table 5.6.

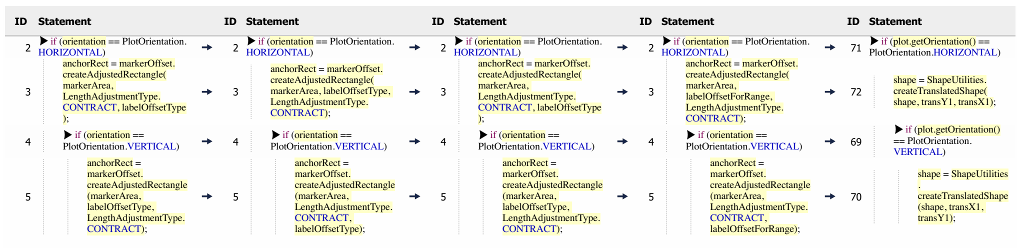
Table 5.6: The effect of clustering based on **Differences** on the initial clusters obtained by clustering based on **Common Structure**

Change	**#Cases**
No changes to the clusters resulting from clustering based on **Common Structure**	25
Removing clone fragments from the clusters resulting from clustering based on **Common Structure** increased the number of refactorable clusters	10
The clusters resulting from clustering based on **Differences** were more and/or smaller from the clusters resulting from clustering based on **Common Structure**	19
Removing a clone fragment from the clusters resulting from clustering based on **Common Structure** increased the number of mapped statements	1

Table 5.6 shows that in 25 cases the clustering based on **Differences** did not affect the initial clusters obtained by clustering based on **Common Structure**. On the other hand, in 30 cases there were changes, which we will discuss in detail next. Before we explain these changes, we will refer to the clusters obtained by clustering

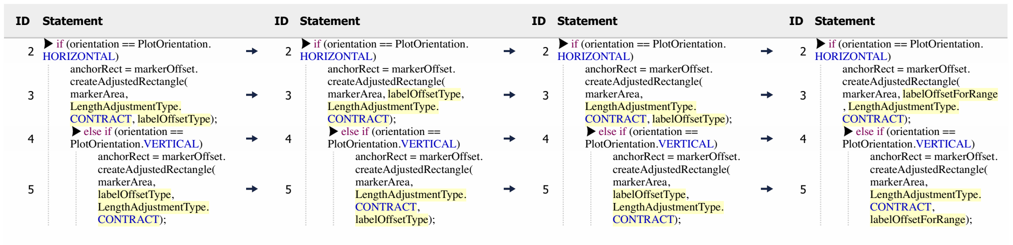
based on **Common Structure** as **original clusters**, and to the clusters obtained by clustering based on **Differences** as **new clusters**.

- There were 10 non-refactorable original clusters, which became refactorable by discarding a single clone fragment or by splitting them into smaller clusters. For example, removing a clone fragment from the non-refactorable original cluster (Figure 5.6a) made it refactorable (Figure 5.6b). The clone fragment that was preventing the refactoring is the rightmost one in Figure 5.6a, which returns an object of type `Shape`, while the rest of the fragments return an object of type `Rectangle2D` and this difference causes a violation of Precondition 6 (Section 2.8).



ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement
2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	71	▶ if (plot.getOrientation() == PlotOrientation.HORIZONTAL)
3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetForRange, LengthAdjustmentType. CONTRACT);	→	72	shape = ShapeUtilities. createTranslatedShape(shape, transY1, transX1);
4	▶ if (orientation == PlotOrientation.VERTICAL)	→	4	▶ if (orientation == PlotOrientation.VERTICAL)	→	4	▶ if (orientation == PlotOrientation.VERTICAL)	→	4	▶ if (orientation == PlotOrientation.VERTICAL)	→	69	▶ if (plot.getOrientation() == PlotOrientation. VERTICAL)
5	anchorRect = markerOffset. createAdjustedRectangle (markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	5	anchorRect = markerOffset. createAdjustedRectangle (markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	5	anchorRect = markerOffset. createAdjustedRectangle (markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	5	anchorRect = markerOffset. createAdjustedRectangle (markerArea, LengthAdjustmentType. CONTRACT, labelOffsetForRange);	→	70	shape = ShapeUtilities . createTranslatedShape (shape, transX1, transY1);

(a) A non-refactorable cluster created from clustering based on Common Structure



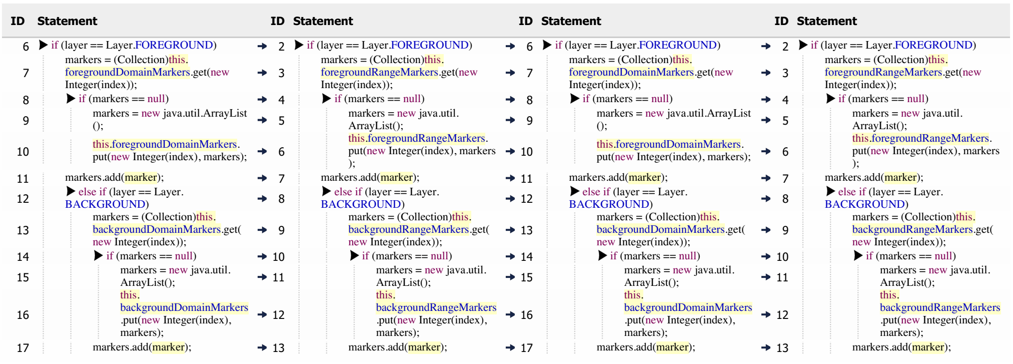
ID	Statement		ID	Statement		ID	Statement		ID	Statement
2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)	→	2	▶ if (orientation == PlotOrientation. HORIZONTAL)
3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	3	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetForRange , LengthAdjustmentType. CONTRACT);
4	▶ else if (orientation == PlotOrientation.VERTICAL)	→	4	▶ else if (orientation == PlotOrientation.VERTICAL)	→	4	▶ else if (orientation == PlotOrientation.VERTICAL)	→	4	▶ else if (orientation == PlotOrientation.VERTICAL)
5	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	5	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetType);	→	5	anchorRect = markerOffset. createAdjustedRectangle(markerArea, labelOffsetType, LengthAdjustmentType. CONTRACT);	→	5	anchorRect = markerOffset. createAdjustedRectangle(markerArea, LengthAdjustmentType. CONTRACT, labelOffsetForRange);

(b) Cluster becomes refactorable after one clone fragment is discarded

Figure 5.6: Change in a cluster after performing clustering based on Differences.

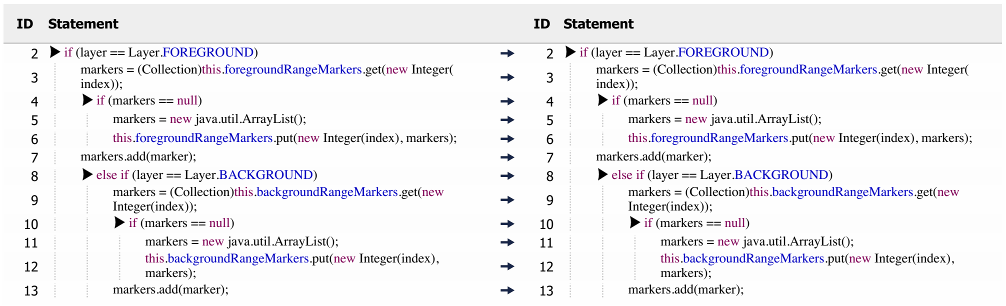
- There were 19 refactorable original clusters, which were further divided into smaller clusters, after applying clustering based on Differences. These changes affected the clone type of the new clusters, and the number of differences appearing in the new clusters.

(1) In 8 cases, further dividing the original clusters improved the quality of the new clusters in terms of clone types. For instance, the original cluster was a Type II clone, and was splitted into two new clusters of Type I clones. An example from our case study is shown in Figure 5.7a, where the original cluster is a Type II clone, but after performing clustering based on differences the new clusters are Type I clones (Figure 5.7b) and Type II clones with less differences (Figure 5.7c), respectively.



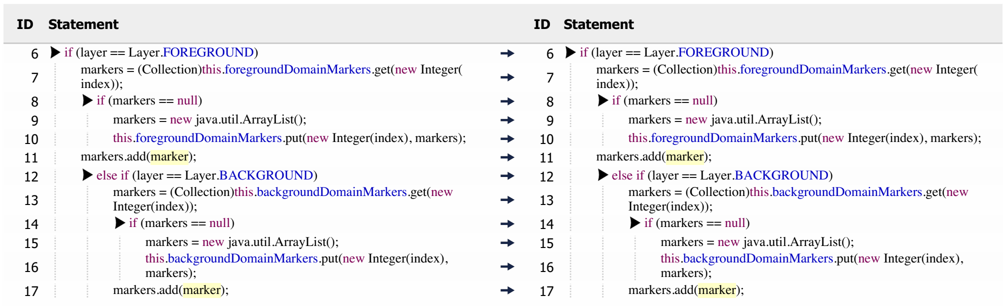
ID	Statement	ID	Statement	ID	Statement	ID	Statement
6	▶ if (layer == Layer.FOREGROUND)	➜ 2	▶ if (layer == Layer.FOREGROUND)	➜ 6	▶ if (layer == Layer.FOREGROUND)	➜ 2	▶ if (layer == Layer.FOREGROUND)
7	markers = (Collection)this.foregroundDomainMarkers.get(new Integer(index));	➜ 3	markers = (Collection)this.foregroundRangeMarkers.get(new Integer(index));	➜ 7	markers = (Collection)this.foregroundDomainMarkers.get(new Integer(index));	➜ 3	markers = (Collection)this.foregroundRangeMarkers.get(new Integer(index));
8	▶ if (markers == null)	➜ 4	▶ if (markers == null)	➜ 8	▶ if (markers == null)	➜ 4	▶ if (markers == null)
9	markers = new java.util.ArrayList();	➜ 5	markers = new java.util.ArrayList();	➜ 9	markers = new java.util.ArrayList();	➜ 5	markers = new java.util.ArrayList();
10	this.foregroundDomainMarkers.put(new Integer(index), markers);	➜ 6	this.foregroundRangeMarkers.put(new Integer(index), markers);	➜ 10	this.foregroundDomainMarkers.put(new Integer(index), markers);	➜ 6	this.foregroundRangeMarkers.put(new Integer(index), markers);
11	markers.add(marker);	➜ 7	markers.add(marker);	➜ 11	markers.add(marker);	➜ 7	markers.add(marker);
12	▶ else if (layer == Layer.BACKGROUND)	➜ 8	▶ else if (layer == Layer.BACKGROUND)	➜ 12	▶ else if (layer == Layer.BACKGROUND)	➜ 8	▶ else if (layer == Layer.BACKGROUND)
13	markers = (Collection)this.backgroundDomainMarkers.get(new Integer(index));	➜ 9	markers = (Collection)this.backgroundRangeMarkers.get(new Integer(index));	➜ 13	markers = (Collection)this.backgroundDomainMarkers.get(new Integer(index));	➜ 9	markers = (Collection)this.backgroundRangeMarkers.get(new Integer(index));
14	▶ if (markers == null)	➜ 10	▶ if (markers == null)	➜ 14	▶ if (markers == null)	➜ 10	▶ if (markers == null)
15	markers = new java.util.ArrayList();	➜ 11	markers = new java.util.ArrayList();	➜ 15	markers = new java.util.ArrayList();	➜ 11	markers = new java.util.ArrayList();
16	this.backgroundDomainMarkers.put(new Integer(index), markers);	➜ 12	this.backgroundRangeMarkers.put(new Integer(index), markers);	➜ 16	this.backgroundDomainMarkers.put(new Integer(index), markers);	➜ 12	this.backgroundRangeMarkers.put(new Integer(index), markers);
17	markers.add(marker);	➜ 13	markers.add(marker);	➜ 17	markers.add(marker);	➜ 13	markers.add(marker);

