Analyzing Clone Evolution for Identifying the Important Clones for Management

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Manishankar Mondal

Permission to Use

Abstract

Code clones (identical or similar code fragments in a code-base) have dual but contradictory impacts (i.e., both positive and negative impacts) on the evolution and maintenance of a software system. Because of the negative impacts (such as high change-proneness, bug-proneness, and unintentional inconsistencies), software researchers consider code clones to be the number one bad-smell in a code-base. Existing studies on clone management suggest managing code clones through refactoring and tracking. However, a software system's code-base may contain a huge number of code clones, and it is impractical to consider all these clones for refactoring or tracking. In these circumstances, it is essential to identify code clones that can be considered particularly important for refactoring and tracking. However, no existing study has investigated this matter. We conduct our research emphasizing this matter, and perform five studies on identifying important clones by analyzing clone evolution history.

In our first study we detect evolutionary coupling of code clones by automatically investigating clone evolution history from thousands of commits of software systems downloaded from on-line SVN repositories. By analyzing evolutionary coupling of code clones we identify a particular clone change pattern, *Similarity Preserving Change Pattern* (SPCP), such that code clones that evolve following this pattern should be considered important for refactoring. We call these important clones the *SPCP clones*. We rank SPCP clones considering their strength of evolutionary coupling. In our second study we further analyze evolutionary coupling of code clones with an aim to assist clone tracking. The purpose of clone tracking is to identify the co-change (i.e. changing together) candidates of code clones to ensure consistency of changes in the code-base. Our research in the second study identifies and ranks the important co-change candidates by analyzing their evolutionary coupling. In our third study we perform a deeper analysis on the SPCP clones and identify their cross-boundary evolutionary couplings. On the basis of such couplings we separate the SPCP clones into two disjoint subsets. While one subset contains the non-cross-boundary SPCP clones which can be considered important for refactoring, the other subset contains the cross-boundary SPCP clones which should be considered important for tracking. In our fourth study we analyze the bug-proneness of different types of SPCP clones in order to identify which type(s) of code clones have high tendencies of experiencing bug-fixes. Such clone-types can be given high priorities for management (refactoring or tracking). In our last study we analyze and compare the late propagation tendencies of different types of code clones. Late propagation is commonly regarded as a harmful clone evolution pattern. Findings from our last study can help us prioritize clone-types for management on the basis of their tendencies of experiencing late propagations. We also find that late propagation can be considerably minimized by managing the SPCP clones. On the basis of our studies we develop an automatic system called AMIC (Automatic Mining of Important Clones) that identifies the important clones for management (refactoring and tracking) and ranks these clones considering their evolutionary coupling, bug-proneness, and late propagation tendencies. We believe that our research findings have the potential to assist clone management by pin-pointing the important clones to be managed, and thus, considerably minimizing clone management effort.

Acknowledgements

First of all, I would like to express my heartiest gratitude to my respected supervisors Dr. Chanchal K. Roy and Dr. Kevin A. Schneider for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. Without them this work would have been impossible.

I would like to thank Dr. Kevin Stanley, Dr. Ian McQuillan, Dr. Ramakrishna Gokaraju, and Dr. Michael W. Godfrey for their willingness to take part in the advisement and evaluation of my thesis work.

Thanks to all of the members of the Software Research Lab with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Muhammad Asaduzzaman, Md. Saidur Rahman, Masudur Rahman, Amit Mondal, Farouq Al. Omari, Shamima Yeasmin, and Judith Fransisca.

I am grateful to the Department of Computer Science of the University of Saskatchewan for their generous financial support through scholarships, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I thank the anonymous reviewers for their valuable comments and suggestions in improving the papers produced from this thesis.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me to reach at this stage. In particular, I would like to thank Gwen Lancaster, Heather Webb, and Sakiba Jalal.

The most wonderful thing in my life is my daughter, Bhaswatee Mondal, whose presence drove me in the right direction in finishing my thesis work. Also, my daughter and my wife Smritikana Mondal are the ones who sacrificed the most for this thesis. I could not be present at their side during their most difficult moments.

I express my heartiest gratitude to my mother Gita Rani Mondal, and my father Monoranjan Mondal who are the architects of my life. Their endless sacrifice has made me to reach at this stage of my life. My parents, my sister Maitrayee Mondal, and my wife have always inspired me in completing my thesis work.

Dedication

I dedicate this thesis to my mother, Gita Rani Mondal, whose affection and inspiration have given me mental strength in every step of my life, to my father, Monoranjan Mondal, whose strong character and philosophical thoughts have always made me take the right decisions, and to my wife, Smritikana Mondal, whose continuous sacrifice has helped me achieve my goal.

Contents

List of Tables

List of Figures

List of Abbreviations

SPCP	Similarity Preserving Change Pattern
SPC	Similarity Preserving Change
SPCO	Similarity Preserving Co-change
AMIC	Automatic Mining of Important Clones
JAR	Java Archive
JDK	Java Development Kit
SQL	Structured Query Language
SVN	Subversion

Chapter 1

Introduction

Code cloning has emerged as a controversial term in the realm of software engineering research and practice, because of its contradictory impacts on software evolution and maintenance. Code cloning generally refers to the frequent copy/paste activities of programmers during implementation. Copying one code fragment from one place of a code-base and pasting it to several other places with or without modification cause the existence of identical or nearly similar code fragments in the code-base. Such code fragments are known as code clones. A group of similar code fragments forms a clone class or a clone group. Two code fragments that are similar to each other form a clone-pair. Although copy/pasting is considered to be the primary reason behind code clones, some other factors such as: repetition of common functionality, programmer laziness, technology limitation, and code understandability may influence clone creation.

Code clones are of significant importance from the perspectives of software maintenance. A great many studies [19, 29, 30, 75, 78, 89, 111, 113, 114, 120, 146, 155, 158, 183, 186, 192, 194–197, 197–199, 201, 203, 211, 215, 220, 229, 232, 233, 248, 272, 275, 276] have been conducted on analyzing clone impacts on software maintenance and evolution. While a number of studies [19, 75, 89, 111, 131–133] identified some positive impacts of code clones (such as: faster software development, and better understandability of source code), there is strong empirical evidence [29, 30, 44, 51, 78, 91, 97, 141, 142, 145, 146, 156, 158, 263] of some negative impacts too. These negative impacts include: hidden bug-propagation [145], unintentional inconsistent changes [29], late propagations [30], and higher instability [78]. Emphasizing the negative impacts, software researchers suggest managing code clones through refactoring [25, 35, 272] and tracking [61, 95].

Clone refactoring refers to the task of merging several clone fragments from a clone class (i.e., a group of code fragments that are similar to one another) into a single one if possible. However, there can be situations where refactoring of clone fragments in a particular class is impossible but the fragments need to be updated together consistently. Clone tracking is important in such situations. Clone tracking [61, 95] means remembering all the clone fragments in a clone class as the software system evolves through changes so that when a programmer makes some changes to a particular clone fragment in that class, the clone tracking system can automatically notify her about the existence of the other clone fragments in the class. The programmer can then decide whether she needs to implement similar changes to these other clone fragments in order to ensure consistency of the software system's code-base.

1.1 Problem Definition

A software system may contain a huge number of code clones. It is impractical to consider all these clones for refactoring or tracking, because some clones are volatile and a considerable proportion of the code clones never change during evolution [119]. Moreover, clone refactoring is often very time consuming and requires interactions from experienced programmers. Clone tracking is resource intensive. In such a situation, it is essential to identify code clones that are important from refactoring and tracking perspectives. However, although a great many studies [26,27,27,35,37,38,56,60–62,82–84,95,134,152,218,219,236,237,244,268,274] have been conducted on clone refactoring and tracking, none of these studies investigate which of the code clones in a software system are important to be refactored or tracked. A number of clone refactoring and tracking tools [27,35,39,61,67,80,83,87,95,122,139,151,152,210,239,244,245] currently exist. However, these tools cannot identify which code clones can be important for refactoring or tracking. Focusing on this drawback of the existing studies and tools, we address the following problem through our research.

Research Problem

Given the huge number of code clones in a software system's code-base, how do we identify the important ones from the perspectives of clone management (clone refactoring and tracking)?

1.2 Addressing the Research Problem

In order to address the problem stated above, we conduct our research for identifying code clones that can be important for refactoring and tracking. In our research, we automatically mine and analyze clone evolution history from thousands of commits of software systems downloaded from on-line SVN repositories. We investigate whether evolutionary coupling and bug-proneness of code clones can be used for prioritizing them for management such as refactoring and tracking. We briefly describe our research behind finding important code clones in the following way.

Our Research behind Addressing the Problem

By manually observing the clone evolution history we realize that not all of them evolve in the same way. While some code clones evolve together (co-evolve) by preserving similarity among them, a considerable portion of them evolve independently (without preserving similarity). Moreover, many code clones never change during evolution. Intuitively, code clones that evolve independently or are rarely changed during evolution should not be considered important for management. Only the code clones that co-evolve by preserving their similarity should be considered important. By observing the evolution of such code clones we discover their change pattern and call this pattern a **Similarity Preserving Change Pattern (SPCP)**. We consider SPCP clones (the code clones that evolve following an SPCP) to be the important ones for refactoring, because they have a tendency of co-evolving consistently (by preserving their similarity). We rank SPCP clones on the basis of the strength of their evolutionary coupling. We further analyze the evolutionary coupling of code clones, and utilize it for predicting and ranking co-change (changing together) candidates for a clone fragment that a programmer attempts to change. While dealing with evolutionary coupling among clone fragments from the same clone class, we suspected that clones might have evolutionary coupling with code fragments beyond their class boundaries. We investigate this and realize that clones having cross-boundary (beyond class boundary) evolutionary coupling should not be considered for refactoring. Refactoring such a clone fragment might negatively affect the evolution of the cross-boundary code fragments that have coupling with it. Thus, such a clone fragment should be considered important for tracking. We mine the cross-boundary evolutionary couplings of SPCP clones, and suggest that SPCP clones having such couplings should be considered important for tracking rather than refactoring. The non-cross-boundary SPCP clones should be considered for refactoring. We rank the non-cross-boundary and cross-boundary SPCP clones for refactoring and tracking on the basis of the strength of their non-cross-boundary and cross-boundary evolutionary coupling respectively. Although we rank the SPCP clones for management on the basis of their evolutionary coupling, we realize that we should also consider their bug-proneness when ranking. Intuitively, code clones with a higher bug-proneness should be given a higher priority for management. Considering this fact, we analyze the bug-proneness of different types (Type 1, 2, and 3) of code clones and realize that Type 3 clones have the highest bug-proneness among the three types. Moreover, the bug-prone clones of Type 3 have the highest possibility of evolving following an SPCP. Thus, Type 3 SPCP clones should be considered the important ones for management. It is suspected that bug-proneness of code clones is primarily influenced by their tendencies of experiencing late propagation. We investigate late propagation in code clones and find that Type 3 clones have the highest tendency of experiencing late propagation among the three clone-types. We finally discover that we can significantly minimize the occurrences of late propagation by managing SPCP clones through refactoring and tracking.

Decomposing Our Research behind Addressing the Problem

We decompose our research behind addressing the research problem into the following five studies.

- **Study 1:** *Identifying code clones that can be considered important for refactoring* [163].
- **Study 2:** *Ranking co-change candidates of code clones* [167].
- **Study 3:** *Identifying code clones that should be considered important for tracking* [162].
- **Study 4:** *Investigating bug-proneness of code clones* [168].
- **Study 5:** *Investigating late propagation in code clones* [171].

We briefly describe each of these studies in the following subsections.

1.2.1 Study 1: Identifying Code Clones that Can Be Important for Refactoring

In this study we investigate how we can identify code clones that can be considered important for refactoring. We have already discussed that none of the existing studies on clone refactoring focused on identifying important clones for refactoring. There are a number of studies [38, 138, 277] on scheduling clone refactoring activity. The purpose of these studies is to determine an optimal schedule for refactoring such that refactoring gain is maximized and refactoring effort is minimized. These studies consider all the code clones in a software system to determine the refactoring schedule. Scheduling is one form of ranking. However, we believe that before determining the refactoring schedule, we need to identify code clones that are important for refactoring. We can then only consider these important clones for scheduling. We do not need to consider the unimportant clones for scheduling. Clone refactoring is time consuming and it may require much effort from the experienced programmers. Thus, discarding the unimportant clones from refactoring considerations is essential to minimize clone refactoring effort.

In our first study, we analyze the clone evolution history of a software system. According to our consideration, if two or more clone fragments from a particular clone class evolve together by preserving their similarity, then these clone fragments should be considered important for refactoring. Emphasizing this fact we define a particular clone change pattern called **Similarity Preserving Change Pattern** (**SPCP**). The code clones that evolve following this pattern (i.e., the **SPCP clones**) are considered to be the important ones for refactoring. We also rank the **SPCP clones** for refactoring on the basis of the strength of their evolutionary coupling (we will discuss evolutionary coupling in Chapter 2). The **non-SPCP clones** (i.e., the code clones that do not evolve following an SPCP) either evolve independently or rarely change during evolution. Thus, such clones should not be considered important for refactoring. Chapter 3 presents a detailed description of our first study.

1.2.2 Study 2: Ranking Co-change Candidates of Code Clones

Clone refactoring is one way of managing code clones. There can be situations where refactoring of code clones in a particular clone class is impractical, however, the clone fragments in the class need to be updated consistently. Clone tracking is important in such situations. As we explained before, a clone tracker remembers the clone fragments in a clone class through system evolution so that it can notify a programmer about the existence of the other fragments in this class when she attempts to make changes to a particular fragment. The programmer can then decide whether she needs to change these other fragments consistently. However, all these other fragments might not need to be changed together (i.e., might not need to be co-changed) consistently with the particular fragment that is going to be changed. Only a few of these other fragments might be important for changing consistently with the particular fragment. Thus, ranking these other fragments on the basis of their importance of changing together with the particular clone fragment that has been attempted to be changed can help a programmer easily identify the important ones to be co-changed.

In our second study we analyze the evolutionary coupling of the clone fragments in a clone class. Let us consider that a programmer is attempting to change a particular clone fragment **CF** in a clone class. The other clone fragments in the clone class are the co-change candidates. We analyze the past change history of all the clone fragments in the clone class and determine their evolutionary coupling. We then determine which other clone fragments in the clone class exhibited evolutionary coupling with **CF**. Such clone fragments have higher possibilities of co-changing with **CF**. We also analyze and compare two types of ranking (we will describe these ranking mechanisms in Chapter 4) of the clone fragments that previously co-changed with **CF**. On the basis of our analysis, we rank these clone fragments considering how recently they co-changed with **CF**. We also rank the clone fragments that did not co-change with **CF** by considering their distances from **CF** in the file system hierarchy. We will elaborate our second study in Chapter 4.

1.2.3 Study 3: Identifying Code Clones that can Be Important for Tracking

From our first study we discovered that **SPCP clones** (code clones that evolved following an SPCP) can be the important ones for refactoring. SPCP clones from a particular clone class have evolutionary coupling among them. However, code clones from a particular clone class might have evolutionary coupling with non-clone fragments, and with clone fragments from other clone classes. Code clones having evolutionary coupling beyond their class boundary should not be considered important for removal through refactoring. Removal of such a clone fragment might negatively affect the evolution of the code fragments that have coupling with it but reside beyond its class boundary. Thus, it is important to identify SPCP clones that have relationships beyond their class boundaries. Such SPCP clones should not be considered for removal through refactoring. These should be considered important for tracking along with their cross-boundary relationships (i.e., relationships beyond their class boundaries). While the cross-boundary SPCP clones should be considered important for tracking, the remaining SPCP clones (i.e., the non-cross-boundary SPCP clones) in a code base can be considered important for refactoring. None of the existing studies on clone analysis and management investigated cross-boundary relationships of code clones.

In our third study, we detect all the SPCP clones in a software system's code-base and then analyze the evolutionary coupling relationships of these SPCP clones beyond their class boundaries. We separate the SPCP clones into two groups: (1) cross-boundary SPCP clones, and (2) non-cross-boundary SPCP clones. While the cross-boundary ones are important for tracking, the non-cross-boundary ones are important for refactoring. In Section 1.2.1, we mentioned that we rank the SPCP clones for refactoring on the basis of the strength of their evolutionary coupling. We apply such a ranking for the non-cross-boundary SPCP clones. We rank the cross-boundary SPCP clones on the basis of the number of cross-boundary coupling links they have. We will describe our third study in Chapter 5.

1.2.4 Study 4: Investigating Bug-proneness of Code Clones

From our previous studies we realize that **SPCP clones** (Code clones that evolved following a similarity preserving change pattern called **SPCP**) are important for refactoring and tracking. We ranked **SPCP** clones on the basis of their evolutionary coupling. However, bug-proneness of code clones should also be considered when prioritizing them for management. There are a number of studies [44, 51, 91, 97, 141] on clone bug-proneness. However, none of these studies compared the bug-proneness of the three types of code clones. Also, there is no study on investigating which type of SPCP clones (i.e., Type 1, Type 2, or Type 3 SPCP clones) have high possibilities of containing bugs. We believe that bug-proneness of code clones should also be considered when prioritizing them for management (refactoring and tracking).

In our fourth study, we analyze the evolutionary history of the clone fragments and identify which of the clone fragments experienced bug-fix changes during evolution. We compare the bug-proneness of the major types (Type 1, Type 2, and Type 3) of code clones. More specifically, we investigate which type of **SPCP** clones (Type 1, Type 2, or Type 3 **SPCP** clones) have high possibilities of experiencing bug-fixes. Although we previously ranked the SPCP clones on the basis of their evolutionary coupling, we can also rank them on the basis of their bug-proneness. We present our clone bug-proneness study in Chapter 6.

1.2.5 Study 5: Investigating Late Propagation in Code Clones

Late propagation is a particular evolutionary pattern of code clones. We will define late propagation in Chapter 7. Existing studies [19, 29, 30] show that this pattern is related to bugs and inconsistencies in the code-base. Researchers have investigated different specific patterns [30] of late propagation and identified which patterns are more related to bugs, faults, and inconsistencies. However, none of the studies investigated the intensities of late propagation in different types of clones separately. Such a study is important because, if late propagation is observed to be more intense in a particular clone type compared to the others, we might consider being more conscious while changing clones of that particular type. Also, we might want to refactor clones of that particular type with higher priority. None of the existing studies investigate the bug-proneness of late propagation in different types of clones separately. Such an investigation is also very important for understanding the comparative harmfulness of late propagation in different clone-types. Our previous studies introduce the importance of **SPCP clones**. We established that **SPCP clones** are the important ones for refactoring and tracking. However, we did not investigate whether managing SPCP clones can help us minimize late propagation in code clones.

In our fifth study, we analyze the evolutionary history of three types of clones separately and identify which code clones experienced late propagations. We then compare the intensities of late propagations in different types (Type 1, Type 2, and Type 3) of code clones. We also investigate and compare the bug-proneness of the late propagation clones of different clone-types. We finally investigate what percentage of the late propagations occur in **SPCP clones**. If most of the late propagations occur in SPCP clones, then it is an implication that managing SPCP clones can minimize the occurrence late propagations. We found that Type 3 clones have higher possibilities of experiencing late propagation compared to Type 1 and Type 2 clones. We also found that late propagations can be minimized considerably by managing the SPCP clones. We describe our fifth study in Chapter 7.

1.3 Outline of the Thesis

Our first chapter (Chapter 1, Introduction) introduces the research problem and presents a short description of our research behind addressing the research problem. The rest of the thesis is organized as follows.

- **Chapter 2** describes the background topics.
- **Chapter 3** elaborates our first study (i.e., **Study 1**) on identifying and ranking code clones that can be considered important for refactoring.
- **Chapter 4** discusses our second study (i.e., **Study 2**) on ranking co-change candidates for clones.
- **Chapter 5** contains the details of our third study (i.e., **Study 3**) on cross-boundary coupling relationships of SPCP clones (the code clones that evolved following an SPCP).
- **Chapter 6** presents our fourth study (i.e., **Study 4**) on comparing the bug-proneness of different types of code clones.
- **Chapter 7** describes our fifth study (i.e., **Study 5**) on the intensity and harmfulness of late propagation in different types of code clones.
- **Chapter 8** concludes the thesis.

Our five studies described in the thesis have already been published in major software engineering conferences [162,163,167,168] and journal [171]. There are a number of other publications [92,159–161,164–166, 169,170,172] from this thesis research. Appendix A contains the list of all the publications out of this thesis. On the basis of our five studies presented in the thesis we develop an automatic system called **AMIC** for mining important clones from the whole code-base of a software system, and also, for ranking these important clones considering their evolutionary coupling, bug-proneness, and late propagation tendencies. AMIC works by automatically mining and analyzing the clone evolution history from thousands of revisions of a candidate software system. We develop AMIC using Java with MySQL as the back-end database server. Chapter 8 presents a conceptual description of AMIC. A user-manual for AMIC is presented in Appendix B. We have also deployed AMIC in the web [173]. The website helps us easily detect and rank the important code clones.

Chapter 2

Background

This chapter presents the terminology that will be used in the rest of the thesis. Section 2.1 defines code clones, Section 2.2 describes different types of code clones, Section 2.3 focuses on the impacts of code clones on software evolution and maintenance, Section 2.4 describes clone refactoring and tracking, and Section 2.5 defines and discusses evolutionary coupling among program entities.

2.1 Code Clone

According to the literature [110,192,203], *if two or more code fragments in a software system's code-base are identical or nearly similar to one another, we call them code clones.*

Clone-Pair

Two code fragments that are similar to each other form a clone pair.

Clone Class

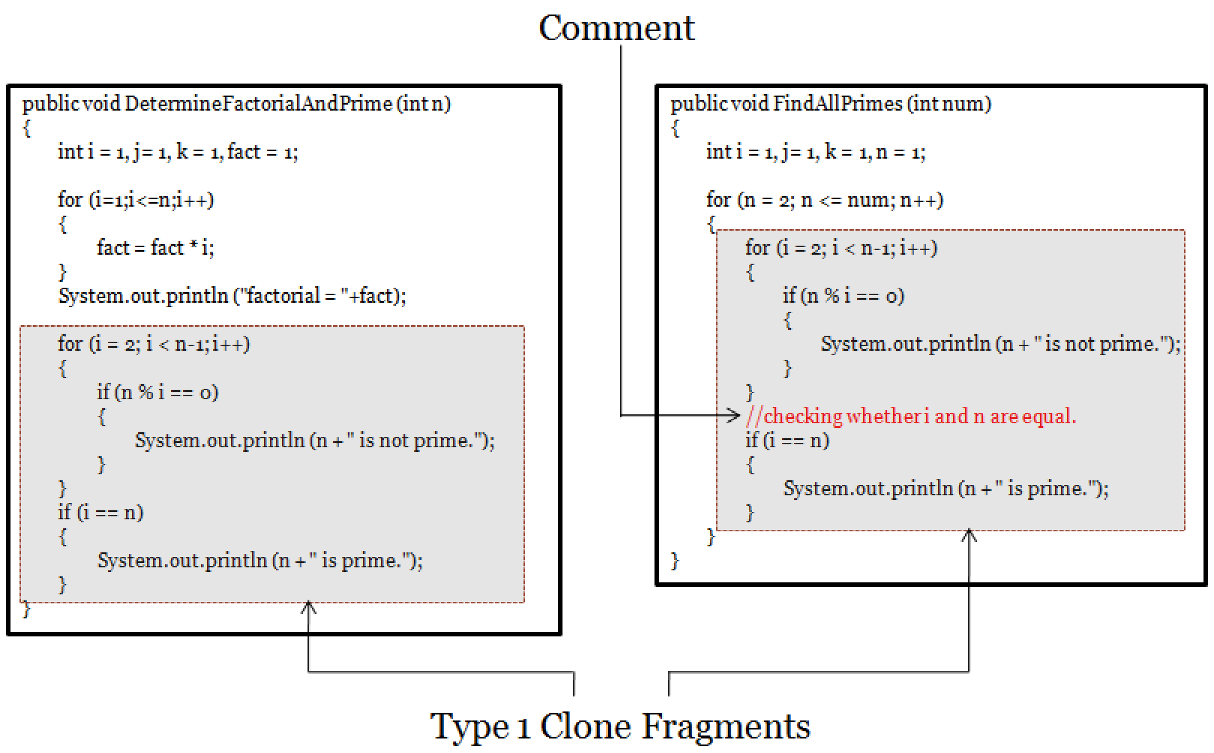
A group of similar code fragments forms a clone class or a clone group.

Clone Fragment

We frequently use the term 'clone fragment' in the thesis. A clone fragment is a particular code fragment which is exactly or nearly similar to one or more other code fragments in a code-base. Each member in a clone class or a clone-pair is a clone fragment.

Cloned Method

If a method contains cloned lines, we call this method a cloned method. If all lines of a method are cloned lines, then this method is a fully cloned method. If a method contains both cloned and non-cloned lines, we call this method a partially cloned method.



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1; i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        //checking whether i and n are equal.
        if (i == n)
        {
            System.out.println (n + " is prime.");
        }
    }
}
```

Figure 2.1: Type 1 (identical) clone pair

Method Clones

If two or more methods are clones of one another, we refer to these as method clones. Method clones are fully cloned methods.

Clone Genealogy

Clone genealogy [119] detection is an integral part of clone analysis. There are a number of studies [19, 24, 77, 131, 146, 179, 207, 208, 208, 209, 241] on clone genealogy detection. We define a clone genealogy in the following way. Let us assume that a clone fragment was created in a particular revision of a software system and was alive in a number of consecutive revisions. Thus, each of these revisions contains a snapshot of the clone fragment. The genealogy of this clone fragment consists of the set of its consecutive snapshots from the consecutive revisions where it was alive. Each clone fragment in a particular revision belongs to a particular clone genealogy. In other words, a particular clone fragment in a particular revision is actually a snapshot in a particular clone genealogy. By examining the genealogy of a clone fragment we can determine how it changed during software evolution.

Clone Visualization

A number of studies [16, 17, 212, 247] have been conducted on visualizing code clones resulting a number of tools. The goal of these studies and tools is to help us visually analyze the clone fragments in a clone class in a particular revision of a software system. Some of these tools also help us visualize the evolution of a clone fragment through different revisions.

2.2 Types of Code Clones

There are four types of code clones as discussed below.

2.2.1 Type 1 Clones

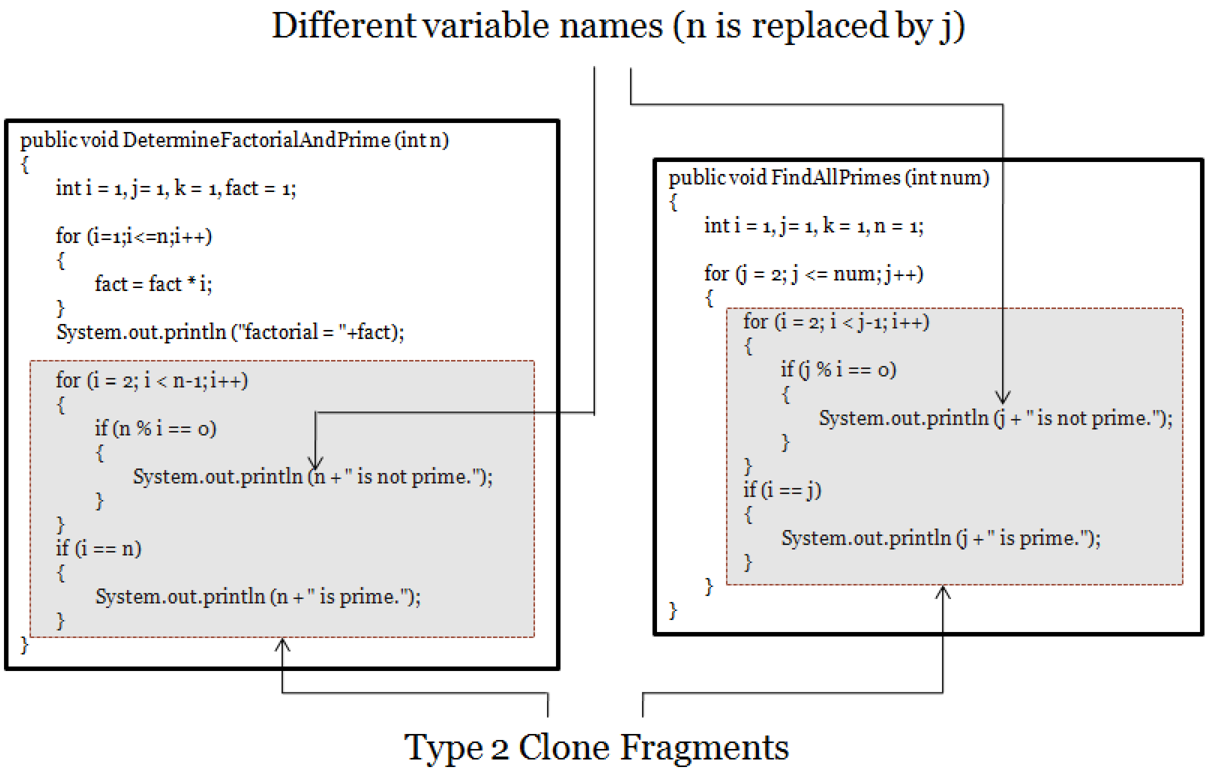
Exactly similar (i.e., identical) code fragments disregarding their comments and indentations are known as Type 1 clones. Fig. 2.1 shows a Type 1 clone-pair. One fragment of the pair resides in the method named 'DetermineFactorialAndPrime', and the other fragment resides in the method 'FindAllPrimes'. The two fragments have been shown in the light gray boxes. We see that the fragment at the right hand side contains a comment. If we disregard this comment, then the two fragments become identical.

2.2.2 Type 2 Clones

Type 2 clones are syntactically similar code fragments. These are mainly created from Type 1 clones because of renaming identifiers and changing data-types. Fig. 2.2 shows a Type 2 clone pair where the two fragments in the pair reside in two methods 'DetermineFactorialAndPrime' and 'FindAllPrimes'. The clone fragments have also been highlighted in the methods. We see that the fragment at the left hand side contains a variable called n. The fragment at the right hand side the corresponding variable has been named as j. Because of this variable renaming, these two fragments make a Type 2 clone pair.

2.2.3 Type 3 Clones

Type 3 clones are created from Type 1 and Type 2 clones because of addition, deletion, or modification of source code lines. Type 3 clones are also known as gapped clones. Fig. 2.3 contains an example of a Type 3 clone pair. The two fragments in the clone pair again reside in the two methods 'DetermineFactorialAnd-Prime' and 'FindAllPrimes'. We see that the fragment at the right hand side contains a line 'k=k+1' for counting the number of primes. However, this line is absent in the fragment at the left hand side. Thus, these two clone fragments make a Type 3 clone pair.



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1;i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

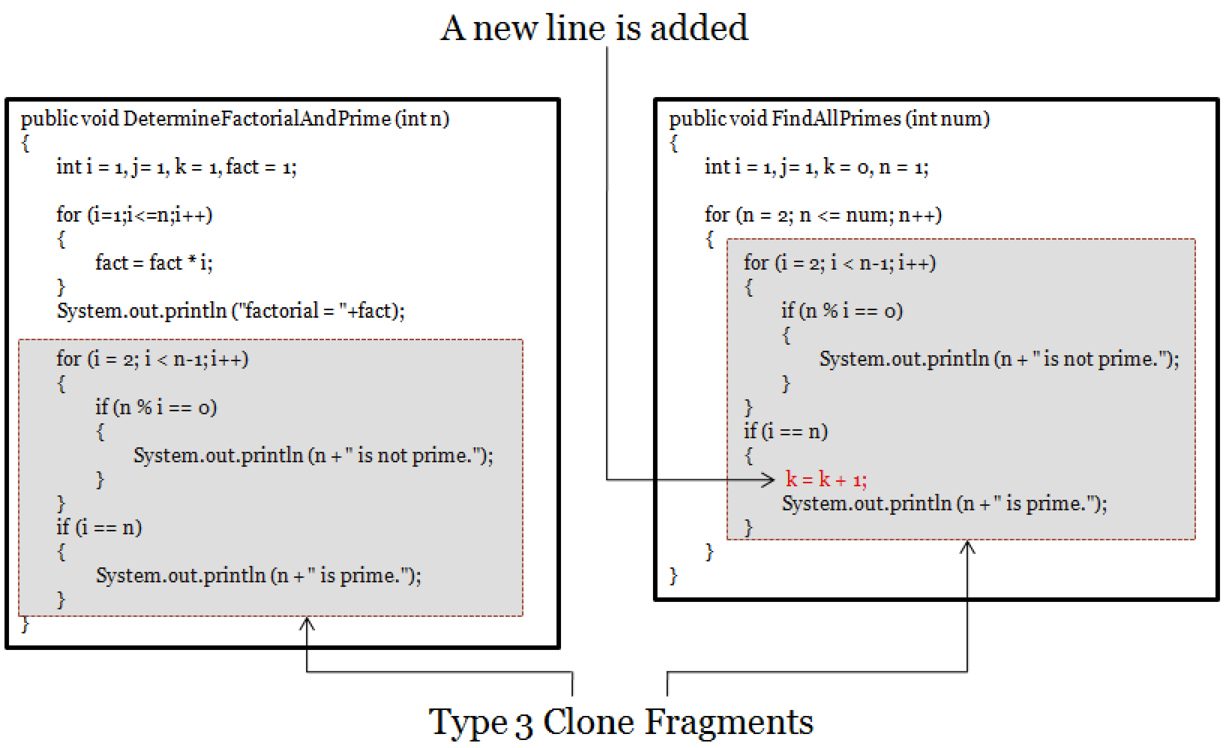
```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 1, n = 1;

    for (j = 2; j <= num; j++)
    {
        for (i = 2; i < j-1; i++)
        {
            if (j % i == 0)
            {
                System.out.println (j + " is not prime.");
            }
        }
        if (i == j)
        {
            System.out.println (j + " is prime.");
        }
    }
}
```

Figure 2.2: Type 2 clone pair

2.2.4 Type 4 Clones

Semantically similar code fragments are known as Type 4 clones. If two or more code fragments perform the same task but are implemented in different ways, these code fragments are called Type 4 clones. Type 4 clones are also known as semantic clones. The two methods ('SumNonRecursive' and 'SumRecursive') in Fig. 2.4 make a Type 4 clone pair. We see that each of these two methods perform the same task (finding the sum from 1 to n), however, they have been implemented in two different ways. When the method at the left hand side finds the sum using a for loop, the method at the right hand side calculates the sum using recursion. Thus, these two methods make a Type 4 clone pair.

Code clones can be of different granularities such as: file clones, class clones, method clones, or arbitrary block clones. Researchers have also investigated on detecting duplications (i.e., clones) in higher level code structures [31, 32, 149], formal models [11, 57, 58, 224, 226], UML sequence diagrams [144, 228], software requirements specifications [101, 102], and Matlab/Simulink models [12, 13, 100, 181, 188, 222, 223, 225].



```
public void DetermineFactorialAndPrime (int n)
{
    int i = 1, j= 1, k = 1, fact = 1;

    for (i=1;i<=n;i++)
    {
        fact = fact * i;
    }
    System.out.println ("factorial = "+fact);

    for (i = 2; i < n-1; i++)
    {
        if (n % i == 0)
        {
            System.out.println (n + " is not prime.");
        }
    }
    if (i == n)
    {
        System.out.println (n + " is prime.");
    }
}
```

```
public void FindAllPrimes (int num)
{
    int i = 1, j= 1, k = 0, n = 1;

    for (n = 2; n <= num; n++)
    {
        for (i = 2; i < n-1; i++)
        {
            if (n % i == 0)
            {
                System.out.println (n + " is not prime.");
            }
        }
        if (i == n)
        {
            k = k + 1;
            System.out.println (n + " is prime.");
        }
    }
}
```

Figure 2.3: Type 3 clone pair

2.2.5 Clone Detection Techniques

A great many clone detection techniques [6, 9, 10, 20–23, 33, 35, 47, 49, 50, 54, 55, 63, 65, 66, 70, 73, 85, 88, 90, 96, 98, 99, 108, 109, 112, 114, 114–118, 121, 124–128, 130, 135–137, 140, 142, 147, 149, 150, 176–178, 185, 189, 192–194, 196, 198, 201, 204, 206, 213, 214, 216, 227, 230, 234, 235, 240, 243, 246, 253, 260, 261, 264, 269, 270, 274, 275] have already been proposed by the researchers resulting a number of clone detection tools. Different techniques are suitable for different purposes. The clone detection techniques as well as tools can be categorized into the following categories on the basis of their underlying detection mechanism.

- **Text similarity based clone detection:** A number of clone detection techniques [98, 99] detect code clones by measuring textual similarity of the candidate code fragments. Such techniques are generally language independent.

- **Token similarity based clone detection:** Such techniques [20, 21, 109, 112, 142] first convert the program into a stream of tokens using lexical analyzers, and then compare the token sequences of the candidate code fragments to determine whether they are clones of one another. Such techniques can detect Type 1 and Type 2 clones.

- **AST (Abstract Syntax Trees) based clone detection:** Some techniques [35, 264] detect code clones by converting the whole program into a parse tree, and then by identifying similar sub-trees.

```
public int  SumNonRecursive (int  n)
{
     int  i = 0, sum = 0;
     for (i = 1; i <= n; i++)
     {
          sum = sum + i;
     }
     return sum;
}
```

```
public int  SumRecursive (int  n)
{
     if ( n == 0)
     {
          return 0;
     }
     else
     {
          return n + SumRecursive ( n - 1);
     }
}
```

Figure 2.4: Type 4 clone pair

- **Hybrid clone detection:** There is a hybrid clone detection technique called NiCad [48] that depends on TXL [1] parser for flexible pretty printing (i.e., normalizing) of the program source code, and then compares pretty-printed code fragments using simple text match.

A number of studies [196,198,201,204,231] have compared the clone detection techniques on the basis of their detection accuracy (i.e., precision, and recall). According to the most recent study [231] performed by Svajlenko and Roy using the Mutation Framework [198], the clone detection tools: NiCad [48], iClones [76], ConQat [103], and SimCad [249] are very good options for detecting the major three types of clones: Type 1, Type 2, and Type 3. A number of techniques [64,115,267] have also been proposed for detecting Type 4 clones (i.e., semantic clones).

2.3 Impacts of Code Clones

According to a great many studies [19,29,30,51,75,78,89,97,111,123,131–133,141,145,146,154,156,158,161, 200,255,256,278] code clones have both positive and negative impacts on software evolution and maintenance. We discuss these impacts in the following paragraphs.

2.3.1 Positive Impacts of Code Clones

According to a number of studies [18,19,75,89,111,131–133,157,184,202] code cloning has some positive impacts on software development. The positive impacts of cloning have been discussed below.

- **Faster software development:** Code cloning can help us in faster development of software systems. Reusing existing code blocks by slightly modifying them for implementing similar functionalities can considerably reduce implementation time and efforts resulting a considerable reduction of software

[1]`http://www.txl.ca/index.html`

development costs. According to a study of Kapser and Godfrey [111] code cloning can often be considered a reasonable development strategy.

- **Program comprehension:** Cloning can also help us in program comprehension. Understanding one clone fragment in a particular clone class might often be sufficient to know what other clone fragments in the same class do. Presence of code clones can also help us in context sensitive code completion [18] as well as context-aware search keywords formulation for handling programming errors and exceptions [184].

- **Reducing the risks of ripple change effect:** Changing a particular code fragment such as a method that is being used by several other methods might require corresponding changes to these other methods. In such a situation, it might be wise to create a copy of the particular method and using this newly created copy by making necessary changes to it leaving the original one as it is [157]. Changing the original fragment is often time consuming because it needs a proper analysis of the change impacts.

2.3.2 Negative Impacts of Code Clones

A number of studies [29, 30, 51, 78, 97, 141, 145, 146, 156, 158] have identified some strong negative impacts of code clones on the evolution and maintenance of software systems. We discuss these in the following points.

- **Hidden bug propagation:** The most important negative impact of the code clones is hidden bug propagation [29]. If one code fragment contains a bug and a programmer copy/pastes it to several places in the code-base without knowing the existence of the bug in the original fragment, the bug gets propagated. If such a hidden bug gets discovered at a particular point of evolution, the fixing should take place in all the copies.

- **Unintentional inconsistencies:** Code clones might create unintentional inconsistencies in the code-base. In general, the clone fragments in a clone class (i.e., a clone group) need to be updated together consistently. However, updating a subset of these fragments leaving the others as they are only because of not knowing the existence of these other fragments might create inconsistency in the code-base [30]. If such inconsistencies get discovered during later evolution, the clone fragments that were previously left unintentionally will need to be updated consistently. Such a phenomenon is also called late propagation [30] in code clones.

- **Higher change-proneness:** The existing studies [145, 146, 156, 158] also show that code clones have higher possibilities of experiencing changes (i.e., code clones are more change-prone) than non-clone code during software evolution. Thus, code clones are expected to require higher maintenance effort compared to non-clone code [158].

- **Code bloat:** Code cloning may lead to an unnecessary increase of code in the code-base [21, 111]. The increased code may require extra effort and cost during maintenance. Code cloning can also cause the existence of unused or dead code fragments in a code-base [111]. Such code fragments reduce code comprehensibility.

2.4 Clone Refactoring and Tracking

Focusing on the impacts of code clones, software researchers suggest to manage code clones through refactoring and tracking so that we can minimize the negative impacts of code clones, while also receiving their positive impacts. A great many studies [26,27,35,37,38,45,56,59–62,74,81–84,95,134,148,152,187,205,217–219,236–238,242,244,257,258,266,268,271,274,279,280] have been done on clone refactoring and tracking resulting a number of techniques and tools.

2.4.1 Clone Refactoring

Clone refactoring is the task of merging (unifying) two or more clone fragments from the same clone class to a single fragment. The objective of clone refactoring is to improve the internal structure of the code-base so that the programmers can maintain it easily. The external behaviour of the software system should not be changed because of refactoring. A number of clone refactoring tools [27,35,39,67,83,87,122,139,151,239,245] currently exist. Given two clone fragments from a clone class, these tools can be used to assess their refactorability, and perform refactoring if they are refactorable.

Clone Categorization from Refactoring Perspective

A number of studies [26,219,268] have categorized code clones from the perspectives of refactoring. Balazinska et al. [26] proposed 18 categories of code clones on the basis of re-engineering/refactoring opportunities. They only considered method clones for categorization. Yu and Ramaswamy [268] categorized code clones into the following three categories: (1) singular concern clones, (2) cross-cutting concern clones, and (3) partial concern clones. They found that partial concern clones are not suitable for refactoring. Schulze et al. [219] categorized code clones for refactoring on the basis of location and type of code in the clone fragments. According to their analysis when clone fragments of a particular clone class are scattered throughout the code-base, AOR (Aspect Oriented Refactoring) is more appropriate for them compared to OOR (Object Oriented Refactoring).

Automatic Refactoring of Code Clones

Balazinska et al. [27] and Meng et al. [151] investigated fully automatic refactoring of code clones. While the tool called CLoRT introduced by Balazinska et al. [27] can refactor method clones only, Meng et al.'s [151] tool, RASE, can refactor code clones of fragment level granularity. Meng et al. also reports that automatic refactoring of code clones cannot eradicate the necessity of manual analysis from the experienced programmers.

Semi-automatic Refactoring of Code Clones

Most of the existing clone refactoring techniques [27, 39, 46, 67, 83, 84, 86, 105, 122, 139, 245, 252] are semi-automatic (i.e., they require user interactions for the actual implementation of refactoring). The most promising tool for seme-automatic clone refactoring is the one that was introduced by Tsantalis et al. [245]. It can automatically assess the refactorability of all major types (Type 1, Type 2, and Type 3) of code clones.

Scheduling for Clone Refactoring

After identifying the code clones for refactoring we can refactor them in different orders. Different refactoring orders will result different extents of gains in terms of system performance, and maintainability. The studies [38, 138, 273, 277] regarding refactoring scheduling propose different schedules (i.e., orders) for refactoring code clones with the goal of achiving the maximum gain while minimizing the refactoring effort.

2.4.2 Clone Tracking

There can be situations where refactoring of some clone fragments is impossible, however, they need to be updated consistently. Clone tracking is important in such situations. Clone tracking refers to the task of remembering all clone fragments from a clone class so that when a programmer attempts to change one fragment in the future, the system can remind her about the existence of the other clone fragments in the same clone class. The programmer can then decide whether these other clone fragments should also be updated consistently or not. Thus, clone tracking is important to update the code-base consistently. A number of clone trackers [61, 80, 95, 152, 210, 244] currently exist. These can track the evolution of the clone fragments in a clone class.

2.5 Evolutionary Coupling of Program Entities

If two or more program entities (such as files, classes, or methods) have evolved by changing together (i.e., by co-changing) during system evolution, then it is an implication that these entities are related and future changes in any of these entities might require the other entities to be changed consistently. In such a situation we say that these entities have evolutionary coupling [7, 71]. A number of studies [8, 14, 15, 28, 34, 41–43, 52, 71, 72, 79, 93, 94, 106, 107, 129, 159, 160, 165, 180, 182, 190, 191, 262, 265, 281] have investigated evolutionary coupling for identifying file level or method level co-change candidates. We can realize evolutionary coupling by using *association rules.*

2.5.1 Association Rule

An association rule [7] is an expression of the form $X \Rightarrow Y$ where X is the antecedent and Y is the consequent. Each of X and Y is a set of one or more program entities. The meaning of such a rule in our context is that if

X gets changed in a particular commit operation, Y also has the tendency of getting changed in that commit operation. We can determine the *confidence* or strength of a particular association rule by determining the *support* of its constituent parts.

2.5.2 Support and Confidence

Support is the number of commit operations in which an entity or a group (two or more) of entities changed together. We consider an example of two entities E1 and E2. If E1 and E2 have ever changed together, we can assume two association rules, $E1 \Rightarrow E2$ and $E2 \Rightarrow E1$, from them. Suppose, E1 changed in four commits: 2, 5, 6, and 10. E2 changed in six commits: 4, 6, 7, 8, 10, and 13. Thus, *support(E1) = 4* and *support(E2) = 6*. However, *support(E1, E2) = 2*, because E1 and E2 changed together in two commits: 6, and 10. Also, *support(E1 => E2) = support(E2 => E1) = support(E1, E2) = 2*.

Confidence of an association rule, $X \Rightarrow Y$, determines the probability that Y will change in a commit operation provided that X changed in that commit operation. We determine the confidence of $X \Rightarrow Y$ in the following way.

$$\textit{confidence}(X \Rightarrow Y) = \textit{support}(X,Y)/\textit{support}(X) \tag{2.1}$$

From the above example of two entities, *confidence (E1 => E2) = support(E1, E2) / support(E1) = 2 / 4 = 0.5* and *confidence(E2 => E1) = 2 / 6 = 0.33*. In our research, we extract and analyze association rules where each of X and Y consist of a single clone or a non-clone fragment. Such a rule can be expressed as $x \Rightarrow y$ where x and y can be: (1) two clone fragments from the same clone class, (2) two clone fragments from two different clone classes, or (3) a clone and a non-clone fragment. We investigate evolutionary coupling considering a finer granularity (fragment level granularity) compared to file level or method level granularities investigated in the existing studies [41, 52, 71, 72, 79]. Moreover, the existing studies on evolutionary coupling did not investigate identifying code clones that can be important for refactoring and tracking. We investigate this issue in our research considering fragment granularity.

Chapter 3
Automatic Ranking of Clones for Refactoring through Mining Association Rules

In this chapter, we present our study on identifying code clones that can be considered important for refactoring. We mine association rules (i.e., evolutionary coupling) among clones in order to detect clone fragments that belong to the same clone class and have a tendency of changing together during software evolution. The idea is that if two or more clone fragments from the same class often change together (i.e., are likely to co-change) by preserving their similarity, they might be important candidates for refactoring. Merging such clones into one (if possible) can potentially decrease future clone maintenance effort.

We define a particular clone change pattern, the **Similarity Preserving Change Pattern (SPCP)**, and consider the cloned fragments that changed according to this pattern (i.e., the SPCP clones) as important candidates for refactoring. By automatically analyzing the clone evolution history of a software system, we identify SPCP clones and mine evolutionary coupling (i.e., association rules) among these. We rank SPCP clones on the basis of the strength of their evolutionary coupling. We apply our implementation on thirteen subject systems and retrieve the refactoring candidates for three types of clones (Type 1, Type 2, and Type 3) separately. Our experimental results show that SPCP clones can be considered important candidates for refactoring. Clones that do not follow SPCP either evolve independently or are rarely changed. By considering SPCP clones for refactoring we not only can minimize refactoring effort considerably but also can reduce the possibility of delayed synchronizations among clones and thus, can minimize inconsistencies in software systems.

The rest of the chapter is organized as follows: Section II describes the significance of our study, Section III describes the terminology, Section IV elaborates on the SPCP (similarity preserving change pattern), Section V presents experimental results and discussion, Section VI mentions some threats to validity, Section VII discusses related work, and Section VIII contains the concluding remarks.

3.1 Introduction

Code cloning is a common yet controversial practice frequently employed by programmers during both development and maintenance of software systems. Cloning involves copying a code fragment from one place and pasting it in one or more additional places in the code-base with or without modifications causing the

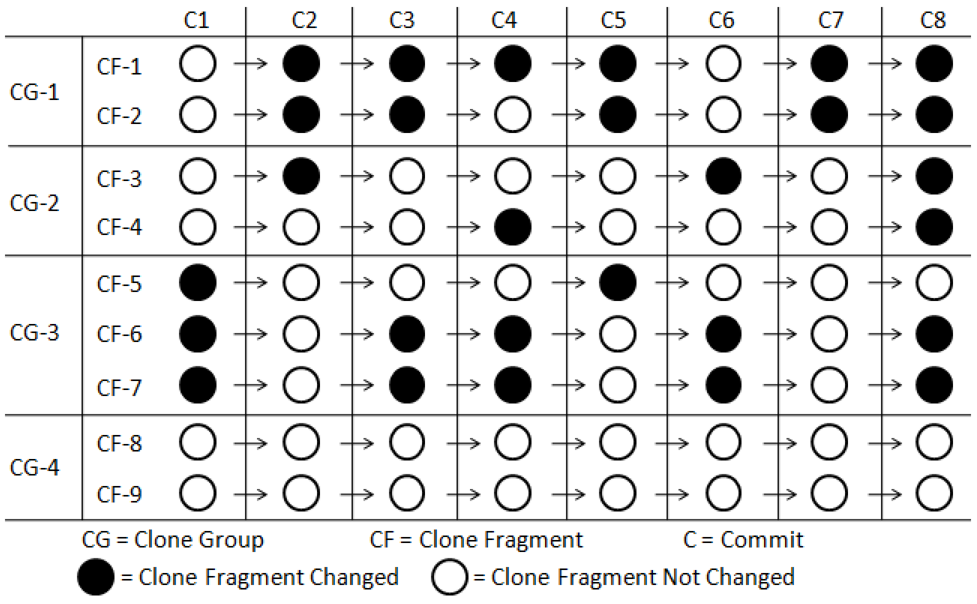


Figure 3.1: Change history of clone fragments

same or similar code fragments to be scattered throughout the system. The original code fragment (i.e., the code fragment from which the copies were created) and the pasted code fragments are clones of one another.

Numerous empirical studies [75, 104, 131–133, 145, 146, 209, 241] have been conducted identifying the possible impact of clones on software maintenance. While a number of studies [131–133] reported that clones are beneficial to both software development and maintenance, there is strong empirical evidence [104, 145, 146] of the negative effects of clones, including hidden bug propagation and unintentional inconsistent changes. Also, higher number of clones indicate higher change-proneness of the software systems [156] as well as higher maintenance effort and cost. The negative effects of clones indicate the necessity of clone refactoring. Clone refactoring refers to the task of merging several clone fragments (that are similar to one another) into a single one (if possible).

Motivation. A number of clone refactoring techniques [26, 105, 219, 239, 277] have been proposed by different studies. Before refactoring, it is important to identify the clones that are our primary refactoring candidates because, there can be a large number of clones in a system and not all of them need to be refactored [119]. That is, we should identify clones that would be important to refactor. Clone refactoring may not be able to be fully automated as it may require critical analysis by the programmer. Thus, a significant amount of effort and cost might sometimes need to be spent for refactoring clones. Identifying and ranking clones according to refactoring need can help us minimize refactoring effort and cost, because we are able to focus on just those clones that are important to be refactored, and we can leave the clones where refactoring is less important or unnecessary. However, there is no existing study that focuses on how we should identify and rank the important clones (from the whole set of clones in a code-base) for refactoring.

A number of studies [38, 138, 277] have investigated scheduling for clone refactoring activity. Given a number of clone fragments for refactoring, these studies aim to find an optimal schedule for refactoring tasks so that refactoring gain is maximized and refactoring effort is minimized. However, these studies cannot identify which of the huge number of code clones in a software system should be given a high importance for refactoring. Our research objective is to identify the important code clones for refactoring and ranking those. We believe that our study can complement the existing clone scheduling studies and techniques by pin pointing the most important ones to be scheduled.

According to our consideration, all clones in a software system might not need to be refactored with equal importance. We use the clone change example presented in Fig. 3.1 to explain this. We see that there are nine code fragments forming four groups (CG-1 through CG-4). The code fragments in a particular group are clones of one another. The change history (in Fig. 3.1) of these code fragments implies the following:

(1) The clone fragments, CF-1 and CF-2 in group CG-1, are likely to be related to each other and they have received corresponding changes. In other words, it is likely that the changes were made to these clones focusing on their consistency. The same is also true for the two code fragments CF-6 and CF-7 in CG-3.

(2) It is likely that the clone fragments, CF-3 and CF-4 in group CG-2, have experienced independent evolution. Because of the independent evolutions, CF-3 and CF-4 might not be regarded as clones of each other after a particular evolution period. Out of the three clone fragments in group CG-3, clone fragments CF-6 and CF-7 seem to be related as they received corresponding changes. However, the other clone fragment CF-5 seems to have a tendency of independent evolution.

(3) The clone fragments, CF-8 and CF-9, are not change-prone and it is likely that in future they will exhibit lower change-proneness compared to the others. Thus, these clones are not likely to add change effort during maintenance.

The example in Fig. 3.1 implies that refactoring clone fragments might not always be appropriate, since clones might evolve independently without preserving similarity. Also, if some clone fragments do not change during evolution we might not consider refactoring those clone fragments, since they do not require additional change effort during software evolution. Thus, when clone fragments belonging to the same clone class change consistently during evolution these clone fragments might be important refactoring candidates. Moreover, a clone class may contain n clone fragments, however, if it is observed that only a subset of the clone fragments change consistently while others are either evolving independently or are rarely changing we should consider refactoring only those consistently changing clone fragments. If the consistently changing clones can be merged through refactoring we can reduce the effort spent for changing clones. The co-change histories of the clones: (1) CF-1 and CF-2 in CG-1, and (2) CF-6 and CF-7 in CG-3 indicate that such clone fragments can be identified by mining association rules among clones.

Contribution. Focusing on the above discussion we define a particular clone change pattern called ***SPCP (Similarity Preserving Change Pattern)*** such that the clone fragments that change following

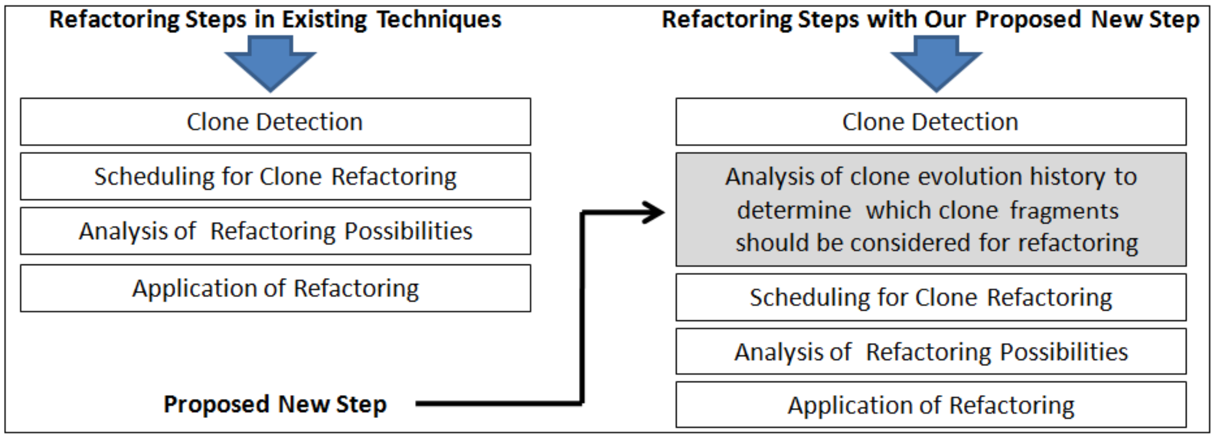

Refactoring Steps in Existing Techniques
Clone Detection
Scheduling for Clone Refactoring
Analysis of Refactoring Possibilities
Application of Refactoring
Proposed New Step
Refactoring Steps with Our Proposed New Step
Clone Detection
Analysis of clone evolution history to determine which clone fragments should be considered for refactoring
Scheduling for Clone Refactoring
Analysis of Refactoring Possibilities
Application of Refactoring

Figure 3.2: Proposed refactoring step

this pattern (i.e., SPCP clones) are likely to be important candidates for refactoring. We describe SPCP in Section 3.3. For the purpose of our study, we develop a prototype tool that can detect all SPCP clones from a subject system and then mines association rules among SPCP clones. It ranks these rules according to their support and confidence values (defined in Chapter 2). According to our investigation on the rules as well as SPCP clones retrieved by our prototype tool,

(1) *SPCP clones can be important candidates for refactoring. The clones that do not follow SPCP either evolve independently or are rarely changed during evolution. Thus, we can mainly focus on the SPCP clones for refactoring.*

(2) *Overall, only 7.04% of the clones existing in a code-base are SPCP clones. Also, for each of 63.44% of the association rules retrieved by our prototype tool, the corresponding SPCP clones are method clones and also, they belong to the same source code file. Thus, automatic identification and ranking of SPCP clones can save a considerable amount of time, effort, and cost for clone refactoring, because we could leave the remaining 92.96% of the clones in the code-base without refactoring.*

(3) *From our manual analysis on 224 association rules from all 13 subject systems (considering the top 10 rules of each clone-type), for overall 64.37% of the association rules we can suggest a standard refactoring technique [68].*

(4) *A considerable amount (overall 11.64%) of SPCP clones can receive resynchronizing changes. Thus, refactoring of SPCP clones can minimize delayed synchronizations among clone fragments and can minimize unwanted inconsistencies in software systems.*

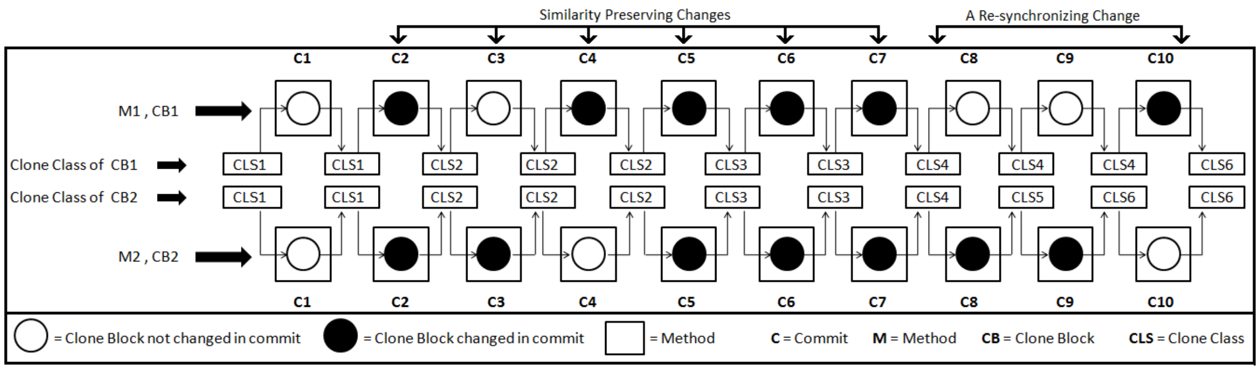


Figure 3.3: Similarity preserving change pattern (SPCP)

3.2 Significance of Our Study

Existing clone refactoring studies and techniques mainly consider four refactoring steps (cf. Fig. 3.2): (1) detection of clones, (2) scheduling of clones for refactoring (3) analysis of refactoring possibilities on the basis of different cloning situations, and (4) application of selected refactoring. Our study differs in that we propose an additional step after clone detection that involves the analysis of the clone evolution history to determine the clones we should consider as the primary refactoring candidates. According to our expectation, this additional step will minimize the clone refactoring effort and task considerably, because this step filters out a significant portion of clones from the refactoring target list based on their change pattern and change-proneness. According to our analysis the excluded clones do not need to be refactored with the same importance as the ones included in the target list, since they either changed rarely or changed independently.

3.3 Similarity Preserving Change Pattern

Our prototype tool mines association rules considering those clone fragments that followed a *similarity preserving change pattern (SPCP)* during evolution. It can also rank these rules on the basis of change-proneness (described in Section 3.3.4) of the participating clones. We define *similarity preserving change pattern* in the following way.

3.3.1 Definition of Similarity Preserving Change Pattern

If two clone blocks received either ***similarity preserving changes*** or ***re-synchronizing changes*** or both during evolution, then we say that these clone blocks follow a similarity preserving change pattern (i.e., are SPCP clones).

Definition of Similarity Preserving Change

We consider two clone blocks $CB1$ and $CB2$ that belong to the same clone class, say $CLS1$, in revision R_i. Suppose, a commit operation C_i on R_i changes any (one or both) of these clone fragments. If in revision R_{i+1} (created because of commit C_i) these two clone blocks, $CB1$ and $CB2$, again remain in one particular clone class (which might not be $CLS1$), then we say that $CB1$ and $CB2$ have received a ***similarity preserving change*** in commit operation C_i. If both of the clone blocks (i.e., $CB1$ and $CB2$) change preserving their similarity in such a commit then we call this change a ***similarity preserving co-change (SPCO)***. If the SPCP of two clone blocks contains SPCOs, then it is likely that the participating clone blocks have changed consistently (evaluated in Section 3.4.1).

Definition of Re-synchronizing Change

Suppose, two clone blocks $CB1$ and $CB2$ belong to the same clone class, $CLS1$, in revision R_i. The commit C_i on revision R_i modified any of these clone blocks in such a way that $CB1$ and $CB2$ could not be considered as clones of each other in revision R_{i+1} (i.e., $CB1$ and $CB2$ may diverge into two different clone classes or one or both of these might not be regarded as a clone fragment). However, in a later commit operation, say C_{i+n} where $n \geq 1$, any one or both of $CB1$ and $CB2$ changed in such a way that $CB1$ and $CB2$ become clones of each other (they converged into one clone class) again. Such a converging change following a diverging change is termed as a ***re-synchronizing change*** in our experiment.

3.3.2 Example of a Similarity Preserving Change Pattern

Fig. 3.3 contains an SPCP followed by the clone blocks, $CB1$ and $CB2$. As indicated in the figure, these clone blocks belong to two methods $M1$ and $M2$ respectively. We see that in each of the commit operations, C2 to C7, the clone blocks $CB1$ and $CB2$ received *similarity preserving change*. In commit C1, none of the clone blocks were changed. We now consider the commit operation C2. Before this commit both of the clone blocks (CB1 and CB2) belonged to the clone class CLS1. After this commit, the clone blocks belonged to CLS2. Although CLS1 and CLS2 are different clone classes, we see that before or after this commit operation both clone blocks belonged to a single clone class. Thus, CB1 and CB2 preserved their similarity after commit C2. In other words, CB1 and CB2 received a *similarity preserving change* in commit C2. We also see that these two clone blocks received *similarity preserving change*s in each of the commits C3 to C7. Moreover, the similarity preserving changes in commits C2, C5, C6, and C7, are SPCOs (similarity preserving co-changes). Finally, the changes in commits C8 to C10 can be considered as an example of re-synchronizing change. Because of the change in C8, $CB2$ diverged into a different clone class, CLS5, and thus, $CB1$ and $CB2$ could not be considered as clones of each other. However, the change in commit C10 is a converging change because, both of the clone blocks again converged into a single clone class after this commit.

3.3.3 Mining Association Rules Considering SPCP

In this experiment, we consider clones residing in methods. Thus, both fully cloned and partially cloned methods have been investigated. For a particular subject system, we collect all of its revisions (mentioned in Table 3.1), then extract the methods in each revision using CTAGS, and then determine method genealogies following the technique proposed by Lozano and Wermelinger [145]. We detect clones in each revision using NiCad [48] and then map the clones to the already detected methods of the corresponding revisions. As method genealogies are already detected, after clone mapping we can easily track the evolution of each clone fragment. Finally, we detect changes between every two consecutive revisions and map these changes to the methods as well as clones located inside the methods.

Extraction of Association Rules

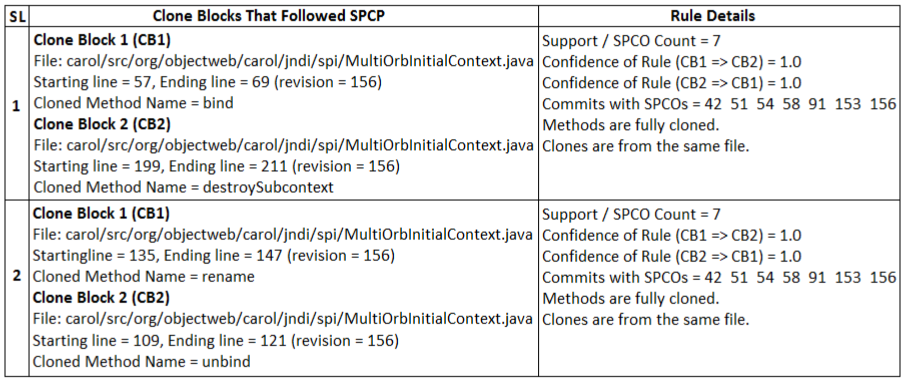
After the preliminary steps we determine all possible pairs of clones. A possible pair of clones consists of two clone fragments $CF1$ and $CF2$ from the same clone class such that they together followed an *SPCP* during the evolution and are alive in the last revision. As we map clones to methods, $CF1$ and $CF2$ reside in methods. From such a clone pair we can determine two association rules: $CF1$ => $CF2$, and $CF2$ => $CF1$. According to the definition, these two rules have the same support value. However, their confidences can be different. We determine support by the number of *similarity preserving co-changes (SPCOs)* in the SPCP followed by the clone fragments in a rule. We extract all association rules with a minimum support of 1.

3.3.4 Ranking of Association Rules

According to our consideration and discussion in the introduction, if a rule detected by our prototype tool has a higher support value (i.e., higher SPCO count) compared to the others, we should assume a higher priority for refactoring the associated clone fragments. Our decision of ranking considering the support value is reasonable from two perspectives.

(1) Higher support value (i.e., higher SPCO count) for a rule (consisting of two SPCP clone fragments) indicates a higher likelihood that the participating clone fragments have changed consistently during evolution. The underlying assumption is that *in a similarity preserving co-change (i.e., in a SPCO), the two participating SPCP clone fragments generally change consistently.* We empirically evaluate this in Section 3.4.1. Intuitively, if two clone fragments have a tendency of changing together consistently, then changes in one fragment in a particular commit operation generally require corresponding changes to the other one in that commit operation.

(2) Higher support value for a rule provides evidence that the corresponding SPCP clones have exhibited higher change-proneness (i.e., changed in higher number of commits) during the past evolution compared to the other SPCP clones in other rules. Thus, considering the existing evolution history we rank the



SL	Clone Blocks That Followed SPCP	Rule Details
1	**Clone Block 1 (CB1)**	Support / SPCO Count = 7
	File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java	Confidence of Rule (CB1 => CB2) = 1.0
	Starting line = 57, Ending line = 69 (revision = 156)	Confidence of Rule (CB2 => CB1) = 1.0
	Cloned Method Name = bind	Commits with SPCOs = 42 51 54 58 91 153 156
	Clone Block 2 (CB2)	Methods are fully cloned.
	File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java	Clones are from the same file.
	Starting line = 199, Ending line = 211 (revision = 156)	
	Cloned Method Name = destroySubcontext	
2	**Clone Block 1 (CB1)**	Support / SPCO Count = 7
	File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java	Confidence of Rule (CB1 => CB2) = 1.0
	Startingline = 135, Ending line = 147 (revision = 156)	Confidence of Rule (CB2 => CB1) = 1.0
	Cloned Method Name = rename	Commits with SPCOs = 42 51 54 58 91 153 156
	Clone Block 2 (CB2)	Methods are fully cloned.
	File: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java	Clones are from the same file.
	Starting line = 109, Ending line = 121 (revision = 156)	
	Cloned Method Name = unbind	

Figure 3.4: Rules of Type 3 clones sorted in decreasing order of support values

association rules according to the decreasing order of change-proneness of the corresponding SPCP clones. As it is difficult to be certain about the future change-proneness of the rules as well as SPCP clones, our decision of ranking relying on the past history is reasonable. We assume higher ranks for those rules as well as SPCP clones that exhibited higher change-proneness in the past. However, higher support might also be an indicator of higher change-proneness of the associated SPCP clones in future. We do not investigate this issue (i.e., future change-proneness) in this research work.

3.3.5 Finding Groups of Refactoring Candidates

Each rule consists of two SPCP clones. This is also possible that more than two clone fragments from the same class are preserving their similarity during evolution following a similarity preserving change pattern (SPCP). Thus, this is convenient to determine groups of refactoring candidates from these rules where a group contains two or more clones (from the same clone class) and all the clones in a group have evolved following a SPCP. Suppose two clone blocks, CB1 and CB2, formed a rule because they followed a SPCP. If there is another rule consisting of the clone blocks CB2 and CB3 (CB2 is common in these two rules), then we can form a group consisting of CB1, CB2, and CB3, because according to the conditions in Section 3.3.1 these three clone blocks together followed a SPCP. Focusing on this fact, we automatically determine groups of refactoring candidates. We term each group as a SPCP clone-class. In Table 3.2 we report the count of groups for each subject system considering each clone-type.

3.3.6 Tool Support

We have developed our prototype tool such that for a particular clone-type of a particular subject system, it generates two XML files. One file contains the ranked rules and the other one contains the SPCP clone groups merging these rules. While each rule in the first file contains only two SPCP clones, a group in the second file may contain more than two SPCP clones. For each SPCP clone (whether in a rule or in a group) we include the starting and ending line numbers of the clone fragment, the name of the method containing

Table 3.1: Subject Systems

	System	Domain	LOC	Revisions
Java	freecol	Game	91,626	1950
	jEdit	Text Editor	1,91,804	4000
	Plandora	Project Management	94,076	73
	Carol	Game	25,092	1699
	OpenYMSG	Yahoo Messenger	15,553	297
C	Ctags	Code Def. Generator	33,270	774
	QMail Admin	Mail Management	4,054	317
	GNUMake Uniproc	Auto-build System for C/C++ Projects	68,095	863
C#	MonoOSC	Formats & Protocols	14,883	355
	GreenShot	Multimedia	37,628	999
Python	Pyevolve	Artificial Intelligence	8.809	200
	Ocemp	Game	57,098	438
	Noora	Development Tool	14,862	140

Table 3.2: Statistics Regarding the Important Cloned Fragments (i.e., the SPCP Clones) Retrieved by Our Prototype Tool

		Type 1						**Type 2**						**Type 3**					
	System	**CC**	**CF**	**SCC**	**SCF**	**PSCF**	**CR**	**CC**	**CF**	**SCC**	**SCF**	**PSCF**	**CR**	**CC**	**CF**	**SCC**	**SCF**	**PSCF**	**CR**
Java	Freecol	93	251	27	56	22.31	31	95	259	28	63	24.32	49	255	799	78	180	22.52	147
	jEdit	1516	4284	3	8	0.18	7	115	531	27	62	11.67	46	399	2009	51	108	5.37	64
	Plandora	77	297	5	10	3.36	5	160	633	2	4	0.63	2	381	1748	19	48	2.74	44
	Carol	31	154	15	51	33.11	39	30	185	17	58	31.35	80	69	300	22	127	42.33	481
	OpenYMSG	10	24	3	8	33.33	7	10	22	0	0	0.0	0	40	112	7	16	14.28	10
C	Ctags	11	29	3	6	20.68	3	19	46	4	8	17.39	4	56	168	15	33	19.64	21
	QMailAdmin	15	49	3	6	12.24	3	11	43	1	2	4.65	1	13	77	4	8	10.38	4
	GNUMake Uniproc	144	308	3	6	1.94	3	28	69	0	0	0.0	0	57	199	2	5	2.51	3
C#	MonoOSC	4	21	3	6	28.57	3	3	6	1	2	33.33	1	9	23	8	17	73.91	10
	GreenShot	168	379	2	4	1.05	2	36	141	11	22	15.60	11	67	279	15	32	11.46	19
Py.	Pyevolve	52	117	0	0	0.0	0	13	54	3	6	11.11	3	25	123	4	11	8.94	13
	Ocemp	19	50	0	0	0.0	0	24	60	0	0	0.0	0	61	167	3	6	3.59	3
	Noora	43	114	4	18	15.78	37	8	40	0	0	0.0	0	26	126	3	10	7.94	12

CC = Number Clone Classes CF = Number of cloned fragments SCC = Number of SPCP Clone classes CR = Count of Rules
SCF = Number of cloned fragments that followed SPCP (i.e., the SPCP clones) PSCF = Percentage of SPCP clones

the clone fragment, and the starting and ending line numbers of this method considering the last (i.e., the latest) revision so that we can easily trace the clone fragment for refactoring in the latest revision of the candidate software system. The six XML files containing the association rules and groups of three types of SPCP clones of our subject system Freecol are available on-line [175].

In order to assist in analyzing the evolution of SPCP clones, our tool also shows important information regarding the ranked rules including (i) the list of commits where the SPCOs occurred, (ii) whether the participating clone blocks are method clones, (iii) whether the clones belong to the same file, (iv) the support and confidence values, (v) the file paths and starting and ending line numbers of the clone blocks and container methods in the revision where the last SPCO occurred. Such a ranking with all information regarding the SPCP clones is presented in Fig. 3.4 that shows the top two rules of total 481 rules retrieved for Type 3 case of our subject system Carol.

3.4 Experimental Results and Discussion

We applied our prototype tool on each of the thirteen subject systems listed in Table 3.1 and detected all SPCP clones and association rules & groups formed by these SPCP clones considering three clone-types (Type 1, Type 2, and Type 3) separately. We manually examined the changes occurred to the SPCP clones. According to our observation, implementation of the changes (*similarity preserving change*s and *resynchronizing change*s) required proper analysis by the responsible programmers. The changes were not tool generated and thus, these were not just reformatting or code transformations (e.g., replacement of for loops by for-each loops). In the following subsections and also in Fig. 3.6 we have mentioned and explained such changes. By analyzing our experimental results we answer the following five research questions.

RQ 1. Can we minimize clone refactoring effort and cost by considering SPCP clones for refactoring?

RQ 2. Are the changes occurring to the clone fragments in SPCOs (similarity preserving co-changes) consistency ensuring changes?

RQ 3. Can SPCP clones be candidates for refactoring?

RQ 4. What are the characteristics of the clones that change following similarity preserving change pattern (SPCP)?

RQ 5. What ratio of the SPCP clones do receive resynchronizing changes?

Statistics of the important refactoring candidates: Table 3.2 shows the statistics of the clone fragments that followed SPCP. We regard these cloned fragments as the important refactoring candidates. For each type of clones of a particular subject system we determine the followings considering all revisions: **(1)** Number of clone classes per revision (**CC**), **(2)** Number of clone fragments per revision (**CF**), **(3)** Number of SPCP clones that is, the number of clone fragments that followed *similarity preserving change pattern* during evolution (**SCF**), **(4)** Number of SPCP clone-classes (**SCC**), **(5)** The percentage of SPCP clones (**PSCF**), and **(6)** Number of association rules formed by the SPCP clones (**CR**). These six measures for each clone type are presented in Table 3.2. Looking at the percentages (**PSCF**) of the cloned fragments that followed SPCP (i.e., the important candidates for refactoring) we realize that *the number of important refactoring candidates can be considerably smaller compared to the total number of clone fragments in a system.*

From Table 3.2, for each clone type, we also determined the overall percentages of the SPCP clones (denoted by *Overall-PSCF*) considering all subject systems according to the following equation.

$$\textit{Overall-PSCF}_{Ti} = \frac{\sum_{for\ all\ systems} SCF_{T1}}{\sum_{for\ all\ systems} CF_{T1}} \tag{3.1}$$

Here, Ti (i = 1, 2, or 3) denotes a particular clone-type (Type 1, Type 2, or Type 3), SCF_{Ti} denotes the number of SPCP clones of a particular type (Ti) of a particular subject system, and CF_{Ti} stands for the total number of clones of type Ti in a particular system. We also calculate the overall percentage of

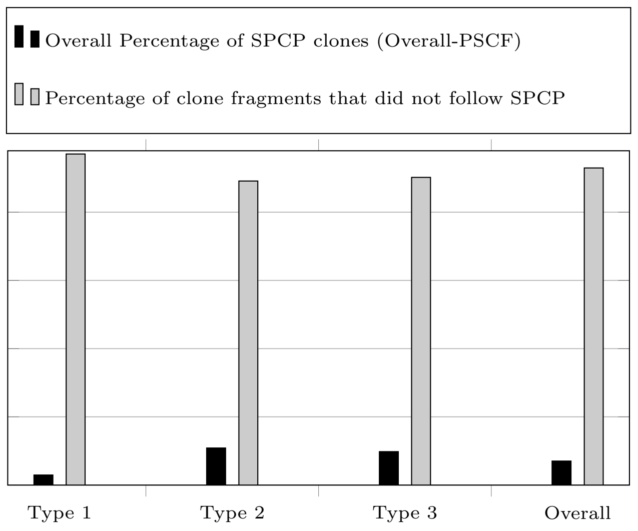


Figure 3.5: Overall percentages of cloned fragments that followed SPCP

SPCP clones considering all subject systems and all clone-types in a similar way. These overall percentages (Overall-PSCFs) are shown Fig. 3.5. From this graph it is clear that *a considerable amount (i.e., 97.05%, 89.13%, 90.2%, and 92.96% for Type 1, Type 2, Type 3, and overall case respectively) of clone fragments (i.e., the clone fragments that did not follow SPCP) can be filtered out from our consideration during refactoring because, these cloned fragments either evolved independently or rarely changed. This reduction in the number of considerable refactoring candidates can minimize refactoring effort and cost.*

We answered the following four research questions by analyzing the SPCP clones and rules retrieved by our prototype tool.

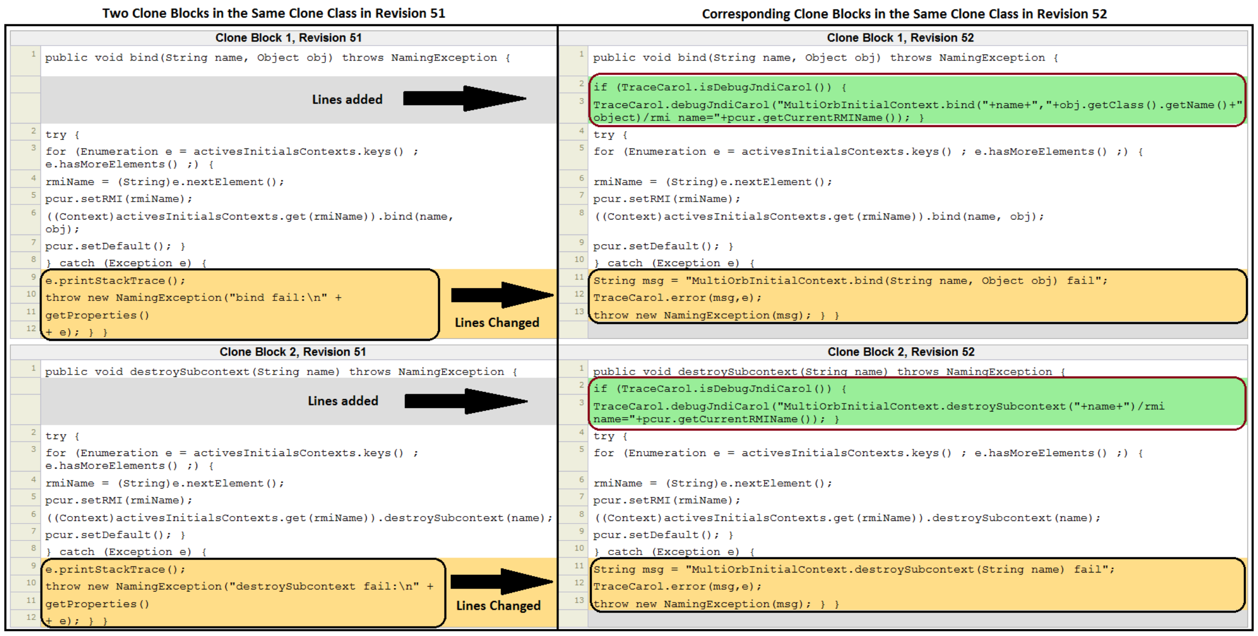
3.4.1 Answer to Research Question RQ 1: *Are the changes occurring to the clone fragments in SPCOs (similarity preserving co-changes) consistency ensuring changes?*

To answer this question, we manually analyzed 485 SPCOs occurred in the SPCPs of 150 rules (considering Type 3 case of Carol) involving 73 SPCP clones. For each of 100 rules, the support value (SPCO count) was greater than 1 (highest support = 7). Each of the remaining rules had a support of 1.

We know that the clone blocks in each of the rules retrieved by our prototype tool followed an SPCP. The support value of a rule is equal to the number of SPCOs in the SPCP of the rule. While examining the SPCP of a rule our tool stores:

(1) The list of commit(s) where the SPCO(s) occurred.

(2) The start and end line numbers of each of the participating clone blocks before and after every commit in the list of commits obtained in Step 1.


Two Clone Blocks in the Same Clone Class in Revision 51
Corresponding Clone Blocks in the Same Clone Class in Revision 52
Lines added
Lines Changed
Lines added
Lines Changed

Clone Block 1, Revision 51

```
public void bind(String name, Object obj) throws NamingException {
try {
for (Enumeration e = activesInitialsContexts.keys() ;
e.hasMoreElements() ;) {
rmiName = (String)e.nextElement();
pcur.setRMI(rmiName);
((Context)activesInitialsContexts.get(rmiName)).bind(name,
obj);
pcur.setDefault(); }
} catch (Exception e) {
e.printStackTrace();
throw new NamingException("bind fail:\n" +
getProperties()
+ e); } }
```

Clone Block 1, Revision 52

```
public void bind(String name, Object obj) throws NamingException {
if (TraceCarol.isDebugJndiCarol()) {
TraceCarol.debugJndiCarol("MultiOrbInitialContext.bind("+name+","+obj.getClass().getName()+"
object)/rmi name="+pcur.getCurrentRMIName()); }
try {
for (Enumeration e = activesInitialsContexts.keys() ; e.hasMoreElements() ;) {
rmiName = (String)e.nextElement();
pcur.setRMI(rmiName);
((Context)activesInitialsContexts.get(rmiName)).bind(name, obj);
pcur.setDefault(); }
} catch (Exception e) {
String msg = "MultiOrbInitialContext.bind(String name, Object obj) fail";
TraceCarol.error(msg,e);
throw new NamingException(msg); } }
```

Clone Block 2, Revision 51