(a) Original cluster is a Type II clone



ID	Statement	ID	Statement
2	▶ if (layer == Layer.FOREGROUND)	➜ 2	▶ if (layer == Layer.FOREGROUND)
3	markers = (Collection)this.foregroundRangeMarkers.get(new Integer(index));	➜ 3	markers = (Collection)this.foregroundRangeMarkers.get(new Integer(index));
4	▶ if (markers == null)	➜ 4	▶ if (markers == null)
5	markers = new java.util.ArrayList();	➜ 5	markers = new java.util.ArrayList();
6	this.foregroundRangeMarkers.put(new Integer(index), markers);	➜ 6	this.foregroundRangeMarkers.put(new Integer(index), markers);
7	markers.add(marker);	➜ 7	markers.add(marker);
8	▶ else if (layer == Layer.BACKGROUND)	➜ 8	▶ else if (layer == Layer.BACKGROUND)
9	markers = (Collection)this.backgroundRangeMarkers.get(new Integer(index));	➜ 9	markers = (Collection)this.backgroundRangeMarkers.get(new Integer(index));
10	▶ if (markers == null)	➜ 10	▶ if (markers == null)
11	markers = new java.util.ArrayList();	➜ 11	markers = new java.util.ArrayList();
12	this.backgroundRangeMarkers.put(new Integer(index), markers);	➜ 12	this.backgroundRangeMarkers.put(new Integer(index), markers);
13	markers.add(marker);	➜ 13	markers.add(marker);

(b) First new cluster becomes Type I clone

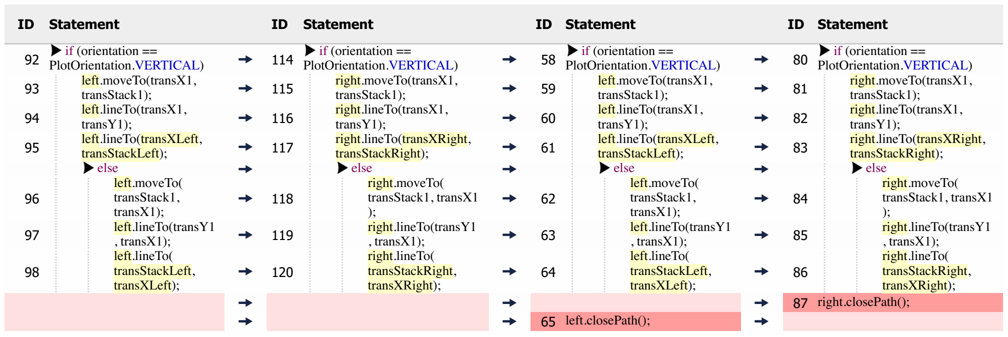


ID	Statement	ID	Statement
6	▶ if (layer == Layer.FOREGROUND)	➜ 6	▶ if (layer == Layer.FOREGROUND)
7	markers = (Collection)this.foregroundDomainMarkers.get(new Integer(index));	➜ 7	markers = (Collection)this.foregroundDomainMarkers.get(new Integer(index));
8	▶ if (markers == null)	➜ 8	▶ if (markers == null)
9	markers = new java.util.ArrayList();	➜ 9	markers = new java.util.ArrayList();
10	this.foregroundDomainMarkers.put(new Integer(index), markers);	➜ 10	this.foregroundDomainMarkers.put(new Integer(index), markers);
11	markers.add(marker);	➜ 11	markers.add(marker);
12	▶ else if (layer == Layer.BACKGROUND)	➜ 12	▶ else if (layer == Layer.BACKGROUND)
13	markers = (Collection)this.backgroundDomainMarkers.get(new Integer(index));	➜ 13	markers = (Collection)this.backgroundDomainMarkers.get(new Integer(index));
14	▶ if (markers == null)	➜ 14	▶ if (markers == null)
15	markers = new java.util.ArrayList();	➜ 15	markers = new java.util.ArrayList();
16	this.backgroundDomainMarkers.put(new Integer(index), markers);	➜ 16	this.backgroundDomainMarkers.put(new Integer(index), markers);
17	markers.add(marker);	➜ 17	markers.add(marker);

(c) Second new cluster remains Type II clone, but has less differences

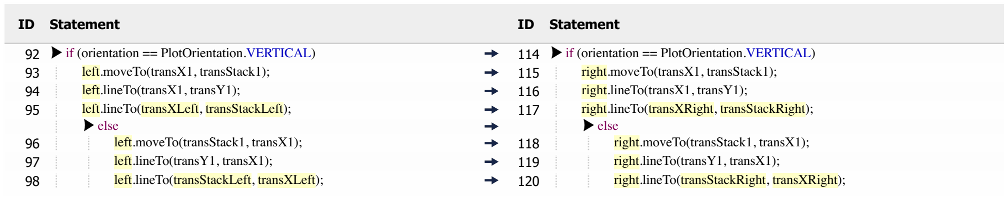
Figure 5.7: The effect on clone types after performing clustering based on Differences.

In another example, the original cluster is a Type III clone (Figure 5.8a), but after splitting it, we obtained two Type II clone clusters (Figures 5.8b, 5.8c), each one of the them containing two clone fragments. Moreover, the gapped statements in the original cluster are mapped in the new cluster (Figure 5.8c).



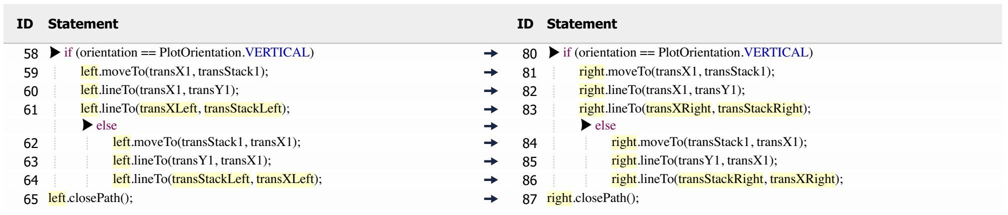
ID	Statement		ID	Statement		ID	Statement		ID	Statement
92	▶ if (orientation == PlotOrientation.VERTICAL)	→	114	▶ if (orientation == PlotOrientation.VERTICAL)	→	58	▶ if (orientation == PlotOrientation.VERTICAL)	→	80	▶ if (orientation == PlotOrientation.VERTICAL)
93	left.moveTo(transX1, transStack1);	→	115	right.moveTo(transX1, transStack1);	→	59	left.moveTo(transX1, transStack1);	→	81	right.moveTo(transX1, transStack1);
94	left.lineTo(transX1, transY1);	→	116	right.lineTo(transX1, transY1);	→	60	left.lineTo(transX1, transY1);	→	82	right.lineTo(transX1, transY1);
95	left.lineTo(transXLeft, transStackLeft);	→	117	right.lineTo(transXRight, transStackRight);	→	61	left.lineTo(transXLeft, transStackLeft);	→	83	right.lineTo(transXRight, transStackRight);
	▶ else	→		▶ else	→		▶ else	→		▶ else
96	left.moveTo(transStack1, transX1);	→	118	right.moveTo(transStack1, transX1);	→	62	left.moveTo(transStack1, transX1);	→	84	right.moveTo(transStack1, transX1);
97	left.lineTo(transY1 , transX1);	→	119	right.lineTo(transY1 , transX1);	→	63	left.lineTo(transY1 , transX1);	→	85	right.lineTo(transY1 , transX1);
98	left.lineTo(transStackLeft, transXLeft);	→	120	right.lineTo(transStackRight, transXRight);	→	64	left.lineTo(transStackLeft, transXLeft);	→	86	right.lineTo(transStackRight, transXRight);
		→			→			→	87	right.closePath();
		→			→	65	left.closePath();	→		

(a) Original cluster is a Type III clone



ID	Statement		ID	Statement
92	▶ if (orientation == PlotOrientation.VERTICAL)	→	114	▶ if (orientation == PlotOrientation.VERTICAL)
93	left.moveTo(transX1, transStack1);	→	115	right.moveTo(transX1, transStack1);
94	left.lineTo(transX1, transY1);	→	116	right.lineTo(transX1, transY1);
95	left.lineTo(transXLeft, transStackLeft);	→	117	right.lineTo(transXRight, transStackRight);
	▶ else	→		▶ else
96	left.moveTo(transStack1, transX1);	→	118	right.moveTo(transStack1, transX1);
97	left.lineTo(transY1, transX1);	→	119	right.lineTo(transY1, transX1);
98	left.lineTo(transStackLeft, transXLeft);	→	120	right.lineTo(transStackRight, transXRight);

(b) First new cluster becomes Type II clone

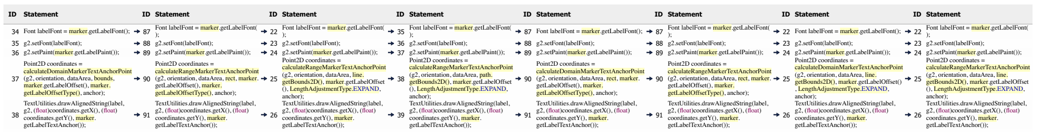


ID	Statement		ID	Statement
58	▶ if (orientation == PlotOrientation.VERTICAL)	→	80	▶ if (orientation == PlotOrientation.VERTICAL)
59	left.moveTo(transX1, transStack1);	→	81	right.moveTo(transX1, transStack1);
60	left.lineTo(transX1, transY1);	→	82	right.lineTo(transX1, transY1);
61	left.lineTo(transXLeft, transStackLeft);	→	83	right.lineTo(transXRight, transStackRight);
	▶ else	→		▶ else
62	left.moveTo(transStack1, transX1);	→	84	right.moveTo(transStack1, transX1);
63	left.lineTo(transY1, transX1);	→	85	right.lineTo(transY1, transX1);
64	left.lineTo(transStackLeft, transXLeft);	→	86	right.lineTo(transStackRight, transXRight);
65	left.closePath();	→	87	right.closePath();

(c) Second new cluster becomes Type II clone

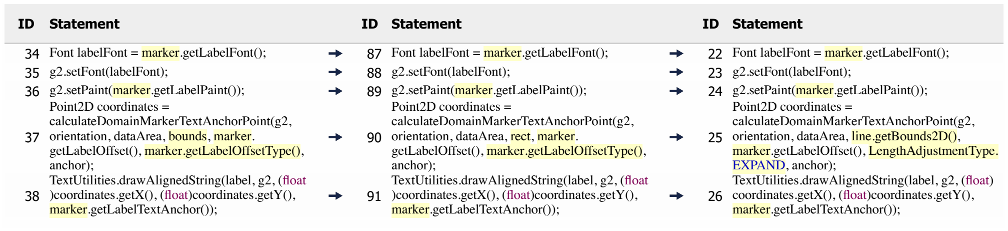
Figure 5.8: The effect on clone types after performing clustering based on Differences.

(2) In 6 cases, the original clusters were further divided into smaller clusters; however, this did not affect the clone type, and refactorability of the new clusters, but only reduced the number of differences in the new clusters. For instance, the original cluster in Figure 5.9a has four differences, namely different subclass types, renamed variables (e.g., `line` and `path`), different method invocations (e.g., `calculateRangeMarkerTextAnchorPoint()` and `calculateDomainMarkerTextAnchorPoint()` have different method names, but the same return type), and different expressions having the same data type (e.g., `bounds` and `line.getBounds2D()`). After splitting the original cluster into two new clusters (Figures 5.9b and 5.9c) most of the differences remain except for the differences in the method invocations.



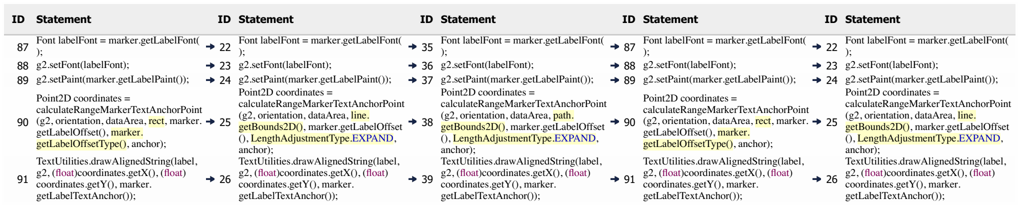
ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement
34	Font labelFont = marker.getLabelFont();	→	87	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();	→	35	Font labelFont = marker.getLabelFont();	→	87	Font labelFont = marker.getLabelFont();	→	87	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();
35	g2.setFont(labelFont);	→	88	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);	→	36	g2.setFont(labelFont);	→	88	g2.setFont(labelFont);	→	88	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);
36	g2.setPaint(marker.getLabelPaint());	→	89	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());	→	37	g2.setPaint(marker.getLabelPaint());	→	89	g2.setPaint(marker.getLabelPaint());	→	89	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());
37	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, bounds, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	90	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	25	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);	→	38	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, path.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);	→	90	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	90	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	25	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);	→	25	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);
38	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	39	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());

(a) Original cluster



ID	Statement		ID	Statement		ID	Statement
34	Font labelFont = marker.getLabelFont();	→	87	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();
35	g2.setFont(labelFont);	→	88	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);
36	g2.setPaint(marker.getLabelPaint());	→	89	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());
37	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, bounds, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	90	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	25	Point2D coordinates = calculateDomainMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);
38	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());

(b) First new cluster



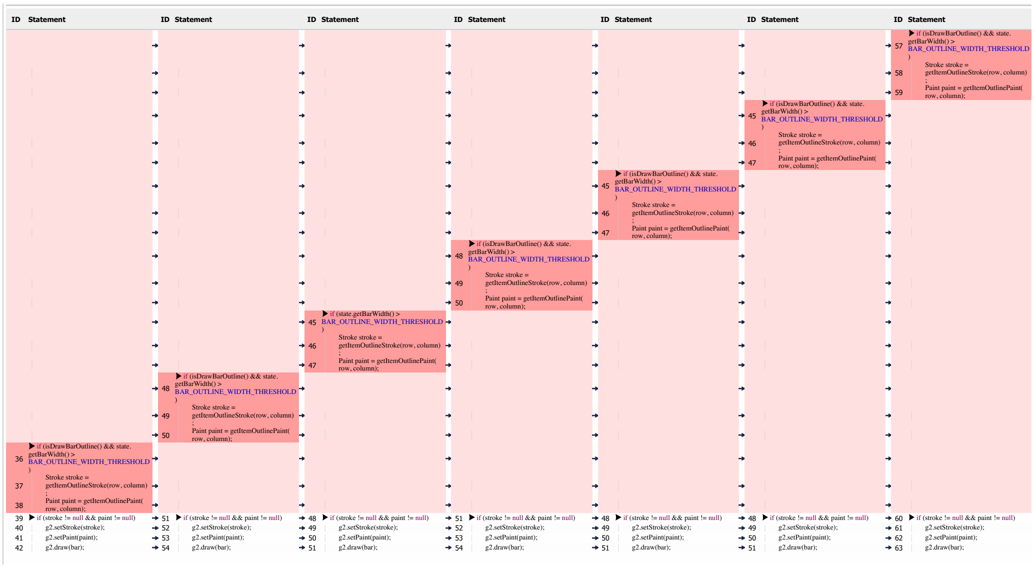
ID	Statement		ID	Statement		ID	Statement		ID	Statement		ID	Statement
87	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();	→	35	Font labelFont = marker.getLabelFont();	→	87	Font labelFont = marker.getLabelFont();	→	22	Font labelFont = marker.getLabelFont();
88	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);	→	36	g2.setFont(labelFont);	→	88	g2.setFont(labelFont);	→	23	g2.setFont(labelFont);
89	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());	→	37	g2.setPaint(marker.getLabelPaint());	→	89	g2.setPaint(marker.getLabelPaint());	→	24	g2.setPaint(marker.getLabelPaint());
90	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	25	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);	→	38	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, path.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);	→	90	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, rect, marker.getLabelOffset(), marker.getLabelOffsetType(), anchor);	→	25	Point2D coordinates = calculateRangeMarkerTextAnchorPoint(g2, orientation, dataArea, line.getBounds2D(), marker.getLabelOffset(), LengthAdjustmentType.EXPAND, anchor);
91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	39	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	91	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());	→	26	TextUtilities.drawAlignedString(label, g2, (float)coordinates.getX(), (float)coordinates.getY(), marker.getLabelTextAnchor());

(c) Second new cluster

Figure 5.9: The number of differences is reduced after performing clustering based on Differences, but the clone type remains the same.

(3) In 5 cases, we have a combination of the improvements discussed in (1) and (2) simply by discarding a single clone fragment from the original cluster. These cases mostly have an odd number of clone fragments in the original cluster.

- The last row in Table 5.6 represents a case where the number of mapped statements increased by removing a single clone fragment. In the original cluster, 3 statements are considered as gaps (Figure 5.10a), and the fragment causing these gapped statements is the third one from the left. The problem is caused from the conditional expression inside the `if` statement (statement 45, third clone fragment), where it has a different AST structure in comparison to the rest of the fragments. For this reason, the `if` statement including the statements nested inside it, are considered as a gap. By removing this fragment from the cluster, we can see that all statements are now mapped (Figure 5.10b).



```
ID  Statement
36  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
37      Stroke stroke = getItemOutlineStroke(row, column);
38      Paint paint = getItemOutlinePaint(row, column);
39  if (stroke != null && paint != null)
40      g2.setStroke(stroke);
41      g2.setPaint(paint);
42      g2.draw(bar);
```

```
ID  Statement
48  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
49      Stroke stroke = getItemOutlineStroke(row, column);
50      Paint paint = getItemOutlinePaint(row, column);
51  if (stroke != null && paint != null)
52      g2.setStroke(stroke);
53      g2.setPaint(paint);
54      g2.draw(bar);
```

```
ID  Statement
45  if (state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
46      Stroke stroke = getItemOutlineStroke(row, column);
47      Paint paint = getItemOutlinePaint(row, column);
48  if (stroke != null && paint != null)
49      g2.setStroke(stroke);
50      g2.setPaint(paint);
51      g2.draw(bar);
```

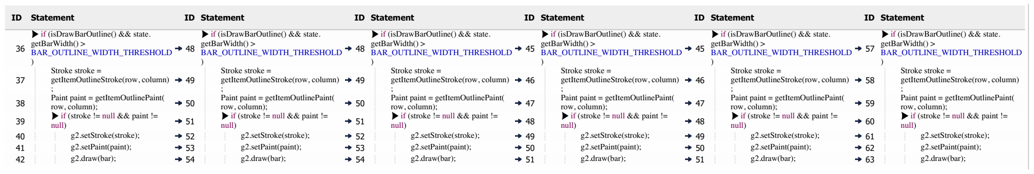
```
ID  Statement
48  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
49      Stroke stroke = getItemOutlineStroke(row, column);
50      Paint paint = getItemOutlinePaint(row, column);
51  if (stroke != null && paint != null)
52      g2.setStroke(stroke);
53      g2.setPaint(paint);
54      g2.draw(bar);
```

```
ID  Statement
45  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
46      Stroke stroke = getItemOutlineStroke(row, column);
47      Paint paint = getItemOutlinePaint(row, column);
48  if (stroke != null && paint != null)
49      g2.setStroke(stroke);
50      g2.setPaint(paint);
51      g2.draw(bar);
```

```
ID  Statement
45  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
46      Stroke stroke = getItemOutlineStroke(row, column);
47      Paint paint = getItemOutlinePaint(row, column);
48  if (stroke != null && paint != null)
49      g2.setStroke(stroke);
50      g2.setPaint(paint);
51      g2.draw(bar);
```

```
ID  Statement
57  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
58      Stroke stroke = getItemOutlineStroke(row, column);
59      Paint paint = getItemOutlinePaint(row, column);
60  if (stroke != null && paint != null)
61      g2.setStroke(stroke);
62      g2.setPaint(paint);
63      g2.draw(bar);
```