```
public void destroySubcontext(String name) throws NamingException {
try {
for (Enumeration e = activesInitialsContexts.keys() ;
e.hasMoreElements() ;) {
rmiName = (String)e.nextElement();
pcur.setRMI(rmiName);
((Context)activesInitialsContexts.get(rmiName)).destroySubcontext(name);
pcur.setDefault(); }
} catch (Exception e) {
e.printStackTrace();
throw new NamingException("destroySubcontext fail:\n" +
getProperties()
+ e); } }
```

Clone Block 2, Revision 52

```
public void destroySubcontext(String name) throws NamingException {
if (TraceCarol.isDebugJndiCarol()) {
TraceCarol.debugJndiCarol("MultiOrbInitialContext.destroySubcontext("+name+")/rmi
name="+pcur.getCurrentRMIName()); }
try {
for (Enumeration e = activesInitialsContexts.keys() ; e.hasMoreElements() ;) {
rmiName = (String)e.nextElement();
pcur.setRMI(rmiName);
((Context)activesInitialsContexts.get(rmiName)).destroySubcontext(name);
pcur.setDefault(); }
} catch (Exception e) {
String msg = "MultiOrbInitialContext.destroySubcontext(String name) fail";
TraceCarol.error(msg,e);
throw new NamingException(msg); } }
```

Figure 3.6: Changes to two Type 3 clone fragments of Carol in commit 51

Suppose the clone blocks regarding a particular rule are $CB1$ and $CB2$ respectively and they received an SPCO in commit C_i applied on revision R_i. Then, for each of these clone blocks we collect the snapshot in revision R_i and the snapshot in revision R_{i+1} using the line numbers. For each clone block we determine the differences of the corresponding snapshots. Then, we manually compare the changes that occurred to $CB1$ with those that occurred to $CB2$ and decide whether the changes were consistent or not.

Investigation details. Among 485 SPCOs that we analyzed manually, in 477 SPCOs (98.35%), the changes to the participating clone blocks (i.e., SPCP clones) were consistent. According to our observation, in each of these 477 SPCOs, the corresponding lines (the same or similar lines) of the two participating SPCP clones were changed in the same or similar way. Thus, the changes in these 477 SPCOs can be termed as consistency ensuring changes to clones. In each of the remaining eight SPCOs, the changes to the clone blocks were not consistent. However, the associated clone blocks still preserved their similarity even after these eight SPCOs.

An example of the consistency ensuring changes that occurred to two clone blocks (corresponding to a rule) in an SPCO on commit 51 of Carol is presented in Fig. 3.6. The method *bind* and the method *destroySubcontext* in Fig. 3.6 are Type 3 clones (full method clone) of each other according to the clone detection results of NiCad. From Fig. 3.4 we see that the rule consisting of these two clone blocks has a support of 7. That is, these two clone blocks received seven SPCOs (i.e., similarity preserving co-changes) in their SPCP (i.e., similarity preserving change pattern). These SPCOs occurred in commits: 42, 51, 54, 58, 91, 153, and 156 respectively. The details of the co-changes in commit 51 (Fig. 3.6) demonstrate that the changes occurred to the two clone blocks (i.e., method clones), *bind* and *destroySubcontext*, are consistent.

As demonstrated in the figure, almost the same if-blocks were added just after the first line in each of these method clones in revision 52. The only differences in these two if-blocks are in the method names and parameters. Also, lines 9 to 12 (in revision 51) in each of the method clones were changed in the same way as can be seen in revision 52. We observed the changes occurred to the method clones in the other six SPCOs too. The changes in each of these SPCOs were also consistent.

We were also interested in identifying the types of changes that mostly occur to the SPCP clones during similarity preserving co-changes (SPCOs). The dominant change types were: addition or deletion of the same or similar statements in both clone fragments; modification of the same or similar corresponding statements in both clone fragments in the same way (e.g., the SPCO occurred in commit 91 on method clones *bind* and *destroySubcontext*); and, addition of the same or similar if-else blocks.

Answer. Thus, *the changes occurring to the SPCP clone fragments in similarity preserving co-changes (SPCOs) generally ensure consistency between the clone fragments. Thus, the higher number of SPCOs in a rule indicates a higher probability that the participating SPCP clones are related and will also change consistently in future commits. So, ranking of rules for refactoring according to the SPCO count is reasonable.*

3.4.2 Answer to Research Question RQ 2: *Can SPCP clones be candidates for refactoring?*

For answering this research question we manually examined the rules and groups of SPCP clones to determine whether we can suggest particular refactoring for the SPCP clones included in a rule or a group (defined in Section 3.3.5). For each clone-type of a subject system we considered the top 10 rules and groups (for the cases with less than 10 rules or groups we considered all) totaling 224 rules and 191 groups from all 13 subject systems. Through our manual analysis on these rules and groups, we determine how many of these are refactorable using standard refactoring mechanisms such as *pull up method*, *extract method*, *remove method*, *parameterize method*, *replace conditional with polymorphism* etc [68]. Then, for each type of clone we determine overall percentage of refactorable rules and groups considering all subject systems. Overall percentages were calculated following a equation similar to Eq. 3.1. Finally, we determine the overall percentage considering all clone-types and subject systems. These percentages are shown in Fig. 3.7.

Fig. 3.7 shows that the proportions regarding Type 1 case are the highest ones compared to the other two cases (Type 2, Type 3). As Type 1 clone fragments are exactly similar clone fragments, it was easier to suggest refactoring techniques for them compared to the other two types. Type 2 clones are syntactically similar with variable renaming and/or changes in data types. According to our investigation some Type 2 clones with different data types were not refactorable. We get the lowest proportions for the Type 3 case, because Type 3 clones often had dissimilar code fragment and we could not suggest refactoring for those. Overall, for 64.37% of the association rules (64.62% of the groups), we could suggest refactoring techniques for the participating SPCP clones. Here, we should mention that the groups are formed from the rules (c.f., Section 3.3.5). As a group may contain more than two SPCP clones while a rule contains only two, the top

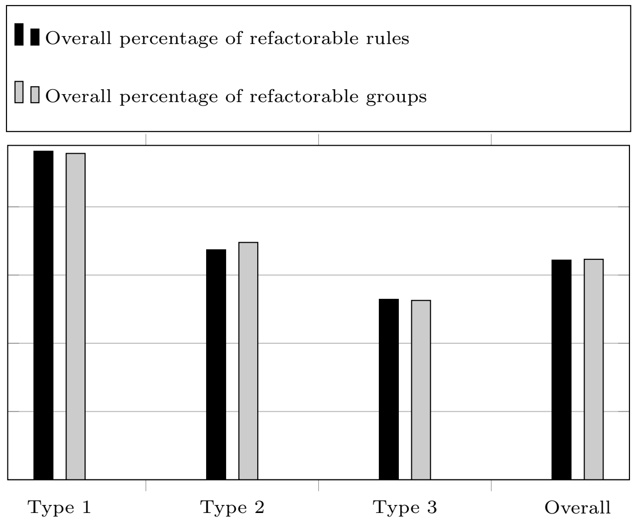


Figure 3.7: Overall percentages of refactorable rules and groups

10 groups regarding a particular clone type of a particular subject system sometimes had more SPCP clones compared to the corresponding top 10 rules. Thus, the percentages regarding the rules and groups are almost the same with little differences.

However, from our investigation we realize that although we could not suggest refactoring for some rules and groups, the participating clone blocks in these rules and groups might often need to be consistently changed. As we generate XML files containing the rules and groups, these files can suggest co-change candidates for any future change in any of the SPCP clones. While changing a particular SPCP clone, the developer can look at the other SPCP clones in the same group to determine whether the changes need to be propagated to these clones too. However, we have not yet automated this feature. We also do not investigate the predictability of future co-change candidates in this research work.

Example of a refactorable rule. As an example, consider the rule (with support / SPCO count of 7) consisting of the method clones, *bind* and *destroySubcontext*, mentioned in Section 3.4.1. The confidence of each of the rules, *bind* => *destroySubcontext* and *destroySubcontext* => *bind*, is one. Thus, these methods (i.e., method clones) always co-changed (i.e., in 7 commits); that means, there is no commit where one changed but the other did not. These methods are almost the same. The only differences are in the method names and parameters (*bind* takes an extra parameter 'obj' of Object type). The first commit where these methods co-changed was applied on revision 42. In this commit, the same statement '*e.prntStackTrace();*' was added after the eighth line of each method. In commit 51 (as mentioned in Section 3.4.1), almost the same if-blocks were added after the first line of each method. Also, the lines 9 to 12 in each of these methods where changed in a similar way (Fig. 3.6) in this commit. The only differences in the added if-blocks and changed lines were in the names of the methods and parameters. In the commit on revision 54, the same changes occurred at the third line of each method. In the same way, in each of the other commits (58, 91, 153, 156) these two methods received the same changes at the same relative line numbers. The changes obviously indicate that those were made focusing on the consistency of these method clones. Now we discuss the possibility of refactoring these method clones (i.e., merging these into one).

We already mentioned that the method *bind* takes one extra parameter *obj* of type *Object*. The other parameter is the same (same name and type) as that of the method *destroySubcontext*. These two method clones remain in the same file[1] and in the same class (Class Name: *MultiOrbInitialContext*). According to our analysis, it is possible to replace these two methods with a single one that takes the two parameters. The callers of *destroySubcontext* can call it using an extra dummy object. Focusing on this possibility we identified the places where *destroySubcontext* was called. We found two places in the same file[1] where both *bind* and *destroySubcontext* remain (and no other places in the code-base) and determined that we can add an extra dummy object in these calls. This is also possible that the new method that will replace the old ones will take an additional context-sensitive string parameter (a third parameter) for printing purpose. We saw that each of these methods, *bind* and *destroySubcontext*, prints an error message containing the method name. The only difference is the method name used in the message. We can replace this method name by the context sensitive string parameter (that can determine what type of failure has occurred depending on the caller) if necessary. So, the rule consisting of the method clones, *bind* and *destroySubcontext*, is an important refactoring candidate.

Example of a refactorable group. An example group of Type 3 clones in Carol contains 4 method clones: *list*, *listBindings*, *rebind*, and *unbind*. These method clones remain in the same file[2], under the same package and also in the same class (Class Name: javaURLContext). According to our analysis, these method clones are important candidates for refactoring and can surely be replaced with a single method. Three of these methods (*list*, *listBindings*, *unbind*) perform exactly the same task taking the same parameter. The remaining one takes an extra parameter. It is possible that we decide to only refactor the first three method clones. For refactoring all four, one can follow the technique described in our previous example.

Answer. Finally, in answer to the second research question we can say that *a considerable amount of association rules (overall 64.37%) and groups (overall 64.62%) formed by the SPCP clones can be refactored using standard refactoring techniques. Thus, SPCP clones can be considered important candidates for refactoring.* From our experience we realize that the refactoring task often requires proper analysis by the expert users. Also, refactoring can be time consuming because the user might need to analyze the evolution history of the target clones. Thus, automatic identification and ranking of important refactoring candidates (i.e., SPCP clones) can help us minimize refactoring time and effort.

[1]File Path: carol/src/org/objectweb/carol/jndi/spi/MultiOrbInitialContext.java
[2]File Path: carol/src/org/objectweb/carol/jndi/enc/java/javaURLContext.java

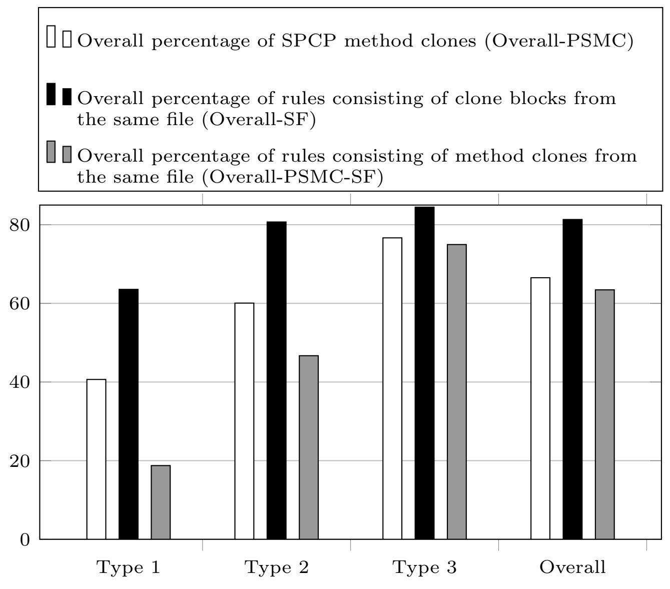


Figure 3.8: Statistics regarding SPCP clones and rules

3.4.3 Answer to Research Question RQ 3: *What are the characteristics of the clones that change following similarity preserving change pattern (SPCP)?*

During manual examination of the rules from Carol we observed that most of the rules consist of method clones (i.e., clone fragments that are full methods). Also, a recent study conducted by Göde [74] demonstrates that developers generally consider removing clones that belong to the same source code file. Considering these two perspectives we determined the followings to answer this research question.

(1) The overall proportion of SPCP clones that are full methods (denoted by *Overall-PSMC*).

(2) The overall proportion of rules where each rule consists of clones belonging to the same file (denoted by *Overall-SF*).

Overall proportions were calculated using a mechanism similar to the one demonstrated in Eq. 3.1. After calculating the overall percentages for each clone-type individually, we also determine the overall proportions considering all clone types. Finally, we calculate the overall proportions of rules consisting of SPCP method clones from the same file (denoted by *Overall-PSMC-SF*). These proportions are shown in Fig. 3.8.

From Fig. 3.8 we see that each of the three measures, Overall-PSMC, Overall-SF, and Overall-PSMC-SF appear in an increasing order from Type 1 case to Type 3 case. Also, Overall-SF is above 60% for each type of clone. Although, Overall-PSMC is below 50% for Type 1 case, this value is above 50% for the other two types with Type 3 case having the highest value (76.66%). From Fig. 3.5 we see that the percentage of clone fragments that follow SPCP is lowest for Type 1 case. Ultimately, in Fig. 3.8 the three bars belonging to overall case (considering all clone types and all subject systems) are mainly influenced by Type 2 and Type 3 rules and the values of all three measures are above 60% for this case.

Answer. According to our experimental result considering all systems and all three clone-types we can draw the following general conclusions: **(1)** *most of the SPCP clones (overall 66.52%) are method clones* and **(2)** *most of the rules (overall 63.44%) consist of SPCP method clones from the same file.*

In this experiment we detected block clones using NiCad. However, NiCad also facilitates the detection of method clones only. In general, clones in the same file might be easier to be refactored. Clones that belong to different files or folders might require the creation of a separate library for refactoring. This may be difficult without programming language support. Thus, SPCP clones that belong to the same file can be promising refactoring candidates. According to our observation, *we can mainly focus on detecting and refactoring SPCP method clones belonging to the same file.*

3.4.4 Answer to Research Question RQ 4: *What ratio of the SPCP clones do receive resynchronizing changes?*

We have already mentioned that SPCP clones can receive two types of changes: *similarity preserving changes* and *resynchronizing changes* (elaborated in Section 3.3.1). Intuitively, resynchronizing changes indicate delayed synchronization among the clone fragments. Delay in synchronization might introduce temporary inconsistency to the functionality of the software systems. We calculated the overall proportions of the SPCP clones that received resynchronizing changes (i.e., delayed synchronizations). For the purpose of calculation we automatically examine the entire evolution histories of the two participating SPCP clones of each association rule and determine whether they received resynchronizing change(s). These proportions are shown in the graph of Fig. 3.9. We see that overall 6.45%, 10.20%, and 13.57% of the Type 1,Type 2, and Type 3 SPCP clones received resynchronizing changes considering all subject systems. If we consider all subject systems and clone-types, this percentage becomes 11.64%.

Answer to RQ 4. According to our investigation, *a considerable amount of the SPCP clones can receive resynchronizing changes during evolution. As delay in synchronizations may introduce temporary inconsistency to the software systems, it is important to identify SPCP clones and refactor them.*

3.5 Threats to Validity

In our experiment we detected clones using NiCad [48]. For different settings of NiCad, the clone detection results may be different. Thus, there might be variations in the association rules as well as the SPCP clones detected for different NiCad settings. However, the settings used in our experiment are considered standard [196,209]. Thus, we think that our findings are significant and can help us minimize clone refactoring

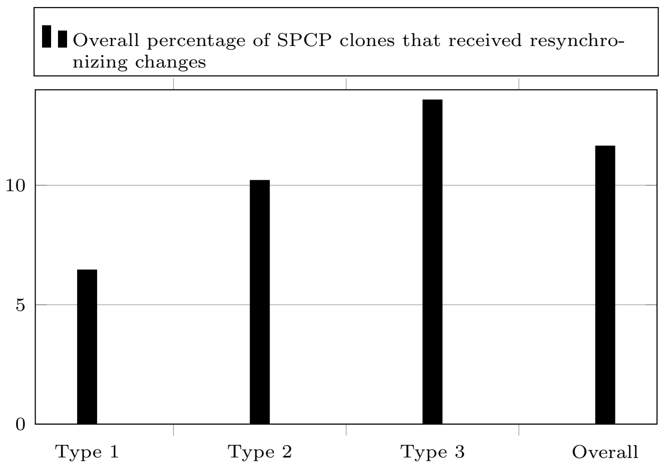


Figure 3.9: Proportions of SPCP clones that received resynchronizing changes

effort considerably. Moreover, the subject systems that we have used in our experiment are of diverse variety in terms of application domains, implementation languages, sizes, and revisions. Thus, we expect that our findings are not biased.

3.6 Related Work

Numerous studies have been conducted regarding the detection, impact analysis [40, 75, 78, 104, 119, 131–133, 145, 146, 208, 241, 259], management [241, 251] and refactoring [25, 74, 105, 219, 239, 277] of clones. As our experiment is centered on clone refactoring, we discuss clone refactoring related studies below.

A number of refactoring approaches [25, 105, 219] select clones for refactoring on the basis of the abstract syntax tree representation of the code base. Higo et al. [83] selected clones for refactoring (implementing a tool called CCShaper) based on the lexical analysis of the source code. Zibran and Roy [277] proposed a conflict aware optimal scheduling algorithm for clone refactoring on the basis of constraint programming. They showed that their scheduling algorithm is superior to the other algorithms those are based on genetic algorithm approaches, greedy approaches and linear programming. Bouktif et al. [38] considered the clone refactoring problem as a constrained knapsack problem where the knapsack consists of all the clones to be refactored. They found an optimal schedule for refactoring the clones in the knapsack by applying a genetic algorithm. Göde [74] performed a case study to determine why clones are removed from the code base. According to his observation, developers often consider removing clones residing in the same source code file. Tairas and Gray [239] developed an Eclipse plug-in, CeDAR, that can forward the detected clones to the Eclipse refactoring engine. The Eclipse engine then handles the refactoring decisions.

We see that none of the existing studies and techniques focused on our proposed refactoring step (c.f., Fig. 3.2) involving the determination of clones that should be considered as important refactoring candidates. Automatic identification of important refactoring candidates can help us minimize refactoring time and effort. We define a particular clone change pattern, SPCP (Similarity Preserving Change Pattern), and show that the clones that changed following this pattern can be important candidates for refactoring. The clones excluded by this pattern either evolved independently or changed rarely during the evolution of the subject system. Thus, these clones should not be our primary targets for refactoring. Our prototype tool can detect all SPCP clones in a system.

3.7 Summary

In this chapter, we present our empirical study on identifying code clones that can be considered as important refactoring candidates. We define a particular clone change pattern, SPCP (Similarity Preserving Change Pattern), such that the clones that changed following this pattern during evolution can be considered as important candidates for refactoring. For the purpose of our study, we implement a prototype tool that mines association rules among clones that follow SPCP.

Using our prototype tool we detected all SPCPs in each of our 13 candidate subject systems considering three clone-types (Type 1, 2, and 3). We also detect association rules among the SPCP clones and rank these SPCP clones for refactoring on the basis of the support values of the rules. More importantly, we determine groups of refactoring candidates by merging the association rules where a group can contain two or more clones that together followed a SPCP. According to our experimental results and manual investigation, we have the following concluding remarks and suggestions.

(1) SPCP clones are important candidates for refactoring. The clones that do not follow SPCP either change independently or are rarely changed. Thus, while taking refactoring decision we suggest to mainly focus on SPCP clones.

(2) On an average only 7.04% of the clones existing in a code-base are SPCP clones. Thus, we can filter-out a significant amount (92.96%) of clones from our refactoring decision. We observe that in case of 63.44% of the association rules (formed by the SPCP clones), the two participating SPCP clones are method clones and moreover, these method clones belong to the same source code file. Thus, automatic identification and ranking of SPCP clones can help us minimize a considerable amount of clone refactoring effort and cost. In presence of ranking, we can decide to primarily refactor the more important SPCP clones.

(3) According to our manual investigation on 224 rules and 191 groups considering all 13 subject systems, overall 64.37% of the rules and 64.62% of the groups can be refactored using standard refactoring techniques.

(4) Refactoring of SPCP clones can minimize the possibility of delayed synchronizations among clone fragments and thus, can minimize unwanted inconsistencies in software systems.

While our investigation in Chapter 3 involves analyzing evolutionary coupling of code clones with a goal of assisting clone refactoring, we further investigate whether we can utilize clone evolutionary coupling in order to assist clone tracking. Chapter 4 contains the details of this investigation.

Chapter 4
Prediction and Ranking of Co-change Candidates for Clones

In the previous chapter (i.e., Chapter 3) we identified code clones that can be considered important for refactoring. We ranked the important refactoring candidates by analyzing their evolutionary coupling (association rules). Clone refactoring is one way of clone management. There can be situations where refactoring of clone fragments from a clone class is impossible, however, the fragments in the class need to be updated together consistently. Clone tracking (defined in Section 2.4.2 of Chapter 3) is important in such situations. Our study presented in Chapter 4 focuses on clone tracking. In this study we again analyze evolutionary coupling of clone fragments in a clone class in order to identify and rank fragments having tendencies of co-changing (i.e., changing together) consistently.

The rest of the chapter is organized as follows. Section 4.2 describes the terminology, Section 4.3 discusses the experimental steps, we present and analyze our experimental results in Section 4.4, Section 4.5 mentions some possible threats to validity, Section 4.6 elaborates on the related work, and Section 4.7 concludes the chapter by mentioning our study findings.

4.1 Introduction and Motivation

Let us consider a clone class that contains a number of clone fragments. We assume that this class is being tracked by a clone tracker so that when a programmer attempts to make changes to a particular fragment in this class, the tracking system can automatically notify the programmer about the existence of the other fragments in the same class. The programmer can then also change these other clone fragments consistently with the fragment that she is going to change. We call these other fragments the co-change (changing together) candidates of the clone fragment that is going to be changed. However, all the clone fragments in a clone class might not need to be changed together consistently because clone fragments might evolve independently. Thus, it is important to have a prior knowledge about which other clone fragments in a clone class need to be consistently co-changed (i.e., need to be changed together consistently) while changing a particular clone fragment in that class. Without such prior knowledge a developer can be overwhelmed while dealing with a clone class with a large number of clone fragments (such as 76 clones in a class of our candidate system jEdit) and might need to spend a significant amount of time and effort in understanding

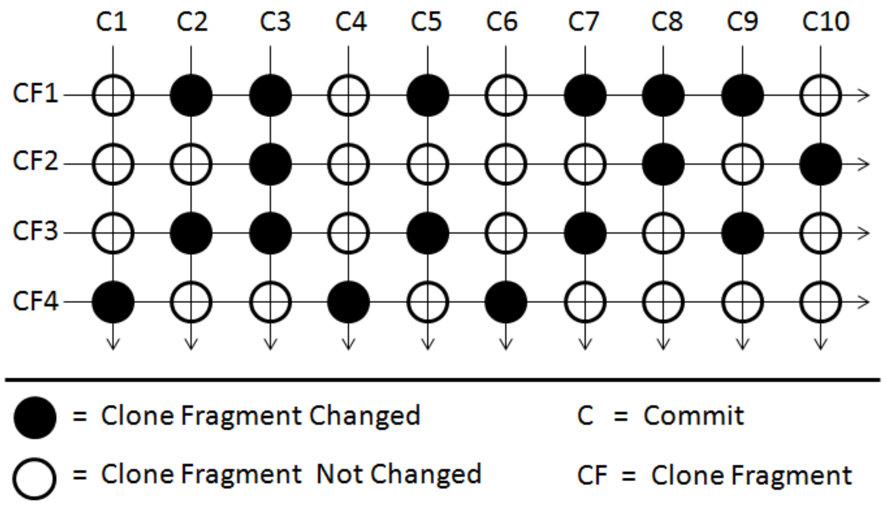


Figure 4.1: Evolution history of four clone fragments from the same clone class

and determining which other clone fragments in the clone class need to be consistently co-changed. Focusing on this issue we propose to automatically rank the other clone fragments (i.e., the co-change candidates) in a clone class according to their probability of being co-changed with the particular clone fragment that is going to be changed from that class by a programmer. For the purpose of ranking, we automatically analyze the evolution history of the clone fragments in the class, and mine evolutionary coupling among the fragments. Our idea of ranking co-change candidates will be described later.

4.1.1 Importance of Our Study with Respect to the Existing Studies

There is no existing study on ranking of the co-change candidates for clones. There is an existing tool called *CloneTracker* [61] that presents the other clones in a clone class while a programmer changes a particular clone fragment in that class. However, this tool does not support ranking of the clone fragments by their need to co-change consistently. There are also several other clone tracking tools [95, 152, 244]. However, none of these tools supports ranking of co-change candidates for clones. There is no previous study on the possibility of ranking the co-change candidates for clones to assist programmers in dealing with consistent updates of clone fragments. Thus, our study presented in this chapter is unique.

4.1.2 The Underlying Idea of Ranking

Our idea of ranking on the basis of the past evolutionary history is illustrated in Fig. 4.1. We can see the evolution history of four clone fragments *CF1*, *CF2*, *CF3*, and *CF4* in a particular clone class *CC* through ten commit operations (*C1* to *C10*). The clone fragments *CF1* and *CF3* changed together (i.e., co-changed) most times. In other words, these two clone fragments have a high tendency of changing together. Thus, it is highly probable that these two clone fragments have co-changed consistently [163]. Also, it is likely that a future change in any one of these two fragments will accompany a corresponding change in the other one. *CF2* has rarely co-changed with *CF1* and *CF3*. *CF4* has never co-changed with any other fragments in its class. In this case, it is likely that *CF4* has a tendency of experiencing independent evolution. After a certain period of evolution, *CF4* might not be considered as a clone in the clone class *CC*.

Given this evolutionary history, if a developer makes a change to the clone fragment *CF1* at a later time, he/she should at first look at *CF3* to check whether *CF3* needs a corresponding change, because *CF3* has co-changed with *CF1* most frequently. CF2 has a comparatively lower probability (compared to CF3) of getting a corresponding change, because *CF2* has co-changed with *CF1* less frequently. Finally, CF4 has the lowest probability. Thus, the co-change candidates of *CF1* can be ordered as: *CF3*, *CF2*, *CF4* according to their tendency of getting co-changed with *CF1*. In this way, as a software system evolves, we can store the past co-change history of the clone fragments and infer this history to make decisions regarding co-change candidates in the future. In this example, we present a ranking on the basis of co-change frequency of *CF* with the other clone fragments. However, we also investigate ranking on the basis of co-change recency (i.e., how lately the other clone fragments co-changed with *CF*). We describe this in Section 4.4.2.

The co-changing tendency of the related program entities (such as files, classes, methods) is known as evolutionary coupling in the literature [71]. We apply the concept of evolutionary coupling in ranking the co-change candidates for clones. Evolutionary coupling will be discussed in Section 4.2.

4.1.3 Findings

We performed our investigation on six subject systems written in two different programming languages (Java and C) and answer four important research questions listed in Table 4.1. According to our observation, ranking of co-change candidates for clones is very important because a clone class may contain a large number of clone fragments. The sizes of the largest clone classes of our subject systems Ctags, QMailAdmin, jEdit, Freecol, Carol, and Jabref are respectively 22, 9, 57, 76, 40, and 65. Also, considering all the systems overall, 4% of the clone classes contain more than 10 clone fragments. Thus, we believe that automatic ranking of co-change candidates for clones can help programmers identify which other clone fragments from a clone class actually need to be co-changed while changing a particular clone in that class with significantly less effort and time. Even if a clone class contains only a few clones (such as three or four), our ranking mechanism can still help programmers by pin-pointing the most likely co-change candidates for a particular clone fragment being changed. According to our experimental results and analysis we can state that:

While changing a particular clone fragment CF from a particular clone class, we can automatically rank its possible co-change candidates (i.e., the other clone fragments in the same clone class) according to its co-change tendency with them. For ranking we automatically retrieve and infer the evolutionary coupling of CF with its possible co-change candidates from the previous evolution history. We propose a composite ranking mechanism for ranking the possible co-change candidates of CF. Our empirical study shows that our proposed ranking mechanism can assign higher ranks to the actual co-change candidates (i.e., the other clone fragments

Table 4.1: Research Questions

SL	Research Question
1	Can we predict co-change candidates for a particular clone fragment by using evolutionary coupling?
2	Can we achieve better ranking of co-change candidates that exhibited evolutionary coupling by considering co-change recency instead of co-change frequency?
3	What are the characteristics of the clone fragments that exhibit evolutionary coupling? Which characteristic can help us in better ranking of co-change candidates that have not yet exhibited evolutionary coupling?
4	How can we rank both types of co-change candidates - (1) the candidates that exhibited evolutionary coupling, and (2) the candidates that did not exhibit evolutionary coupling for a particular clone fragment?

from the same clone class that actually co-changed with CF) so that a developer attempting to change CF can identify the more likely co-change candidates with less effort and time. The ranking mechanism is our primary contribution. The state of the art techniques and tools [61, 95, 152, 244] do not rank the possible co-change candidates for clones. We believe that our proposed ranking mechanism can complement existing clone tracking tools and techniques.

4.2 Terminology

4.2.1 Evolutionary Coupling

We defined evolutionary coupling in Chapter 2. However, we again provide a brief description about it here for the ease of our discussion.

During the evolution of a software system if two or more program entities (such as files, classes, methods) appear to change together (i.e., co-change) frequently (i.e., in many commits) then we say that these entities exhibit evolutionary coupling. It is likely that these entities are related and a future change to any one of these entities will accompany corresponding changes to the other entities. By mining and analyzing evolutionary coupling we can discover the underlying relationships among program entities in a software system [71, 281]. Evolutionary coupling helps us predict the co-change candidates (i.e., the other entities that might also need to be changed) while changing a particular entity [281]. In this research work, we apply the concept of evolutionary coupling to discover the underlying relationships among clones and to predict the co-change candidates while changing a particular clone fragment in a particular clone class.

Table 4.2: Subject Systems

Subject Systems	**Language**	**Application Domains**	**LOC in Last Revision**	**Revisions**
Ctags	C	Code Def. Generator	33,270	774
QMail Admin	C	Mail Management	4,054	317
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Manager	45,515	1545

In order to mine evolutionary coupling among clone fragments we determine all possible pairs of co-changed clone fragments by examining the software evolution history.

4.2.2 Pair of Co-changed Clone Fragments (PCCF)

A pair of co-changed clone fragments (i.e., a *PCCF*) consists of two clone fragments $CF1$ and $CF2$ from the same clone class such that they changed together (i.e., co-changed) in at least one commit operation during system evolution. During the evolution of a software system, if $n \geq 2$ clone fragments from a particular clone class changed together (co-changed) in a particular commit, we determine all possible pairs from these n clone fragments. Each of these pairs is a *PCCF*. For every *PCCF* that we obtain by mining the whole evolution history, we determine the number of times (i.e., the number of commits) the constituent clones co-changed. We call this number the co-change frequency of a *PCCF*.

4.3 Experimental Steps

In this experiment, we consider clones residing in methods. Thus, both fully cloned and partially cloned methods have been investigated. For a particular subject system, we collect all of its revisions (mentioned in Table 4.2), then extract the methods in each revision using CTAGS, and then determine method genealogies following the technique proposed by Lozano and Wermelinger [145]. We detect clones in each revision using NiCad [48] and then map the clones to the already detected methods of the corresponding revisions. As method genealogies are already detected, after clone mapping we can easily track the evolution of each clone fragment residing inside a method. Finally, we detect changes between every two consecutive revisions and map these changes to the methods as well as clones located inside the methods. We detected both exact and near-miss block clones using the NiCad clone detector considering a dissimilarity threshold of 20% with blind renaming. This setting of NiCad is considered standard for detecting near-miss clones [196].

After the preliminary steps we determine all possible pairs of co-changed clone fragments (*PCCF*s) by examining all the commit operations. For each of the *PCCF*s we determine its co-change frequency. We rank the possible co-change candidates of a particular clone fragment on the basis of this co-change frequency. However, we also rank the possible co-change candidates on the basis of co-change recency (i.e., on the basis of how lately a co-change occurred). We describe and compare these ranking mechanisms in Section 4.4.2.

Table 4.3: Statistics Regarding Evolutionary Coupling among Clones

Systems	NC	NCRC	NCEC	NPCCF
Ctags	694	412	89	79
QMailAdmin	137	128	34	347
jEdit	12329	1356	326	324
Freecol	2265	1659	489	577
Carol	3040	1365	616	2041
Jabref	3708	1552	509	684

NC = No. of Clones created during system evolution.

NCRC = No. of Clones that Received Changes.

NCEC = No. of Clones that showed Evolutionary Coupling.

NPCCF = No. of the pairs of co-changed clone fragments.

4.4 Experimental Results and Analysis

We perform our investigation on each of the subject systems listed in Table 4.2. We determine all pairs of co-changed clone fragments (*PCCF*s) from these candidate systems. For each of the systems we determine four measures - (1) total number of clones created during system evolution, (2) total number of clones that received changes (at least once) during evolution, (3) total number of clones that exhibited evolutionary coupling, and (4) total number of *PCCF*s and show these in Table 4.3. If a clone fragment is included in at least one *PCCF*, we consider that it has exhibited evolutionary coupling, because it has co-changed with at least one of the other clone fragments in its clone class during evolution.

We also determine the percentage of modified clones (i.e., the clones that received changes at least once during evolution) that exhibited evolutionary coupling. This percentage for each subject system is shown in Fig. 4.2. The figure shows that in case of each of the subject systems, a considerable percentage of modified clones exhibited evolutionary coupling. The overall percentage considering all subject systems is 31.85%. Thus, it seems that overall 68.15% of the modified clones (i.e., clone fragments that received changes) evolved independently. However, according to our observation, overall 70.82% of the total clone fragments of a subject system never changed during evolution.

We automatically retrieve evolutionary coupling from each of the candidate subject systems. The XML files containing the pairs of co-changed clone fragments are available on-line[1]. Our primary goal in this research work is to investigate whether we can predict and rank co-change candidates for clones using evolutionary coupling. In the following subsections, we answer four research questions regarding this.

[1]XML Files: `https://homepage.usask.ca/~mam815/ongoingresearch.php`

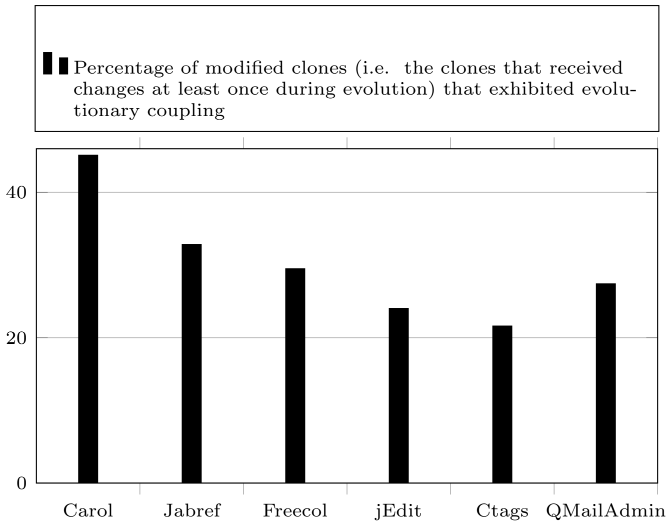


Figure 4.2: Proportion of modified clones (i.e., the clones that received changes at least once during evolution) that exhibited evolutionary coupling

4.4.1 Answering Research Question 1

RQ 1: Can we predict co-change candidates for a particular clone fragment using evolutionary coupling?

Finding the answer to this research question is the central objective of our research work. Here, we should note that by the term ‘co-change candidates’ for a particular clone fragment we mean the other clone fragments in the same clone class containing the particular clone fragment. However, non-clone fragments can also co-change with a clone fragment. We do not investigate this in this research work. We mainly focus on the efficient tracking as well as proper management of code clones. We target to minimize the drawbacks of the existing clone tracking techniques and tools. We plan to investigate the non-clone co-change fragments as future work. In the following paragraphs we describe our investigation methodology for answering **RQ 1**.

Methodology. For answering this research question we automatically analyze each of the commits of a subject system. Let us consider a particular clone class *CC* in a particular revision *R* of a subject system. A commit operation *C* was applied on revision *R* and more than one clone fragments of the clone class *CC* co-changed (i.e., changed together) in this commit. We consider a particular clone fragment *CF* from *CC* such that *CF* changed in commit *C*. So, we know which other clone fragments from clone class *CC* actually co-changed with *CF* in commit *C*. However, we want to determine whether and to what extent we can predict these true co-change candidates for *CF* (in commit *C*) by analyzing the evolutionary coupling exhibited by *CF* during the previous commits 1 to $C - 1$.

Our prediction mechanism is presented in Fig. 4.3. In this figure, we see a clone class *CC* in revision *R*. The clone class *CC* contains eight clone fragments *CF* to *CF7*. A commit operation *C* applied on revision *R* modified five clone fragments - *CF*, *CF1*, *CF4*, *CF6*, and *CF7* from this class. The corresponding clone class in revision *R*+1 (i.e., after the application of the commit operation *C*) is also shown in the figure. We consider

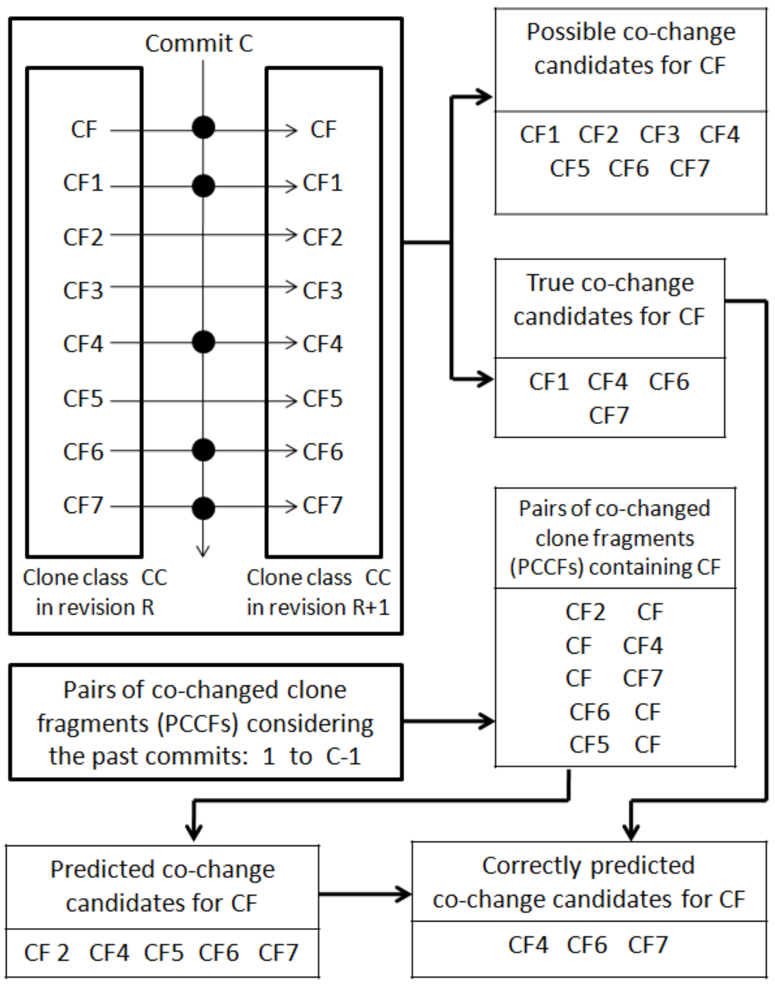

Commit C
CF
CF1
CF2
CF3
CF4
CF5
CF6
CF7
CF
CF1
CF2
CF3
CF4
CF5
CF6
CF7
Clone class CC in revision R
Clone class CC in revision R+1
Possible co-change candidates for CF
CF1 CF2 CF3 CF4 CF5 CF6 CF7
True co-change candidates for CF
CF1 CF4 CF6 CF7
Pairs of co-changed clone fragments (PCCFs) containing CF
CF2 CF
CF CF4
CF CF7
CF6 CF
CF5 CF
Pairs of co-changed clone fragments (PCCFs) considering the past commits: 1 to C-1
Predicted co-change candidates for CF
CF 2 CF4 CF5 CF6 CF7
Correctly predicted co-change candidates for CF
CF4 CF6 CF7

Figure 4.3: Prediction of co-change candidates using evolutionary coupling

the clone fragment *CF*. From the figure we can determine the actual co-change candidates (*CF1*, *CF4*, *CF6*, and *CF7*) for *CF*. We want to determine the extent that we can correctly predict these actual co-change candidates for *CF* by analyzing its evolutionary coupling during the past commits (i.e., the commits from 1 to $C-1$). For the purpose of describing, we define the following four sets considering the clone fragment *CF* and the commit operation C.

Possible Co-change Candidates. All the clone fragments of the clone class *CC* excluding *CF* are termed as the set of *possible co-change candidates* of *CF*. Each of these clone fragments has a possibility of co-changing with *CF*. However, all of these possible co-change candidates might not co-change with *CF* in a particular commit.

True Co-change Candidates. All those clone fragments (from the clone class *CC*) that actually co-changed with *CF* in commit operation C are termed as the set of *true co-change candidates* of *CF* in commit C.

Predicted Co-change Candidates. All those clone fragments (in the clone class *CC*) that we can predict as the co-change candidates for *CF* by analyzing the evolutionary coupling of *CF* in previous commits (from commit 1 to $C-1$) are termed as the set of *predicted co-change candidates.*

Correctly Predicted Co-change Candidates. All those clone fragments from the set of *predicted co-change candidates* that actually co-changed with *CF* in commit C are termed as the set of *correctly predicted co-change candidates.*

Fig. 4.3 shows these four sets for clone fragment *CF* considering commit C. We determine the set of *predicted co-change candidates* for *CF* in the following way. We at first retrieve all the pairs of co-changed clone fragments (*PCCF*s) considering all the commits from 1 to $C-1$. We select those *PCCF*s where the clone fragment *CF* appears. Fig. 4.3 shows five such *PCCF*s. From these *PCCF*s we determine all other clone fragments beside *CF*. These clone fragments (*CF2*, *CF4*, *CF5*, *CF6* and *CF7* as shown in Fig. 4.3) are the set of *predicted co-change candidates* for *CF*. We get these by analyzing evolutionary coupling of *CF*.

Finally, we determine the set of *correctly predicted co-change candidates* for *CF* in commit C. Fig. 4.3 shows three correctly predicted co-change candidates - *CF4, CF6, CF7.* We determine the following six measures for *CF* considering the commit operation C.

(1) The number of possible co-change candidates for *CF*,

(2) The number of possible co-change candidates that exhibited evolutionary coupling with *CF* in the previous commits 1 to $C-1$. This is the number of predicted co-change candidates for *CF* in commit C

(3) The number of possible co-change candidates that did not exhibit evolutionary coupling with *CF* in the past commits. We call these co-change candidates the non-predicted co-change candidates for *CF* in commit C, because we could not predict them by analyzing evolutionary coupling.

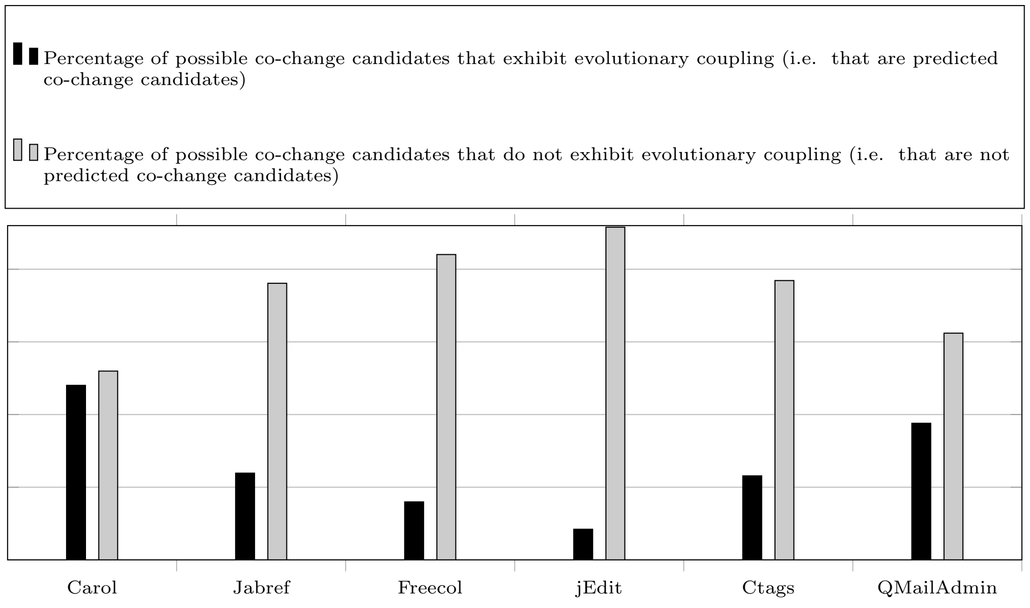


Figure 4.4: Comparison between the proportions of predicted co-change candidates and non-predicted co-change candidates

(4) The number of predicted co-change candidates that actually co-changed with *CF* in commit *C*. This is the number of correctly predicted co-change candidates for *CF*.

(5) The number of non-predicted co-change candidates that co-changed with *CF* in commit *C*.

(6) The number of true co-change candidates (i.e., the possible co-change candidates that actually co-changed with *CF*) for *CF* in commit *C*.

We examine all the commit operations where more than one clone fragments from the same clone class changed together (i.e., co-changed). For each of the clone fragments (*CF*) that changed in such a commit, we determine the above six measures. Considering all the commit operations of a particular subject system we determine the summation for each of the respective measures. Then, we calculate the following percentages using these measures.

(1) The percentage of possible co-change candidates that were selected as the predicted co-change candidates. In other words, the percentage of possible co-change candidates that exhibited evolutionary coupling with *CF*.

(2) The percentage of possible co-change candidates that were selected as the non-predicted co-change candidates. In other words, the percentage of possible co-change candidates that did not exhibit evolutionary coupling with *CF*.

(3) The percentage of predicted co-change candidates that are true co-change candidates. We also call this the *precision in predicting true co-change candidates by analyzing evolutionary coupling.*

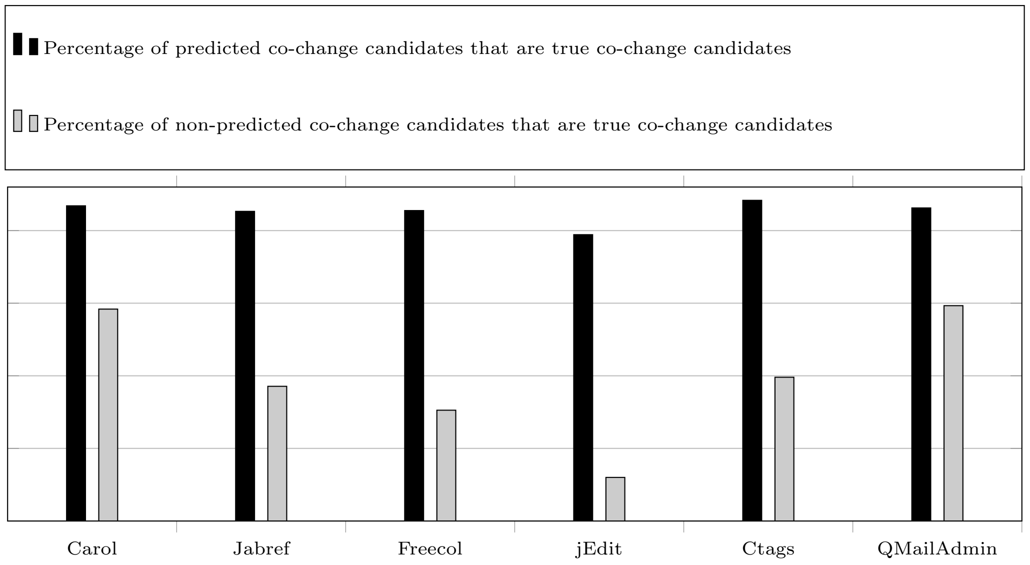


Figure 4.5: Comparison between the percentage of predicted co-change candidates that were true co-change candidates (i.e., the precision) and the percentage of non-predicted co-change candidates that were true co-change candidates

(4) The percentage of non-predicted co-change candidates that are true co-change candidates.

(5) The percentage of true co-change candidates that we could predict (i.e., by analyzing evolutionary coupling). We also call this percentage the *recall in predicting true co-change candidates by analyzing evolutionary coupling.*

Fig. 4.4 compares the first two percentages - (1) the percentage of possible co-change candidates that exhibited evolutionary coupling, and (2) the percentage of possible co-change candidates that did not exhibit evolutionary coupling, for each of the subject systems. We see that the proportion of possible co-change candidates that exhibited evolutionary coupling is much lower than its counter part for most of the subject systems. The overall values of these percentages are, 26% and 74% respectively.

However, from the comparison scenario in Fig. 4.5 (comparing the third and fourth percentages) we see that the proportion of predicted co-change candidates that were true co-change candidates (i.e., that were correctly predicted) is much higher compared to the proportion of non-predicted co-change candidates that were selected as true co-change candidates for most of the subject systems. *Thus, the predicted co-change candidates have a much higher probability of being true co-change candidates. In other words, the possible co-change candidates that exhibited evolutionary coupling have a much higher probability of being true co-change candidates compared to the possible co-change candidates that did not exhibit evolutionary coupling.*

Finally, Fig. 4.6 demonstrates that *for each of the subject systems a considerable proportion of true co-change candidates can be predicted by analyzing evolutionary coupling.* The overall proportion considering all subject systems is 43.17%. As we mentioned before, this percentage is the recall in predicting true co-

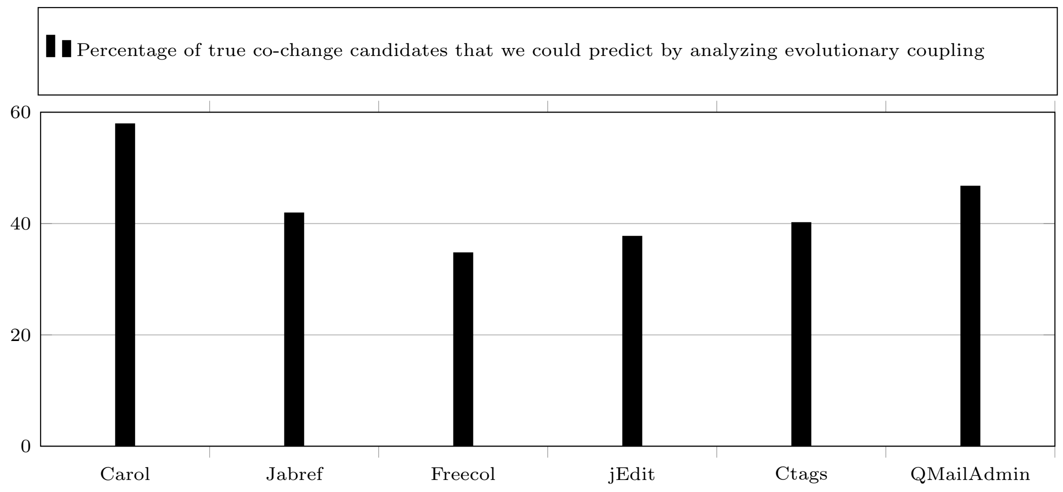


Figure 4.6: The proportion of true co-change candidates that we could predict by analyzing evolutionary coupling (i.e., the recall)

change candidates by analyzing evolutionary coupling. We also determine the overall precision (i.e., the third percentage) in predicting true co-change candidates considering all systems. This overall precision is 85.18%.

Answer to RQ 1. From our analysis and discussion we can say that evolutionary coupling can help us predict true co-change candidates for a particular clone fragment with considerable accuracy in terms of precision (= 85.18%) and recall (= 43.17%).

4.4.2 Answering Research Question 2

RQ 2: Can we achieve better ranking of co-change candidates by considering co-change recency instead of co-change frequency?

From the first research question we understand that we can predict a considerable amount of true co-change candidates for clones by analyzing evolution coupling. For answering this research question we rank predicted co-change candidates (i.e., predicted by analyzing evolutionary coupling) in the following two ways.

(1) On the basis of co-change frequency

(2) On the basis of co-change recency (i.e., how lately a co-change occurred)

Then, we determine which ranking system generally gives better ranks for the correctly predicted co-change candidates. In the following paragraphs we describe the ranking mechanisms and compare those.

Description of the ranking mechanisms. We at first assume a particular clone class CC in a particular revision R of a particular candidate system. More than one clone fragments from this class co-changed in the commit operation C applied on revision R. We consider a particular clone fragment CF from CC that changed in this commit operation C. For the clone fragment CF, we determine the following two sets of co-change candidates considering commit operation C following the methodology described in the previous subsection.

(1) **True co-change candidates.** The clone fragments from clone class *CC* that actually co-changed with *CF*

(2) **Predicted co-change candidates.** The clones in clone class *CC* that exhibited evolutionary coupling with *CF*

We rank these predicted co-change candidates in two ways. The ranking procedures are as follows.

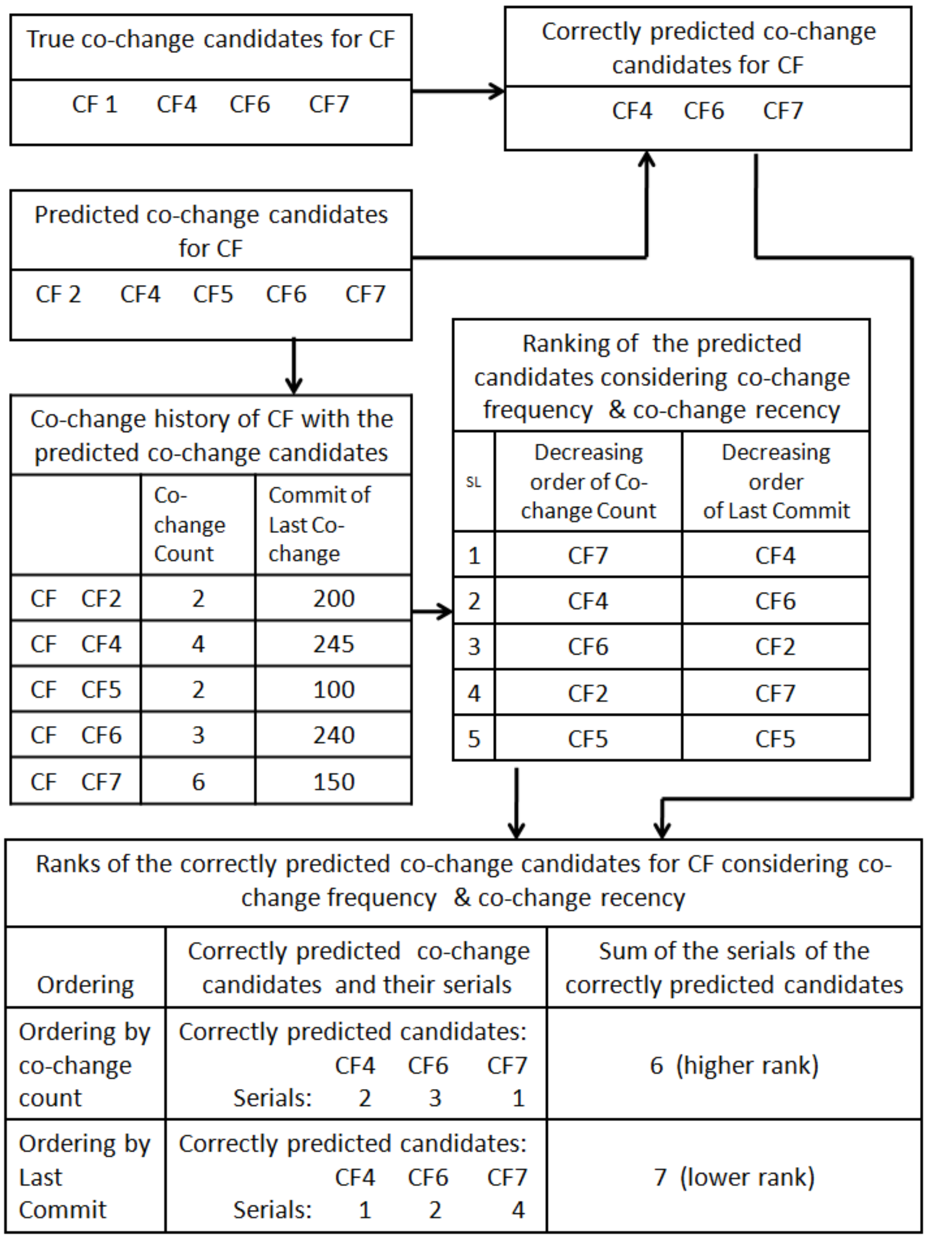
Ranking predicted co-change candidates on the basis of co-change frequency. We know that each of the predicted co-change candidates has previously (in the commits preceding the commit *C*) co-changed with the clone fragment *CF*. We assume higher ranks for those predicted co-change candidates that previously co-changed with *CF* more frequently. We sort these predicted co-change candidates in decreasing order of their co-change counts with CF.

Ranking predicted co-change candidates on the basis of co-change recency. In this case the predicted co-change candidates that co-changed with *CF* more recently are given higher ranks. For each of the predicted co-change candidates, we determine the last commit operation where it co-changed with *CF*. We sort the predicted co-change candidates in decreasing order of their last commits.

An example describing the two ways of ranking. Fig. 4.7 shows an example of the two ways of ranking of the predicted co-change candidates for a clone fragment *CF*. The sets - **(1)** true co-change candidates, **(2)** predicted co-change candidates, and **(3)** correctly predicted co-change candidates for *CF* are taken from Fig. 4.3. In this figure (i.e., Fig. 4.7) we show a possible co-change history of *CF* with each of the predicted co-change candidates. The co-change history consists of two pieces of information - **(1)** The number of times *CF* co-changed with a predicted co-change candidate, and **(2)** The last commit operation where *CF* co-changed with a predicted co-change candidate.

These are the indicators of co-change frequency and co-change recency respectively. We automatically collect this information from the co-change history of *CF*. From the figure (Fig. 4.7) we see that *CF* co-changed with *CF7* the highest number of times (i.e., 6 as shown in the figure). However, among all the predicted co-change candidates, *CF4* co-changed with *CF* most recently. This co-change occurred in commit 245 as shown in the figure. Rankings (i.e., orderings) of the predicted co-change candidates in two ways (i.e., on the basis of co-change frequency and co-change recency) are also shown in Fig. 4.7. We see that while *CF7* is assigned the highest rank (i.e., its serial number is 1) on the basis of co-change frequency, *CF4* is assigned the highest rank on the basis of co-change recency. In both cases, *CF5* gets the lowest rank.

Comparison of the ranking mechanisms. We see that each of the two ranking systems described above is based on evolutionary coupling of the clone fragments. However, we want to determine which one is better. The ranking system that provides better ranks for the correctly predicted co-change candidates should be considered as the superior one.



True co-change candidates for CF
CF 1 CF4 CF6 CF7

Correctly predicted co-change candidates for CF
CF4 CF6 CF7

Predicted co-change candidates for CF
CF 2 CF4 CF5 CF6 CF7

Co-change history of CF with the predicted co-change candidates

	Co-change Count	Commit of Last Co-change
CF CF2	2	200
CF CF4	4	245
CF CF5	2	100
CF CF6	3	240
CF CF7	6	150

Ranking of the predicted candidates considering co-change frequency & co-change recency

SL	Decreasing order of Co-change Count	Decreasing order of Last Commit
1	CF7	CF4
2	CF4	CF6
3	CF6	CF2
4	CF2	CF7
5	CF5	CF5

Ranks of the correctly predicted co-change candidates for CF considering co-change frequency & co-change recency

Ordering	Correctly predicted co-change candidates and their serials	Sum of the serials of the correctly predicted candidates
Ordering by co-change count	Correctly predicted candidates: CF4 CF6 CF7 Serials: 2 3 1	6 (higher rank)
Ordering by Last Commit	Correctly predicted candidates: CF4 CF6 CF7 Serials: 1 2 4	7 (lower rank)

Figure 4.7: Ranking of predicted co-change candidates by co-change frequency and co-change recency

For each clone fragment *CF* changed in a commit operation *C* we determine its predicted co-changed candidates and rank these in two ways to get two different rankings (or orderings) of the predicted co-change candidates. Then we determine the correctly predicted co-change candidates and locate them in each of the rankings of the predicted co-change candidates. We determine the serial numbers (i.e., position values) of the correctly predicted co-change candidates from each ranking. We get two sets of serial numbers from the two ranking systems. We determine the summation of the serial numbers obtained from each ranking system. A lower summation indicates better ranking.

In Fig. 4.7, the widest table (i.e., the bottom one) shows the comparison of the ranking systems considering the predicted co-change candidates for the clone fragment *CF*. There are five predicted co-change candidates for *CF*. According to the example, three (*CF4, CF6, CF7*) of these predicted co-change candidates have actually co-changed with *CF* in commit operation *C*. We show the serial numbers of the predicted co-change candidates in each of the rankings. We see that from the ranking on the basis of co-change frequency, we get the serial numbers - 2, 3, and 1 for the correctly predicted co-change candidates *CF4*, *CF6*, and *CF7* respectively. The serial numbers obtained from the other ranking system are 1, 2, and 4 respectively. The summations (6 considering co-change frequency, and 7 considering co-change recency) of the serial numbers are also shown in the figure. According to our explanation, the ranking system on the basis of co-change frequency provides better ranks to the correctly predicted co-change candidates.

For each of the clone fragments changed in each of the commits we determine which ranking system provides better ranks to the correctly predicted co-change candidates. We determine two percentages - **(1)** The percentage of cases where the ranking on the basis of co-change frequency gives us better ranks for the correctly predicted co-change candidates, and **(2)** The percentage of cases where the ranking on the basis of co-change recency provides us better ranks for the correctly predicted co-change candidates

We show these percentages in Fig. 4.8. In case of each of the subject systems, for most of the cases the two ranking systems provide the same ranks to the correctly predicted co-change candidates. However, if we consider the remaining cases, we see (c.f., Fig. 4.8) that *for each of the subject systems, co-change recency provides better ranks to the correctly predicted co-change candidates compared to co-change frequency.*

Answer to RQ 2. From our discussion we decide that we can achieve better ranking of co-change candidates by considering co-change recency instead of co-change frequency.

4.4.3 Answering Research Question 3

RQ 3: What are the characteristics of the clone fragments that exhibit evolutionary coupling? Which characteristic can help us in better ranking of the co-change candidates that have not yet exhibited evolutionary coupling?

Answering this research question is important. From our discussion and analysis while answering *RQ 1* we understand that the percentage of predicted co-change candidates for a particular clone fragment *CF* (i.e., the percentage of possible co-change candidates that exhibit evolutionary coupling with *CF*) is always smaller

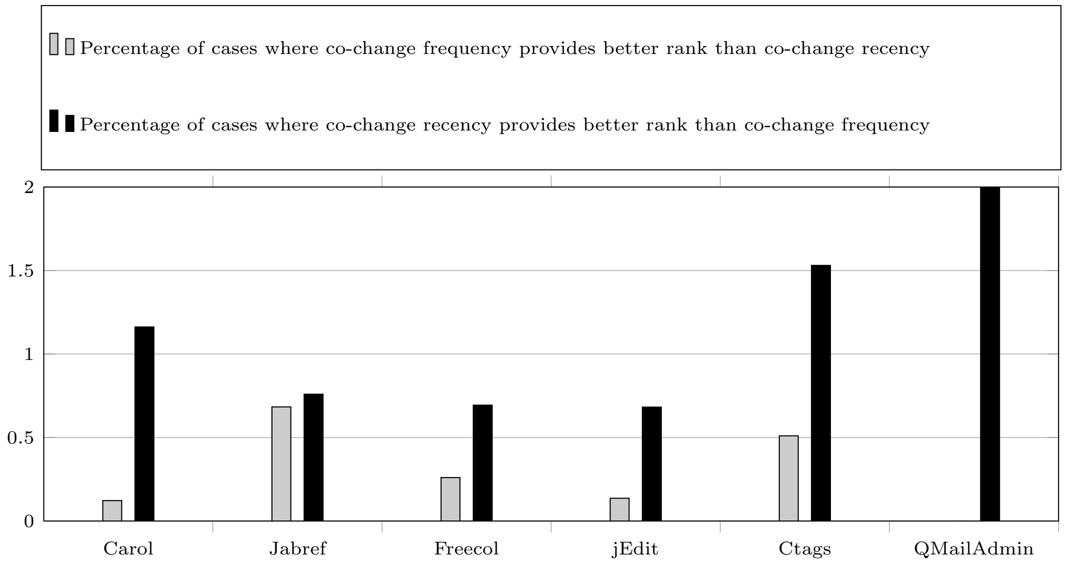


Figure 4.8: Comparison of the two ranking systems (ranking using co-change frequency and co-change recency)

compared to the percentage of non-predicted co-change candidates (i.e., the possible co-change candidates that have not yet exhibited evolutionary coupling with *CF*). We can rank the predicted co-change candidates. However, we are also interested in ranking the non-predicted co-change candidates. For this purpose we analyze the characteristics of the clone fragments that exhibit evolutionary coupling. Let us assume that we have discovered a dominant characteristic of the clone fragments that exhibit evolutionary coupling. If we see that some of the non-predicted co-change candidates also possess this characteristic, we can assume better ranks for these non-predicted co-change candidates.

Methodology. For answering this research question, we analyze the following two characteristics of the clone fragments that exhibit evolutionary coupling.

(1) **Regarding clone type.** We analyze whether the clone fragments exhibiting evolutionary coupling are method clones or block clones.

(2) **Regarding the proximity of the clone fragments.** We analyze whether two clone fragments exhibiting evolutionary coupling generally remain in close proximity to each other or not.

Considering each of the subject systems we determine all the pairs of co-changed clone fragments (*PCCF*s). For each *PCCF* we determine the types (method clone or block clone) of the two participating clone fragments. We also determine whether the two clone fragments remain in the same file or in different files. For each of the subject systems we determine the following two percentages - **(1)** The percentage of *PCCF*s (i.e., the pairs of co-changed clone fragments) where the participating clone fragments remain in the same file. **(2)** The percentage of *PCCF*s where both of the participating clone fragments are method clones. We present these two percentages in Fig. 4.9. From the black bars we see that for most of the subject systems (except

QMailAdmin), the participating clone fragments in most of the *PCCF*s (i.e., above 55% of the *PCCF*s) remain in the same file. We also observe (from the white bars) that in the case of four subject systems (Carol, Freecol, jEdit, and Ctags), both of the participating clone fragments in most of the *PCCF*s are method clones (i.e., clones are full methods). However, if we compare these two characteristics, we see that file proximity is the more dominant one for most of the subject systems (except Carol).

We rank the non-predicted co-change candidates (for a particular clone fragment *CF* in a particular commit *C*) in the following two ways.

(1) Considering clone file proximity and

(2) Considering clone type (method clones or block clones)

In case of ranking considering clone file proximity, we provide higher (i.e., better) ranks to those non-predicted co-change candidates that are nearer (i.e., in closer proximity) to the clone fragment *CF*. The clone file proximity between *CF* and a particular non-predicted co-change candidate is determined by the distance between the corresponding container files in the file system structure as was done by Alali et. al [14]. In case of ranking considering clone type, we provide higher ranks to those non-predicted co-change candidates that are method clones. We compared these two ranking systems following the same way described while answering RQ 2. In this case, from the ranking (of the non-predicted co-change candidates) obtained from each ranking system, we determine the summation of the position values (i.e., serial numbers) of the non-predicted true co-change candidates. The ranking system that provides the lower summation (i.e., the higher rank) is the better one.

Considering each of the clone fragments changed in each of the commits (where more than one clone fragments from the same clone class co-changed) of a particular subject system we determine and rank the non-predicted co-change candidates using the above two ranking systems and determine which ranking system provides better ranks to the non-predicted true co-change candidates. We determine two percentages - **(1)** The percentage of cases where the ranking system on the basis of clone file proximity provides better ranks, and **(2)** The percentage of cases where the ranking system on the basis of clone type provides better ranks

These two percentages for each of the subject systems is shown in Fig. 4.10. We see that for the case of each of the subject systems, *the percentage of cases where ranking by clone file proximity provides better ranks is much higher than the percentage of cases where ranking by clone type provides better ranks to the non-predicted true co-change candidates.* However, we observed that for 56% to 87% of cases (considering all the subject systems), the two ranking systems provided the same ranks. Finally, we consider the ranking system on the basis of clone file proximity to be the better for ranking the non-predicted co-change candidates.

Answer to RQ 3. Generally, the clone fragments that exhibit evolutionary coupling - (1) remain in close proximity (i.e., in the same file) to each other, and (2) are method clones. According to our analysis, consideration of clone file proximity can help us in better ranking of the non-predicted co-change candidates for a particular clone fragment.

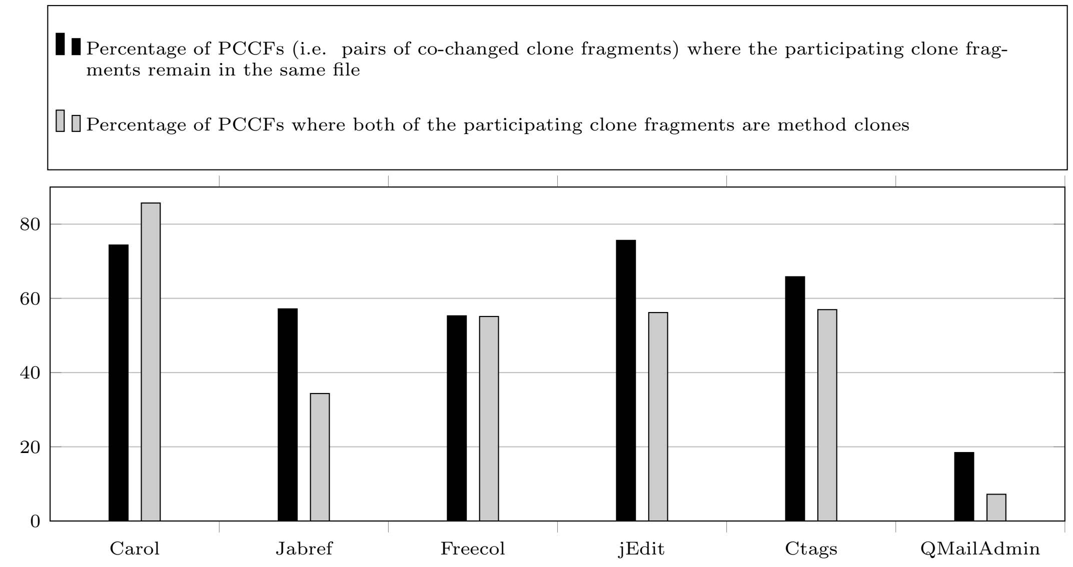


Figure 4.9: Comparing the characteristics of clone fragments that exhibit evolutionary coupling

4.4.4 Answering Research Question 4

RQ 4: How can we rank both types of co-change candidates - (1) the candidates that exhibited evolutionary coupling, and (2) the candidates that did not exhibit evolutionary coupling for a particular clone fragment?

Answering this research question is important. From the answers to the previous research questions we understand that the possible co-change candidates for a particular clone fragment can broadly be divided into two disjoint sets - (1) the predicted co-change candidates (i.e., the candidates that exhibited evolutionary coupling with the particular clone fragment), and (2) non-predicted co-change candidates (i.e., the candidates that have not yet exhibited evolutionary coupling with the particular clone fragment). For the first set we decide a ranking mechanism on the basis of co-change recency as the better one. For the second we found the ranking mechanism on the basis of clone file proximity to be the better one. Thus intuitively, a combination of the two ranking mechanisms (ranking by co-change recency for the predicted candidates and ranking by clone file proximity for the non-predicted candidates) can possibly be used for better ranking of all co-change candidates of a particular clone fragment. We verify this in this research question in the following way.

Methodology. We rank all the possible co-change candidates for a particular clone fragment *CF* changed in a particular commit *C* in the following two ways:

(1) Ranking of all possible co-change candidates using two ranking mechanisms: ranking of the predicted candidates by co-change recency, and ranking of the non-predicted candidates by clone file proximity,

(2) Ranking of all possible co-change candidates by clone file proximity.

We compared these two ways of ranking to determine which one can provide better ranks to all the true co-change candidates for a particular clone fragment *CF* in a particular commit *C*. We performed our

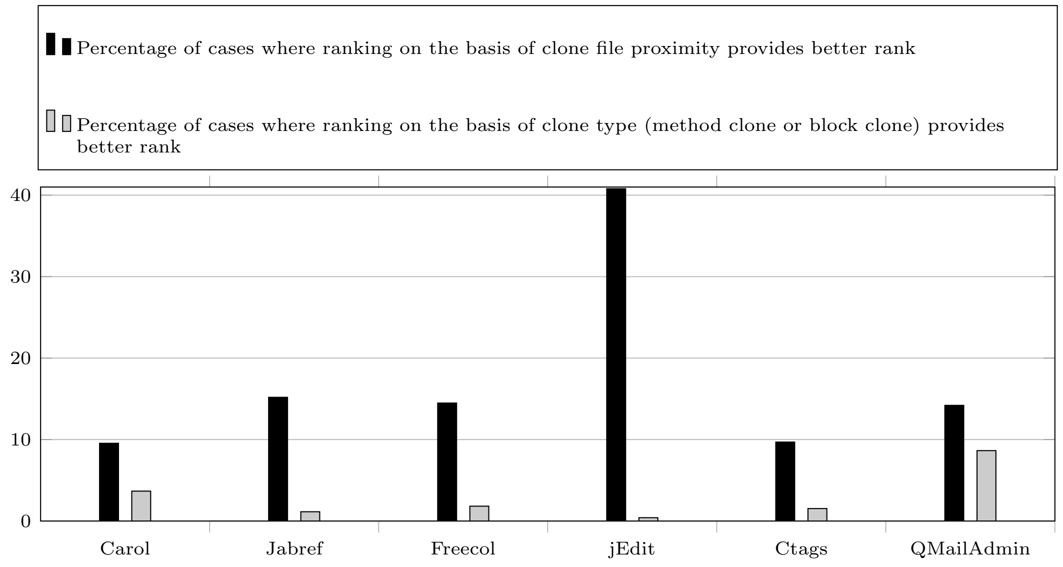


Figure 4.10: Comparison of the two ranking systems for the non-predicted co-change candidates

comparison in the same way (i.e., considering all commit operations) as is described while answering RQ 2. We determine the percentage of cases where we get better ranks for the true co-change candidates using the combined ranking mechanism (the first way mentioned above) and also the percentages of cases where the second way of ranking (ranking all candidates by clone file proximity) provides better ranks. These percentages are shown in Fig. 4.11. From Fig. 4.11 we see that in the case of each of the subject systems, the percentage of cases where the combined ranking provides better ranks is much higher compared to the percentage of cases where ranking by only clone file proximity provides better ranks to the true co-change candidates. Thus, we decide that the combined ranking mechanism yields in better ranking of the true co-change candidates compared to the file proximity ranking.

Answer to RQ 4. From our discussion and analysis presented above, we suggest a combined ranking (ranking of predicted co-change candidates by co-change recency, and ranking of non-predicted co-change candidates by clone file proximity) of the possible co-change candidates for a particular clone fragment.

4.5 Threats to Validity

We used the NiCad clone detector [48] for detecting clones. For different settings of NiCad, the statistics that we present might be different. Wang et. al [254] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [196] and with these settings NiCad can detect clones with high precision and recall [198, 201].

For determining the prediction accuracy of our implemented prediction system, we have used the precision and recall measures. Shepperd and MacDonell [221] conducted a fine grained study on the evaluation of the

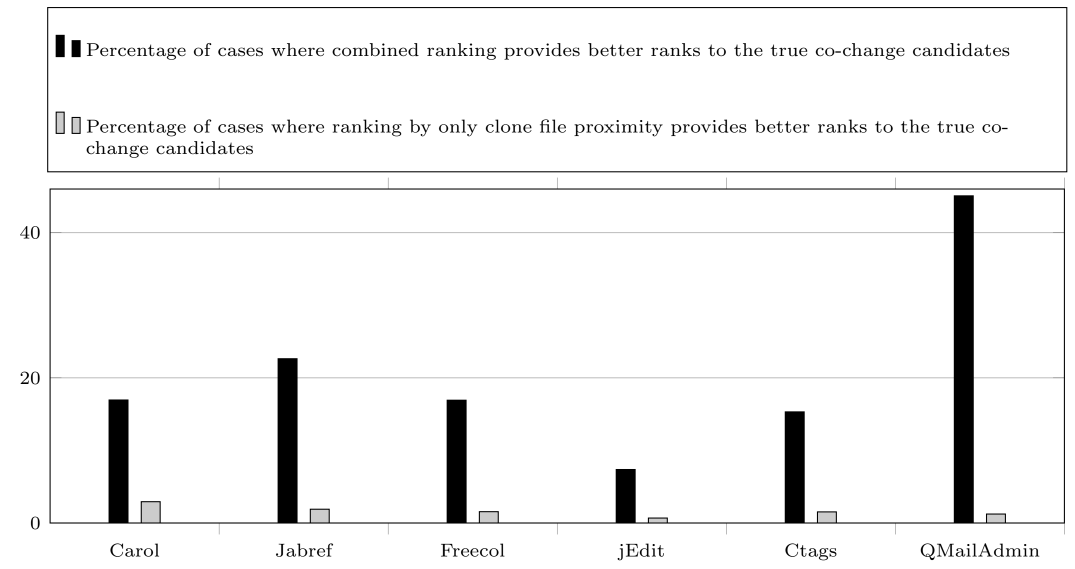


Figure 4.11: Comparison between the combined ranking (i.e, ranking predicted candidates by co-change recency and non-predicted co-change candidates by clone file proximity) and ranking considering only clone file proximity

prediction systems. According to their observation different prediction systems might give us conflicting results. They advised researchers not to use biased accuracy measures for the purpose of determining prediction accuracy. However, precision and recall are well known and extensively used measures and they represent the exact scenario (in term of accuracy) when reported together. Thus, we believe that our reported findings in the form of precision and recall are important.

The subject systems that we have studied in this experiment are not enough to take a concrete decision regarding the ranking of co-change candidates for clones. However, our candidate systems are of diverse variety in terms of application domains, sizes and revisions. Thus, our findings cannot be attributed to chance. Finally, we believe that our findings are important and can help us in better clone management.

4.6 Related Work

A great many studies have already been done regarding the detection, evolution [19,119,241], impact analysis [29,78,111,131,132,145,146,156,158], and maintenance [60,95,152,163,244] of code clones. Although there are some positive impacts [75,111,131,132] of cloning on both software development and maintenance, a number of studies [29,145,146,156] have shown empirical evidence of a strong negative impact of clones on software evolution. Focusing on the negative impacts, software researchers have emphasized proper maintenance of code clones through clone refactoring [25,25,38,163] or tracking [60,61,95,152,244]. As clone refactoring is not always possible [119], tracking of clones becomes very important for better software maintenance. As our research work is focused on clone tracking, we discuss the existing clone tracking techniques and studies.