(a) Original cluster with gapped statements



```
ID  Statement
36  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
37      Stroke stroke = getItemOutlineStroke(row, column);
38      Paint paint = getItemOutlinePaint(row, column);
39      if (stroke != null && paint != null)
40          g2.setStroke(stroke);
41          g2.setPaint(paint);
42          g2.draw(bar);
```

```
ID  Statement
48  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
49      Stroke stroke = getItemOutlineStroke(row, column);
50      Paint paint = getItemOutlinePaint(row, column);
51      if (stroke != null && paint != null)
52          g2.setStroke(stroke);
53          g2.setPaint(paint);
54          g2.draw(bar);
```

```
ID  Statement
48  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
49      Stroke stroke = getItemOutlineStroke(row, column);
50      Paint paint = getItemOutlinePaint(row, column);
51      if (stroke != null && paint != null)
52          g2.setStroke(stroke);
53          g2.setPaint(paint);
54          g2.draw(bar);
```

```
ID  Statement
45  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
46      Stroke stroke = getItemOutlineStroke(row, column);
47      Paint paint = getItemOutlinePaint(row, column);
48      if (stroke != null && paint != null)
49          g2.setStroke(stroke);
50          g2.setPaint(paint);
51          g2.draw(bar);
```

```
ID  Statement
45  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
46      Stroke stroke = getItemOutlineStroke(row, column);
47      Paint paint = getItemOutlinePaint(row, column);
48      if (stroke != null && paint != null)
49          g2.setStroke(stroke);
50          g2.setPaint(paint);
51          g2.draw(bar);
```

```
ID  Statement
57  if (isDrawBarOutline() && state.getBarWidth() > BAR_OUTLINE_WIDTH_THRESHOLD)
58      Stroke stroke = getItemOutlineStroke(row, column);
59      Paint paint = getItemOutlinePaint(row, column);
60      if (stroke != null && paint != null)
61          g2.setStroke(stroke);
62          g2.setPaint(paint);
63          g2.draw(bar);
```

(b) New cluster after discarding a single clone fragment

Figure 5.10: The effect on the number of mapped statements after performing clustering based on Differences.

In conclusion, the clustering based on Differences increased the number of refactorable clusters by 11. However, it is difficult to generalize the findings in terms of the improvements obtained by clustering based on Differences (i.e., more suitable clone types for refactoring, increase in the number of mapped statements, decrease in the number of differences), since the number of examined cases is very small.

5.2.4 Clone Group level Evaluation

In the previous sections, we evaluated the accuracy and performance of our work in comparison to Tsantalis et al. at clone pair level. In this section, we will discuss the refactorability and the total execution time for the clone groups containing 3 or more clone instances.

5.2.4.1 Group Refactorability

The number of clusters that contain three fragments or more in this evaluation can be seen in Table 5.7. The second column shows the number of clusters categorized based on their size. The third column shows the number of refactorable clusters. Figure 5.11 is an example of a refactorable cluster that contains 3 Type II clone instances. The clone type for a group is determined based on the clone types of all pairs:

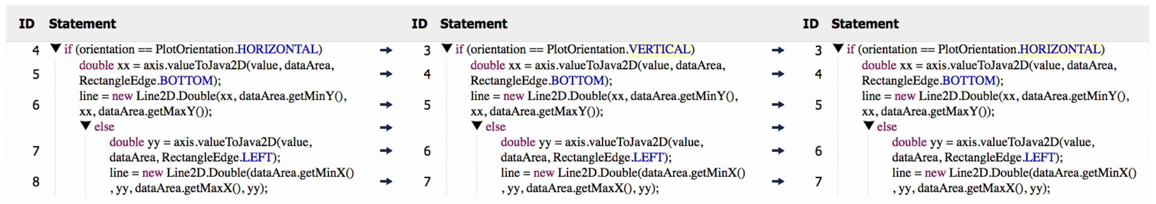
Clone Type I All clones pairs are Type I clones, i.e., there is no differences between the clone fragments in the cluster.

Clone Type II At least one clone pair is Type II clone, i.e., the cluster may contain Type I clone pairs, but there is at least one Type II clone pair.

Clone Type III At least one clone pair is Type III clone, i.e., the cluster may contain Type I and Type II clone pairs, but there is at least one Type III clone pair.

Table 5.7: Cluster refactorability results

Cluster Size	#Clusters	#Refactorable Clusters
3	44	22
4	37	14
5	8	5
6	2	2
7	4	3
8	1	1
9	1	1
>9	1	0



ID	Statement		ID	Statement		ID	Statement
4	▼ if (orientation == PlotOrientation.HORIZONTAL)	→	3	▼ if (orientation == PlotOrientation.VERTICAL)	→	3	▼ if (orientation == PlotOrientation.HORIZONTAL)
5	double xx = axis.valueToJava2D(value, dataArea, RectangleEdge.BOTTOM);	→	4	double xx = axis.valueToJava2D(value, dataArea, RectangleEdge.BOTTOM);	→	4	double xx = axis.valueToJava2D(value, dataArea, RectangleEdge.BOTTOM);
6	line = new Line2D.Double(xx, dataArea.getMinY(), xx, dataArea.getMaxY());	→	5	line = new Line2D.Double(xx, dataArea.getMinY(), xx, dataArea.getMaxY());	→	5	line = new Line2D.Double(xx, dataArea.getMinY(), xx, dataArea.getMaxY());
	▼ else	→		▼ else	→		▼ else
7	double yy = axis.valueToJava2D(value, dataArea, RectangleEdge.LEFT);	→	6	double yy = axis.valueToJava2D(value, dataArea, RectangleEdge.LEFT);	→	6	double yy = axis.valueToJava2D(value, dataArea, RectangleEdge.LEFT);
8	line = new Line2D.Double(dataArea.getMinX() , yy, dataArea.getMaxX(), yy);	→	7	line = new Line2D.Double(dataArea.getMinX() , yy, dataArea.getMaxX(), yy);	→	7	line = new Line2D.Double(dataArea.getMinX() , yy, dataArea.getMaxX(), yy);

Figure 5.11: Example of a refactorable cluster with three clone instances.

5.2.4.2 Group Execution Time

The total execution time for our approach shown in **Figure 5.12** is computed starting from extracting the required information (Section 4.3) from each clone in the group, and ending to assessing the refactorability for all resulting clusters (Section 4.7). Also, for the sake of a fair comparison, we performed all pairwise combinations when computing the mapping of statements between the clone fragments of the group. For the approach by Tsantalis et al. the time is computed as the sum of the execution time for all pairwise combinations $(Clone_1, Clone_2), \ldots, (Clone_1, Clone_n), \ldots,$ $(Clone_{n-1}, Clone_n)$, where n is the group size.

Table 5.8: Groups based on Group Size

Group Size	Number of Groups
2	398
3	73
4	74
5	15
6	8
7	5
8	11
9	3
10	3
11	1
12	2

Figures 5.12 and 5.13 show that there is a correlation between the median execution time and the number of clone fragments in a group. We can see that our approach has a better overall performance as the number of fragments increases, particularly for groups containing 6 or more fragments. However, this comparison might be inaccurate, because our approach in practice applies the statement mapping process for much less combinations of clone pairs than Tsantalis et al., due to the decomposition of the initial clone group into smaller clusters in the clustering step. Despite of the inaccuracy in the comparison, we examine if there are any statistically significant differences for both Group-1 (groups containing 2-5 clone instances) and Group-2 (groups containing 6-12 clone instances) in Figures 5.12 and 5.13, and the p-values are $2.2 * 10^{-16}$ and $1.53 * 10^{-06}$, respectively. The p-value for Group-1 indicate that there is a significant difference in-favor of Tsantalis et al., while the p-value for Group-2 indicates that there is a significant difference in-favor of our work. As for the effect size, for Group-1 is medium, while for Group-2 is large.

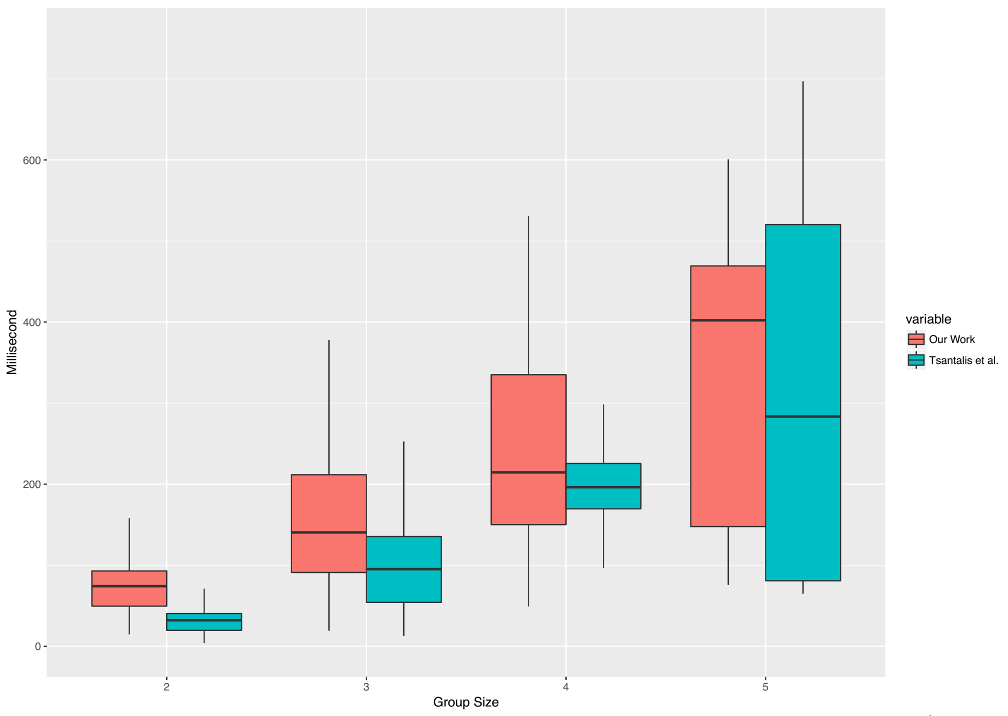


Figure 5.12: Execution time for groups containing between 2-5 clones (Group-1)

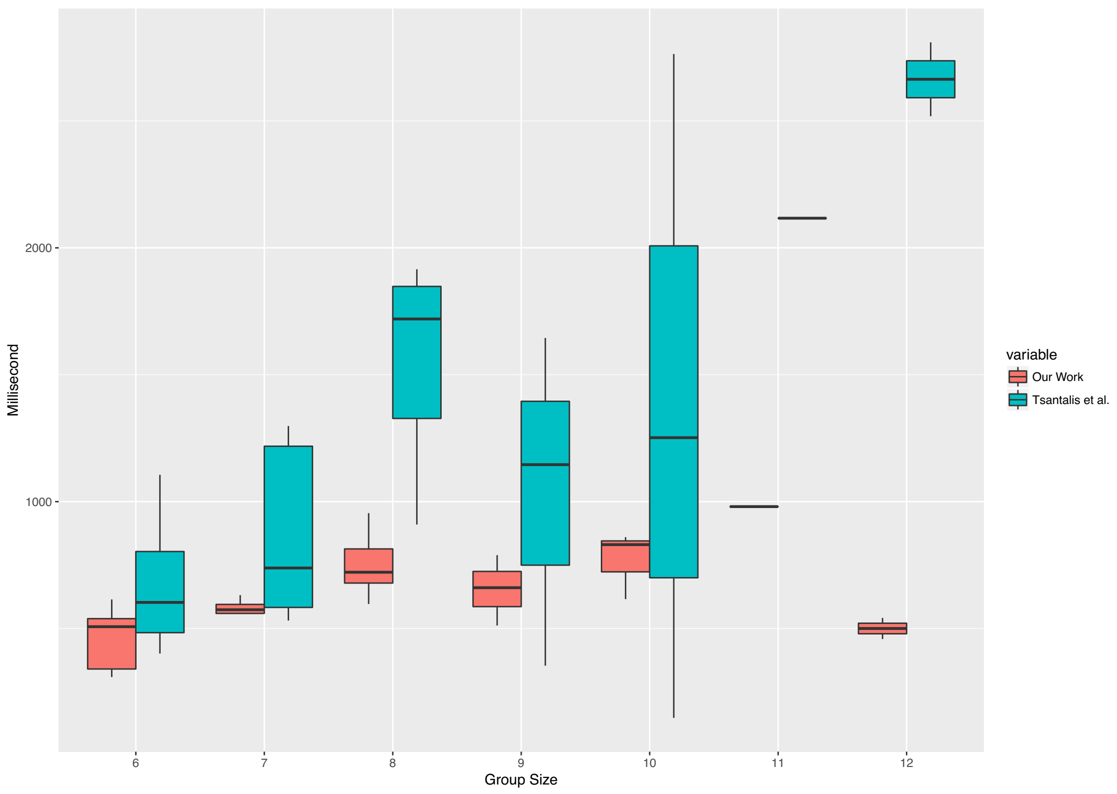


Figure 5.13: Execution time for groups containing between 6-12 clones (Group-2)

Chapter 6

Empirical Study

The goal of this chapter is to present an extensive evaluation of our work through a large-scale empirical study. This chapter is divided into two sections. In the first section, we present the examined projects, the used clone detection tools, and setup of our experiment. In the second section, we present the results of the empirical study by comparing our approach with [TMK15].

6.1 Experiment Setup

We executed our and Tsantalis et al. implementation on the clones detected by four clone detection tools in nine open-source projects.

6.1.1 Projects

The nine projects examined in our study are from different domains, and have different age, and code size. Table 6.1 shows the projects used in our experiment, along with their version.

Figure 6.1: Examined Projects [Tsab]

Project	Domain	Age†	Size*
Apache Ant 1.7.0	Java application build tool	6 1/2	67
Columba 1.4	email client	1 1/2	75
EMF 2.4.1	modeling framework	5 1/2	118
JMeter 2.3.2	server performance testing tool	7 1/4	54
JEdit 4.2	text editor	5	51
JFreeChart 1.0.10	chart library	7 1/2	76
JRuby 1.4.0	programming language	3 1/2	101
Hibernate 3.3.2	Java persistence framework	7 1/2	209
SQuirreL SQL 3.0.3	universal SQL client	8	141

† years of development from the initial release to the examined release of the project
* in KLoC

6.1.2 Clone Detection Tools

We used four clone detection tools in the experiment, and these tools can be found at [Tsab] along with the clone detection tool configuration parameters, such as the minimum number of tokens in the detected clones.

6.1.2.1 CCFinder

A well-known token-based clone detection technique proposed by Kamiya et al. [KKI02]. According to a study done by Bellon et al. [Bel+07], CCFinder had the highest precision and recall among eight clone detection techniques. Also the study compared the eight techniques in terms of execution time and memory requirements, and CCFinder was one of the most efficient techniques.

6.1.2.2 Deckard

A tree-based clone detection technique proposed by Jiang et al. [Jia+07]. Deckard, transforms the abstract syntax trees of the code into a feature vector, and then uses Locality Sensitive Hashing to cluster the clone fragments. The fragments within the same cluster are considered similar. Jiang et al. [Jia+07] compared Deckard to CloneDR (a tree-based technique), and CP-Miner (a token-based technique) and concluded that Deckard is faster than CloneDR when the similarity threshold is 0.999 and has similar performance to CP-Miner.

6.1.2.3 CloneDR

A tree-based clone detection technique proposed by Baxter et al. [Bax+98]. The abstract syntax tree is partitioned into subtrees based on a hash function, and then the subtrees are compared though tree matching. In a study done by Bellon et al. [Bel+07], CloneDR had the highest precision (100%) and at the same time had the lowest recall (9%).

6.1.2.4 NiCad

A text-based clone detection technique proposed by Roy et al. [RC08a]. NiCad applies code normalization, source code transformation, and code filtering by incorporating a lightweight tree-based structural analysis. Roy et al. [RC09] evaluated NiCad through an automated framework that creates randomly mutated clones from the original code and injects them into random places throughout the code base. The results show that NiCad (Full version, where the former capabilities are all enabled) had 100% recall and over 96% precision for Type-2 and Type-3 injected clones. We configured NiCad with two different options, namely *consistent* renaming, where the same identifiers are replaced with a single pseudo-variable (X_{index}) using an order-sensitive indexing scheme, and *blind* renaming, where all identifiers are replaced with a single pseudo-variable (X).

6.1.3 Experiment Machine

The specifications of the machine we used to run our experiments can be seen in Table 6.1.

Table 6.1: Machine Specifications

Part	**Specification**
Processor	Quad-core Intel Core i5-2400, 3.10GHz
Hard disk	Samsung EVO SSD
RAM	16GB DDR3
OS	Windows 10 Professional

6.2 Results and Discussion

This section is a replication of the Qualitative study we did on a wider range of projects and clone detection tools. Table 6.2 gives the percentage of groups containing more than two clone instances as detected by each of the four clone detection tools in the nine projects. The percentage of clone groups containing more than two clone instances ranges from 12% to 44%, while the remaining clone groups contain a single pair of clone fragments.

Table 6.2: Percentage of groups containing more than 2 clone instances.

Project	Tool									
	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	Groups	>2	Groups	>2	Groups	>2	Groups	>2	Groups	> 2
Apache Ant	958	28%	1473	25%	767	20%	582	31%	558	37%
Columba	792	23%	1332	28%	742	27%	546	35%	511	35%
EMF	1470	34%	1892	28%	325	40%	941	38%	990	36%
Hibernate	1346	30%	2161	25%	1322	28%	734	33%	665	32%
Jakarta	760	23%	104	44%	789	25%	552	31%	541	31%
JEdit	358	18%	642	22%	400	12%	343	31%	312	26%
JFreechart	2113	43%	2507	28%	1820	39%	818	40%	823	41%
JRuby	1314	20%	1311	25%	591	36%	878	35%	793	31%
SQuirreL SQL	1627	32%	2236	36%	723	31%	827	32%	846	43%

6.2.1 Statement Mapping Comparison

This section compares our work with [TMK15] with respect to the statement mapping solutions produced by each approach. **Table 6.3** shows the total number of clone pairs being compared. In the next subsections, we will discuss the results separately for each clone type.

Table 6.3: Total Clone Pairs being compared

Project	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	2,191	3,104	860	1,689	2,545
Columba	1,008	4,192	1,042	2,765	2,700
EMF	10,115	4,216	2,365	4,366	7,317
Hibernate	1,917	3,500	1,111	13,540	3,348
JEdit	234	698	271	325	302
JFreechart	79,109	42,181	62,974	81,298	48,215
JMeter	681	537	965	1,188	938
JRuby	1,646	2,505	807	5,540	1,887
SQuirreL SQL	3,314	7,185	2,113	7,061	8,874
Total	100,215	68,118	72,508	117,772	76,126

6.2.1.1 Clone Type I

Table 6.4 shows the total number of Type I clone pairs being compared per project. Table 6.5, shows the percentage of clone pairs that have an identical statement mapping with Tsantalis et al., and as it can be observed in all cases both approaches produce an identical statement mapping solution. Table 6.6 presents the median time for our approach and Tsantalis et al. approach. It appears that the latter is faster in pairwise statement mapping for the clones reported by CCFinder, Deckard, and CloneDR and slower for the clones reported by NiCad (with Blind and Consistent renaming option).

Table 6.4: Number of Type I clones pairs per project

Project	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	290	1,888	124	286	364
Columba	310	2,134	210	182	258
EMF	332	2,291	88	312	458
Hibernate	732	1760	264	222	264
JEdit	58	332	77	64	77
JFreechart	1635	3,265	787	798	851
JMeter	175	248	152	149	160
JRuby	314	470	100	167	196
SQuirreL SQL	1,029	3,768	654	1,098	1,216
Total	4875	16156	2456	3278	3844

Table 6.5: Pairwise Statement Mapping comparison for Clone Type I

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)
Apache Ant	100	100	100	100	100	100	100	100	100	100
Columba	100	100	100	100	100	100	100	100	100	100
EMF	100	100	100	100	100	100	100	100	100	100
Hibernate	100	100	100	100	100	100	100	100	100	100
JEdit	100	100	100	100	100	100	100	100	100	100
JFreechart	100	100	100	100	100	100	100	100	100	100
JMeter	100	100	100	100	100	100	100	100	100	100
JRuby	100	100	100	100	100	100	100	100	100	100
SQuirreL SQL	100	100	100	100	100	100	100	100	100	100
Average	100	100	100	100	100	100	100	100	100	100

P: Pairs that our work and Tsantalis et al. have the same mapping
S: Statements that our work and Tsantalis et al. have the same mapping

Table 6.6: Median Execution time for Pairwise Statement Mapping for Clone Type I

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	O	T	O	T	O	T	O	T	O	T
Apache Ant	32.21	34.47	43.62	39.37	34.64	38.26	31.37	30.31	32.86	31.41
Columba	24.92	23	22.62	21.97	21.34	22.75	19.8	19.74	25.48	23.44
EMF	126.41	75.72	121.89	75.91	176.71	102.9	135.02	129.74	75.42	73.21
Hibernate	54.37	51.51	62.67	54.31	37.42	31.21	31.23	30.82	39.06	37.54
JEdit	61.18	48.74	44.48	40.35	31.54	31.85	36.01	51.13	40.61	57.74
JFreechart	55.95	51.63	28.3	26.07	54.77	53.36	50.5	50.69	50.51	50.56
JMeter	34.93	27.38	29.42	28.33	34.12	33.42	27.76	31.59	31.6	32.18
JRuby	115.48	74.96	89.22	61.36	52.17	73.28	62.52	71.02	66.51	75
SQuirreL SQL	31.76	33.63	31.21	29.85	31.68	32.98	27.2	27.72	32.25	33.88
Average	59.69	46.78	52.60	41.95	52.71	46.67	46.82	49.20	43.81	46.11

O: Our execution time in milliseconds
T: Tsantalis et al. execution time in milliseconds

6.2.1.2 Clone Type II

This clone type is the most dominant compared to Type I and Type III in terms of the number of pairs examined in this empirical study. This is an indication that developers mostly copy-paste code and perform changes to identifiers, literals, and types. Table 6.7 presents the number of clone pairs being compared, and the results of the comparison are shown in Table 6.8. As we can see from the last row in Table 6.8 (1) The lowest percentage of clone pairs with an identical statement mapping to Tsantalis et al. is in NiCad (Blind) with 85.16% and the highest is in CloneDR with 96.97%; (2) The lowest percentage of identically mapped statements with Tsantalis et al. is in CCFinder (94.84%), and the highest in CloneDR (98.39%).