The most recent study on clone tracking was conducted by Duala-Ekoko and Robillard [60]. They introduced the concept of *clone region descriptor*. On the basis of this concept they proposed a technique for tracking clones in evolving software. They implemented a tool called 'CloneTracker' [61] as an Eclipse plug-in for tracking clones. The tool provides supports for two tasks - (1) change notifications and (2) simultaneous editing of clones. After modifying a particular clone tracked by CloneTracker, the programmer is notified about the other clone fragments in the same group that contains the modified clone. However, 'CloneTracker' does not support any type of ranking of these other clone fragments. In other words, this tool does not have any prior knowledge about which other clone fragments have higher probability of co-changing with the particular clone fragment. Our research presented in this chapter (Chapter 4) focuses on ranking of these other clone fragments so that the responsible programmer can easily pinpoint those other clone fragments that are more likely to consistently co-change with the particular clone fragment.

Jablonski and Hou [95] developed a tool called *CReN* to track copy-paste code clones and support consistent renaming of identifiers. Miller and Myer [152] proposed a technique for simultaneous editing in multiple clone fragments in the same clone class to minimize the task of repetitive editing. They implemented their technique in a text editor called *LAPIS*. There is also another clone tracking tool called *Codelink* developed by Toomin et. al [244]. However, none of these existing clone trackers supports ranking of the co-change candidates for clones.

We previously conducted a study [163] on the detection and ranking of *SPCP clones* (i.e., clones that evolve following a similarity preserving change pattern) through analysis of evolutionary coupling. *SPCP clones* are important candidates for refactoring. We ranked these *SPCP clones* using evolutionary coupling to prioritize refactoring tasks. However, as clones are not always refactorable, it is important to track them efficiently. Our prime focus in this research work is to support the existing clone trackers by providing facilities for automatic prediction and ranking of the likely co-change candidates when we change a particular clone fragment.

From our discussion above we believe that our study presented in Chapter 4 is important and unique. Our experimental result has the potential to assist in better management of code clones, and thus can help us in better software maintenance.

4.7 Summary

In this research work we present an in-depth investigation into the possibility of predicting and ranking co-change candidates for clones through analysis of evolutionary coupling among clone fragments. We empirically studied six subject systems written in two programming languages (Java and C). We used the NiCad clone detector for detecting clones. Our experimental results and analysis imply that while changing a particular clone fragment in a particular clone class

(1) We can predict which other clone fragments in the same clone class will also co-change with the particular clone fragment with considerable accuracy (precision = 85.18%, recall = 43.17%) by analyzing the evolutionary coupling of the particular clone fragment.

(2) We can automatically rank the possible co-change candidates (all other clone fragments in the same clone class) on the basis of their evolutionary coupling with the particular clone fragment such that the co-change candidates that are more likely to co-change with the particular clone fragment get higher ranks.

We propose a composite ranking mechanism for ranking the possible co-change candidates of a particular clone fragment. In the presence of such a ranking, the responsible programmer can easily pinpoint the likely co-change candidates while changing a particular clone. Thus, our ranking mechanism can complement existing clone tracking techniques and tools for better management of code clones.

While our investigation in this chapter (i.e., Chapter 4) and in the previous one (i.e., Chapter 3) involves analyzing evolutionary coupling among code clones from the same clone class, we suspect that a clone fragment in a particular clone class might have evolutionary coupling with code fragments beyond its class boundary. We investigate this matter in Chapter 5.

Chapter 5
Automatic Identification of Important Clones for Refactoring and Tracking

In Chapter 3 we presented our study on identifying code clones that are important for refactoring. We discovered a particular clone change pattern called Similarity Preserving Change Pattern (SPCP), and proposed a mechanism for detecting the SPCP clones (i.e., the code clones that evolved following an SPCP). The clone fragments that do not follow this pattern either evolve independently or rarely change during evolution. Thus, these non-SPCP clone fragments cannot be important candidates for management. We proposed that the SPCP clone fragments can be considered important for refactoring. In Chapter 5 we further analyze the evolutionary coupling of SPCP clones, and divide those into two disjoint subsets on the basis of our analysis. While clone fragments in one subset should be considered important for tracking, the clone fragments in the remaining subset can be considered important for refactoring.

The rest of the chapter is organized as follows. Section 5.2 describes the terminology, Section 5.3 elaborates on the cross-boundary relationships of SPCP clones, Section 5.4 describes the methodology, Section 5.5 answers the research questions based on experimental results, Section 5.6 mentions possible threats to validity, Section 5.7 discusses the related work, and Section 5.8 concludes the chapter by summarizing our findings.

5.1 Introduction and Motivation

In our previous studies we only analyzed evolutionary coupling among clone fragments from the same clone class. However, a clone fragment in a particular clone class can have evolutionary coupling with clone fragments from other clone classes, and also, with non-clone fragments. Such couplings across the class boundaries should also be considered when selecting code clones for refactoring or tracking. In Fig. 5.1 we provide an example of such couplings.

The figure shows that an SPCP clone fragment in *Clone Class 1* has couplings with four code fragments beyond its class boundary. Two code fragments are clone fragments from two other clone classes, *Clone Class 2*, and *Clone Class 3*. The remaining two code fragments are non-clone fragments. Removal of this SPCP clone fragment might have ripple change effects on these four code fragments remaining outside of *Clone Class 1* and also, might negatively affect their future evolution. Thus, according to our consideration, such an SPCP clone fragment having relationships across its class boundary should not be considered for removal

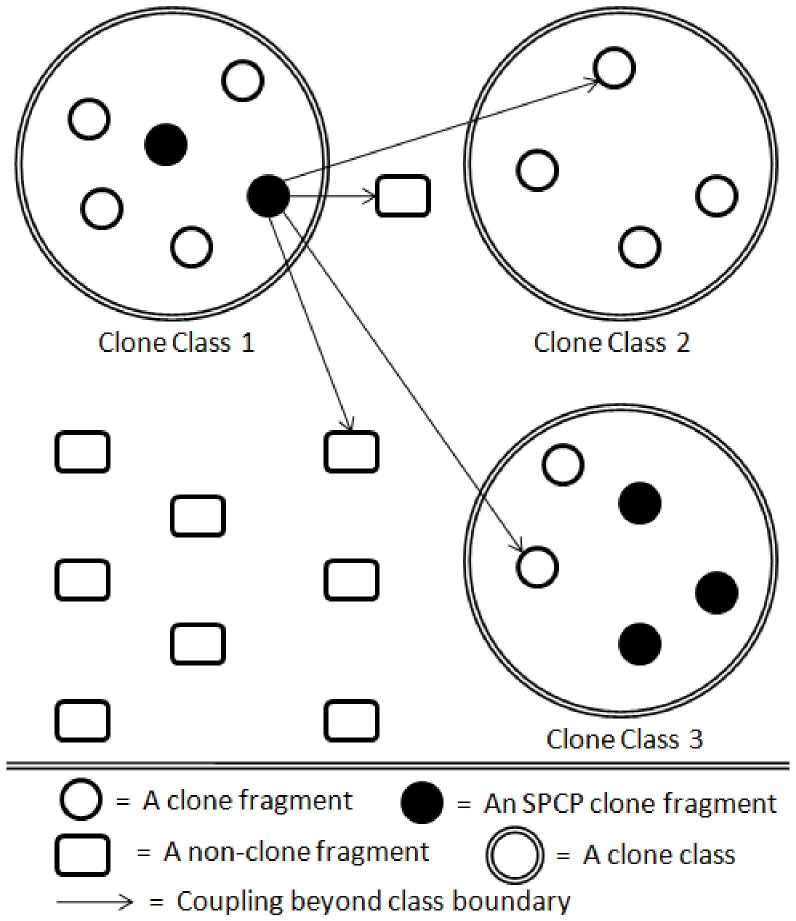


Figure 5.1: Example of an SPCP clone fragment having relationships beyond its class boundary

through refactoring. Rather, these should be important candidates for tracking, because they evolve by maintaining consistency not only with other SPCP clones in their own clone classes but also with non-clone fragments as well as with clone fragments from other clone classes beyond their class boundaries. We should track these clone fragments along with their cross-boundary relationships so that we can update them (i.e., the SPCP clone fragment as well as the code fragments beyond its class boundary) consistently in the future. However, if we want to refactor such an SPCP clone fragment, then we must analyze its relationships beyond its class boundary so that removal of it does not leave the related code fragments (beyond class boundary) in an inconsistent state and does not negatively affect the future evolution of these related code fragments.

Thus, it is important to automatically identify SPCP clone fragments that have couplings beyond their class boundaries so that we can discard these from consideration while taking refactoring decision and consider them as important candidates for tracking and future change prediction. Focusing on this important issue, in this study, we automatically extract and analyze the evolutionary coupling (i.e., change coupling) of SPCP clones and identify those SPCP clones each of which has change coupling with non-clone fragments and/or with clone fragments from other clone classes rather than its own clone class. Finally, we separate the whole set of SPCP clones into two disjoint subsets: **(1)** one contains the cross-boundary SPCP clones (i.e., the SPCP clones having cross boundary relationships), and **(2)** the other contains the non-cross-boundary SPCP clones. We perform an in-depth empirical study on both cross-boundary and non-cross-boundary SPCP clones.

According to our investigation considering both exact (Type 1) and near-miss (Type 2, Type 3) clones in thousands of revisions of six diverse subject systems covering two programming languages (Java, and C) we answer four research questions listed in Table 5.1 and come to the following decisions.

(1) The cross-boundary SPCP clones should be considered as the important candidates for tracking. Removal of these clone fragments without taking proper care of their cross-boundary relationships might negatively affect the future evolution of the related code fragments. We should track cross-boundary SPCP clones along with their coupled code fragments so that we can update them consistently in the future. However, if such an SPCP clone fragment needs to be refactored, then we should be careful of their relationships across their class boundaries. Overall, 10.27% of all clones in a software system are cross-boundary SPCP clones.

(2) The non-cross-boundary SPCP clones should be considered as the important candidates for refactoring. Overall, 13.20% of all clones in a software system are non-cross-boundary SPCP clones.

(3) The cross-boundary SPCP clones have much higher change-proneness than the non-cross-boundary SPCP clones. The main reason behind this higher change-proneness is that the cross-boundary SPCP clones are generally highly coupled with other code fragments beyond their class boundaries. Thus, such SPCP clones are the places in a software system where we can think of possible restructuring to minimize their coupling.

Table 5.1: Research Questions

SL	Research Question
1	What proportion of the SPCP clones have cross-boundary relationships?
2	Should we discard the cross-boundary SPCP clones from consideration while taking refactoring decisions and also, consider them for tracking?
3	Do cross-boundary SPCP clones have higher change-proneness than the non-cross-boundary SPCP clones?
4	Which types (i.e., Type 1, Type 2, or Type 3) of SPCP clone fragments have higher possibility of having cross-boundary relationships?

(4) Considerable proportions of Type 2 and Type 3 SPCP clones have cross-boundary relationships. However, cross-boundary SPCP clones are rare in Type 1 case.

As the non-SPCP clone fragments either evolved independently or rarely changed during evolution, we can exclude them from management considerations. Overall 43% of the SPCP clones of a subject system can have cross-boundary relationships. Our implemented system can automatically identify these and thus, can help us identify the important candidates for tracking as well as for refactoring.

5.2 Terminology

We conduct our experiment considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones) defined in Chapter 2. Our study evolves analyzing evolutionary coupling of code clones. We discussed evolutionary coupling in Chapter 6.

Corresponding change. Let us assume that two code fragments have co-changed (changed together) in a particular commit operation. If the changes are related such that changes in one code fragment required changes to the other fragment to ensure consistency between them, then we say that the changes to these two code fragments are corresponding changes. Here, a code fragment can be a clone fragment from a particular clone class or a non-clone fragment (defined in Section 5.4).

5.3 SPCP Clones having Relationships beyond their Class Boundaries

In this section, we at first discuss SPCP clones and then describe how we detect those SPCP clones each having change coupling with other code fragments (clone and/or non-clone fragments) beyond its class boundary.

5.3.1 SPCP Clones

The elaboration of SPCP is *Similarity Preserving Change Pattern.* As we defined in our earlier work [163], if two or more clone fragments from the same clone class evolve by receiving only *Similarity Preserving Changes* and/or *Re-synchronizing changes*, then we say that these clone fragments follow a *Similarity Preserving Change Pattern.* We call these clone fragments the *SPCP Clone Fragments* or *SPCP Clones.*

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied on this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered as clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. Let us consider two code fragments that are clones of each other in a particular revision. A commit operation was applied on this revision, and any one of both of the fragments received some changes in such a way that the code fragments were not considered as clones of each other in the next revision. However, in a later commit operation any one or both of the code fragments received some changes, and because of these changes the code fragments again became clones of each other. Such a converging change followed by a diverging change is termed as a re-synchronizing change.

5.3.2 SPCP Clones having Cross-Boundary Relationships

In our previous work [163] we mentioned that if two or more clone fragments from a particular clone class are identified as SPCP clones, then they might be important candidates for refactoring, because they evolve by receiving similarity preserving changes or re-synchronizing changes. However, merging these clone fragments into one, that is, removal of all these fragments by a single one is tricky. Here, we should not only think of whether we can merge them but also think about whether we should merge them. If we remove a code fragment from a code-base without being conscious about its relationships with other code fragments, then it is likely that the related code fragments will be negatively affected. These relationships might not be structural. Most of the recent IDEs are capable of pin-pointing the violations or breaking of structural relationships. However, code fragments might have evolutionary coupling relationships, and the IDEs cannot identify if we are going to break such a relationship between two code fragments. By analyzing the evolutionary history of the clone fragments of our subject systems, we observe that an SPCP clone fragment from a particular clone class can also have evolutionary coupling relationships with other two types of code fragments: (1) clone fragments from other clone classes and, (2) non-clone fragments such that the SPCP clone fragment and these other code fragments often need to be changed together (co-changed) correspondingly. In presence of such co-change relationships, also known as change couplings, this SPCP clone fragment should not be removed by refactoring. Removal of this fragment can negatively affect the other related fragments. Thus, this is important to identify SPCP clones with relationships beyond their class boundaries so that we can filter-out them from consideration while taking refactoring decisions and can keep track of them along with their cross-boundary relationships for updating them consistently in the future. We detect cross-boundary SPCP clones by analyzing evolutionary coupling. The detection procedure is described below in detail.

5.3.3 Detecting Cross-Boundary SPCP Clones by Mining their Evolutionary Coupling

We at first detect all the SPCP clones from a subject system following the procedure we proposed in our earlier work [163]. We then automatically examine the evolutionary history of the subject system in order to extract the evolutionary coupling of each SPCP clone with non-clone fragments as well as with clone fragments from other clone classes rather than its own clone class. By examining the evolutionary history, we determine pairs of co-changed code fragments that can be categorized into the following two categories.

(1) ***Different Class Category:*** *Each pair in this category consists of two clone fragments: (1) one is an SPCP clone fragment from a particular clone class, and (2) the other one is a clone fragment (may be an SPCP clone fragment or not) from a different clone class rather than the clone class of the first one such that these two clone fragments co-changed (i.e., changed together) during the past evolution.*

(2) ***Clone Non-clone Category:*** *A pair in this category consists of two code fragments: (1) one is an SPCP clone fragment from a particular clone class, and (2) the other one is a non-clone fragment such that these two code fragments co-changed during the past evolution.*

We examine each of the commit operations and determine the pairs of co-changed code fragments. A particular pair may appear more than once. We count the number of commits a pair appears. From a particular pair (CF1, CF2) of co-changed code fragments we determine two association rules $CF1 => CF2$ and $CF2 => CF1$ along with their support and confidence values. We consider only those association rules each of which satisfies the following two conditions.

Condition 1. The rule has a support (i.e., the number of times the constituent code fragments co-changed) of at least 2. We discard the lowest support rules (i.e., the rules with support of 1) from consideration because the constituent code fragments in such a rule has a very low probability of having change coupling between them. Such kind of discarding of rules has been done by previous studies [41, 107].

Condition 2. The rule has a confidence of 1 (i.e., the highest confidence). An association rule *CF1 => CF2* with highest confidence indicates that *each commit operation where CF1 received some changes, CF2 also received some changes.* Thus, it is very much likely that *CF1* and *CF2* have change coupling, and a future change in one fragment will trigger a corresponding change to the other one. The term corresponding change is defined in Section 5.2.

We determine the set of SPCP clone fragments involved in those rules that satisfy the above two conditions. We consider these SPCP clones as the cross-boundary SPCP clones. By excluding these cross-boundary SPCP clones from the whole set of SPCP clones of a subject system, we get the non-cross-boundary SPCP clones.

Table 5.2: Subject Systems

System	Language	Domain	LOC	Revisions
Ctags	C	Code Def. Generator	33,270	774
QMailAdmin	C	Mail Management	4,054	317
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Manager	45,515	1545

Revisions = Number of revisions investigated

5.4 Methodology

Table 5.2 lists the six open source subject systems that we investigate in our study. We consider all the revisions (as noted in Table 5.2) beginning with the first one for each of the systems.

We first download all the revisions as noted in Table 5.2 for all the subject systems from their open-source SVN repository[1]. Then, for each system we perform nine experimental steps as follows: (1) Method detection and extraction from each of the revisions using CTAGS [2], (2) Detection and extraction of code clones from each revision using the NiCad clone detector [48], (3) Detection of changes between every two consecutive revisions using *diff*, (4) Locating these changes to the already detected methods as well as clones of the corresponding revisions, (5) Locating the code clones detected from each revision to the methods of that revision, (6) Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [145], (7) Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy, (8) Detection of SPCP clone fragments following technique we proposed in our earlier work [163], and (9) Mining the evolutionary coupling of SPCP clone fragments to identify the cross-boundary SPCP clones.

Detecting the method-genealogy for a particular method involves identifying each instance of that method in each of the revisions where the method was alive. By detecting the genealogy of a method, we can determine how it changed during evolution. We detect clone genealogies by locating the clones detected from each revision to the already detected methods of that revision. The genealogy of a particular clone fragment also helps us determine how it evolved through the commits. We assign unique IDs to the method genealogies and clone genealogies to recognize them across revisions. We use NiCad [48] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [198, 201]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of

[1]Open source SVN repository. http://sourceforge.net/

[2]CTAGS: `http://ctags.sourceforge.net/`

a minimum size of 5 LOC with 20% dissimilarity threshold and blind renaming of identifiers. These settings are considered standard [196].

In this experiment we consider clone and non-clone fragments that reside within methods. We already mentioned that after detecting clones we locate them in the methods. A clone fragment is recognized by its starting and ending line numbers. However, non-clone fragments require explanation. If a method contains one or more clone fragments, then we consider the remaining code in the method as a non-clone fragment. If a method does not contain any clone fragment, then we consider the full method as a non-clone fragment. A fully cloned method does not contain a non-clone fragment.

5.5 Experimental Results and Discussion

We apply our implemented system on each of the six subject systems listed in Table 5.2. Our implemented system automatically determines the SPCP clones, and then identifies those SPCP clones that have relationships across their class boundaries. By analyzing these SPCP clones we answer the research questions mentioned in Table 5.1.

5.5.1 RQ 1: What Proportion of the SPCP Clones of a Subject System Have Cross-boundary Relationships?

Answering this research question is important. If it is observed that generally a significant proportion of the SPCP clones of a subject system have cross boundary relationships, then we can consider these clone fragments for tracking along with their cross-boundary relationships in order to update them consistently in future. However, if the number of cross-boundary SPCP clones is too low compared to the total number of SPCP clones in a system, then the detection of such SPCP clones might just be an overhead.

Methodology. We have already mentioned that in our previous study [163] we detected SPCP clone pairs. We merged these pairs to form groups of SPCP clone fragments. While detecting SPCP clone pairs in our previous work, we considered the following two constraints: **(1)** the two constituent SPCP clone fragments in a pair must co-change at least once during evolution, and **(2)** the two SPCP clone fragments in a pair must remain in two different methods. However, in this research work, we did not apply these constraints. Firstly, according to the definition of similarity preserving change pattern (SPCP) [163], two clone fragments from a particular clone class can follow an SPCP without being co-changed at all. So, in this experiment we detect all those SPCP clone pairs where the constituent clone fragments in a particular pair might not co-change at all. Secondly, two clone fragments remaining in the same method can also follow a similarity preserving change pattern and can be important candidates for refactoring. So, in this experiment we consider the same method case too. We show the amount of SPCP clones detected from each of the subject systems in Table 5.3. The proportion of SPCP clones having cross-boundary relationships is shown in Fig. 5.2. Fig. 5.2 shows that *in case of each of the subject systems a considerable amount of the SPCP clones have change couplings*

Table 5.3: Number of SPCP Clone Fragments

	Ctags	QMail.	Freecol	jEdit	Carol	Jabref
No. of SPCP Clones	229	31	860	902	238	418
% of SPCP Clones w.r.t all clones in the system	43.04	59	55.69	12	29.45	43.63
No. of CBSPCP Clones	116	28	340	442	125	121
% of CBSPCP Clones w.r.t all clones in the system	21.80	53.29	22.02	5.88	15.47	12.63
% of NCBSPCP Clones w.r.t all clones in the system	21.24	5.70	33.67	6.12	13.98	31

SPCP Clones = The clone fragments that evolved following SPCP

No. of CBSPCP Clones = Number of Cross-Boundary SPCP clone fragments

CBSPCP Clones = Cross-boundary SPCP clone fragments

NCBSPCP Clones = Non-cross-boundary SPCP clone fragments

with code fragments beyond their class boundaries. According to our consideration, such SPCP clones should not be considered for removal through refactoring. They should be tracked along with their cross-boundary relationships. Moreover, we have already mentioned that we detect cross-boundary SPCP clones considering the highest confidence rules. As higher confidence indicates stronger change coupling between the constituent code fragments in a rule, we can expect that each of our detected cross-boundary SPCP clones has strong change coupling with non-clone fragments and/or with clone fragments (may be SPCP clone fragments or not) from clone classes other than its own clone class.

For each of our candidate systems we develop two separate XML files containing respectively the cross-boundary and non-cross-boundary SPCP clones from that system. For each cross-boundary SPCP clone fragment, we list the other code fragments (along with their file and starting and ending line numbers) that are related to it. We determine groups of non-cross-boundary SPCP clones following the procedure we proposed in our previous study [163] and include the groups in the XML files. These XML files are available on-line[3].

Answer to RQ 1: In general, *a considerable proportion (overall 43% considering all six subject systems) of the SPCP clones of a subject system have strong change couplings with code fragments beyond their class boundaries. These cross-boundary SPCP clones should be considered as the important candidates for tracking. We should track them along with their cross-boundary couplings so that we can update them consistently during future evolution. Overall 10.27% of all clones in a software system are cross-boundary SPCP clones. The non-cross-boundary SPCP clones are conservative in the sense that*

[3]`http://goo.gl/r6gDm2`

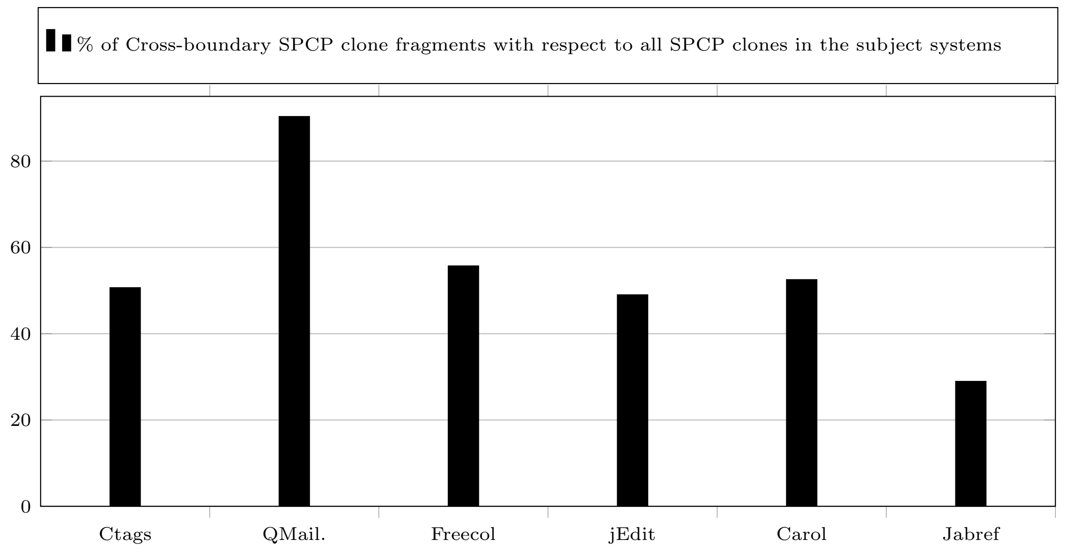


Figure 5.2: Percentage of cross-boundary SPCP clone fragments

they do not have change couplings with code fragments beyond their class boundaries. Thus, these should be considered as the important candidates for refactoring. According to our statistics considering all subject systems, overall 13.20% of all clones in a software system are non-cross-boundary SPCP clones.

5.5.2 RQ 2: Should We Discard the Cross-boundary SPCP Clones from Consideration While Taking Refactoring Decisions and also, Consider them for Tracking?

Answering this research question is the central objective of our research work. However, it is difficult to answer this question, because we do not have any empirically established set of characteristics such that a clone fragment that does not have any of the characteristics in the set should not be considered for removal through refactoring. In our previous study we considered that two or more clone fragments from a particular clone class might be important for refactoring only if they evolve following an SPCP (Similarity Preserving Change Pattern). The clone fragments that do not follow SPCP either evolve independently or rarely change during evolution. However, another important characteristic for a clone fragment to be considerable for refactoring is that it is not expected to have change couplings with other code fragments beyond its class boundary. We did not consider this issue in our previous study. If an SPCP clone fragment has such couplings, then it is better not to refactor it, rather it should be tracked along with its cross-boundary couplings so that the maintenance engineers can update them (i.e., the SPCP clone fragment and the code fragments coupled with it beyond its class boundary) consistently in the future.

In Fig. 5.1 (explained in the introduction) we showed an SPCP clone fragment having couplings beyond its class boundary. We also explained that removal of such an SPCP clone fragment might have ripple change effects[1] on the related code fragments remaining outside of its class boundary and also, might negatively affect the future evolution of these related code fragments. Thus, possibly we should not consider refactoring such an SPCP clone fragment. Here we should note that it is impossible to calculate how much negative effect can be caused in future because of the removal of such an SPCP clone fragment. However, we can have an idea from the number of change coupling relationships an SPCP clone fragment currently has. According to our consideration, if a number of fragments are related to a particular code fragment, then removal of that particular fragment might negatively affect the future evolution of all the related code fragments. Thus, by looking at the number of cross-boundary code fragments (i.e., the code fragments beyond class boundary) a particular SPCP clone fragment is related to we can have an idea of how much negative effect might be caused for the removal of that particular SPCP clone fragment.

In this research question, we determine how many change couplings an SPCP clone fragment can have beyond its class boundary. We automatically identify these change couplings by mining association rules as mentioned previously and then manually investigate the association rules to determine whether an SPCP clone fragment is really coupled with code fragments beyond its class boundary. If an SPCP clone fragment has change couplings with code fragments across its class boundary, then we can possibly decide that removal of this clone fragment can negatively affect these related code fragments and thus, can be harmful for future software evolution.

Methodology. We determine all the association rules associating an SPCP clone fragment with other code fragments (non-clone fragments or clone fragments from other clone classes rather than its own class) beyond its class boundary. As we previously mentioned, we consider each of those rules that have the highest confidence (confidence of 1) to ensure the likeliness of change coupling between the code fragments constituting a rule. Considering all the cross-boundary SPCP clone fragments in a subject system, we determine the average number of code fragments (clone fragments from other clone classes, and non-clone fragments) a cross-boundary SPCP clone fragment is associated to by association rules beyond its class boundary. The average numbers for each of the subject system are shown in Fig. 5.3.

From the figure we see that a cross-boundary SPCP clone fragment can exhibit strong evolutionary coupling (i.e., change coupling) with a considerable number of other code fragments beyond its class boundary. Most of these code fragments are clone fragments from different clone classes rather than its own clone class. We sort the cross-boundary SPCP clone fragments in decreasing order of the number of cross-boundary code fragments they are associated to and then analyze the association rules of the top ten SPCP clone fragments from each subject system. In the case of each of the association rules, we determine the commit operations where the constituent code fragments (a cross-boundary SPCP clone fragment and a code fragment beyond its class boundary) co-changed and whether they co-changed correspondingly. According to our manual investigation, each of our investigated cross-boundary SPCP clone fragments was actually related to the code fragments beyond its class boundary. We provide an example of a corresponding change in the following paragraphs.

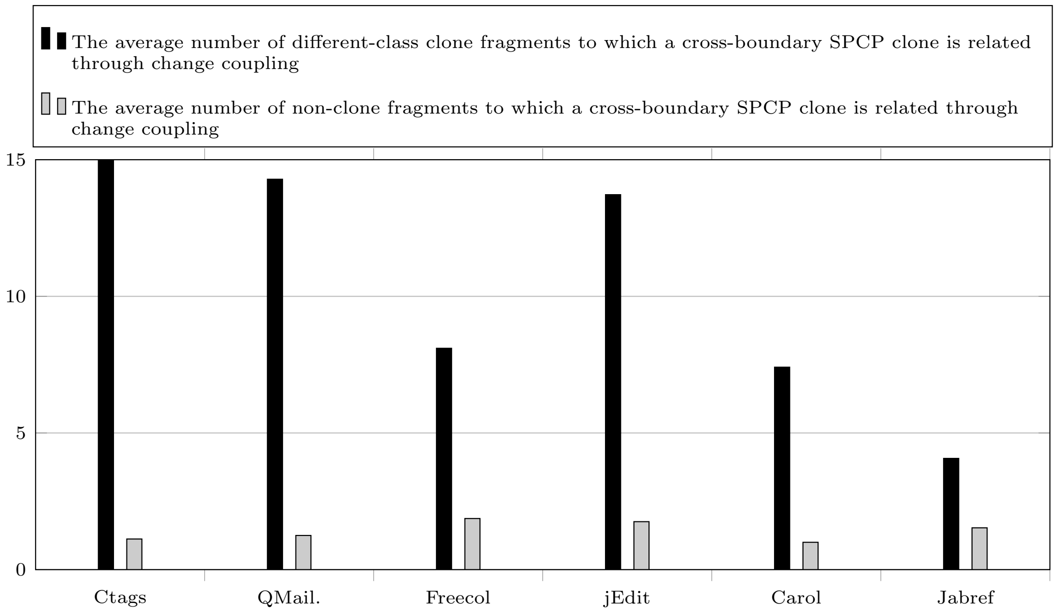
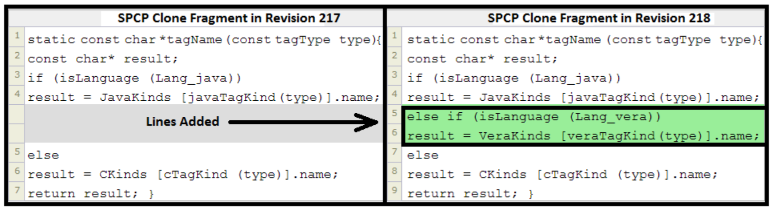


Figure 5.3: The average number of code fragments to which a cross-boundary SPCP clone fragment is related beyond its class boundary

Example: We provide an example of a corresponding change between an SPCP clone fragment and a non-clone fragment from our subject system Ctags. The SPCP clone fragment is a method called 'tagName' with signature *const char * tagName (const tagType type)*. The non-clone fragment is also a method with name 'includeTag' and signature *boolean includeTag(const tagType type, const boolean isFileScope)*. These two code fragments belong to the same file '*c.c*'. They co-changed in two commit operations applied on the revisions 217, and 242. Also, both of the association rules constructed from these two code fragments have the highest confidence (confidence = 1). Thus, these two code fragments always co-changed (changed together), and also, there is no commit operation where one fragment changed but the other did not. In such a situation, we can expect strong change coupling between these two code fragments (an SPCP clone fragment, and a non-clone fragment). We manually analyze the changes occurred to these code fragments in both of the commit operations. According to our analysis, the changes occurred to the two code fragments in each of the commit operations were corresponding and thus, the code fragments have change coupling. We show the changes occurred to the two code fragments in the commit operation applied on revision 217 in two figures: Fig. 5.4, and Fig. 5.5.

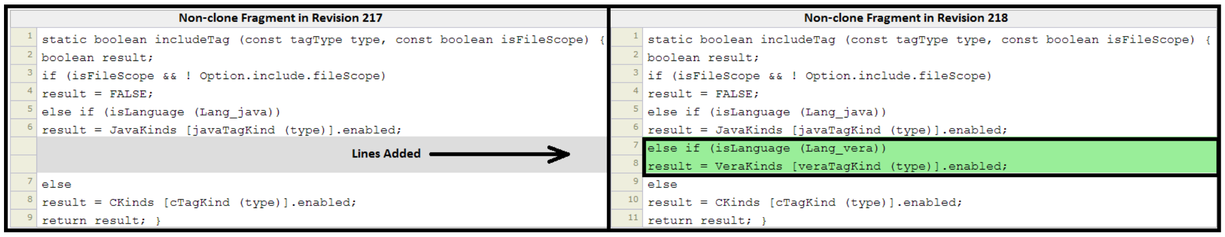
Fig. 5.4 shows the changes occurred to the SPCP clone fragment (*tagName*), and Fig. 5.5 shows the changes occurred to the non-clone fragment (*includeTag*). Each figure shows the two instances of the respective code fragment in two revisions 217, and 218. We also highlight the changes between these two instances. From Fig. 5.4 we see that two lines (statements) were added to the SPCP clone fragment because of the commit on revision 217. If we take a look at Fig. 5.5, we can see that the same statements were


SPCP Clone Fragment in Revision 217
SPCP Clone Fragment in Revision 218
Lines Added