Table 6.7: Number of Type II clones pairs per project

Project	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	1,891	1,210	645	1,238	2,067
Columba	686	2,044	796	2,348	2,333
EMF	9,778	1,923	2,264	3,867	5,642
Hibernate	1,129	1,736	718	6,508	1,927
JEdit	170	364	169	228	207
JFreechart	76,612	38,912	60,803	59,641	44,021
JMeter	504	288	754	845	731
JRuby	1,320	2,035	686	3,994	1,584
SQuirreL SQL	2,264	3,415	1,396	5,651	6,774
Total	94354	51927	68231	84320	65286

Table 6.8: Pairwise Statement Mapping comparison for Clone Type II

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)
Apache Ant	94.20	97.01	98.10	98.68	84.58	95.32	85.45	95.49	96.23	98.21
Columba	73.50	89.09	97.41	99.07	95.20	99.21	72.08	92.46	72.87	92.36
EMF	95.70	97.81	98.17	99.39	95.14	98.16	92.58	98.16	91.76	97.63
Hibernate	81.58	92.30	94.24	97.31	75.42	94.01	83.20	92.49	85.46	94.85
JEdit	77.06	93.87	96.70	98.46	94.61	98.33	85.09	95.85	87.92	96.66
JFreechart	96.09	98.09	99.69	98.80	96.84	98.76	93.13	97.77	96.14	97.10
JMeter	89.76	96.56	94.79	97.70	87.67	95.12	87.22	95.51	88.78	95.93
JRuby	93.20	95.70	95.48	97.05	86.01	96.41	74.74	93.10	87.97	96.68
SQuirreL SQL	86.58	93.12	98.18	99.02	93.33	96.13	92.97	95.78	93.83	97.69
Average	87.52	94.84	96.97	98.39	89.86	96.83	85.16	95.18	89.00	96.34

P: Pairs that our work and Tsantalis et al. have the same mapping
S: Statements that our work and Tsantalis et al. have the same mapping

The last table (Table 6.9) shows that Tsantalis et al. approach has a better performance in pairwise statement mapping, especially for CloneDR where the execution time is much faster by around 8ms.

Table 6.9: Median Execution time for Pairwise Statement Mapping for Clone Type II

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	O	T	O	T	O	T	O	T	O	T
Apache Ant	37.69	42.92	29.16	32.18	23.63	30.22	37.16	43.71	41.03	46.13
Columba	30.83	30.18	18.03	16.32	24.9	35.66	22.86	24.44	23.65	24.66
EMF	90.06	62.64	119.14	104.48	100.32	55.96	109.02	81.64	76.41	48.01
Hibernate	66.79	63.68	51.04	47.11	39.76	48.89	24.47	27.97	23.25	24.62
JEdit	49.82	41.9	49.68	40.52	54.17	35.81	47.48	39.62	45.56	34.48
JFreechart	24	25.55	27.05	24.2	28.78	27.64	26.52	26.86	23.98	24.39
JMeter	34.44	28.85	29.11	36.05	34.44	32.75	30.52	30.6	28.76	28.74
JRuby	98.72	120.64	112.82	68.69	56.99	57.6	92.01	124.75	88.95	97.86
SQuirreL SQL	33.79	36.55	36.87	33.38	32.28	37.38	32.64	29.44	36.88	35.41
Average	51.79	50.32	52.54	44.77	43.92	40.21	46.96	47.67	43.16	40.48

O: Our execution time in milliseconds
T: Tsantalis et al. execution time in milliseconds

6.2.1.3 Clone Type III

This clone type has the least number of pairs examined in this empirical study compared to Type I and Type II, as we can see in Table 6.10. Also the table shows that the largest number of Type III clones pairs were detected in JFreechart. We can also see that NiCad detected the largest number of Type III clone pairs among the tools, while CloneDR detected the least number of Type III clone pairs.

Contrary to the other clone types, the similarity in pairwise statement mapping to Tsantalis et al. is lower, as can be seen in Table 6.11, especially at pair level ranging between 60%-85%, but at statement level it ranges between 87%-93%. Our work has better performance in pairwise statement mapping for the clones detected by Deckard and Nicad, while Tsantalis et al. has better performance in pairwise statement mapping for the clones detected by CCFinder and CloneDR.

Table 6.10: Number of Type III clones pairs per project

Project	CCFinder	CloneDR	Deckard	NiCad Blind	NiCad Consistent
Apache Ant	10	6	91	165	114
Columba	12	14	36	235	109
EMF	5	2	13	187	1,217
Hibernate	56	4	129	6,810	1,157
JEdit	6	2	25	33	18
JFreechart	862	4	1,384	20,859	3,343
JMeter	2	1	59	194	47
JRuby	12	0	21	1379	107
SQuirreL SQL	21	2	63	312	884
Total	986	35	1,821	30,174	6,996

Table 6.11: Pairwise Statement Mapping comparison for Clone Type III

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)	P(%)	S(%)
Apache Ant	90.00	89.39	83.33	96.88	67.42	90.60	67.28	86.32	58.04	86.32
Columba	81.82	96.81	100	100	63.89	91.01	87.01	96.38	85.32	95.10
EMF	100	100	50.00	95.83	58.33	95.06	68.65	94.81	84.80	94.93
Hibernate	83.93	91.30	100	100	47.20	80.83	80.36	85.69	79.43	90.54
JEdit	100	100	100	100	72.00	93.99	84.85	97.37	77.78	96.08
JFreechart	98.37	99.00	100	100	60.71	95.68	75.15	95.89	60.28	94.44
JMeter	50.00	77.78	0.00	94.74	46.55	82.01	75.77	91.38	82.98	94.37
JRuby	91.67	96.00	0.00	0.00	61.90	84.88	78.39	90.25	82.24	93.61
SQuirreL SQL	76.19	85.56	100	100	69.84	90.67	80.45	79.40	87.67	94.48
Average	85.78	92.87	70.37	87.49	60.87	89.41	77.55	90.83	77.62	93.32

P: Pairs that our work and Tsantalis et al. have the same mapping
S: Statements that our work and Tsantalis et al. have the same mapping

Table 6.12: Median Execution time for Pairwise Statement Mapping for Clone Type III

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent	
	O	T	O	T	O	T	O	T	O	T
Apache Ant	29.89	30.69	29.1	28.43	46.35	54.1	33.65	44.92	30.37	36.62
Columba	30.04	26.66	27.5	21.2	30.89	39.35	24.67	31.96	28.26	27.14
EMF	61.53	57.95	123.3	68.02	98.82	58.77	132.9	133.89	53.11	63.73
Hibernate	54.69	50.14	144.72	101.74	53.42	78.47	21.42	22.29	12.49	14.89
JEdit	61.54	50	37.86	23.68	46.9	48.66	51.55	41.67	18.11	15.76
JFreechart	24.59	21.92	25.65	62.81	19.6	30.79	29.32	33.98	19.92	28.27
JMeter	27.56	23.66	50.11	59.48	45.3	69.88	29.06	36.36	32.36	32.73
JRuby	177.27	95.22	0	0	64.96	196.9	45.95	51.59	37.3	36.65
SQuirreL SQL	31.88	25.78	52.49	42.47	29.44	30.35	28.23	28.5	28.78	32.18
Average	55.44	42.45	54.53	45.31	48.41	67.47	44.08	47.24	28.97	32.00

O: Our execution time in milliseconds
T: Tsantalis et al. execution time in milliseconds

6.2.2 Clone Group Refactorability

Table 6.13 shows the percentage of the clone subgroups that contain three fragments or more and are refactorable. As it can be observed, CloneDR has the highest percentage of refactorable subgroups among the other tools with an average of 59.5%, while the rest of the clone detection tools have an average around 33.2%. The projects with the highest percentage of refactorable groups are (1) Columba, and Apache Ant

(> 50%); (2) Hibernate, JMeter, and SQuirreL SQL (between 40%-50%); (3) The rest of the projects have a refactorability of less than 40%.

Table 6.13: Refactorable Subgroups

Project	CCFinder		CloneDR		Deckard		NiCad Blind		NiCad Consistent		Average (Project)
	T	R	T	R	T	R	T	R	T	R	
Apache Ant	115	50.4%	195	72.3%	67	38.81%	96	35.42%	121	38.84%	51.52%
Columba	100	45%	185	55.7%	82	69.51%	114	48.25%	111	54.95%	54.22%
EMF	186	24.2%	232	59.9%	71	8.45%	185	27.57%	229	24.02%	32.78%
Hibernate	201	40.8%	220	61.8%	81	37.04%	137	24.09%	113	35.4%	42.69%
JEdit	26	26.9%	54	51.9%	16	25%	34	32.35%	24	29.17%	37.01%
JFreechart	596	18.8%	436	56%	346	24.28%	266	25.56%	293	27.65%	30.41%
JMeter	69	49.3%	21	38.1%	67	41.79%	79	37.97%	82	41.46%	42.14%
JRuby	151	29.1%	153	47.1%	96	29.17%	163	17.79%	143	23.78%	29.32%
SQuirreL SQL	205	37.1%	394	64.5%	96	45.83%	274	39.42%	292	41.1%	47.74%
Average(Tool)	34.4%		59.5%		33.3%		31.1%		34.0%		

T: Total subgroups

R: Refactorable subgroups

6.2.3 Threats to Validity

One threat to the internal validity of our empirical study comes from the configuration of the clone detection tools that were used. Each of the clone detection tools that we used implements a different technique in detecting clone instances, and uses different kinds of parameters, thus it was difficult to configure the tools with similar settings. For instance, the minimum clone size is interpreted by NiCad as minimum lines of code, by CCFinder and Deckard as minimum number of tokens, and by CloneDR as minimum number of AST nodes. Moreover, even slight changes in the configuration of the parameters, would lead to different results. For example, in our study we used two configurations for NiCad, where *Blind renaming* is the default option and replaces all variable identifiers with the same pseudo-variable (X), while *Consistent renaming* replaces consistently each variable identifier with a unique pseudo-variable (X_{index}). Both NiCad configurations yielded different results in terms of the number of detected clone pairs, statement mapping similarity, execution time, and refactorability (Blind 31.1% vs. Consistent 34%). However, to address this issue we used the default or recommended configurations of the tools, which have been used by many researchers in different empirical studies.

Another internal threat is that we did not actually refactor the clones in the refactorable subgroups, in order to assess that the refactoring does not introduce compilation errors, and all tests are passing after refactoring. However, since the refactorability assessment for subgroups is based on Tsantalis et al. [TMK15] work (we extended their code to accept more than two clone fragments), where they refactored 610 refactorable clone pairs and all of them passed the unit tests and none of them caused any compilation error, we can assume that if there are no precondition violations, then the subgroup is safe to be refactored.

The clustering step imposes another threat to internal validity. By breaking-down the original clone groups to smaller subgroups, we might create some redundant clusters or discard some refactorable fragments from the clusters. Therefore, we might have ended up with smaller than the optimally sized subgroups in the refactorability assessment. Finally, some other threats to internal validity come from the algorithms and heuristics that we used in our work, which might not be optimal, and thus further investigation and experiments need to be performed.

The external threats to our empirical study hinder the ability to generalize our findings beyond the nine open-source projects we examined and the four clone detection tools we used. However, the diversity in the application domain, language specifications, size, and age of the examined projects, as well as the diversity in the techniques applied by the selected clone detection tools mitigate this threat to a large extent.

Chapter 7

Conclusion and Future Work

This work is the first step towards an effective and efficient clone group refactoring approach. To this point we developed and implemented an approach that (1) Uses data and control dependencies, as well data types to map the statements within clone fragments; (2) Finds refactorable clusters within clone groups; (3) Assesses the refactorability of clones at pair and group (cluster) level. During our work on mapping clone statements we tried two approaches. In the Metric approach, for each statement, we generate a feature vector representing the statement's structure in terms of AST without incorporating data type information. However, the results show that this approach does not perform well, as we compared it to the state-of-the-art-work [TMK15] regarding the similarity of the statement mappings they produce, and the results at pair level were for: Clone Type I, 98.1%; Clone Type II, 89%; Clone Type III, 43.5%. After that we tried another approach that relies on data types and we were able to achieve a higher percentage of similarity in pair matching for the same clone set, and the results are: Clone Type I, 100%; Clone Type II, 94%; Clone Type III, 62.5% .

We compared our work to the state-of-the-art-work [TMK15] in pairwise statement mapping, using the later approach in mapping clones statements, for a large number of clones detected by 4 different clone detection tools in 9 open-source projects. For Clone Type I, we achieved an accuracy of (100%) in pairwise statement mapping. For Clone Type II, we achieved an identical statement mapping between 85.2%-97% at clone pair level, and 94.9%-98.9% at statement level. Finally, for Clone Type III, we achieved an identical statement mapping between 60.8%-85.8% at clone pair level, and 87.5%-93.3% at statement level.

Clustering can improve the overall execution time by minimizing the number of clones that need to be compared. It also serves as a filtration step by excluding clone instances that have control structure and statement differences. Moreover, we found that by removing a single clone instance from a group, the group becomes refactorable, or it increases the number of statements to be extracted. We also assessed the refactorability of the clusters that we found within the clone groups detected by clone detection tools. We achieved a refactorability of 59.5% for the clones detected by CloneDR, and a refactorability ranging between 31.1%-34.4% for the clones detected by the rest of the tools.

As future work, we are planning to (1) Address some of the internal threats to validity; (2) Extend our work to support clone refactoring with Lambda expressions Tsantalis et al. [TMR17]; (3) Perform an in-depth evaluation for the clone pairs examined in the Empirical study to further improve some steps of our approach; (4) Create an Eclipse plug-in for clone group refactoring;

References

[AGF11] Roberta Arcoverde, Alessandro Garcia, and Eduardo Figueiredo. “Understanding the longevity of code smells: preliminary results of an explanatory survey”. In: *Proceedings of the 4th Workshop on Refactoring Tools*. ACM. 2011, pp. 33–36.

[Bak95] B. S. Baker. “On finding duplication and near-duplication in large software systems”. In: *Proceedings of 2nd Working Conference on Reverse Engineering*. July 1995, pp. 86–95. DOI: `10.1109/WCRE.1995.514697`.

[Bal+99] M. Balazinska et al. “Measuring clone based reengineering opportunities”. In: *Proceedings Sixth International Software Metrics Symposium (Cat. No.PR00403)*. 1999, pp. 292–303. DOI: `10.1109/METRIC.1999.809750`.

[Bax+98] Ira D. Baxter et al. “Clone Detection Using Abstract Syntax Trees”. In: *Proceedings of the International Conference on Software Maintenance*. ICSM ’98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 368–. ISBN: 0-8186-8779-7. URL: `http://dl.acm.org/citation.cfm?id=850947.853341`.

[Bec+99] Kent Beck et al. “Refactoring: improving the design of existing code”. In: *Refactoring Improving the Design of Existing Code, Addison-Wesley Professional* (1999), pp. 75–88.

[Bel+07] Stefan Bellon et al. “Comparison and Evaluation of Clone Detection Tools”. In: *IEEE Trans. Softw. Eng.* 33.9 (Sept. 2007), pp. 577–591. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70725. URL: http://dx.doi.org/10.1109/TSE.2007.70725.

[Bia+13] Yixin Bian et al. “SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones”. In: *Journal of Systems and Software* 86.8 (2013), pp. 2077–2093.

[BJD13] Mathieu Bouchard, Anne-Laure Jousselme, and Pierre-Emmanuel Doré. “A proof for the positive definiteness of the Jaccard index matrix”. In: *International Journal of Approximate Reasoning* 54.5 (2013), pp. 615–626.

[BKR02] Abraham Bookstein, Vladimir A Kulyukin, and Timo Raita. “Generalized hamming distance”. In: *Information Retrieval* 5.4 (2002), pp. 353–375.

[CC90] Elliot J. Chikofsky and James H Cross. “Reverse engineering and design recovery: A taxonomy”. In: *IEEE software* 7.1 (1990), pp. 13–17.

[Cha99] Ned Chapin. “Software maintenance: A different view”. In: *Managing Requirements Knowledge, International Workshop on* 00.undefined (1899), p. 507. DOI: doi.ieeecomputersociety.org/10.1109/AFIPS.1985.53.

[Col+94] D. Coleman et al. “Using metrics to evaluate software system maintainability”. In: *Computer* 27.8 (Aug. 1994), pp. 44–49. ISSN: 0018-9162. DOI: 10.1109/2.303623.

[Deh+11] Najim Dehak et al. “Front-end factor analysis for speaker verification”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 19.4 (2011), pp. 788–798.

[Dek92] Sasa Dekleva. “Delphi study of software maintenance problems”. In: *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE. 1992, pp. 10–17.

[DH02] C. Ding and Xiaofeng He. “Cluster merging and splitting in hierarchical clustering algorithms”. In: *2002 IEEE International Conference on Data Mining, 2002. Proceedings*. 2002, pp. 139–146. DOI: `10.1109/ICDM.2002.1183896`.

[Din+01] Chris HQ Ding et al. “A min-max cut algorithm for graph partitioning and data clustering”. In: *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE. 2001, pp. 107–114.

[DR08] Ekwa Duala-Ekoko and Martin P Robillard. “Clonetracker: tool support for code clone management”. In: *Proceedings of the 30th international conference on Software engineering*. ACM. 2008, pp. 843–846.

[DRD99] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. “A language independent approach for detecting duplicated code”. In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 109–118.

[Foua] Eclipse Foundation. *AST View Plugin*. `http://www.eclipse.org/jdt/ui/astview/`. Accessed: Jan 27, 2017.

[Foub] Eclipse Foundation. *ASTVisitor Eclipse API*. `http://help.eclipse.org/mars/index.jsp?topic=\%2Forg.eclipse.jdt.doc.isv\%2Freference\%2Fapi\%2Forg\%2Feclipse\%2Fjdt\%2Fcore\%2Fdom\%2FASTVisitor.html`. Accessed: Jan 27, 2017.

[Fow+99] Martin Fowler et al. *Refactoring: Improving the design of existing programs*. 1999.

[FOW84] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. "The program dependence graph and its use in optimization". In: *International Symposium on Programming*. Springer. 1984, pp. 125–132.

[GK05] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[Gra] GrammaTech. *CodeSurfer*. `https://www.grammatech.com/products/codesurfer`. Accessed: Jan 27, 2017.

[GT03] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. 2nd. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 2003. ISBN: 9789812384256, 9812384251.

[Har16] Simon Harris. *Simian, Clone Detection Tool.* `http://www.harukizaemon.com/simian/`. Accessed: Jan 27, 2017. 2016.

[HHK12] K. Hotta, Y. Higo, and S. Kusumoto. "Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph". In: *2012 16th European Conference on Software Maintenance and Reengineering*. Mar. 2012, pp. 53–62. DOI: `10.1109/CSMR.2012.16`.

[HK11] Yoshiki Higo and Shinji Kusumoto. "Code clone detection on specialized PDGs with heuristics". In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE. 2011, pp. 75–84.

[HKI08] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. "A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system". In: *Journal of Software Maintenance and Evolution: Research and Practice* 20.6 (2008), pp. 435–461.

[HS88] Desmond G Higgins and Paul M Sharp. "CLUSTAL: a package for performing multiple sequence alignment on a microcomputer". In: *Gene* 73.1 (1988), pp. 237–244.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988. ISBN: 0-13-022278-X.

[JDH09] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. "CloneDetective - A Workbench for Clone Detection Research". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 603–606. ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070566`. URL: `http://dx.doi.org/10.1109/ICSE.2009.5070566`.