```
static const char *tagName (const tagType type){
const char* result;
if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].name;
else
result = CKinds [cTagKind (type)].name;
return result; }
```

```
static const char *tagName (const tagType type){
const char* result;
if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].name;
else if (isLanguage (Lang_vera))
result = VeraKinds [veraTagKind (type)].name;
else
result = CKinds [cTagKind (type)].name;
return result; }
```

Figure 5.4: Changes occurred to an SPCP clone fragment in the commit operation applied on revision 217 of our subject system Ctags.


Non-clone Fragment in Revision 217
Non-clone Fragment in Revision 218
Lines Added

```
static boolean includeTag (const tagType type, const boolean isFileScope) {
boolean result;
if (isFileScope && ! Option.include.fileScope)
result = FALSE;
else if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].enabled;
else
result = CKinds [cTagKind (type)].enabled;
return result; }
```

```
static boolean includeTag (const tagType type, const boolean isFileScope) {
boolean result;
if (isFileScope && ! Option.include.fileScope)
result = FALSE;
else if (isLanguage (Lang_java))
result = JavaKinds [javaTagKind (type)].enabled;
else if (isLanguage (Lang_vera))
result = VeraKinds [veraTagKind (type)].enabled;
else
result = CKinds [cTagKind (type)].enabled;
return result; }
```

Figure 5.5: Changes occurred to a non-clone fragment in the commit operation applied on revision 217 of our subject system Ctags.

also added to the non-clone fragment in the same commit operation. The changes occurred to the two code fragments imply that those (i.e., the changes) were made focusing on the consistency of the two code fragments. In other words, the SPCP clone fragment and the non-clone fragment co-changed correspondingly in the commit operation. The changes occurred to these two code fragments in the commit 242 are also corresponding according to our manual investigation. Thus, we can expect that these two code fragments are related although there is no caller-callee relationship. According to our analysis, these code fragments have a tendency of co-changing consistently. In such a situation, deletion of the SPCP clone fragment (‘tagName’) through refactoring might negatively affect the future evolution of the non-clone fragment. However, if we still want to remove this SPCP clone fragment, then we must take the relationship between this fragment and the non-clone fragment (‘includeTag’) into consideration so that removal of the SPCP clone fragment does not leave the non-clone fragment in an inconsistent state and also does not affect the future evolution of the non-clone fragment.

Answer to RQ 2. From our discussion and analysis we can say that *the SPCP clone fragments having cross-boundary relationships should not be considered for removal through refactoring. They should be tracked along with their relationships for their consistent updates in future. However, in case we want to remove such a clone fragment, we must consider and analyze its relationships (change couplings in our experiment) with the other code fragments beyond its class boundary so that these other code fragments are not left in an inconsistent state because of the removal of the SPCP clone fragment.*

5.5.3 RQ 3: Do Cross-boundary SPCP Clones Have Higher Change-proneness Than the Non-cross-boundary SPCP Clones?

Motivation. From our answer to the previous research question we understand that a cross-boundary SPCP clone fragment has a tendency of maintaining consistency (i.e., changing consistently) with non-clone fragments as well as with clone fragments from other clone classes. Also, as it is an SPCP clone fragment, it also maintains consistency with other SPCP clone fragment(s) in its own clone class. From this we suspect that possibly cross-boundary SPCP clones have higher change-proneness compared to the non-cross-boundary SPCP clones. Also, our detection mechanism of cross-boundary SPCP clone fragments described in Section 5.3.3 ensures that each cross-boundary SPCP clone fragment must co-change at least twice with a non-clone fragment or with a clone fragment from other clone class. In these circumstances it is highly likely that cross-boundary SPCP clones will exhibit higher change-proneness than the non-cross-boundary SPCP clones. However, we investigate this matter in detail in this research question. Such an investigation is important from the perspective of software maintenance.

Literature [145, 146, 156] shows that code clones have higher change-proneness compared to non-cloned code. However, some studies [119, 163] also show that not all clone fragments in a software system are highly change-prone, and even some clone fragments never change during software evolution. Thus, this is important to identify which clones are highly change-prone and to investigate the reasons behind their high change-proneness. If our investigation shows that cross-boundary SPCP clones have significantly higher change-proneness than the non-cross-boundary SPCP clones, then the cross-boundary SPCP clones should be regarded as the hot spots in a software system. In general, highly change-prone code fragments have higher possibilities of introducing bugs and inconsistencies to the software system if the changes occurred to them are not properly propagated to the other related code fragments because of unconsciousness. We know that the cross-boundary SPCP clones are related with many other code fragments beyond their class boundaries. If such an SPCP clone fragment appears to be highly change-prone, then we should be more conscious about the changes that occurred to it. If a change occurred to it is not properly propagated to the other related code fragments both inside and outside of its class, then the software system might become inconsistent. Thus, if cross-boundary SPCP clones have high change-proneness, then it is very much important that they should be tracked along with their relationships so that we can change them consistently during future evolution. In this research question we investigate whether cross-boundary SPCP clone fragments have significantly higher change-proneness compared to the non-cross-boundary SPCP clones.

Methodology. To quantify the change-proneness of a particular SPCP clone fragment, we measure the number of times it changed (i.e., the number of commits where it changed) during the whole evolution period of the software system as we did in a previous study [160]. We at first determine all the SPCP clone fragments of a particular subject system, and then separate them into two groups. One group contains the cross-boundary SPCP clone fragments. We call this group the *cross-boundary-group*. The other group

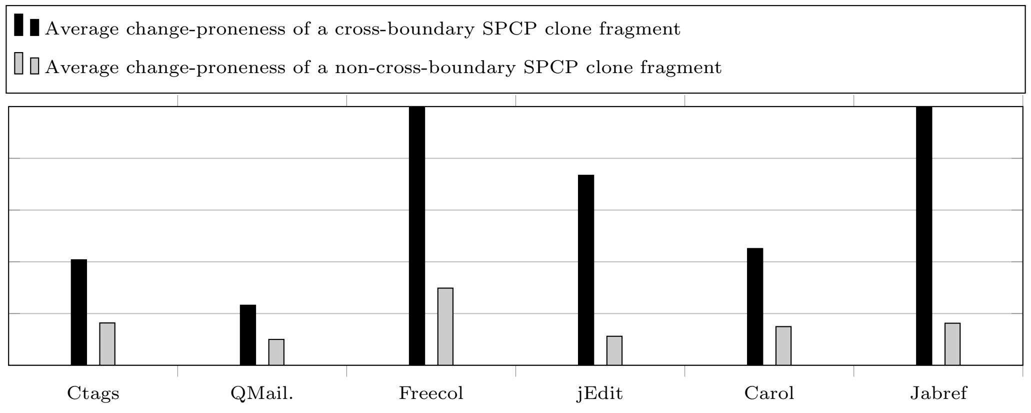


Figure 5.6: Comparison of change-proneness between cross-boundary and non-cross-boundary SPCP clone fragments

contains the non-cross-boundary SPCP clone fragments. We call this group the *non-cross-boundary-group.* We measure the change-proneness of each of the SPCP clone fragments included in each of the two groups. We then determine the average change-proneness per group. These average values are shown in Fig. 5.6.

Fig. 5.6 shows that in case of each of the subject systems the change-proneness of the cross-boundary SPCP clones is much higher than the change-proneness of the non-cross-boundary SPCP clones. This is expected according to our previous discussion. However, we also wanted to investigate whether the cross-boundary SPCP clones exhibit significantly higher change-proneness than the non-cross-boundary SPCP clones. We perform our investigation in the following way.

In case of this investigation, we did not rely on the average change-proneness values, rather we used the actual change-proneness value of each of the SPCP clone fragments in each of the groups. For each of the subject systems, we performed the *Mann-Whitney-Wilcoxon* Tests [3, 4] on the change-proneness values of the SPCP clone fragments of the two groups: *cross-boundary-group*, and *non-cross-boundary-group.* We determine whether the change-proneness values in the *cross-boundary-group* are significantly higher than the change-proneness values in the *non-cross-boundary-group.* For our six subject systems we performed six tests and observed that in case of each test, the difference between the change-proneness values in the two groups was highly significant with *p-value* < 0.001 (for both one-tailed and two-tailed tests). Thus, we can say that the change-proneness values in the *cross-boundary-group* are significantly higher than those in the *non-cross-boundary-group.* The test details are shown in Table 5.4.

Answer to RQ 3. According to our analysis *cross-boundary SPCP clones have significantly higher change-proneness than the non-cross-boundary SPCP clones. Thus, cross-boundary SPCP clones should be tracked along with their cross-boundary couplings with high priority so that we can change them consistently during future evolution.*

Table 5.4: Mann-Whitney-Wilcoxon Tests Regarding the Significance of Difference between Change-proneness Values of Cross-Boundary-Group and Non-Cross-Boundary-Group

	Ctags	QMail.	Freecol	jEdit	Carol	Jabref
SCBG	116	28	340	442	125	121
SNCBG	113	3	520	460	113	297
p-Value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001

SCBG = No. of Samples (SPCP clone fragments) in Cross-Boundary-Group

SNCBG = No. of Samples in Non-Cross-Boundary-Group

p-Value = Probability Value. If it is less than 0.05 then the difference between the two samples are considered significant.

The main reason behind this high change-proneness of the cross-boundary SPCP clones is that each of these has change couplings beyond its class boundary. From Fig. 5.3 we see that a cross-boundary SPCP clone fragment is generally coupled with a high number of code fragments beyond its class. In other words, these clone fragments are highly coupled. Generally highly coupled code fragments are not desirable in a software system, because changes to such a code fragment might cause ripple change effects to the related code fragments. Thus, the cross-boundary SPCP clones are the possible places in a software system where we can think of possible restructuring to minimize their couplings. Also, as the cross-boundary SPCP clones are not suitable for removal through refactoring, we propose efficient tracking of these clone fragments along with their cross-boundary relationships.

5.5.4 RQ 4: Which Types of SPCP Clones Have Higher Possibility of Having Cross-boundary Relationships?

In this research question we wanted to see a comparative scenario of the proportions of SPCP clone fragments as well as of the proportions of cross-boundary SPCP clone fragments in different types of clones of our candidate systems. The findings from this research question can help us understand two important things: **(1)** which type(s) of SPCP clone fragments have lower possibilities of having cross-boundary relationships and thus, are more suitable for refactoring, and **(2)** which type(s) of SPCP clones have higher possibility of having cross-boundary relationships and thus, are more suitable for tracking. We perform our analysis in the following way.

Methodology. We detect each of the three major types (Type 1, Type 2, and Type 3) of clone fragments separately using the NiCad [48] clone detector and then automatically determine the SPCP clone fragments having cross-boundary change couplings in each clone type. We determine the percentage of such SPCP clone fragments with respect to the total number of SPCP clone fragments considering each clone type. Table 5.5 shows the numbers of SPCP clone fragments and cross-boundary SPCP clone fragments in three clone types. The percentages of the cross-boundary SPCP clones are shown in the graph of Fig. 5.7.

Table 5.5: Cross-Boundary SPCP Clones in Three Clone-types

	Type 1		Type 2		Type 3	
	SPCP	CB-SPCP	SPCP	CB-SPCP	SPCP	CB-SPCP
Ctags	6	0	42	12	133	65
QMailAdmin	0	0	11	9	10	9
Freecol	75	2	143	42	534	164
jEdit	146	66	69	10	458	179
Carol	2	0	48	21	142	58
Jabref	13	0	84	12	236	70

SPCP = Count of SPCP clone fragments

CB-SPCP = Count of cross-boundary SPCP clone fragments

From Table 5.5 we see that the number of SPCP clone fragments is generally the highest in Type 3 case among all three types except QMailAdmin. Also, for three subject systems Ctags, Carol, and Jabref we did not get any cross-boundary SPCP clone fragments in Type 1 case. Fig. 5.7 shows that for most of the subject systems except jEdit and Carol, the percentage of cross-boundary SPCP clones is the highest in Type 3 clones. However, the percentages of cross-boundary SPCP clones are also considerable for the Type 2 cases. In case of jEdit and Carol, the percentages regarding Type 1 and Type 2 cases are the highest ones respectively. Considering all the subject systems it seems that the percentage of cross-boundary SPCP clones is generally the lowest in Type 1 case and the highest in Type 3 case.

Answering RQ 4. According to our investigated subject systems, *the SPCP clones in both Type 2 and Type 3 cases have high possibilities of having cross-boundary relationships. However, the proportions of cross-boundary SPCP clones are generally very low for the Type 1 case.*

As Type 1 SPCP clones have lower probabilities of having cross-boundary relationships compared to the SPCP clones in the other two types (Type 2, and Type 3), Type 1 SPCP clones should be considered for refactoring with higher priority. According to our observation, we can even exclude Type 1 clones from consideration while detecting cross-boundary SPCP clones. However, the amounts of cross-boundary SPCP clones in the other two types are considerable. According to our analysis, we should detect cross-boundary SPCP clones considering these two clone types (Type 2, and Type 3) so that we can exclude such SPCP clones from refactoring decision and also, can keep track of them along with their relationships for consistent updates in future.

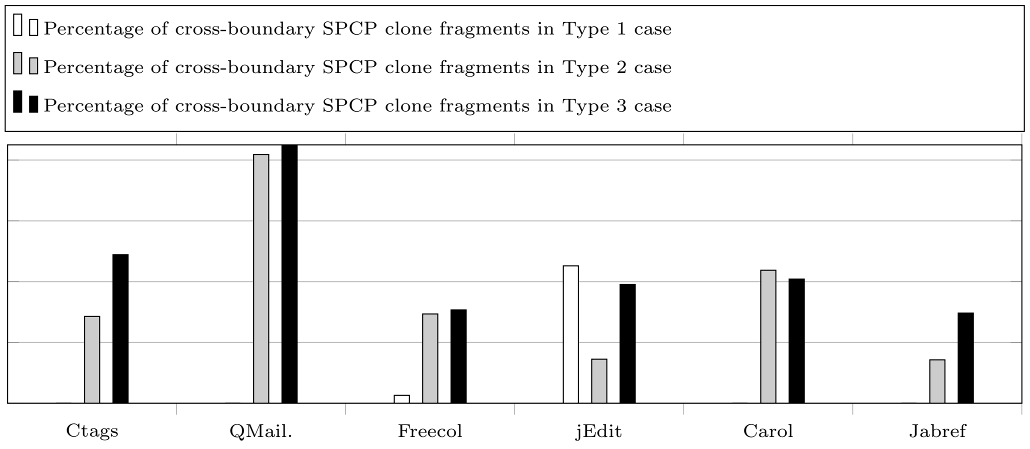


Figure 5.7: Comparison of change-proneness between cross-boundary and non-cross-boundary SPCP clone fragments

5.5.5 Ranking of SPCP Clones for Tracking and Refactoring

From our previous analysis it is clear that we recommend the cross-boundary SPCP clones as the important candidates for tracking and the non-cross-boundary SPCP clones as the important candidates for refactoring. In our previous study we ranked the SPCP clones for refactoring on the basis of their similarity preserving co-changes [163]. According to our findings in this research work, we can apply such a ranking for refactoring the non-cross-boundary SPCP clones. However, the cross-boundary SPCP clones should be treated in a different way. We suggest to rank them on the basis of the number of cross-boundary code fragments they are related to. An SPCP clone fragment having a higher number of cross-boundary relationships can be given a higher rank compared to the other SPCP clone fragments having comparatively a lower number of cross-boundary relationships. In general, the more a code fragment is coupled, the more it is challenging to be changed. Tracking of such a code fragment with all of its couplings can help us change it consistently in future. Thus, we believe that our consideration regarding the ranking of cross-boundary SPCP clones for tracking is reasonable. We rank the cross-boundary SPCP clones detected from each of our candidate systems considering our proposed ranking mechanism. The XML files containing these ranked cross-boundary SPCP clones are available on-line[4].

5.6 Threats to Validity

We used the NiCad clone detector [48] for detecting clones. For different settings of NiCad, the statistics that we obtain might be different. Wang et al. [254] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [196] and with these settings NiCad can detect clones with high precision and recall [198, 201].

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the cross-boundary relationships of SPCP clones. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important and can help us to better manage code clones by minimizing clone refactoring effort and suggesting important candidates for tracking.

5.7 Related Work

Numerous studies have been conducted regarding the detection, impact analysis [75,78,104,119,131,132,145,146], management [251], refactoring [25,105,239,277] and tracking [61,95,152,244] of code clones.

A number of refactoring approaches [25, 105] select clones for refactoring on the basis of the abstract syntax tree representation of the code base. Higo et al. [83] selected clones for refactoring (implementing a tool called CCShaper) based on the lexical analysis of the source code. Bouktif et al. [38] proposed an optimal schedule for refactoring clones using genetic algorithm. Zibran and Roy [277] proposed a conflict aware optimal scheduling algorithm for clone refactoring using constraint programming. Tairas and Gray [239] developed an Eclipse plug-in, CeDAR, for the purpose of clone refactoring.

A number of techniques for clone tracking also exist. Duala-Ekoko and Robillard [61] implemented an Eclipse plug-in 'CloneTracker' for tracking clones. Jablonski and Hou [95] developed a tool called CReN to track copy-paste activities. Miller and Myer [152] proposed a technique for simultaneous editing of multiple clone fragments. Toomin et al. [244] developed a clone tracking tool called *Codelink*.

We see that a number of studies and techniques for clone refactoring and tracking already exist. These techniques and tools consider all clones in a software system for refactoring and tracking. However, our study in Chapter 5 is different and unique in the sense that we detect only those clones that are important for refactoring or tracking.

Previously we conducted a study [163] to identify important clones for refactoring. We defined a particular clone change pattern called SPCP (*Similarity Preserving Change Pattern*) and proposed a mechanism for detecting clones that follow SPCP. The non-SPCP clones are not important for refactoring or tracking, because they either evolve independently or rarely change during evolution. We proposed that the SPCP clones could be important candidates for refactoring. However, an SPCP clone fragment can have couplings with other code fragments beyond its class boundary. Such cross-boundary SPCP clones should not be considered for removal through refactoring. They should be considered as important candidates for tracking. In this research work we propose a mechanism for automatically identifying cross-boundary SPCP clones. We also propose a particular ranking mechanism for prioritizing the cross-boundary SPCP clones for tracking. We perform an in-depth empirical study on both the cross-boundary and non-cross-boundary SPCP clones. From our empirical evaluation we suggest that cross-boundary SPCP clones are the important candidates for tracking and the non-cross-boundary SPCP clones are the important candidates for refactoring.

In a previous study [164], we analyzed the evolutionary couplings of clone fragments in order to predict their future co-change candidates. However, our study in Chapter 5 is different in the sense that here we focus on identifying clones that are important for tracking or refactoring. Our implemented system in this research work can automatically extract the important clones for refactoring (non-cross-boundary SPCP clones) as well as for tracking (cross-boundary SPCP clones). Thus, our study is important for better management of code clones.

5.8 Summary

In Chapter 5, we perform an in-depth empirical study on the identification of clone fragments that are important for refactoring or tracking. We first detect all the SPCP clones (i.e., the clone fragments that evolved following a similarity preserving change pattern) in a software system. We analyze the evolutionary coupling of the SPCP clones and identify those SPCP clones that have change couplings (i.e., evolutionary couplings) with other code fragments beyond their class boundaries. According to our consideration, these cross-boundary SPCP clones should not be considered for removal through refactoring, because removal of such clone fragments might negatively affect the future evolution of the related code fragments beyond the class boundaries. We consider these as the important candidates for tracking. We suggest the non-cross-boundary SPCP clones to be the important candidates for refactoring. Our implemented prototype tool can automatically identify both cross-boundary and non-cross-boundary SPCP clones by analyzing software evolution history.

We apply our prototype tool on six diverse subject systems written in two programming languages and detect the cross-boundary and non-cross-boundary SPCP clones. According to our empirical study involving rigorous manual analysis, overall 43% of the SPCP clone fragments have cross-boundary relationships. Cross-boundary SPCP clones exhibit significantly higher change-proneness than the non-cross-boundary SPCP clones. The reason behind this higher change-proneness is that cross-boundary SPCP clones are generally highly coupled. As lower coupling is always desirable in software systems, cross-boundary SPCP clones are possible places in a software system where we can think of possible restructuring to minimize their coupling. We also observe that the percentage of cross-boundary SPCP clones is generally the lowest in Type 1 case, and the highest in Type 3 case. We believe that automatic detection of cross-boundary as well as non-cross-boundary SPCP clones will help us in better management of code clones in terms of both tracking and refactoring.

While our investigations in Chapter 3, 4, and 5 involve identifying and ranking important clones for refactoring and tracking on the basis of their evolutionary coupling, we believe that bug-proneness of code clones should also be considered when prioritizing them for management. Our next chapter (Chapter 6) presents our investigation on clone bug-proneness.

Chapter 6

A Comparative Study on the Bug-Proneness of Different Types of Code Clones

In our previous chapters we identified code clones that are important for refactoring or tracking. We called these code clones SPCP clones (i.e., code clones that evolved by following a Similarity Preserving Change Pattern called SPCP). We ranked these important clones on the basis of their evolutionary coupling. However, we realize that bug-proneness of code clones should also be considered when prioritizing or ranking them for management. Focusing on this we perform a comparative study on the bug-proneness of different types of code clones so that clone-types with higher bug-proneness can be given higher priorities for management. We also analyze which type of bug-prone clones have high possibilities of evolving following an SPCP. Chapter 6 contains the details of our clone bug-proneness study.

The rest of the chapter is organized as follows. Section 6.2 describes the terminology, Section 6.4 discusses the experimental steps, Section 6.5 answers the research questions by presenting and analyzing the experimental results, Section 6.6 mentions the possible threats to validity, Section 6.7 discusses the related work, and finally, Section 6.8 concludes the chapter by summarizing our study findings.

6.1 Introduction and Motivation

According to a number of studies [29, 44, 51, 78, 91, 97, 141, 142, 263], code clones are directly related to bugs and inconsistencies in a software system. However, although there are different types of code clones, none of the existing studies investigate the comparative bug-proneness of these different clone-types. Such an investigation is important because it can help us identify which type(s) of clones have the highest tendency of exhibiting bug-proneness and thus, should be considered to be the important ones for management such as refactoring and tracking. Focusing on this issue in this research work we investigate the comparative bug-proneness of the major types of code clones: Type 1, Type 2, and Type 3 (defined in Chapter 2). In particular, we answer four important research questions listed in Table 6.1. According to our in-depth investigation on thousands of revisions of seven diverse subject systems written in two different programming languages (C and Java) we can state that:

Table 6.1: Research Questions

SL	Research Question
RQ 1	Which clone types have a higher possibility of experiencing bug fixing changes?
RQ 2	Do the clone fragments from the same clone class co-change (i.e., change together) consistently during a bug-fix?
RQ 3	What proportion of the clone fragments that experienced bug-fixing changes are SPCP clones?

(1) Type 3 clones have a higher bug-proneness compared to Type 1 and Type 2 clones. The bug-proneness of Type 1 clones is the lowest among the three clone-types. Our statistical significance tests show that Type 3 clones have a significantly higher bug-proneness than Type 1 clones.

(2) Type 3 clones have the highest likeliness of being co-changed (i.e., getting changed together) consistently among the three clone-types when changed to fix a bug.

(3) Type 3 bug-fix clones have the highest possibility of evolving following a *Similarity Preserving Change Pattern* called SPCP. According to our previous studies [162, 163], SPCP clones (i.e., clones that evolve following a Similarity Preserving Change Pattern) are the important ones to consider for clone management.

Our experimental results imply that Type 3 clones should be given a higher priority than the other two clone-types when making clone management decisions (such as clone refactoring, or tracking) and our findings (points 2 and 3 above) can be used to rank code clones during clone management. In our previous studies [162, 163] we detected and ranked SPCP clones for refactoring and tracking on the basis of their co-change tendencies. However, we should also consider their bug-proneness. Our implemented prototype tool is capable of automatically detecting SPCP clones that exhibited bug-proneness during evolution. Thus, it can help us rank clones considering their bug-proneness too.

6.2 Terminology

We conduct our experiment considering both exact (Type 1) and near-miss clones (Type 2 and Type 3 clones). We defined these clone-types in Chapter 2.

6.3 Similarity Preserving Change Pattern (SPCP)

In our previous studies [162,163] we showed that the code clones that evolve following a *Similarity Preserving Change Pattern* (SPCP) are the important ones for refactoring or tracking. A *Similarity Preserving Change Pattern* consists of a *Similarity Preserving Change* and/or a *Re-synchronizing Change*.

Table 6.2: Subject Systems

Systems	**Lang.**	**Domains**	**LLR**	**Revisions**
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
BRL-Cad	C	3-D Modeling	39,309	735
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545

LLR = LOC in the Last Revision

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied to this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received a *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. A re-synchronizing change consists of a *diverging change* followed by a *converging change*. Let us consider two code fragments that are clones of each other in a particular revision. A commit operation C_i was applied to this revision, and any one or both of the fragments received some changes in such a way that the code fragments were not considered clones of each other in the next revision. We say that the code fragments experienced a *diverging change*. However, in a later commit operation C_{i+n} ($n \geq 1$) any one or both of the code fragments received some changes, and because of these changes the code fragments again became clones of each other. We say that the code fragments experienced a *converging change* in commit C_{i+n}. A *diverging change* followed by a *converging change* is termed a *re-synchronizing change*.

6.4 Experimental Steps

We perform investigation on seven subject systems (Table 6.2) downloaded from an on-line SVN repository [5].

6.4.1 Preliminary Steps

We perform the following preliminary steps before analyzing bug-proneness: **(1)** Extraction of all revisions (as mentioned in Table 6.2) of each of the subject systems from the online SVN repository; **(2)** Method detection and extraction from each of the revisions using CTAGS [2]; **(3)** Detection and extraction of code clones from each revision by applying the NiCad [48] clone detector; **(4)** Detection of changes between every two consecutive revisions using *diff*; **(5)** Locating these changes to the already detected methods as well as clones of the corresponding revisions; **(6)** Locating the code clones detected from each revision to the methods of that revision; **(7)** Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [145]; **(8)** Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy; and **(9)** Detection of SPCP clone fragments by analyzing clone change patterns. For completing these steps we use the tool SPCP-Miner [170].

We use NiCad [48] for detecting clones because it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [198,201]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold and blind renaming of identifiers. For different settings of a clone detector the clone detection results can be different and thus, the findings regarding the bug-proneness of code clones can also be different. Thus, selection of reasonable settings (i.e., detection parameters) is important. We used the mentioned settings in our research, because in a recent study [231] Svajlenko and Roy show that these settings provide us with better clone detection results in terms of both precision and recall.

Clone Genealogies of Different Clone-Types. SPCP-Miner [170] detects clone genealogies considering each clone-type (Type 1, Type 2, and Type 3) separately. Considering a particular clone-type it first detects all the clone fragments of that particular type from each of the revisions of the candidate system. Then, it performs origin analysis of these detected clone fragments and builds the genealogies. Thus, all the instances in a particular clone genealogy are of a particular clone-type. An instance is a snap-shot of a clone fragment in a particular revision. A detailed elaboration of the genealogy detection approach is presented in our previous study [163]. As we obtain three separate sets of clone genealogies for three different clone-types, we can easily determine and compare the bug-proneness of these clone-types.

Tackling Clone-Mutations. Xie et al. [263] found that mutations of the clone fragments (i.e., a particular clone fragment may change its type) might occur during evolution. If a particular clone fragment is considered of a different clone-type during different periods of evolution, SPCP-Miner extracts a separate clone-genealogy for this fragment for each of these periods. Thus, even with the occurrences of clone-mutations, we can clearly distinguish which bugs were experienced by which clone-types.

6.4.2 Bug-proneness Detection Technique

For a particular candidate system, we first retrieve the commit messages by applying the 'SVN log' command. A commit message describes the purpose of the corresponding commit operation. We automatically infer the commit messages using the heuristic proposed by Mockus and Votta [153] in order to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) was necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those clone fragments that are related to the bug-fix. We also determine the number of changes that occurred to such a clone fragment in a bug-fix commit using the UNIX *diff* command.

The way we detect the bug-fix commits was also previously followed by Barbour et al. [29]. Barbour et al. [29] detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to bug-fix. In our study we detect bug-fix commits in the same way as they detected, however, our study is not limited to the late propagation clones only. We investigate the bug-proneness of all clone fragments in a software system. Also, Barbour et al. [29] did not investigate Type 3 clones in their study. We consider Type 3 clones in our bug-proneness analysis. Moreover, we compare the bug-proneness of different types of code clones from different perspectives. None of the existing studies do such comparisons.

6.5 Experimental Results and Analysis

We present and analyze our experimental results in the following subsections in order to answer the research questions mentioned in Table 6.1.

RQ 1: Which clone types have a higher possibility of experiencing bug fixing changes?

Rationale. It is important to know which types of clones have a higher probability of experiencing bug-fix changes compared to the others. The code clones exhibiting higher bug-proneness should be given higher priorities when making clone management decisions (such as refactoring and tracking). Refactoring or tracking of such clone fragments (i.e., highly bug-prone clones) could help us minimize the probability of the occurrences of bugs or inconsistencies in these fragments in the future. In a previous study [158] we found Type 1 and Type 2 clones to be more unstable (i.e., change-prone) than Type 3 clones. However, there is no empirical study on the correlation between change-proneness and bug-proneness of code clones. Thus, we should not infer the bug-proneness of clone types from their change-proneness. A comparative study on the bug-proneness of different types of code clones is important. We perform our investigations for answering *RQ 1* in the following two ways.

- **Investigation 1:** Investigation regarding the proportion of bug-fix changes experienced by the code clones.
- **Investigation 2:** Investigation regarding the proportion of code clones experiencing bug-fix changes.

Table 6.3: Percentage of Changes Related to Bug-fix

	Type 1		Type 2		Type 3	
Systems	TNC	PCB	TNC	PCB	TNC	PCB
Ctags	40	10%	84	11.90%	161	14.29%
Camellia	21	9.52%	20	0%	259	14.67%
BRL-Cad	322	0.93%	41	19.51%	215	7.9%
Freecol	134	20.89%	126	21.43%	766	30.42%
jEdit	1594	25.91%	145	47.59%	1265	43.08%
Carol	245	14.69%	279	21.86%	1123	23.15%
Jabref	304	4.27%	244	6.56%	1164	6.44%

TNC = Total Number of Changes that occurred to the Clones

PCB = Percentage of Changes related to a Bug-fix.

Investigation 1. *Investigating what proportion of the changes that occurred to the clone fragments of different clone-types are related to a bug-fix.*

Considering the code clones of a particular clone-type of a particular subject system, we first determine how many changes occurred to the code clones during the period of evolution (consisting of the revisions mentioned in Table 6.2). Then we identify which of these changes were related to a bug-fix. Finally, we calculate the percentage of changes related to a bug-fix considering each clone-type of each of the candidate systems using Eq. 6.1.

$$PCB = NBC * 100 \ / \ TNC \qquad (6.1)$$

In Eq. 6.1, TNC is the total number of changes that occurred to the code clones of a particular clone type of a particular subject system, NBC is the number of bug-fix changes that occurred to those code clones, and lastly, PCB denotes the percentage of changes related to a bug-fix with respect to all the changes (TNC) that occurred to those code clones. Table 6.3 shows the TNC and PCB for each clone-type of each of the subject systems. We also plot the percentages (PCB) in Fig. 6.1 to get a visual understanding regarding their comparison.

From Fig. 6.1 we see that for six out of seven subject systems (i.e., except Camellia) the percentage of bug-fix changes is the lowest for the Type 1 case. For four systems (Ctags, Camellia, Freecol, and Carol) the percentage regarding the Type 3 case is the highest among the three cases (Type 1, Type 2, and Type 3). For the remaining three systems, the percentage regarding the Type 2 case is the highest. The figure also shows the overall percentages (i.e., measured over all the subject systems) for the three clone-types. We see that the percentage of bug-fix changes is the highest in Type 3 case. The overall percentages regarding the other two cases (Type 1, and Type 2) are almost the same. We calculate the overall percentages using the following equation.

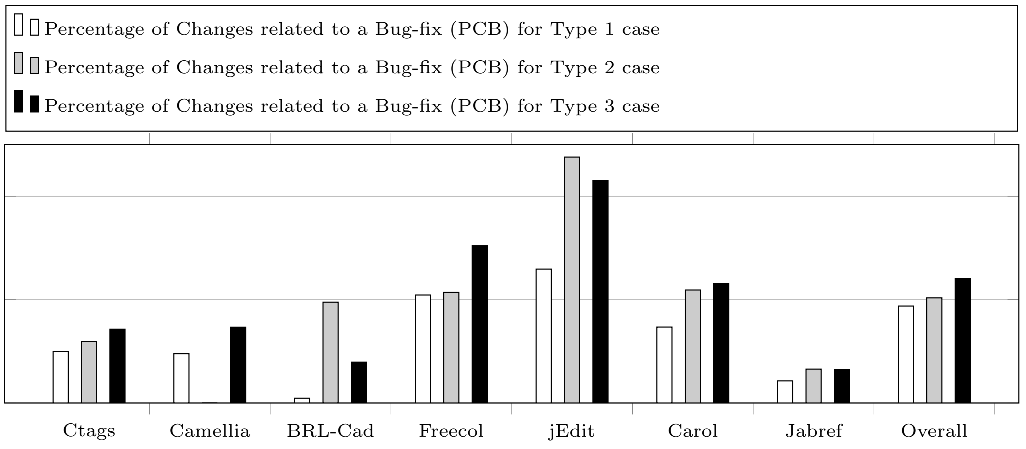


Figure 6.1: Comparison regarding the percentage of bug-fix changes that occurred to the clone fragments.

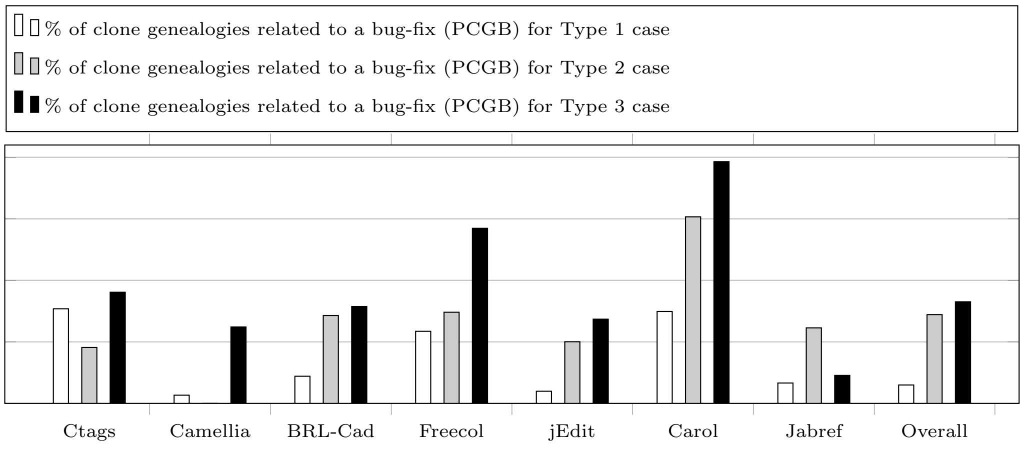


Figure 6.2: Comparison regarding the percentage of clone fragments that experienced bug-fixing changes.

$$OP_{Type\ i} = \frac{100 * \sum_{for\ all\ systems} NBC_{Type\ i}}{\sum_{for\ all\ systems} TNC_{Type\ i}} \tag{6.2}$$

$OP_{Type\ i}$ is the overall percentage of bug-fix changes occurred to the $Type\ i$ clones. $NBC_{Type\ i}$ is the number of bug-fix changes to the $Type\ i$ clones of a particular subject system. $TNC_{Type\ i}$ is the total number of changes that occurred to the $Type\ i$ clones of a subject system.

Investigation 2. *Investigating what proportion of the clone fragments in different clone-types are related to bug-fix changes?*

We mentioned (in Section 6.4) that we determine the genealogies of the detected clone fragments. Considering each clone-type of each of the subject systems we determine how many clone genealogies were created during the evolution and how many of these experienced a bug-fix. From these two values we determine the

Table 6.4: Percentage of Clones Related to Bug-fix

	Type 1		Type 2		Type 3	
Systems	TNCG	PCGB	TNCG	PCGB	TNCG	PCGB
Ctags	52	7.69%	88	4.55%	155	9.03%
Camellia	300	0.67%	48	0%	177	6.21%
BRL-Cad	136	2.2%	28	7.14%	127	7.87%
Freecol	239	5.86%	162	7.41%	752	14.23%
jEdit	7398	0.99%	399	5.01%	2688	6.85%
Carol	415	7.47%	211	15.17%	682	19.65%
Jabref	483	1.66%	228	6.14%	1363	2.27%

TNCG = Total Number of Clone Genealogies created during evolution.

PCGB = Percentage of Clone Genealogies related to a Bug-fix.

percentage of clone genealogies that experienced bug-fixing changes using a similar equation to Eq. 1. Table 6.4 shows the total number of clone genealogies (the column *TNCG*) as well as the percentage of bug-fix clone genealogies (the column *PCGB*) for each clone-type of each of the candidate systems. We also plot the percentages (*PCGB*) in the graph of Fig. 6.2 for easily understanding the comparison of bug-proneness among the three clone-types. The figure also shows the overall percentages of clone genealogies related bug-fix for each clone-type. Overall percentages were calculated using a similar equation to Eq. 2.

From Fig. 6.2 we see that for all of the subject systems except Jabref, the percentage of clones related to bug-fix is the highest in the Type 3 case. Also, the percentage of bug-fix clones is the lowest in the Type 1 case for most of the systems except Ctags, and Camellia. The overall percentages of the bug-fix clones in the three clone-types provide such implications. Finally, the graph in Fig. 6.2 implies that Type 3 clones generally have a much higher tendency of experiencing bug-fixing changes compared to the clone fragments of the other two clone-types.

Statistical Significance Tests. We were also interested to investigate whether Type 3 clones have a significantly higher tendency of experiencing bug-fixing changes compared to the clones of the other two types. We performed Mann-Whitney-Wilcoxon (MWW) tests [3, 4] considering the percentages of the bug-fix clone genealogies of the three cases (Type 1, Type 2, and Type 3) as recorded in Table 6.4. We first determine whether the percentages regarding the Type 3 case are significantly higher than those of the Type 1 case. Our MWW test result implies that *the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 case with a* p-value *of 0.026 (for two tailed test) which is less than 0.05.* However, we observe that the percentages for the Type 3 case are not significantly higher than those of the Type 2 case. The MWW test is non-parametric and does not require the samples to be normally distributed [3]. This test can be applied to both small and large sample sizes [3]. In our research, we perform this test considering a significance level of 5%. Finally, it appears that *the percentage of Type 3 clones that experience bug-fixing changes is significantly higher than the percentage of bug-fix clones in the Type 1 case.*

Answer to RQ 1. From our investigations we can state that *while Type 3 clones have a higher bug-proneness compared to the other two clone-types in general, the bug-proneness of Type 1 clones is the lowest for most of our subject systems.* Our statistical significance test results indicate that *Type 3 clones have a significantly higher bug-proneness compared to Type 1 clones.*

In general, the total number of Type 3 clones in a software system is higher compared to the other two clone-types as is evident in Table 6.4 (except Camellia, jEdit, and BRL-Cad). Also, our investigation results indicate that Type 3 clones have the highest possibility of introducing bugs. Finally, our findings imply that possibly Type 3 clones should be managed (i.e., refactored or tracked) with the highest priority.

A possible reason behind why Type 3 clones exhibit the highest bug-proneness is that these are gapped clones (i.e., there are some non-clone lines in the Type 3 clone fragments). Thus, copy-pasting and consistently changing a Type 3 clone fragment is not as straight forward as in the cases of Type 1 and Type 2 clones. Also, because of the gaps in the Type 3 clones, refactoring of such clones might sometimes be difficult, and it causes an increased number of Type 3 clones in the software systems (i.e., as can be seen from our experimental results). Because of the existence of the gaps, possibly tracking is the best suitable management technique for Type 3 clones.

RQ 2: Do the clone fragments from the same clone class co-change (i.e., change together) consistently during a bug-fix?

Rationale. From our answer to *RQ 1*, we understand that code clones of each clone-type have a tendency of experiencing bug-fixing changes, and Type 3 clones have the highest tendency. However, it is also important to know whether two or more clone fragments from the same clone class co-changed (i.e., changed together) consistently (i.e., the clone fragments were modified in the same way) during bug-fixes. Such clones are more important for clone management than those clones that did not experience consistent co-change during bug-fixes for the following reasons.

(1) If more than one clone fragments from the same clone class are changed together consistently during a bug-fix, then it is an implication that those clone fragments contained the same bug and fixing of that bug required those clone fragments to be modified together consistently. Unification of these clone fragments (i.e., that co-changed consistently during bug-fixes) into a single one through refactoring can possibly help us fix future bugs or inconsistencies with reduced effort, because in that case the bug-fixing changes will require to be implemented in a single code fragment rather than implementing/propagating the same changes to multiple similar code fragments.

(2) If only a single clone fragment from a particular clone class is modified for fixing a bug leaving the other fragments in that class as they are, then it is an implication that this particular clone fragment does

not require to maintain consistency with the other clone fragments in its class, and it has a tendency of evolving independently. Such a fragment might not be regarded as a member of the class if it continues to evolve independently, and in that case it should not be considered for clone management.

For this research question we investigate whether clone fragments from the same clone class have a tendency of co-changing consistently during a bug-fix, and if so, how this tendency differs across the clone-types. The clone-type with a higher tendency should be given a higher priority for management.

Methodology. In a previous study [163] we showed that if two or more clone fragments from the same clone class experience a *similarity preserving co-change* (we define it in the next paragraph) in a particular commit operation, then it is an implication that they co-changed consistently (i.e., they were changed in the same way) in that commit. Considering this fact we answer this research question by automatically examining the bug-fix commits and determining whether two or more clone fragments from the same clone class experienced *similarity preserving co-change*s in these commits. If such clone fragments really exist, then these should be given higher priorities for management as we have just discussed.

Similarity Preserving Co-change. Let us consider that two code fragments *CF1* and *CF2* are clones of each other in revision R. A commit operation C was applied on this revision and both of these two code fragments were changed (i.e., the clone fragments were co-changed) in this commit. If in revision *R+1* (created because of the commit operation C) these two code fragments are again considered as clones of each other (i.e., if they preserve their similarity), then we say that *CF1* and *CF2* experienced a *similarity preserving co-change* in the commit operation C.

Considering each clone-type of each of the subject systems we determine which clone fragments experienced bug-fix commits and which of these clone fragments received similarity preserving co-changes in the bug-fix commits. Finally, we determine the percentage of clone fragments that received similarity preserving co-changes in the bug-fix commits with respect to all clone fragments related to bug-fix. Table 6.5 shows the total number of clones related to bug-fix and the percentage of bug-fix clones that experienced similarity preserving co-changes during bug-fix commits. We also show these percentages in Fig. 6.3 to do a visual comparison of the percentages regarding different clone-types.

From Fig. 6.3 we see that there are no vertical bars for Type 2 and Type 3 cases of Ctags, and also, for Type 2 case of Camellia. The reason is that the number of bug-fix clones that experienced similarity preserving co-changes is zero for each of these cases. This is also evident from Table 6.5. From the overall percentages we see that bug-fix clones of Type 3 have the overall highest tendency of experiencing similarity preserving co-changes in the bug-fix commits. The tendency for Type 2 case is also very near to that of the Type 3 case. Bug-fix clones of Type 1 have the lowest tendency of experiencing similarity preserving co-changes during bug-fix.

We also manually analyzed all the similarity preserving co-changes that occurred to the bug-fix clones of each clone-type of Freecol during the bug-fix commits to see whether the clone fragments were really modified consistently (i.e., whether the clone fragments were modified in the same way). According to our

Table 6.5: Percentage of Bug-fix Clones that Experienced Similarity Preserving Co-change in the Bug-fix Commits

	Type 1		Type 2		Type 3	
Systems	CGBF	BFCS	CGBF	BFCS	CGBF	BFCS
Ctags	4	50%	4	0%	14	0%
Camellia	2	100%	0	0%	11	45.45%
BRL-Cad	3	66.67%	2	100%	10	60%
Freecol	14	57.14%	12	50%	107	51.4%
jEdit	73	8.21%	20	30%	184	24.45%
Carol	31	38.7%	32	50%	134	50.74%
Jabref	8	25%	14	28.57%	31	67.74%

CGBF = Number of Clone Genealogies related to a Bug-fix.

BFCS = Percentage of Bug-fix Clone genealogies that experienced similarity preserving co-change in bug-fix commits.

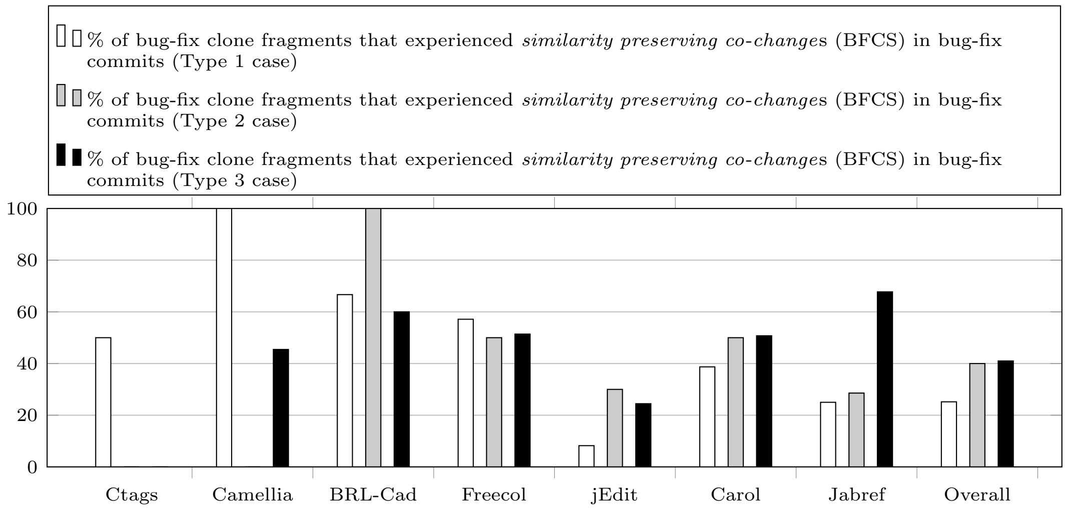
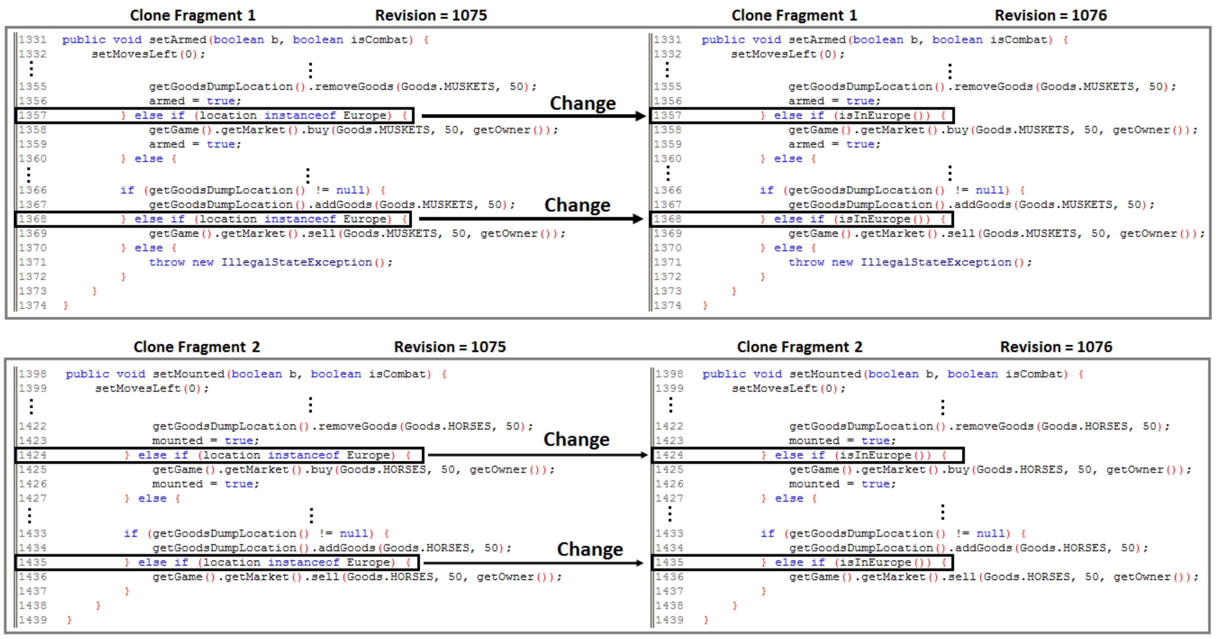


Figure 6.3: Comparison regarding the percentage of bug-fix clones that experienced *similarity preserving co-changes* during bug-fix commits.


Clone Fragment 1 Revision = 1075
Clone Fragment 1 Revision = 1076
Change
Change
Clone Fragment 2 Revision = 1075
Clone Fragment 2 Revision = 1076
Change
Change

```
public void setArmed(boolean b, boolean isCombat) {
    setMovesLeft(0);
 ⋮
            getGoodsDumpLocation().removeGoods(Goods.MUSKETS, 50);
            armed = true;
        } else if (location instanceof Europe) {
            getGame().getMarket().buy(Goods.MUSKETS, 50, getOwner());
            armed = true;
        } else {
 ⋮
        if (getGoodsDumpLocation() != null) {
            getGoodsDumpLocation().addGoods(Goods.MUSKETS, 50);
        } else if (location instanceof Europe) {
            getGame().getMarket().sell(Goods.MUSKETS, 50, getOwner());
        } else {
            throw new IllegalStateException();
        }
    }
}
```

```
public void setArmed(boolean b, boolean isCombat) {
    setMovesLeft(0);
 ⋮
            getGoodsDumpLocation().removeGoods(Goods.MUSKETS, 50);
            armed = true;
        } else if (isInEurope()) {
            getGame().getMarket().buy(Goods.MUSKETS, 50, getOwner());
            armed = true;
        } else {
 ⋮
        if (getGoodsDumpLocation() != null) {
            getGoodsDumpLocation().addGoods(Goods.MUSKETS, 50);
        } else if (isInEurope()) {
            getGame().getMarket().sell(Goods.MUSKETS, 50, getOwner());
        } else {
            throw new IllegalStateException();
        }
    }
}
```

```
public void setMounted(boolean b, boolean isCombat) {
    setMovesLeft(0);
 ⋮
            getGoodsDumpLocation().removeGoods(Goods.HORSES, 50);
            mounted = true;
        } else if (location instanceof Europe) {
            getGame().getMarket().buy(Goods.HORSES, 50, getOwner());
            mounted = true;
        } else {
 ⋮
        if (getGoodsDumpLocation() != null) {
            getGoodsDumpLocation().addGoods(Goods.HORSES, 50);
        } else if (location instanceof Europe) {
            getGame().getMarket().sell(Goods.HORSES, 50, getOwner());
        }
    }
}
```