[JFr] JFreeChart. *JFreechart-1.0.10*. `https://users.encs.concordia.ca/~nikolaos/TSE_2015/projects/jfreechart-1.0.10.7z`. Accessed: Jan 27, 2017.

[JH07] Nicolas Juillerat and Beat Hirsbrunner. "Toward an implementation of the" form template method" refactoring". In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE. 2007, pp. 81–90.

[Jia+07] Lingxiao Jiang et al. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones". In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. ISBN: 0-7695-2828-7. DOI: `10.1109/ICSE.2007.30`. URL: `http://dx.doi.org/10.1109/ICSE.2007.30`.

[Jue+09] Elmar Juergens et al. "Do code clones matter?" In: *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society. 2009, pp. 485–495.

[KH01] Raghavan Komondoor and Susan Horwitz. "Using slicing to identify duplication in source code". In: *International Static Analysis Symposium*. Springer. 2001, pp. 40–56.

[KH03] Raghavan Komondoor and Susan Horwitz. "Effective, automatic procedure extraction". In: *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE. 2003, pp. 33–42.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670.

[Kon01] Georges Golomingi Koni-N′Sapu. "A scenario based approach for refactoring duplicated code in object oriented systems". In: *Master's thesis, University of Bern* (2001).

[Kos08] Rainer Koschke. "Frontiers of software clone management". In: *Frontiers of Software Maintenance* 2008 (2008), pp. 119–128.

[Kri01] Jens Krinke. "Identifying similar code with program dependence graphs". In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE. 2001, pp. 301–309.

[Lag+97] B. Lague et al. "Assessing the benefits of incorporating function clone detection in a development process". In: *1997 Proceedings International Conference on Software Maintenance*. Oct. 1997, pp. 314–321. DOI: 10.1109/ICSM.1997.624264.

[Lev66] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (Feb. 1966), p. 707.

[Li+06] Zhenmin Li et al. “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code”. In: *IEEE Trans. Softw. Eng.* 32.3 (Mar. 2006), pp. 176–192. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.28. URL: http://dx.doi.org/10.1109/TSE.2006.28.

[Lin+14] Yun Lin et al. “Detecting differences across multiple instances of code clones”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 164–174.

[Liu+06] Chao Liu et al. “GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’06. Philadelphia, PA, USA: ACM, 2006, pp. 872–881. ISBN: 1-59593-339-5. DOI: 10.1145/1150402.1150522. URL: http://doi.acm.org/10.1145/1150402.1150522.

[LS81] Bennet P Lientz and E Burton Swanson. “Problems in application software maintenance”. In: *Communications of the ACM* 24.11 (1981), pp. 763–769.

[LW08] Angela Lozano and Michel Wermelinger. “Assessing the effect of clones on changeability”. In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE. 2008, pp. 227–236.

[Mar] cre] Marco Torchiano [aut. *Efficient Effect Size Computation*. https://cran.r-project.org/web/packages/effsize/effsize.pdf. Accessed: March 5, 2017.

[Men+15] Na Meng et al. "Does automated refactoring obviate systematic editing?" In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 392–402.

[Mic] University of Michigan. *The COBOL Programming Languaget*. `http://groups.engin.umd.umich.edu/CIS/course.des/cis400/cobol/cobol.html`. Accessed: Jan 27, 2017.

[MKM13] Na Meng, Miryung Kim, and Kathryn S McKinley. "LASE: locating and applying systematic edits by learning from examples". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 502–511.

[MLM96] J. Mayrand, C. Leblanc, and E. M. Merlo. "Experiment on the automatic detection of function clones in a software system using metrics". In: *1996 Proceedings of International Conference on Software Maintenance*. Nov. 1996, pp. 244–253. DOI: `10.1109/ICSM.1996.565012`.

[Mob90] R.Keith Mobley. *An introduction to predictive maintenance*. English. ID: 19740087. New York, NY: Van Nostrand Reinhold, 1990. ISBN: 0442318286 9780442318284.

[MR92] Bill N Maggard and David M Rhyne. "Total productive maintenance: a timely integration of production and maintenance". In: *Production and Inventory Management Journal* 33.4 (1992), p. 6.

[MRS12] Manishankar Mondal, Chanchal K Roy, and Kevin A Schneider. "An empirical study on clone stability". In: *ACM SIGAPP Applied Computing Review* 12.3 (2012), pp. 20–36.

[MT04] T. Mens and T. Tourwe. “A survey of software refactoring”. In: *IEEE Transactions on Software Engineering* 30.2 (Feb. 2004), pp. 126–139. ISSN: 0098-5589. DOI: 10.1109/TSE.2004.1265817.

[Ngu+12] Hoan Anh Nguyen et al. “Clone management for evolving software”. In: *Software Engineering, IEEE Transactions on* 38.5 (2012), pp. 1008–1026.

[Opd92] William F Opdyke. “Refactoring: A program restructuring aid in designing object-oriented application frameworks”. PhD thesis. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[PJ09] Hae-Sang Park and Chi-Hyuck Jun. “A simple and fast algorithm for K-medoids clustering”. In: *Expert Systems with Applications* 36.2 (2009), pp. 3336–3341.

[Pro] ProjectBauhaus. *CCDIMAL, Clone Detection Tool.* http://www.iste.uni-stuttgart.de/en/ps/project-bauhaus.html. Accessed: Jan 27, 2017.

[RC07] Chanchal Kumar Roy and James R Cordy. *A survey on software clone detection research.* Tech. rep. Technical Report 541, Queen′s University at Kingston, 2007.

[RC08a] Chanchal K. Roy and James R. Cordy. “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization”. In: *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension.* ICPC ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 172–181. ISBN: 978-0-7695-3176-2. DOI: 10.1109/ICPC.2008.41. URL: http://dx.doi.org/10.1109/ICPC.2008.41.

[RC08b] Chanchal K Roy and James R Cordy. "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization". In: *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE. 2008, pp. 172–181.

[RC09] C. K. Roy and J. R. Cordy. "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools". In: *2009 International Conference on Software Testing, Verification, and Validation Workshops*. Apr. 2009, pp. 157–166. DOI: 10.1109/ICSTW.2009.18.

[RCK09] Chanchal K Roy, James R Cordy, and Rainer Koschke. "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". In: *Science of Computer Programming* 74.7 (2009), pp. 470–495.

[RM05] Lior Rokach and Oded Maimon. "Clustering methods". In: *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 321–352.

[Rom+06] Jeanine Romano et al. "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen′sd indices the most appropriate choices". In: *In annual meeting of the Southern Association for Institutional Research*. 2006.

[Rou87] Peter J Rousseeuw. "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis". In: *Journal of computational and applied mathematics* 20 (1987), pp. 53–65.

[SGP04] David Shepherd, Emily Gibson, and Lori Pollock. "Design and Evaluation of an Automated Aspect Mining Tool". In: *Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS '04)*. IEEE, Apr. 2004.

[SP03] David Shepherd and Lori Pollock. *Ophir: A Framework for Automatic Mining and Refactoring of Aspects*. Tech. rep. University of Delaware, Oct. 2003.

[Sta] Laerd Statistics. *Wilcoxon Signed-Rank Test using SPSS Statistics*. https://statistics.laerd.com/spss-tutorials/wilcoxon-signed-rank-test-using-spss-statistics.php. Accessed: March 5, 2017.

[STV16] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. "Why We Refactor? Confessions of GitHub Contributors". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 858–870. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950305. URL: http://doi.acm.org/10.1145/2950290.2950305.

[SWA03] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. "Winnowing: local algorithms for document fingerprinting". In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 76–85.

[TG12] Robert Tairas and Jeff Gray. "Increasing clone maintenance support by unifying clone detection and refactoring activities". In: *Information and Software Technology* 54.12 (2012), pp. 1297–1307.

[TMK15] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. "Assessing the refactorability of software clones". In: *IEEE Transactions on Software Engineering* 41.11 (2015), pp. 1055–1090.

[TMR17] Nikolaos Tsantalis, Davood Mazinanian, and Shahriar Rostami. "Clone Refactoring with Lambda Expressions". In: *Proceedings of the 39th Inter-*

national Conference on Software Engineering. ICSE 2017. Buenos Aires, Argentina, 2017.

[Tsaa] Nikolaos Tsantalis. *DataSet*. `https://users.encs.concordia.ca/~nikolaos/TSE_2015/results/jfreechart-1.0.10-deckard-tests.7z`. Accessed: Jan 27, 2017.

[Tsab] Nikolaos Tsantalis. *Projects*. `https://users.encs.concordia.ca/~nikolaos/TSE_2015/`. Accessed: Jan 27, 2017.

[Tsu+15] Masateru Tsunoda et al. "Benchmarking Software Maintenance Based on Working Time". In: *Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence (ACIT-CSI), 2015 3rd International Conference on*. IEEE. 2015, pp. 20–27.

[WB87] Michael Wolfe and Utpal Banerjee. "Data dependence and its application to parallel processing". In: *International Journal of Parallel Programming* 16.2 (1987), pp. 137–178.

[Wes14] Peter H. Westfall. "Kurtosis as Peakedness, 1905–2014. R.I.P." In: *The American Statistician* 68.3 (2014), pp. 191–195. DOI: `10.1080/00031305.2014.917055`. eprint: `http://dx.doi.org/10.1080/00031305.2014.917055`. URL: `http://dx.doi.org/10.1080/00031305.2014.917055`.

[WG14] Wei Wang and Michael W Godfrey. "Recommending clones for refactoring using design, context, and history". In: *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2014, pp. 331–340.

[Whe95] Donald J Wheeler. *Advanced topics in statistical process control*. Vol. 470. SPC press Knoxville, TN, 1995.

[Wil45] Frank Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 00994987. URL: http://www.jstor.org/stable/3001968.

[Wir89] Terry Wireman. “World class maintenance management.” In: *AUTOFACT’89* (1989), p. 1989.

[WSG16] S. Wehaibi, E. Shihab, and L. Guerrouj. “Examining the Impact of Self-Admitted Technical Debt on Software Quality”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. Mar. 2016, pp. 179–188. DOI: 10.1109/SANER.2016.72.

[YM13] Aiko Yamashita and Leon Moonen. “Surveying developer knowledge and interest in code smells through online freelance marketplaces”. In: *User Evaluations for Software Engineering Researchers (USER), 2013 2nd International Workshop on*. IEEE. 2013, pp. 5–8.

[ZKF05] Ying Zhao, George Karypis, and Usama Fayyad. “Hierarchical clustering algorithms for document datasets”. In: *Data mining and knowledge discovery* 10.2 (2005), pp. 141–168.