```
public void setMounted(boolean b, boolean isCombat) {
    setMovesLeft(0);
 ⋮
            getGoodsDumpLocation().removeGoods(Goods.HORSES, 50);
            mounted = true;
        } else if (isInEurope()) {
            getGame().getMarket().buy(Goods.HORSES, 50, getOwner());
            mounted = true;
        } else {
 ⋮
        if (getGoodsDumpLocation() != null) {
            getGoodsDumpLocation().addGoods(Goods.HORSES, 50);
        } else if (isInEurope()) {
            getGame().getMarket().sell(Goods.HORSES, 50, getOwner());
        }
    }
}
```

Figure 6.4: An example of a similarity preserving co-change of two Type 3 clone fragments (i.e., Clone Fragment 1, and Clone Fragment 2) of Freecol in a bug-fix commit operation applied to revision 1075. Each of these two clone fragments is a method clone (i.e., the whole method is a clone fragment). The figure shows that they were changed consistently in the bug-fix commit and were again considered as Type 3 clones of each other in revision 1076.

manual analysis in each case of similarity preserving co-change, the clone fragments were changed together consistently. Fig. 6.4 shows an example of similarity preserving co-change of two Type 3 clone fragments in the bug-fix commit operation applied to revision 1075 of Freecol. We show the instances of these two clone fragments in revisions 1075 and 1076 and highlight the changes that occurred to them. We see that the clone fragments changed together consistently (i.e., in the same way) in the bug-fix commit operation. The commit log as stated by the programmer is "*Fixes a bug relating to giving units equipment while onboard a carrier in Europe*". We see that the bug-description is relevant to the context. Fig. 6.4 shows that both the clone fragments contained the same bug and were fixed in the same way. The example reveals the fact that unification of these two clone fragments into a single one could help us fix future bugs with reduced effort.

During our manual investigation of the bug-fixes that occurred to code clones, the categories of bug-fixes that appeared frequently are as follows: fixing the same semantically incorrect implementation in multiple clone fragments from the same class, addition of the same missing implementations in multiple clone fragments of the same class, and fixing the same GUI related error in multiple clone fragments.

Answer to RQ 2. Our investigation results show that *clone fragments from the same clone class have a tendency of co-changing (i.e., changing together) consistently during the bug-fix commit operations.* Consdering all the subject systems, bug-fix clones of Type 1 exhibit the lowest tendency. The tendencies regarding both Type 2 and Type 3 cases are higher compared to Type 1 case. Thus, we should possibly prioritize Type 3 and Type 2 clones over Type 1 clones when making clone refactoring or tracking decisions.

Through our investigation of this research question (*RQ 2*) we suggest to consider higher priorities for managing those clones that experienced similarity preserving co-changes during bug-fixes. Our findings are important for ranking clones for both refactoring and tracking. However, we require further investigations of the evolution histories of the bug-fix clones because of the following two issues.

Issue 1. The clone fragments that experienced *similarity preserving co-changes* in bug-fix commits might evolve independently afterwards. In that case we should possibly not consider these clone fragments important for management.

Issue 2. A clone fragment that was changed in a bug-fix commit without experiencing a *similarity preserving co-change* might co-evolve consistently with the other fragments in its class afterwards. In that case this clone fragments should be considered important for management.

In order to address these two issues, we need to investigate the entire evolution histories of the bug-fix clones to analyze whether they co-evolved with the other clone fragments in their respective clone classes following a *similarity preserving change pattern* which we called SPCP in our previous studies [162,163]. We perform such an investigation in RQ 3.

RQ 3: What proportion of the clone fragments that experienced bug-fixing changes are SPCP clones?

Rationale. From our discussion at the end of *RQ 2* we realize that it is important to analyze whether the clone fragments that experienced bug-fixes also have the tendencies of evolving following a *similarity preserving change pattern* called *SPCP* (defined in Section 6.2). As the bug-fix clones have tendencies of experiencing *similarity preserving co-change*s (revealed from *RQ 2*), we suspect that they might have tendencies of following SPCP too. In other words, bug-fix clones might also be regarded as SPCP clones. In our previous studies [162, 163] we empirically showed that SPCP clones are important candidates for refactoring or tracking. The clone fragments that do not follow SPCP either evolve independently or are rarely changed during evolution. Thus, the non-SPCP clones should not be considered important for clone management.

To address research question we investigate which of the bug-fix clones are also SPCP clones. Such clone fragments (i.e., the SPCP clones that experienced bug-fixes) should be given the highest priorities for management. In our previous studies [162,163] we ranked the SPCP clones on the basis of their co-change tendencies. We did not consider the bug-proneness of the SPCP clones. We believe that bug-proneness should also be considered for ranking the SPCP clones. However, ranking of SPCP clones considering both bug-proneness and co-change tendencies is not our main focus in this research. We focus on investigating whether bug-fix clones also have the possibility of following an SPCP, and if so, how this possibility differs across different clone-types.

A clone fragment that experienced a bug-fix (whether through a *similarity preserving co-change* or not) might not evolve following an SPCP afterwards (related to **Issue 1** stated in *RQ 2*). In this case we understand that the particular clone fragment evolved independently and thus, is not important from the

Table 6.6: No. of Bug-fix Clones that Evolved by Following an SPCP (Similarity Preserving Change Pattern)

	Type 1				Type 2				Type 3			
Systems	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL	CGBF	CGSPCP	CGBFSPCP	CGBFSPCPL
Ctags	4	20	4	2	4	27	0	0	14	85	6	1
Camellia	2	4	2	2	0	2	0	0	11	36	10	0
BRL-Cad	3	42	2	2	2	8	2	2	10	41	8	4
Freecol	14	43	9	3	12	49	10	2	107	331	80	10
jEdit	73	50	0	0	20	63	11	2	184	614	157	96
Carol	31	82	16	0	32	73	20	3	134	325	117	22
Jabref	8	104	5	2	14	51	11	0	31	293	25	10

CGBF = Total number of Clone Genealogies (i.e., clones) that are related to a Bug-fix.
CGSPCP = Total number of Clone Genealogies that followed an SPCP (*Similarity Preserving Change Pattern*).
CGBFSPCP = Total number of bug-fix clones (i.e., the clones that were changed in bug-fix commits) that followed an SPCP.
CGBFSPCPL = Total number of bug-fix clones that followed an SPCP and are also alive in the last revision.

perspectives of clone management.

Methodology. Considering each clone-type of each of the subject systems we determine the SPCP clones using SPCP-Miner [170]. We also determine those clone fragments that experienced bug-fixes following the procedure described in Section 6.4. Then we identify which of these bug-fix clones also appear in the list of SPCP clones. Finally, we determine the percentage of bug-fix clones that have also been selected as the SPCP clones. We determine the following four measures for each clone-type of each candidate system and show these measures in Table 6.6.

- **Measure 1:** The total number of bug-fix clones (The column **CGBF** in Table 6.6).
- **Measure 2:** The total number of SPCP clones (The column **CGSPCP** in Table 6.6).
- **Measure 3:** The total number of bug-fix clones which have also been selected as SPCP clones (The column **CGBFSPCP** in Table 6.6).
- **Measure 4:** The total number of bug-fix clones which have been selected as SPCP clones and are alive in the last revision (The column **CGBFSPCPL** in Table 6.6). We determine and present this measure because while making refactoring or tracking decisions we are primarily concerned with those clone fragments that are alive in the last revision (i.e., the most recent revision) of the system.

It might be the case that only a single clone fragment from a clone class got changed in a bug-fix commit operation however, the clone fragment later co-evolved with the other clone fragments in its class by preserving similarity and thus, can be selected as an SPCP clone fragment (related to **Issue 2** stated in *RQ 2*). Such examples are evident in Type 3 case of Ctags. From Table 6.5 we see that the bug-fix clone fragments (14 in total) of Type 3 case of Ctags did not experience similarity preserving co-changes. However, Table 6.6 shows that some of these clone fragments (6 in total) evolved following SPCPs (similarity preserving change patterns). If we compare Table 6.5 and 6.6 we can discover some other examples of such cases.

We also determine the following two percentages from the above four measures considering each clone-type of each of the candidate system.

(1) The percentage of the bug-fix clones that are selected as SPCP clones. This percentage (Measure 3 * 100 / Measure 1) is shown in Fig 6.5.

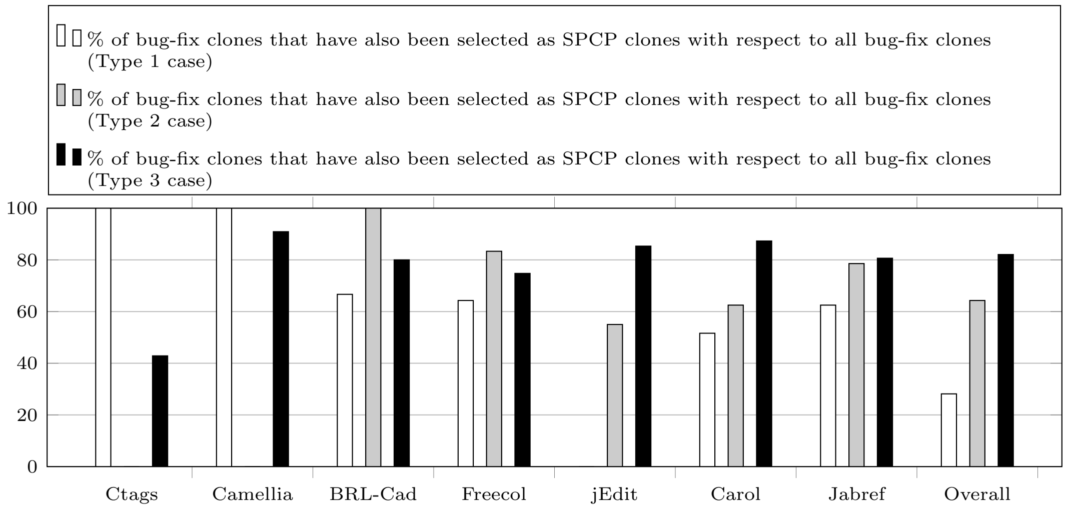


Figure 6.5: Comparison regarding the percentage of clone fragments that have experienced bug-fixes and have also been selected as SPCP clones.

(2) The percentage of the bug-fix clones that have been selected as SPCP clones and are also present in the last revision with respect to all bug-fix clones. This percentage (Measure 4 * 100 / Measure 1) is shown in Fig. 6.6.

From Fig. 6.5 we see that for most of the subject systems except Ctags and Camellia, the percentages regarding Type 2 and Type 3 cases are higher compared to the percentage regarding Type 1 case. The overall percentages for the three clone-types also reflect this. From these overall percentages we can see that the bug-fix clones of the Type 3 case have the highest possibility of evolving following an SPCP (Similarity Preserving Change Pattern). The possibility regarding the Type 1 case is the lowest among the three cases. Such an overall scenario can also be observed in Fig. 6.6.

Answer to RQ 3. From our investigations we can state that *a considerable proportion of the clone fragments that experienced bug-fixing changes have a tendency of evolving by following a similarity preserving change pattern (SPCP) and thus, are the important candidates for refactoring or tracking.* We also observe that the bug-fix clones of the Type 3 case generally have the highest probability of following an SPCP. Thus, we again infer that Type 3 clones should be given the highest priority for management.

Our findings from Fig. 6.6 also imply that for most of the subject systems a considerable proportion of the bug-fix clones that evolve following a *similarity preserving change pattern* remain alive in the last revision (i.e., the most recent revision) of the subject systems. Such clones should be given the highest importance for management, because programmers are mostly concerned with the last revision of the code-base (i.e., the working copy). The findings from this research question and also, from the previous one are important for ranking clones considering their bug-proneness. In the future, on the basis of these findings we would like to propose a clone ranking mechanism considering both the co-change tendencies and bug-proneness of code clones.

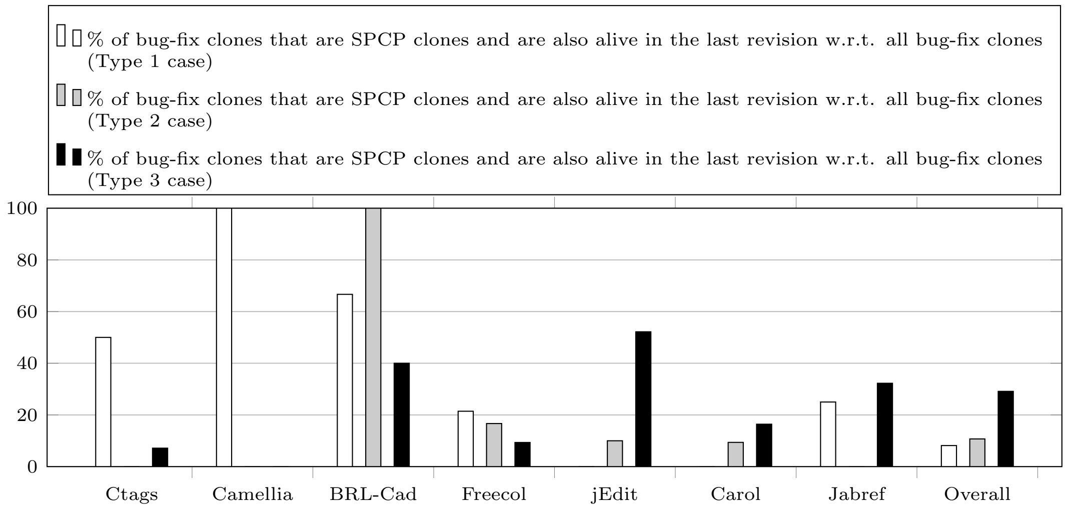


Figure 6.6: Comparison regarding the percentage of bug-fix clones that have been selected as SPCP clones and are also present in the last revision.

6.6 Threats to Validity

We used the NiCad clone detector [48] for detecting clones. For different settings of NiCad, the statistics that we obtain might be different. Wang et al. [254] defined this problem as the *confounding configuration choice problem* and conducted an empirical study to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard [196] and with these settings NiCad can detect clones with high precision and recall [198,201,231]. Thus, we believe that our findings on the bug-proneness of code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al. [30]. Such a technique proposed by Mocus and Votta [153] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [30] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

In our experiment we did not study enough subject systems to be able to generalize our findings regarding the comparative bug-proneness of clone-types. However, our candidate systems were of diverse variety in terms of application domains, sizes and revisions. Thus, we believe that our findings are important from the perspectives of clone management and can help us in better ranking of code clones for refactoring and tracking.

6.7 Related Work

Bug-proneness of code clones has already been investigated by a number of studies. Li and Ernst [141] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [142] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [51] investigated on finding instances of incompletely fixed bugs in near-miss code clones by investigating a broad range of features of such clones involving machine learning. Göde and Koschke [78] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes occurred to the code clones are unintentionally inconsistent. Chatterji et al. [44] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [97] performed a study on the context based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al. [91] developed a tool called 'CloneInspector' in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs. Xie et al. [263] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution, and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky.

Rahman et al. [183] found that bug-proneness of cloned code is less than that of non-cloned code on the basis of their investigation on the evolution history of four subject systems using DECKARD [96] clone detector. However, they considered monthly snap-shots (i.e., revisions) of their systems and thus, they have the possibility of missing buggy commits. In our study, we consider all the snap-shots/revisions (i.e., without discarding any revisions) of a subject system as mentioned in Table 6.2 from the beginning one. Thus, we believe that we are not missing any bug-fix commits. Moreover, our goal in this study is different. We compare the bug-proneness of different types of code clones.

Selim et al. [220] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider seven diverse subject systems in our investigation. Also, we compare the bug-proneness of different types of clones. Selim et al. [220] did not perform a type centric analysis in their study.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [19] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [29] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base.

We see that different studies have investigated clone related bugs in different ways and have developed different bug detection tools. However, none of these studies make a comparison of the bug-proneness of different types of code clones. Comparing the bug-proneness of different clone-types is important from the perspectives of clone management. The clone-type with a higher bug-proneness can be given a higher priority when making clone management decisions. Focusing on this issue we make a comparison of the bug-proneness of the major types (Type 1, Type 2, Type 3) of clones from different perspectives and identify which types of clones have a higher bug-proneness and thus, should be given a higher priority for management. None of the existing studies made such a comparison. Our study also provides useful implications regarding ranking of code clones for refactoring and tracking.

6.8 Summary

In Chapter 6 we present an empirical study on the comparative bug-proneness of different types of code clones. According to our investigation on the major types of code clones: Type 1 (Exact clones), Type 2 (Near-miss clones), and Type 3 (Near-miss clones) in thousands of revisions of seven diverse subject systems written in two different programming languages (C, and Java) we can state that:

(1) Type 3 clones exhibit the highest bug-proneness among the three clone-types. The bug-proneness of Type 3 clones is significantly higher than that of Type 1 clones.

(2) Also, Type 3 clones have the highest likeliness of co-changing (i.e., changing together) consistently during the bug-fixing changes.

(3) Moreover, the bug-fix clones of Type 3 exhibit the highest tendencies of evolving following a *similarity preserving change pattern* (SPCP). The existing studies [162, 163] show that the SPCP clones (i.e., the clone fragments that evolve following a similarity preserving change pattern) are important for refactoring and tracking.

Our experimental results imply that Type 3 clones should be given the highest priority when making clone management decisions. Our findings regarding the consistent co-change of bug-prone clones and also, regarding their tendencies of following SPCP can be considered for ranking code clones for management.

While our investigation in Chapter 6 involves comparing bug-proneness of different clone-types, there is a common belief that bug-proneness of code clones is primarily caused by a particular clone evolution pattern called *late propagation*. Thus, late propagation in code clones should also be considered when prioritizing them for refactoring or tracking. In Chapter 7 we present our comparative study on the late propagation tendencies of different types of code clones.

Chapter 7
A Comparative Study on the Intensity and Harmfulness of Late Propagation in Near-Miss Code Clones

In Chapter 6 we investigated bug-proneness of different types of code clones with the goal of prioritizing clone-types for management on the basis of their bug-proneness. It is suspected that bug-proneness of code clones is often related to a particular clone evolution pattern called late propagation. Existing studies [29,30] show that this particular evolution pattern of code clones may introduce bugs and inconsistencies in the code-base. In Chapter 7 we compare the intensities of late propagation in different types of code clones with a goal of prioritizing clone-types for management considering their late propagation tendencies. We also analyze whether we can minimize late propagation in code clones by considering the SPCP clones (code clones that evolved following a Similarity Preserving Change Pattern called SPCP defined in Chapter 3) for management such as refactoring and tracking.

The rest of the chapter is organized as follows. Section 7.2 describes the related terminology, Section 7.3 elaborates on the detection of late propagation in clones, the experimental results are presented and analyzed to answer the research questions in Section 7.4, Section 7.5 discusses the related work, Section 7.6 mentions possible threats to validity and finally, we conclude this chapter by mentioning future work in Section 7.7.

7.1 Introduction and Motivation

There are strong empirical evidences [19, 30, 241] that late propagation is related to bugs [19, 241] and inconsistencies [30] in the code-base. Researchers have investigated different specific patterns [30] of late propagation and identified which patterns are more related to bugs, faults, and inconsistencies. However, the existing studies regarding late propagation in clones have the following limitations.

(1) None of the studies investigate the intensities of late propagation in different types of clones separately. Such a study is important because, if late propagation is observed to be more intense in a particular clone type compared to the others, we might consider being more conscious while changing clones of that particular type. Also, we might want to refactor clones of that particular type with higher priority.

(2) None of the existing studies investigate the bug-proneness of late propagation in different types of clones separately. Such an investigation is also very important for understanding the comparative harmfulness of late propagation in different clone-types.

Table 7.1: Research Questions

Serial No.	Research Question
1	What percentage of late propagations in Type 3 clones occur only because of the changes in the non-cloned portions of the participating clone fragments?
2	Are the intensities of late propagation different in different types of clones?
3	Late propagations in which types of clones are more related to bugs?
4	Clones of which clone-type(s) have higher possibilities of experiencing bug-fixing changes at the time of convergence?
5	Do the participating clone fragments in a clone pair that experience late propagation generally remain in different files?
6	Do block clones or method clones exhibit higher intensity of late propagation?
7	Do late propagations mainly occur to the SPCP clones or non-SPCP clones?

Focusing on these issues, we investigate late propagation in three types of clones (Type 1, Type 2, and Type 3) separately and answer seven important research questions presented in Table 7.1. According to our experimental results on thousands of revisions of eight diverse subject systems written in two different programming languages we can state that:

- The percentage of late propagations in Type 3 clones occurred only because of the changes in the non-matched (i.e., non-cloned) portions of the clone fragments is very low (less than one) for most of our candidate systems. However, this proportion can sometimes be considerable (for example our subject system jEdit). Such late propagations should be ignored when making clone management decisions. Our implemented system can automatically detect such ignorable late propagations. We perform our investigations related to research questions RQ 2 to RQ 7 (Table 7.1) by disregarding these ignorable late propagations.

- The intensity of late propagations in Type 3 clones is higher compared to the other two clone-types.

- Type 3 clones have a higher possibility of experiencing buggy late propagations compared to the clone fragments of the other two clone-types.

- Almost all of the clone fragments that experience late propagations are block clones. According to our statistical significance tests, the percentage of block clones that experience late propagations is significantly higher than the corresponding percentage of method clones. It seems that creating block clones is more risky than creating method clones.

- Around 89% of the late propagations involve SPCP clones [163] (i.e., the clone fragments that evolved following a Similarity Preserving Change Pattern called SPCP). In other words, late propagations mainly occur to the SPCP clones. By refactoring and tracking SPCP-clones we can possibly minimize future occurrences of late propagations considerably.

The research work presented in this chapter is a significant extension of our earlier work [166]. In our previous study [166] we detected late propagations in three types of clones separately and answered three research questions (Table 7.1): RQ 2, RQ 5, and RQ 6. We extend this work with a number of investigations: (1) investigating which proportion of late propagations in Type 3 clones occurred only because of the changes in the non-cloned portions of the participating clone fragments, (2) analyzing and comparing the bug-proneness of the late propagations in three types of clones, and (3) investigating late propagation in SPCP clones. We perform these investigations for answering the four new research questions: RQ 1, RQ 3, RQ 4, and RQ 7.

7.2 Terminology

We conduct our experiment regarding late propagation considering exact (Type 1) and near-miss clones (Type 2 and Type 3 clones). Chapter 2 defines these clone-types.

7.2.1 Late Propagation in a Clone Pair

Let us consider a pair of clone fragments. We say that this clone pair has experienced late propagation if it receives a diverging change followed by a converging change [30].

- ***Diverging Change.*** Let us assume a particular commit C_i where one or both of these two clone fragments were changed. Because of this change, the fragments were not considered as clones of each other. In other words, the clone fragments diverged. Such a change is called a *diverging change* for the clone pair.

- ***Converging Change.*** Let us assume a later commit C_{i+n} ($n >= 1$) where one or both of these fragments were changed, and because of this change, the fragments were again considered as clones of each other. In other words, the fragments converged. The change for which the fragments converged is termed as a *converging change*.

A particular clone pair may experience late propagation more than once during evolution. Fig. 7.1 shows a possible example of late propagation experienced by a clone pair (*CF1*, *CF2*). The commit C_i applied on revision R_i modified *CF1* and as a result, *CF1* and *CF2* diverged. However, in commit C_{i+2}, the fragment *CF2* changed and *CF1* and *CF2* again became clones of each other in R_{i+3}.

7.2.2 SPCP Clones

In a previous study [163] we empirically showed that SPCP clones are important for refactoring or tracking. SPCP-Clones are those clone fragments that evolved following a particular change pattern called *Similarity*

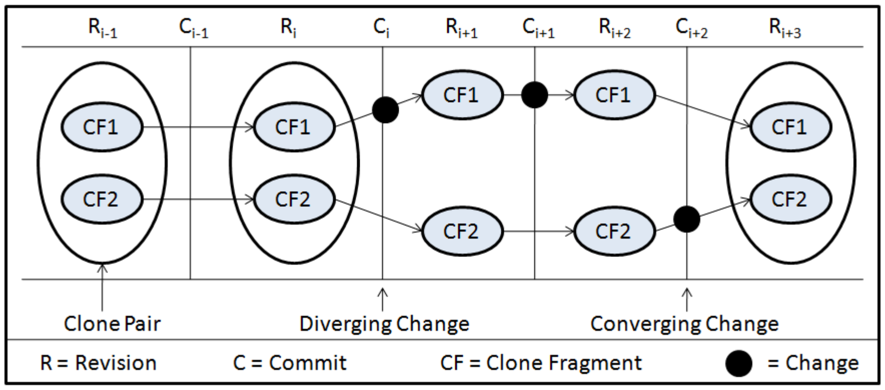


Figure 7.1: A possible example of late propagation

Preserving Change Pattern (SPCP). A *Similarity Preserving Change Pattern* consists of only *Similarity Preserving Change* and/or *Re-synchronizing Change.*

Similarity Preserving Change. Let us consider two code fragments that are clones of each other in a particular revision of a subject system. A commit operation was applied on this revision, and any one or both of these code fragments (i.e., clone fragments) received some changes. However, in the next revision (created because of the commit operation) if these two code fragments are again considered as clones of each other (i.e., the code fragments preserve their similarity), then we say that the code fragments received *Similarity Preserving Change* in the commit operation.

Re-synchronizing Change. Let us consider two code fragments that are clones of each other in a particular revision. If these two clone fragments experience a *diverging change* followed by a *converging change*, then we say that they experienced a re-synchronizing change. A re-synchronizing change can also be termed as a late propagation.

Here, we should clarify that two clone fragments (i.e., a clone-pair) might experience late propagation (i.e., re-synchronizing change) a number of times during evolution, however they might not be regarded as SPCP clones. Let us consider that two clone fragments experienced late propagation(s) during evolution. It might be the case that after the last occurrence of late propagation they again diverged, but did not converge again. In such a case, these two fragments will not be regarded as SPCP clones, because they did not preserve their similarity lastly. As they did not preserve their similarity, their evolution pattern is not a *Similarity Preserving Change Pattern* (i.e., is not an SPCP). Examples of such cases are evident from our answer to RQ 7.

We performed an empirical study [162] on SPCP clones where we separated all the SPCP clones in a software system into two disjoint groups. The clone fragments in one group are important for refactoring whereas, the clone fragments in the other group are important for tracking. The clone fragments that do not follow SPCP either evolve independently or are rarely changed during evolution. Thus, these non-SPCP clone fragments should not be considered important for management.

7.2.3 Granularity of Late Propagation

We should note that we conduct our late propagation study considering the granularity of clone pairs as was done by each of the previous studies. While it would be good to conduct such a study considering clone classes, consideration of clone classes might cause the loss of important information regarding late propagation. Let us assume a clone class consisting of six clone fragments in revision R_i. The subsequent commits might affect only two of these six clone fragments leaving the other four fragments as they are. There is a possibility that these two clone fragments (that are getting changed) will experience a late propagation (i.e., the changes occurred in one clone fragment will propagate to the other one with some delay) in future evolution. However, the other four clone fragments might not require to be changed during the whole period of evolution. In other words, the changes occurred to the two clone fragments might not ever need to be propagated to the other four clone fragments. In such a situation, consideration of all these six clone fragments for late propagation is not reasonable. While pairs of clone fragments in a particular class might experience late propagation, the whole class might not. Thus, we believe that investigating late propagation considering clone pairs is reasonable.

7.3 Detection of Late Propagation

We detect and experiment late propagations from eight subject systems listed in Table 7.2. We downloaded the revisions of each of these systems from an open-source SVN repository SourceForge[1]. For each of the subject systems we considered each of the revisions beginning from the first one. We select these systems in our research focusing on the diversity of their application domains (i.e., the systems belong to seven application domains), sizes (i.e., the subject systems are of different sizes, from very small to large), and the number of revisions. Thus, we believe that our reported experimental results are not affected by these parameters.

7.3.1 Preliminary Steps

Detection of late propagation by mining the revisions of a particular subject system requires the following preliminary steps to be done sequentially - (i) Extraction of methods from each of the revisions, (ii) Detection of method genealogies, (iii) Extraction of clones from each of the revisions, (iv) Locating these clones to the already detected methods, (v) Extraction of changes between every two consecutive revisions, and (vi) Reflecting these changes to the already detected methods and clones residing in these methods.

We extract methods using Ctags[2]. For detecting method genealogies we follow the procedure proposed by Lozano and Wermelinger [145]. The genealogy of a particular method helps us to understand how a

[1]Sourceforge: `http://www.sourceforge.net`

[2]Ctags: `http://sourceforge.net/projects/ctags/?source=directory`

Table 7.2: Subject Systems

Systems	Language	Application Domains	LOC	Revisions
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	207
BRL-CAD	C	3D Modeling	40,941	735
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545
Java-ML	Java	Java Machine Learning Library	16,428	1200

particular method evolved during software evolution. As we detect method genealogies, we also detect clone genealogies by locating the propagation of the clone fragments through the methods.

Clone Genealogy. By the term *clone genealogy* we mean the genealogy of a particular code fragment which is also regarded as a clone fragment by the clone detector. By detecting the genealogy of a particular clone fragment we can easily determine how that fragment was changed during the evolution. A particular clone class may contain two or more clone fragments. We determine a separate genealogy for each of these clone fragments.

We use NiCad clone detector for detecting and extracting clones from each revision of a subject system. The main purpose of choosing NiCad is that it can detect clones of different clone-types separately including Type 3 with high precision and recall [198, 201]. For detecting Type 3 clones, we considered a dissimilarity threshold of 20% with blind renaming of identifiers. Here, we should note that before using the NiCad outputs for Type-2 and Type-3 cases, we pre-processed them in the following way.

(1) Every Type-2 clone class that exactly matched any Type-1 clone class was excluded from Type-2 outputs.

(2) Every Type-3 clone class that exactly matched any Type-1 or Type-2 class was excluded from Type-3 outputs.

We performed these because we wanted to investigate each of the three types of clones separately. The above two pre-processing steps ensure that the set of Type 2 clone classes that we investigate does not contain any Type 1 clone class. Also, the set of our investigated Type 3 clone classes does not contain any Type 1 or Type 2 clone classes. The detection mechanism of late propagation clone-pairs is described in the following subsection.

7.3.2 Detection of Clone-Pairs that Experienced Late Propagation

After completing the preliminary steps described above we automatically mine the late propagation clone-pairs. At the very beginning, we assume a global list of clone pairs each of which has the potential of experiencing late propagation. We call such a clone pair a CPLP (Clone Pair having the potential of experiencing Late Propagation). We call this list the GLOBAL LIST.

Clone Pair having the potential of experiencing Late Propagation (CPLP). We consider a pair of code fragments, (*CF1* and *CF2*), which are clones of each other in revision R_i. A commit operation C_i was applied on R_i and one or both of these fragments changed. However, because of this change, *CF1* and *CF2* were not considered as clones of each other in revision R_{i+1}. In other words, the change in C_i is a diverging change for the pair (*CF1*, *CF2*). This pair is considered as a CPLP because, there is a possibility that in a future commit operation, the fragments *CF1* and *CF2* will converge (i.e., *CF1* and *CF2* will again be considered as clones of each other).

The GLOBAL LIST remains empty initially. We examine the commit operations sequentially from the very beginning one. We only consider those commits where there were changes to one or more clone fragments of a particular clone type. As we examine the commit operations, we update the GLOBAL LIST and mark some clones pairs (i.e., some CPLPs) in this list as the late propagation clone pairs. Suppose, C_i is such a commit which was applied on revision R_i and the immediate next revision R_{i+1} was created as a result. We perform the following steps sequentially considering C_i.

Step 1. Determining the list of affected clone fragments. We identify the list of clone fragments (in revision R_i) that received some changes during C_i. We call this list the LIST OF AFFECTED CLONE FRAGMENTS.

Step 2. Determining the list of affected clone pairs. We make a list of clone pairs that involve one or more clone fragments in the LIST OF AFFECTED CLONE FRAGMENTS. We denote this list of clone pairs as the LIST OF AFFECTED CLONE PAIRS.

Step 3. Updating the Global List using the List of Affected Clone Fragments. We identify those clone pairs in the GLOBAL LIST each of which involves any of the clone fragments in the LIST OF AFFECTED CLONE FRAGMENTS. There is a possibility that such a clone pair in the GLOBAL LIST has converged. In order to check this we determine whether the fragments in such a pair are considered as clones of each other in revision R_{i+1} which was created because of commit C_i. If this is true, then we understand that this clone pair in the GLOBAL LIST has experienced late propagation. We mark this clone pair as a late propagation pair.

Step 4. Updating the Global List using the List of Affected Clone Pairs. If any pair in the LIST OF AFFECTED CLONE PAIRS already appears in the GLOBAL LIST, we do not need to consider this pair because, this has already been handled in the previous step. Considering the remaining pairs (in the LIST OF AFFECTED CLONE PAIRS), we determine the CPLPs (i.e., the clone pairs that have the potential of experiencing late propagation). If the two fragments in a remaining pair are not considered as clones of each other in revision R_{i+1}, then this pair is a CPLP. We include the CPLPs in the GLOBAL LIST.

For each of the commit operations we follow the above four steps, update the GLOBAL LIST and mark some CPLPs in this list as the late propagation pairs if they converge. After examining all the commit operations we get all the late propagation clone pairs of a particular clone type.

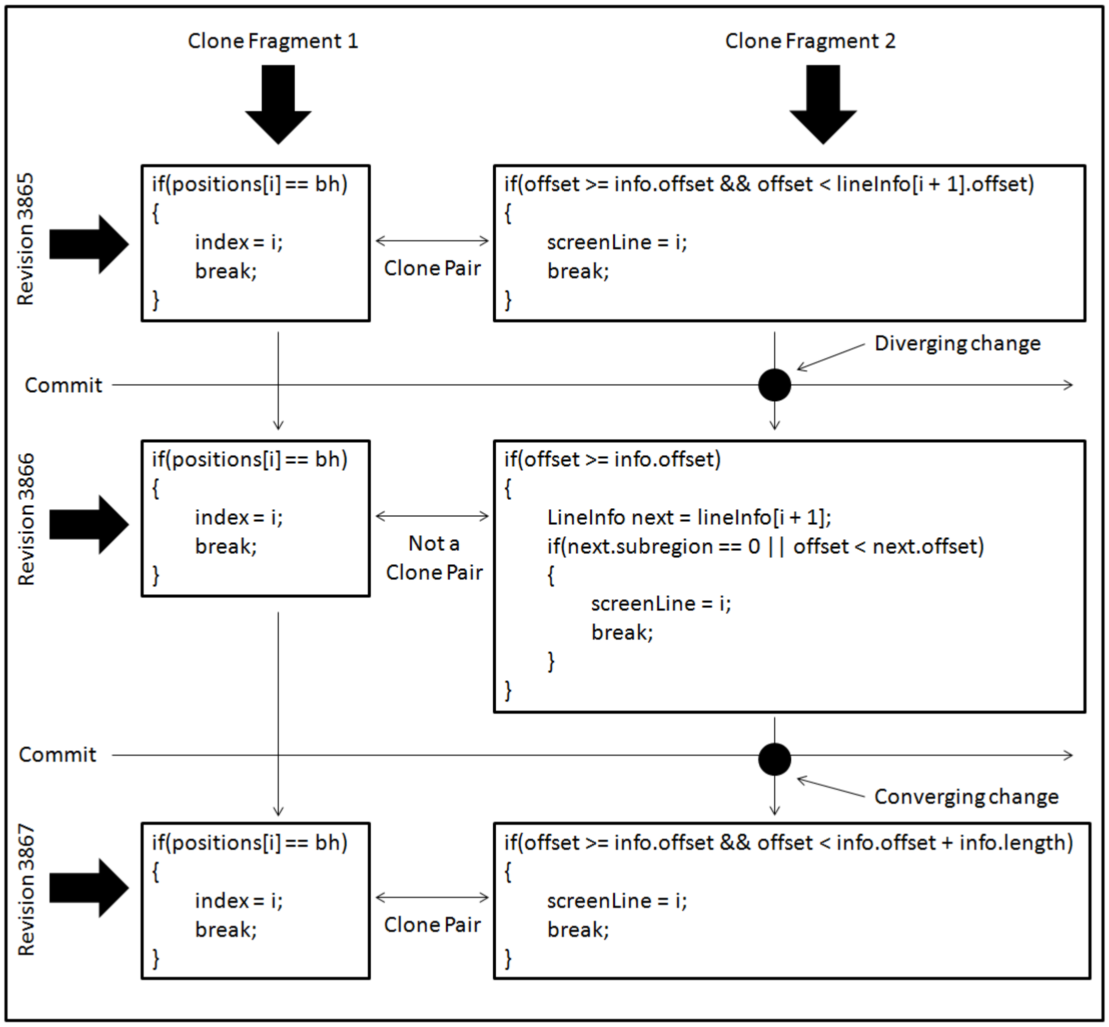
Now, let us assume that a particular pair in the GLOBAL LIST has been marked as a late propagation pair during the examination of the commit operation C_i. This pair has the following three possibilities during future evolution.

- The pair may again experience late propagation. In our experiment we detect each of the occurrences of late propagations a particular clone pair experienced during evolution.
- The fragments in the pair may evolve independently. However, independent evolution of such fragments without any convergence is not our concern in this research work.
- One or both fragments may form new pair(s) with other fragments of the same or other clone types. In this case, our implementation considers the new pairs in calculation because they can experience late propagation.

Detection of late propagation considering an individual clone-type. Suppose we are detecting late propagation considering the clones of Type j where j = 1, 2, or 3. The clone fragments *CF1* and *CF2* are clones of this type in revision R_i. Because of the commit C_i on revision R_i, the fragments *CF1* and *CF2* diverged. Let us assume that in commit C_{i+n}, the fragments converged and they were again considered as clones of Type j. Then, we consider this late propagation as a late propagation of Type j. It might be the case that after converging, *CF1* and *CF2* were not considered as clones of Type j. They were considered as clones of Type k where k= 1, 2, or 3 and $j \neq k$. In this case we do not consider a late propagation, because the fragments changed their types. While detecting late propagation in the clones of Type k, the fragments *CF1* and *CF2* are considered to determine whether they experienced a late propagation of Type k. However, we plan to investigate the intensity of such mixed type late propagations (i.e., where the participating fragments were considered of one clone type before divergence but of another clone type after convergence) as a future work.

An example of late propagation in Type 3 clones. We present an example of late propagation that occurred to a Type 3 clone pair of our subject system jEdit. We automatically detect this late propagation by applying our late propagation detection tool. We present Fig. 7.2 for describing the late propagation example.

In Fig. 7.2 we see a Type 3 clone pair in revision 3865 of our candidate system jEdit. As we can see, the participating clone fragments (denoted as *Clone Fragment 1* and *Clone Fragment 2*) are two if-blocks. NiCad detects these Type 3 clones by considering a dissimilarity threshold of 20% and applying blind renaming of



Revision 3865, Clone Fragment 1:

```
if(positions[i] == bh)
{
      index = i;
      break;
}
```

Revision 3865, Clone Fragment 2:

```
if(offset >= info.offset && offset < lineInfo[i + 1].offset)
{
      screenLine = i;
      break;
}
```

Revision 3866, Clone Fragment 1:

```
if(positions[i] == bh)
{
      index = i;
      break;
}
```

Revision 3866, Clone Fragment 2:

```
if(offset >= info.offset)
{
      LineInfo next = lineInfo[i + 1];
      if(next.subregion == 0 || offset < next.offset)
      {
            screenLine = i;
            break;
      }
}
```

Revision 3867, Clone Fragment 1:

```
if(positions[i] == bh)
{
      index = i;
      break;
}
```

Revision 3867, Clone Fragment 2:

```
if(offset >= info.offset && offset < info.offset + info.length)
{
      screenLine = i;
      break;
}
```

Figure 7.2: An example of late propagation in a Type 3 clone pair from subject system jEdit

identifiers. These two clone fragments belong to two different source code files[3] [4]. The names of the container methods of these two clone fragments are *removePosition* and *getScreenLineForOffset* in revision 3865 . The commit operation applied on revision 3865 changed the clone fragment at the right hand side (i.e., Clone Fragment 2). Because of this change they were not considered as a clone pair in revision 3866. Thus, this change is a diverging change for the clone pair. However, the commit operation applied on revision 3866 again changed the fragment at the right hand side and the fragments converged (i.e., became a clone pair) in revision 3867. Thus, this clone pair experienced a late propagation.

7.4 Experimental Results and Discussion

We applied our implementation on each of the eight subject systems in Table 7.2 and identified the clone pairs that experienced late propagation considering each of the three clone types (Type 1, Type 2, Type 3). In the following subsections, we answer the research questions mentioned in the introduction by presenting and analyzing our experimental results. In our previous study [166] we did not consider late propagation between clone fragments remaining in the same method. We consider such late propagations in this extended study. Also, during the period of divergence of a particular late propagation, any one or both of the two participating code fragments (i.e., the code fragments that were considered as a clone-pair before divergence) might become non-clone fragments or be considered as clone fragments of different clone classes. We identify late propagations considering both of these cases in this extended research work.

7.4.1 RQ 1: What Percentage of Late Propagations in Type 3 Clones Mainly Occur Because of the Changes in the Non-matched Portions of the Participating Clone Fragments?

Rationale. We know that Type 3 clones have both cloned and non-cloned portions. If a late propagation in Type 3 clones occur because of the changes in the non-matched (i.e., non-cloned) portions only, then this late propagation might not be important from the perspective of clone management. If it is observed that a significant portion of the late propagations in Type 3 clones occur because of the changes in the non-matched portions, then it is important to identify and discard these late propagations while doing investigations regarding clone management. We answer RQ 1 in the following way.

Methodology. Let us consider that a pair of Type 3 clone fragments has experienced a late propagation. If the matched portions of none of these two clone fragments were modified during the diverging and converging change, and also, during the period of divergence, then we decide that this late propagation can be ignored when making clone management decisions. We at first detect all the occurrences of late propagations in Type 3 clones, and then determine which late propagations occurred only because of the changes in the non-matched portions. A particular clone pair can experience late propagation more than once. We detect

[3]Source code file for Clone Fragment 1: trunk/org/gjt/sp/jedit/buffer/OffsetManager.java
[4]Source code file for Clone Fragment 2: trunk/org/gjt/sp/jedit/textarea/ChunkCache.java

and check all those for our investigation regarding RQ 1. We automatically check a late propagation in the following way.

Let us assume that two code fragments *CF1* and *CF2* are Type 3 clones of each other. They experienced a late propagation where the diverging change occurred in commit C_i and the converging change occurred in commit C_j ($C_j > C_i$). We check all these commits from C_i to C_j to determine whether any one or both of *CF1* and *CF2* changed in these commits and whether the changes occurred in the matched or non-matched portions of the fragments. Suppose, we are going to check the commit operation C where $C_i <= C <= C_j$. We extract the two instances of the two code fragments *CF1* and *CF2* before this commit. Let us assume that these instances are $CF1_{before}$ and $CF2_{before}$ respectively. We also determine the two instances, $CF1_{after}$ and $CF2_{after}$, after the commit. We blind-rename the instances $CF1_{before}$ and $CF2_{before}$ and then find the differences of these blind-renamed instances using *diff* command. From *diff* output we determine the matched and non-matched portions of $CF1_{before}$ and $CF2_{before}$. We then determine the differences between $CF1_{before}$ and $CF1_{after}$ using *diff* command to identify the changes occurred to $CF1_{before}$ in terms of additions, deletions, and modifications. We determine whether these changes occurred to matched or non-matched portions of $CF1_{before}$ using the line numbers of the changes. In the same way we determine whether any changes occurred to the matched or non-matched portions of $CF2_{before}$. If in each of the commit operations from C_i to C_j the two code fragments *CF1* and *CF2* received changes only in their non-matched portions, then we consider this late propagation as an ignorable one.

Considering the late propagations occurred to the Type 3 clones of each of the subject systems we determine what proportion of late propagations occurred only because of the changes in the non-matched portions and thus, are ignorable. Table 7.3 shows these proportions for our subject systems. We see that in case of five subject systems (Ctags, BRL-CAD, Freecol, Carol, and Java-ML) this percentage is zero. For the remaining three subject systems: Camellia, jEdit, and Jabref these percentages are 0.14%, 7.22%, and 0.07% respectively.

Answer to RQ 1: *According to our experimental results, the percentage of late propagations in Type 3 clones occurred only because of the changes in the non-matched portions of the participating clone fragments is very low (i.e., less than one) in most of the subject systems (i.e., seven out of eight systems) we have studied.* However, our analysis is based on only eight subject systems which are not enough to generalize our findings. The percentage of ignorable late propagations can be considerable for some systems (for example, 7.22% in case of our subject system jEdit). We should ignore these late propagations when making clone management decisions. Our implemented prototype tool can automatically detect such ignorable late propagations so that we can discard them from considerations. For answering the remaining research questions we ignore these ignorable late propagations occurred in Type 3 clones.

Table 7.3: Late Propagations in Type 3 Clones

	Ctags	**Camellia**	**BRL-CAD**	**Freecol**	**jEdit**	**Carol**	**Jabref**	**Java-ML**
LP	5	728	1593	1062	83	644	3028	65
LPNM	0	1	0	0	6	0	2	0
PLPNM	0%	0.14%	0%	0%	7.22%	0%	0.07%	0%

LP = Total number of late propagations in Type 3 clones

LPNM = Number of late propagations in Type 3 clones occurred only because of the changes in the non-matched portions

PLPNM = Percentage of late propagations in Type 3 clones occurred only because of the changes in the non-matched portions

Table 7.4: Statistics Regarding Late Propagations in Different Clone-types

	Type 1				Type 2				Type 3			
System	CG	CPL	LG	LP	CG	CPL	LG	LP	CG	CPL	LG	LP
Ctags	146	0	0	0	170	0	0	0	549	5	6	5
Camellia	584	9	10	61	189	84	30	476	571	230	89	727
BRL-CAD	343	0	0	0	171	0	0	0	586	324	31	1593
Freecol	6593	232	47	3498	675	177	54	8903	4748	272	133	1062
jEdit	42778	14	21	14	1536	0	0	0	9756	50	68	77
Carol	1969	12	8	12	751	1	2	2	3022	225	133	644
Jabref	4262	7	9	7	895	3	4	3	5849	601	280	3026
Java-ML	429	4	5	4	404	33	19	110	1103	33	23	65

CG = Total number of clone genealogies

CPL = Total number of clone-pairs that experienced late propagation

LG = Number of distinct clone genealogies that experienced late propagation

LP = Total number of late propagations (discarding the ignorable ones in Type 3 case)

7.4.2 RQ 2: Are the Intensities of Late Propagation Different in Different Types of Clones?

Rationale. If it is observed that late propagation in a particular clone-type is more intense compared to the other clone-types, then it is an implication that clones of that particular clone type have a higher probability of introducing bugs and inconsistencies to the code-base compared to the other types. Thus, it would be beneficial if we could refactor clones of that particular type with higher priority. By minimizing these clones we can minimize the possibility of faults and inconsistencies to the code-base.

Methodology. For answering this research question we applied our prototype tool on each of the candidate systems and determine the following measures considering each of the three types of clones of each of the subject systems.

- The total number of clone genealogies
- The number of clone-pairs (i.e., pair of clone genealogies) that experienced late propagation
- The number of distinct clone genealogies that experienced late propagation
- The total number of late propagations occurred to the clone fragments. A particular pair of clone genealogies can experience late propagation more than once. We determine all of the occurrences of late propagations experienced by each clone pair.

We show these measures in Table 7.4. We investigate the intensity of late propagations in different clone types in the following two ways.

- **Investigation 1:** By determining and comparing the probability that a clone genealogy of a particular clone type will experience a late propagation.
- **Investigation 2:** By determining and comparing how often a pair of clone genealogies of a particular clone type experienced a late propagation.

Investigation 1: *Comparison of the probability that a clone genealogy of a particular clone type will experience a late propagation.*

We calculate probability as percentage. Considering each clone-type of each of the subject systems we determine the percentage of clone genealogies that experienced late propagation. These percentages are shown in Table 7.5. We calculate these percentages from Table 7.4.

From Table 7.5 we see that for five out of eight subject systems (except Camellia, Freecol, and Java-ML) Type 3 clones exhibit the highest intensity of late propagation in comparison with the other two clone types (Type 1, and Type 2). For three systems (i.e., Camellia, Freecol, and Java-ML), Type 2 clones exhibit the highest intensity.

Table 7.5: Percentage of Clone Genealogies that Experienced Late Propagations from Different Clone-types

System	PCGLP - Type 1	PCGLP - Type 2	PCGLP - Type 3	CTHP
Ctags	0%	0%	1.09%	Type 3
Camellia	1.71%	15.87%	15.59%	Type 2
BRL-CAD	0%	0%	5.29%	Type 3
Freecol	0.71%	8%	2.8%	Type 2
jEdit	0.05%	0%	0.7%	Type 3
Carol	0.4%	0.27%	4.4%	Type 3
Jabref	0.21%	0.45%	4.79%	Type 3
Java-ML	1.16%	4.7%	2.09%	Type 2

PCGLP = Percentage of clone genealogies that experienced late propagations

CTHP = Clone-type with the highest percentage

Statistical significance test regarding the intensity of late propagation. We were also interested to investigate whether the intensity of late propagation in Type 3 clones is significantly higher than the intensity of late propagation in Type 1 or Type 2 clones. We perform our investigation using Mann-Whitney-Wilcoxon (MWW) tests [3,4]. We have already determined the percentage of clone genealogies that experienced late propagation considering each clone type of each of the subject systems. These percentages are shown in Table 7.5. Thus, for a particular clone-type we get eight percentages from eight subject systems. We performed MWW tests [4] for each pair of clone-types. For example, in case of the pair (Type 1, Type 3), we determine whether the eight percentages regarding Type 1 are significantly different than the eight percentages regarding Type 3. Here, we should note that MWW test is non-parametric and does not require the samples to be normally distributed [3]. This test can be applied to both small and large sample sizes [3]. In our research, we perform this test considering a significance level of 5%.

According to our MWW test result, the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 case with *p-value* = 0.01 (approximately) for both one-tailed and two-tailed tests, and with an effect size (r) of 0.71. We see that *p-value* < 0.05. Also, the effect size is large [1]. The effect size calculation procedure for the MWW test is available on-line [69]. Finally, we can say that the intensity of late propagation in Type 3 clones is significantly higher than the intensity of late propagation in Type 1 clones. However, we did not get significant differences for any of the other two pairs: (Type 1, Type 2) and (Type 2, Type 3).

Investigation 2: *Comparison of how often a clone-pair of a particular clone type will experience a late propagation.*

Table 7.6: The Average Number of Times a Alone-pair Experienced Late Propagation from Different Clone Types

System	**AT - Type 1**	**AT - Type 2**	**AT - Type 3**	**CTHAT**
Ctags	0	0	1	Type 3
Camellia	6.77	5.66	3.16	Type 1
BRL-CAD	0	0	4.92	Type 3
Freecol	15.07	50.29	3.9	Type 2
jEdit	1	0	1.54	Type 3
Carol	1	2	2.86	Type 3
Jabref	1	1	5.03	Type 3
Java-ML	1	3.33	1.96	Type 2

AT = Average number of times a clone-pair experienced late propagations

CTHAT = Clone-type with the highest average number of times

We perform this investigation considering the clone-pairs that experienced late propagations. Considering each clone-type of each of the subject systems we determine how many times a particular pair of clone fragments experienced late propagation on an average. Table 7.6 shows this average value for each clone-type of each of the subject systems. We see that for five out of eight subject systems (except Freecol, Camellia, and Java-ML), the average number of late propagations received by a Type 3 clone-pair is higher than the average number of late propagations experienced by a Type 1 or Type 2 clone-pair. For Camellia, Type 1 clone pairs exhibit the highest average. For both of the subject systems Freecol and Java-ML, the highest average values are exhibited by the Type 2 clone pairs.

Answer to RQ 2. *According to our investigation results, the intensity of late propagation in Type 3 clones is higher compared to the intensity of late propagation in the other two clone-types for five out of eight candidate systems. Also, according to the MWW test results, Type 3 clones exhibit a significantly higher intensity of late propagations than Type 1 clones. Thus, possibly Type 3 clones have a higher probability of introducing faults and inconsistencies to a code-base than the clones of the other two clone-types.*

7.4.3 RQ 3: Late Propagations in Which Types of Clones Are More Related to Bugs?

Rationale. In a previous study Barbour et al. [29] found that late propagations in clones are related to bugs. However, they studied only Type 1 and Type 2 clones. Moreover, they did not report bugs for these two cases separately and thus, did not draw a comparative scenario between the bug-proneness of late propagations in Type 1 and Type 2 clones. We believe that understanding the comparative bug-proneness of the late propagations in three types of clones is important. If it is observed that the late propagations in a particular type of clones have a very low probability of being related to bugs compared to the other clone-types, then the late propagations of that particular clone-type could be ignored. In our study we investigate the bug-proneness of the late propagations in three types of clones (Type 1, Type 2, and Type 3) separately and show a comparative scenario considering these clone types.

Methodology. Previously Barbour et al. [29] performed a similar kind of investigation. They at first determined those clone pairs that experienced late propagations. Then, they determined whether any of these pairs ever experienced a fault fix during evolution. We perform a more in-depth investigation where we determine whether a fault fix occurred during the period of divergence of a particular late propagation. We believe that a particular late propagation occurred to a particular clone pair can only be related to a bug-fix if the bug-fix occurred during the late propagation period (i.e., the period of divergence). A pair of clones that experienced a late propagation can also experience a bug-fix however, the bug-fix might not occur during the period of late propagation. In this case we cannot relate this late propagation to the bug-fix.

We at first extract the SVN commit logs for each of the subject systems. The log contains the purpose why each of the revisions was created. If a revision was created because of a bug-fix, the corresponding log mentions it and generally includes the bug-ID. We identify the bug-fix commits from the commit log of a subject system using the heuristic proposed by Mockus and Votta [153]. Barbour et al. [29] also used the same heuristic in order to identify the bug-fix commits. As an example, if a commit message contains the word 'bug', then we consider the commit as a bug-fix commit. However, such an heuristic might cause false positives (i.e., identifying a commit as a bug-fix commit which was not actually done because of a bug-fix). According to the investigation results of Barbour et al. [29], this heuristic can help us detect bug-fix commits with a precision of 87%. We also perform manual investigations on the bug-fix commits detected from our candidate systems using this heuristic. From our analysis on 400 bug-fix commits (the first 50 bug-fix commits from each of the eight subject systems) we find that around 84% of these commits are true positives (i.e., are really bug-fix commits).

We detect late propagations considering each type of clones. A particular pair of clone fragments might experience late propagation more than once. In this investigation we detect all the late propagations that a particular clone pair experienced during evolution. A late propagation consists of a clone pair, a diverging commit, and a converging commit. We identify a late propagation to be related to a bug-fix if the following two conditions are satisfied: (1) at least one of the bug-fix commits (detected using our heuristic) occurred in between the diverging and converging commits of the late propagation, and (2) at least one clone fragment of the clone-pair was changed in this bug-fix commit. In this way we determine how many late propagations were related to bug fix. Considering each type of clones we determine the number of late propagations that were related to bug-fix. In case of Type 3 clones we disregard all those late propagations that occurred because of the changes in the non-matched portions of the clones. We compare the intensity of buggy late propagations in three types of clones in the following two ways.

- **Investigation 1:** By determining and comparing the possibility that a late propagation occurred to a clone-pair of a particular clone-type will be a buggy late propagation.
- **Investigation 2:** By determining and comparing the possibility that a clone fragment of a particular clone-type will experience a buggy late propagation.

Investigation 1: Comparison of the possibility that a late propagation occurred to a clone-pair of a particular clone-type will be a buggy late propagation.

Considering each clone-type of the each of the subject systems we determine the total number of late propagations, and the number of late propagations that are related to bug-fix. The percentage of buggy late propagations (i.e., the late propagations related to bug-fix) for each clone-type of each candidate system is shown in Table 7.7. Here we should note that more than one late propagations might be related to the same bug-fix. In case of Type 3 clones we consider only those late propagations that occurred because of the changes in matched portions of the clone fragments.

From Table 7.7 we see that Type 1 and Type 2 clones of Ctags and BRL-CAD, and also, Type 2 clones of jEdit did not experience any late propagations. In this table we also show the type of clones that experienced the highest proportion of bug-fix late propagations. We see that for four systems (BRL-CAD, Carol, Jabref, and Java-ML) out of eight subject systems, Type 3 clones experienced the highest proportion of buggy late propagations. In case of two subject systems (Camellia, Freecol) of the remaining ones, Type 2 clones experienced the highest proportion of bug-fix late propagations. In case of jEdit, Type 1 clones received the highest percentage of buggy late propagations. Thus, we see that for most of the subject systems Type 3 clones experienced the highest percentage of buggy late propagations.

Investigation 2: Comparison of the possibility that a clone fragment of a particular clone-type will experience a buggy late propagation.

Considering each type of clone of each of our candidates systems we determine the total number of clone genealogies created during system evolution, and the number of clone genealogies that experienced a late propagation related to a bug-fix. The percentage of these buggy late propagation genealogies with respect to all clone genealogies in case of each clone-type of each of the candidates systems is shown Table 7.8. In case of each of the subject systems, the table also shows the clone-type from which the highest proportion of clone genealogies experienced buggy late propagations.

From Table 7.8 we see that none of the three clone-types in Ctags received late propagations that are related to bug-fix. For most of the remaining subject systems (four systems out of seven), the proportion of Type 3 clones that experienced buggy late propagations is the highest compared to the proportions regarding the other two clone-types.

Statistical significance tests regarding the probability of experiencing buggy late propagation. We also wanted to investigate whether Type 3 clones have a significantly higher probability of experiencing buggy late propagations compared to Type 1 and Type 2 clones. Considering each clone type for each of the candidate systems we determined the percentage of clone-genealogies that experienced buggy

Table 7.7: Percentage of Late Propagations Related to Bug-fix

	Type 1		Type 2		Type 3		
System	**LP**	**PLPBF**	**LP**	**PLPBF**	**LP**	**PLPBF**	**TCHP**
Ctags	0	0%	0	0%	5	0	n/a
Camellia	61	3.28%	476	22.9%	727	15.13%	Type 2
BRL-CAD	0	0%	0	0%	1593	22.15%	Type 3
Freecol	3498	35.79%	8903	40.08%	1062	39.64%	Type 2
jEdit	14	71.43%	0	0%	77	70.13%	Type 1
Carol	12	0%	2	0%	644	34.63%	Type 3
Jabref	7	0%	3	0%	3026	41.21%	Type 3
Java-ML	4	0%	110	29.09%	65	36.92%	Type 3

LP = Total number of late propagation(s).

PLPBF = Percentage of late propagations that experienced bug-fix.

TCHP = The type of clones that experienced the highest proportion of bug-fix late propagations.

Table 7.8: Percentage of Clone Genealogies that Experienced Buggy Late Propagations

	Type 1		Type 2		Type 3		
System	**CG**	**PGBL**	**CG**	**PGBL**	**CG**	**PGBL**	**CTHP**
Ctags	146	0%	170	0%	549	0%	n/a
Camellia	584	0.51%	189	10.05%	571	8.58%	Type 2
BRL-CAD	343	0%	171	0%	586	4.77%	Type 3
Freecol	6593	0.71%	675	5.93%	4748	2.10%	Type 2
jEdit	42778	0.04%	1536	0%	9756	0.56%	Type 3
Carol	1969	0%	751	0%	3022	2.66%	Type 3
Jabref	4262	0%	895	0%	5849	4.07%	Type 3
Java-ML	429	0%	404	3.21%	1103	0.73%	Type 2

CG = Total number of clone genealogies of a particular clone-type.

PGBL = Percentage of genealogies that experienced a buggy late propagation.

CTHP = The clone-type where the highest proportion of clone genealogies experienced buggy late propagations.

late propagations. These percentages are recorded in Table 7.8. We perform MWW tests [4] to determine whether the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 and Type 2 cases. We perform the tests considering the significance level of 5%. According to our test results, the percentages for Type 3 case are significantly higher than the percentages for Type 1 case with a *p-value* < 0.01 for both two-tailed test and one-tailed test, and with an effect size of 0.67. We see that the *p-value* is smaller than 0.05. Thus, we can say that Type 3 clones exhibit a significantly higher probability of experiencing buggy late propagations compared to Type 1 clones. However, there is no significant difference between probabilities regarding Type 2 and Type 3 clones and also, between the probabilities for Type 1 and Type 2 clones.

Answer to RQ 3. *From our investigations we can state that Type 3 clones have a higher possibility of experiencing buggy late propagations compared to the clone fragments of the other two clone-types. Moreover, the probability of experiencing buggy late propagations for Type 3 clones is significantly higher compared to Type 1 clones.*

7.4.4 RQ 4: Clones of Which Clone-type(s) Have Higher Possibilities of Experiencing Bug-fixing Changes at the time of Convergence?

In the previous research question we investigated those late propagations each of which experienced a bug-fix. A bug-fix can occur at any commit operation during the period of late propagation. However, we believe that it is important to know whether the bug-fix occurred at the time of convergence (i.e., at the converging commit) or not. If a bug-fix commit affects two previously diverged clone fragments in such a way that they converge together, then we can understand that for fixing of the bug it was necessary to ensure consistency of the diverged clone fragments. Thus, these clone fragments might be considered for management with high priority. If not refactorable, then these clone fragments should always be tracked to maintain their consistency. The existing clone tracker *CloneTracker* [60, 61] does not automatically track all the clone fragments in a subject system. It allows programmers to select a subset of clones for tracking. Moreover, *CloneTracker* does not prioritize clones for tracking. Thus, a programmer is responsible to infer the more important clones for tracking and let *CloneTracker* know about this. In such a situation automatic prioritization of clone fragments for tracking can help programmers a lot. Our implemented system can automatically identify those clone fragments that received bug-fixing changes at the converging commit operation. Possibly these clone fragments can be prioritized for tracking. Here, we should note that in this research we only investigate the past evolution history of the clone fragments. This might not be the case that a clone fragment that experienced bug-fixing changes during a late propagation at past will also experience bugs in the future. We would like to investigate the likeliness of occurrence of such a phenomenon as a future work. In RQ 4 we investigate whether clone fragments experience bug-fixing changes at the time of convergence, and if so, how the intensity of this phenomenon differs across clone-types. We answer RQ 4 in the following way.

Methodology. We at first select all those late propagations each of which experienced a bug-fix using the methodology described in the previous research question. Then we automatically check each of these late propagations to determine whether the bug-fix occurred at the converging commit (i.e., the commit operation where two previously diverged clone fragments converged because of the changes in any one or both of the fragments). We determine the number of clone fragments that experienced such late propagations. Table 7.9 shows the percentage of clone fragments that experienced this type of late propagations with respect to all clone fragments in each clone-type of each of the candidate systems.

From Table 7.9 we see that in case of most of the subject systems disregarding Ctags, the proportion of clones that experienced bug-fixing changes at the time of convergence is the highest in Type 3 case. We disregard Ctags because none of the clone fragments in Ctags experienced bug-fixing changes.

Statistical significance test regarding the possibility of experiencing bug fixing changes at the time of convergence. As we have done previously, we wanted to investigate whether Type 3 clones exhibit a significantly higher possibility of experiencing bug-fixing changes at the time of convergence compared to Type 1 and Type 2 clones. We perform MWW tests [4] to determine whether the percentages of Type 3 clone-genealogies experiencing bug-fixing changes at the time of convergence are significantly different than the corresponding percentages for Type 1 and Type 2 case. According to our tests considering a significance level of 5%, the percentages regarding Type 3 case are significantly higher than the percentages regarding Type 1 case with *p-value* < 0.01 for both one-tailed and two-tailed tests, and with an effect size of 0.68. We see that the *p-value* is less than 0.05, and also, the effect size is large [69]. Thus, we can say that Type 3 clones exhibit a significantly higher possibility of experiencing bug-fixing changes at the time of convergence compared to Type 1 clones. However, the difference between the percentages regarding the Type 2 and Type 3 cases is not statistically significant. The same is true for the Type 1 and Type 2 cases.

Answer to RQ 4: According to our investigation, *Type 3 clones have higher possibilities of experiencing bug-fixing changes at the time of converging compared to the clone fragments in each of the other two clone types. Moreover, Type 3 clones exhibit a significantly higher probability of experiencing bug-fixing changes at the converging commits compared to Type 1 clones.* We have already discussed that the clone fragments that experience bug-fixing changes at the time of convergence should be considered important for tracking. Our implemented prototype tool can automatically identify such clone fragments by analyzing clone evolution history and thus, can help programmers identify the important tracking candidates while dealing with *CloneTracker*.

7.4.5 RQ 5: Do the Participating Clone Fragments in a Clone Pair that Experience Late Propagation Generally Remain in Different Files?

Rationale. According to a number of studies [53, 250], the program entities that often need to be changed together (i.e., that often require corresponding changes) should remain in close proximity to each other so that while changing a particular entity the developer does not miss to look at other entities that may require corresponding changes. Considering this fact we suspect that possibly the clone fragments in a clone pair that exhibit late propagation generally remain in two different files and as a result, the developers often forget to make corresponding changes to these clone fragments. We investigate this matter in the following way.

Methodology. We have already said that considering each of the clone types of each of the subject systems we identify the clone pairs that experienced late propagation. For each of these pairs we determined whether the participating clone fragments remain in different files or in the same file. We determined two percentages - (i) the percentage of the clone pairs having clone fragments from different source code files and (ii) the percentage of clone pairs consisting of clone fragments from the same file. These percentages are shown in Table 7.10.

Analysis. From Table 7.10 we see that for ten cases (for example Type 1 case of jEdit, Type 2 case of Jabref) the percentage of late propagation clone pairs each having clone fragments from different files is higher than the percentage of late propagation clone pairs each having clone fragments from the same file. However, the opposite is true for nine cases (for example Type 1 case of Freecol, Type 1 case of Carol). The clone fragments in the remaining five cases (Type 1 and Type 2 cases of Ctags and BRL-CAD, and Type 2 case of jEdit) did not experience any late propagations.

Table 7.9: Percentage of Clone Genealogies that Experienced Bug-fixing Changes at the Time of Convergence

	Type 1		Type 2		Type 3		
System	**CG**	**PCGB**	**CG**	**PCGB**	**CG**	**PCGB**	**CTHP**
Ctags	146	0%	170	0%	549	0%	n/a
Camellia	584	0%	189	9.52%	571	7.9%	Type 2
BRL-CAD	343	0%	171	0%	586	3.92%	Type 3
Freecol	6593	0.71%	675	4.59%	4748	1.58%	Type 2
jEdit	42778	0.03%	1536	0%	9756	0.42%	Type 3
Carol	1969	0%	751	0%	3022	1.16%	Type 3
Jabref	4262	0%	895	0%	5849	3.54%	Type 3
Java-ML	429	0%	404	3.21%	1103	0.73%	Type 2

CG = Total number of clone genealogies of a particular clone-type.

PCGB = Percentage of clone genealogies that received bug-fixing changes at the time of convergence.

CTHP = The clone-type where the highest proportion of clone genealogies experienced bug-fixing changes at the time of convergence.

Table 7.10: Percentage of Late Propagation Clone-pairs Each Having Clone Fragments from the Same File or from Different Files

	Type 1				Type 2				Type 3			
System	**CPL**	**SF**	**DF**	**MLP**	**CPL**	**SF**	**DF**	**MLP**	**CPL**	**SF**	**DF**	**MLP**
Ctags	0	0%	0%	n/a	0	0%	0%	n/a	5	100%	0%	S
Camellia	9	44.4%	55.6%	D	84	19%	81%	D	230	39.6%	60.4%	D
BRL-CAD	0	0%	0%	n/a	0	0%	0%	n/a	324	100%	0%	S
Freecol	232	94.8%	5.2%	S	177	93.2%	6.8%	S	272	46.3%	53.7%	D
jEdit	14	14.3%	85.7%	D	0	0%	0%	n/a	50	14%	86%	D
Carol	12	100%	0%	S	1	100%	0%	S	225	20.9%	79.1%	D
Jabref	7	100%	0%	S	3	0%	100%	D	601	26.6%	73.4%	D
Java-ML	4	0%	100%	D	33	87.9%	12.1%	S	33	81.8%	18.2%	S

CPL = Total number of clone pairs that experienced late propagations.

SF = Percentage of late propagation pairs each having clone fragments from the same file

DF = Percentage of late propagation pairs each having clone fragments from different files

MLP = The situation that occurs for most of the late propagation pairs. This filed can have either of the two values: ‘S’ or ‘D’

S = For most of the late propagations, the two clone fragments belong to the same file

D = For most of the late propagations, the two clone fragments belong to different files

Answer to RQ 5. From our investigation we understand that whether the two participating clone fragments in a particular clone pair remain in different files or in the same file, the clone pair can experience late propagation. Proximity of the constituent clone fragments in a clone pair possibly does not have any significant effect on the occurrence of late propagations to that pair.

7.4.6 RQ 6: Do Block Clones or Method Clones Exhibit Higher Intensity of Late Propagation?

Rationale. Intuitively, copying a block of statements from one method and pasting that block to several other methods is more difficult compared to copy-pasting a whole method. While pasting a block of statements into a method, the variable names and data types in the block might need to be changed in accordance with the variables and data types in that method. If there is a problem in making such correspondence and as a result, the variables are not changed correctly, then this will create inconsistency in future evolution. If a number of block clones (forming a clone class) are created with such inconsistencies, these inconsistencies in different clone fragments will be discovered at different times during evolution and as a result, late propagation will happen. Also, blocks might not have well defined boundaries as of methods. For this reason, keeping track of block clones might seem to be more difficult compared to method clones to a programmer.

Methodology. We perform the following two investigations for answering this research question.

- **Investigation 1:** *Investigating what proportions of the late propagations involve block-clones or method-clones.*

- **Investigation 2:** *Investigating what proportions of the block-clones and method-clones experienced late propagation.*

Investigation 1: *Investigating what proportions of the late propagations involve block-clones or method-clones.*

Considering each of the clone types of each of the subject systems, we at first determine those clone pairs that exhibited late propagation and then we determine whether the clone fragments in such a pair are block clones or method clones. We calculate: (i) the percentage of late propagation clone pairs each consisting of at least one block clone, and (ii) the percentage of late propagation clone pairs consisting of method clones only. These percentages regarding each clone-type of each subject system are shown in Table 7.11.

Analysis. From Table 7.11 we see that for almost all of the cases, the percentage of late propagation clone pairs involving block clones is much higher (100 % in many cases such as Type 2 case of Freecol) than the percentage of late propagation pairs consisting of only method clones. Although we determine the percentage of late propagation clone pairs having at least one block clone, for most of the cases we observed that both of the clones in such a pair are block clones. However, for a very few cases (such as Type 1 and Type 2 cases of Ctags) we did not get any clone pair experiencing late propagation.

Table 7.11: Percentage of Late Propagation Clone-pairs Each Consisting of Method Clones or Block Clones

	Type 1				Type 2				Type 3			
System	**CPL**	**BC**	**MC**	**MLP**	**CPL**	**BC**	**MC**	**MLP**	**CPL**	**BC**	**MC**	**MLP**
Ctags	0	0%	0%	n/a	0	0%	0%	n/a	5	100%	0%	B
Camellia	9	100%	0%	B	84	100%	0%	B	230	99.1%	0.87%	B
BRL-CAD	0	0%	0%	n/a	0	0%	0%	n/a	324	100%	0%	B
Freecol	232	100%	0%	B	177	100%	0%	B	272	97.1%	2.9%	B
jEdit	14	57.1%	42.9%	B	0	0%	0%	n/a	50	90%	10%	B
Carol	12	100%	0%	B	1	100%	0%	B	225	95.6%	4.4%	B
Jabref	7	100%	0%	B	3	0%	100%	M	601	98.2%	1.8%	B
Java-ML	4	0%	100%	M	33	87.9%	12.1%	B	33	90.9%	0.1%	B

CPL = Total number of clone pairs that experienced late propagations.

BC = Percentage of late propagation pairs each consisting of at least one block clone

MC = Percentage of late propagation pairs each consisting of method clones only

MLP = The situation that occurs for most of the late propagation pairs. This filed can have either of the two values: 'B' or 'M'

B = For most of the late propagation pairs, at least one of the two clone fragments is a block clone

M = For most of the late propagation pairs, both clone fragments are method clones

Investigation 2: *Investigating what proportions of the block-clones and method-clones experienced late propagation.*

Considering each clone-type of each of the subject systems we first identify all the clone genealogies created during evolution, and then separate those into two disjoint subsets: (1) block clone genealogies, and (2) method clone genealogies. We also determine which of these block clone genealogies as well as which of the method clone genealogies experienced late propagations. Finally, we calculate the following two percentages:

- The percentage of block clones (i.e., block clone genealogies) that experienced late propagation with respect to all block clones.
- The percentage of method clones that experienced late propagation with respect to all method clones.

Table 7.12 shows the following four measures considering each clone-type of each of the subject systems: (1) the number of block clones, (2) the number of block clones that experienced late propagation, (3) the number of method clones, and (4) the number of method clones that experienced late propagations during evolution. Table 7.13 shows the percentages of block clones as well as method clones that experienced late propagations with respect to all block clones and method clones respectively. The percentages in this table were calculated from the values in Table 7.12. Table 7.13 shows that for most of the cases, the percentage of method clones that experienced late propagations is smaller compared to the corresponding percentage of block clones. We found only two cases (i.e., Type 1 case of Java-ML, and Type 2 case of Jabref) where the percentage of method clones that experienced late propagations is higher than the corresponding percentage of block clones.

Statistical Significance Tests. We wanted to determine whether the percentages of block clones that experienced late propagations are significantly higher than the corresponding percentages of method clones. Table 7.13 contains 24 cases (8 systems $\times$ 3 clone-types) in total. We perform Mann-Whitney-Wilcoxon tests [3,4] to determine whether the percentages regarding block clones in these cases are significantly higher compared to the percentages regarding method clones. We consider a significance level of 5%. According to our test, the percentages of block clones that experienced late propagations are significantly higher than the corresponding percentages of method clones with *p-value* = 0.02 for two-tailed test and 0.01 for one-tailed test, and with an effect size of 0.32. We see that the *p-value*s are smaller than 0.05, and thus, the percentages of block clones that experienced late propagations are significantly higher than the corresponding percentages of method clones.

Answer to RQ 6: *The clone pairs that experience late propagation generally consist of block clones instead of method clones.* Our second investigation shows that *block clones exhibit a significantly higher tendency of experiencing late propagation than method clones.* Such an observation implies that block clones possibly have higher probability of introducing inconsistencies to a code-base compared to the method clones. Thus, creating block clones is more risky than creating method clones. We should possibly consider refactoring (if possible) block clones with higher priority.

Table 7.12: The Number of Block Clones and Method Clones that Experienced Late Propagations

	Type 1				Type 2				Type 3			
System	**NB**	**NBL**	**NM**	**NML**	**NB**	**NBL**	**NM**	**NML**	**NB**	**NBL**	**NM**	**NML**
Ctags	85	0	61	0	99	0	71	0	336	6	213	0
Camellia	398	10	186	0	177	30	12	0	520	86	51	3
BRL-CAD	311	0	32	0	153	0	18	0	520	31	66	0
Freecol	5721	47	872	0	508	54	167	0	3545	116	1203	18
jEdit	16828	10	25950	11	767	0	769	0	6437	46	3319	22
Carol	907	8	1062	0	296	2	455	0	1728	112	1294	21
Jabref	2262	9	2000	0	640	0	255	4	3797	246	2052	36
Java-ML	224	0	205	5	240	13	164	6	553	18	550	5

NB = Number of block clone genealogies created during evolution.

NBL = Number of block clone genealogies that experienced late propagation.

NM = Number of method clone genealogies created during evolution.

NML = Number of method clone genealogies that experienced late propagation.

Table 7.13: Percentage of Block Clones and Method Clones that Experienced Late Propagations

	Type 1			Type 2			Type 3		
System	**PBC**	**PMC**	**HP**	**PBC**	**PMC**	**HP**	**PBC**	**PMC**	**HP**
Ctags	0%	0%	n/a	0%	0%	n/a	1.79%	0%	PBC
Camellia	2.51%	0%	PBC	16.94%	0%	PBC	16.53%	5.88%	PBC
BRL-CAD	0%	0%	n/a	0%	0%	n/a	5.96%	0%	PBC
Freecol	0.82%	0%	PBC	10.62%	0%	PBC	3.27%	1.49%	PBC
jEdit	0.06%	0.04%	PBC	0%	0%	n/a	0.71%	0.66%	PBC
Carol	0.88%	0%	PBC	0.67%	0%	PBC	6.48%	1.62%	PBC
Jabref	0.39%	0%	PBC	0%	1.56%	PMC	6.47%	1.75%	PBC
Java-ML	0%	2.43%	PMC	5.42%	3.65%	PBC	3.25%	0.91%	PBC

PBC = Percentage of block clones that experienced late propagation.

PMC = Percentage of method clones that experienced late propagation.

HP = The larger one of the above two percentages. This field can have either of the two values: PBC, and PMC.

Table 7.14: Statistics Regarding SPCP clones and Late Propagations

	Type 1			Type 2			Type 3		
System	**SPCP**	**LP**	**LPSPCP**	**SPCP**	**LP**	**LPSPCP**	**SPCP**	**LP**	**LPSPCP**
Ctags	26	0	n/a	16	0	n/a	118	5	3
Camellia	64	61	61	21	476	216	78	727	642
BRL-CAD	80	0	n/a	17	0	n/a	111	1593	1559
Freecol	143	3498	3498	94	8903	8885	180	1026	598
jEdit	486	14	14	21	0	n/a	128	77	25
Carol	84	12	12	110	2	2	498	644	336
Jabref	50	7	7	84	3	3	544	3026	2111
Java-ML	119	4	4	34	110	104	355	65	59

SPCP = The number of SPCP clone fragments.

LP = The total number of late propagations

LPISPCP = The number of late propagations involving SPCP clones

7.4.7 RQ 7: Do Late Propagations Mainly Occur to the SPCP Clones or Non-SPCP Clones?

Rationale. In a previous study [163] we showed that SPCP (Similarity Preserving Change Pattern) clones are the important ones from the perspectives of clone management (such as clone tracking or refactoring). We suggested to mainly focus on managing SPCP clones when taking clone management decisions [162,163]. In this research question (i.e., RQ 7) we investigate whether late propagations mostly occur to the SPCP clones. In such a case we can say that proper management of SPCP clones (i.e., through refactoring or tracking) can help us minimize late propagations. We perform our investigation in the following two ways.

- **Investigation 1:** By investigating what proportions of late propagations occur to the SPCP clones.
- **Investigation 2:** By investigating the frequency of the occurrences of late propagations to the SPCP clones and non-SPCP clones.

Investigation 1: *Investigation on the proportion of late propagations occurred to the SPCP clones.*

If it is observed that most of the late propagation occur to the SPCP clones rather than non-SPCP clones, then we can decide that managing SPCP clones through refactoring and/or tracking can help us minimize the occurrences of late propagations considerably. We investigate in the following way.

We at first determine the SPCP clones in the code-base by applying our detection mechanism elaborated in our previous study [162]. Then, we determine all the occurrences of late propagations. We automatically

Table 7.15: Percentage of Late Propagations Involving SPCP Clones

System	**PLS - Type 1**	**PLS - Type 2**	**PLS - Type 3**
Ctags	0%	0%	60%
Camellia	100%	45.38%	88.31%
BRL-CAD	0%	0%	97.86%
Freecol	100%	99.8%	58.28%
jEdit	100%	0%	32.48%
Carol	100%	100%	52.17%
Jabref	100%	100%	69.76%
Java-ML	100%	94.54%	90.76%

PLS = Percentage of late propagations involving SPCP clones

OPPS = Overall percentage per system

check each of these late propagations and determine which late propagations involve SPCP clones (i.e., any of the two participating clone fragments in the late propagation are SPCP clones). Considering each clone-type of each of the candidate systems we determine how many of the corresponding late propagations involve SPCP clones. Table 7.14 shows the total number of SPCP clones, total number of late propagations, and the number of late propagations that involve SPCP clones. The percentage of late propagations involving SPCP clones considering each clone-type of each of the candidate systems is shown in Table 7.15.

From Table 7.15 we see that the percentage of late propagations involving SPCP clones is zero for Type 1 and Type 2 cases of Ctags and BRL-CAD, and also, for Type 2 case of jEdit. The reason is that we did not get any late propagations for these cases. This is also evident from Table 7.14. However, from this table (i.e., Table 7.14) we see that in each of these cases we found SPCP clones. Table 7.15 shows that in case of each of the subject systems, all of the late propagations in Type 1 clones involved SPCP clones. The same is true for Type 2 cases of most of the subject systems except Camellia. From Table 7.15 and 7.14 we understand that a number of late propagations in Type 2 and Type 3 cases do not involve SPCP clones. In Section 7.2 we explained that two clone fragments might experience late propagation(s), however, they will not be regarded as SPCP clones if they finally diverge and do not converge again. The late propagations that do not involve SPCP clones were experienced by such non-SPCP clone pairs.

Considering all clone types of all the candidate systems we found 20253 occurrences of late propagations in total, and 18139 of these involved SPCP clones. Thus, around 89.56% of the total late propagations involved SPCP clones. Such a finding implies that late propagations mainly occur to the SPCP clones. From this we come to the decision that we can considerably minimize the future occurrences of late propagations by managing the SPCP clones through refactoring and tracking.

Table 7.16: Frequency of Late Propagations in SPCP and Non-SPCP Clones

	Type 1			Type 2			Type 3		
System	**FLS**	**FLNS**	**HF**	**FLS**	**FLNS**	**HF**	**FLS**	**FLNS**	**HF**
Ctags	0	0	n/a	0	0	n/a	1	1	n/a
Camellia	6.78	0	FLS	2.27	6.87	FLNS	2.89	3.73	FLNS
BRL-CAD	0	0	n/a	0	0	n/a	4.87	8.5	FLNS
Freecol	15.07	0	FLS	54.18	1.38	FLS	5.43	2.86	FLS
jEdit	1	0	FLS	0	0	n/a	1.77	1.35	FLS
Carol	1	0	FLS	2	0	FLS	2.52	3.34	FLNS
Jabref	1	0	FLS	1	0	FLS	6.04	3.63	FLS
Java-ML	1	0	FLS	3.59	1.5	FLS	2.19	1	FLS

FLS = Frequency of late propagations in SPCP clones.

FLNS = Frequency of late propagations in non-SPCP clones.

HF = Higher Frequency. This field can have two values: FLS or FLNS

Investigation 2: *Investigation on the frequency of the occurrences of late propagations to the SPCP clones and non-SPCP clones.*

From the previous investigation we understand that late propagations mainly occur to the SPCP clones rather than the non-SPCP clones. In this investigation we determine whether late propagations are more frequent to the SPCP clones or to the non-SPCP clones. If the frequency of late propagations to the SPCP clones is higher compared to the frequency of late propagations to the non-SPCP clones, then we can again decide that refactoring and tracking of SPCP clones can help us minimize late propagations. We perform our investigation in the following way.

Considering each clone-type of each of the candidate systems we determine the following four measures:

- **Measure 1:** The number of distinct clone pairs that involve SPCP clones and experienced late propagations,
- **Measure 2:** The number of distinct clone pairs that do not involve SPCP clones and experienced late propagations,
- **Measure 3:** The total number of late propagations each involving SPCP clone(s), and
- **Measure 4:** The total number of late propagations involving only non-SPCP clones.

We then determine the following two frequencies from the above measures.

- The frequency of late propagations in SPCP clones by dividing **Measure 3** by **Measure 1** .
- The frequency of late propagations in the non-SPCP clones by dividing the fourth measure (**Measure 4**) by the second one (**Measure 2**).

These frequencies for each clone-type of each of the subject systems are shown in Table 7.16. The table shows that for most of the cases (i.e., except Type 2 and Type 3 cases of Camellia, Type 3 case of Carol, and Type 3 case of BRL-CAD) the frequency of late propagations in SPCP clones is higher than the frequency of late propagations in non-SPCP clones.

Statistical significance test regarding the frequency of late propagations in SPCP and non-SPCP clones: We also wanted to investigate whether the frequency of late propagations in SPCP clones is significantly higher than the frequency of late propagations in non-SPCP clones. If we consider all three clone types of all eight candidate systems we get 24 cases (3 clone-types x 8 candidate systems) in total. We have already mentioned that we did not get any late propagations for Type 1 and Type 2 cases of Ctags and BRL-CAD, and Type 2 case of jEdit. Considering the remaining 19 cases we determine two sets of frequencies. One set contains the frequencies of late propagations in SPCP clones in these 19 cases. The other set contains the frequencies of late propagations in non-SPCP clones. We perform the Mann-Whitney-Wilcoxon test [3,4] to determine whether the samples in these two sets are significantly different. We consider a significance level of 5%. According to the test results, the difference between these two sets is significant with *p-value* (probability value) = 0.04 (< 0.05) for two-tailed test and 0.02 for one-tailed test, and with an effect size of 0.33. We see that the *p-value*s are smaller than 0.05. Thus, we can say that the frequency of late propagations in SPCP clones is significantly higher than the frequency of late propagations in non-SPCP clones.

Answer to RQ 7: *According to our investigations we can state that most of the late propagations (around 89.56%) involve SPCP clones. In other words, late propagations mainly occur to the SPCP clones. Also, the frequency of the occurrence of late propagations in SPCP clones is significantly higher compared to non-SPCP clones. These findings imply that by managing SPCP clones through refactoring and tracking we can minimize the occurrences of late propagations considerably.* Moreover, SPCP clones not only include those clone fragments that experienced late propagations but also other clone fragments that are important to be updated consistently [163]. Thus, we should primarily focus on managing the SPCP clones. Management of SPCP clones through refactoring and tracking will not only help us minimize late propagations but also will help us in better maintenance of software systems.

7.4.8 Discussion

In our research we answered seven research questions. In this section we mention and discuss our important findings regarding late propagation from the answers to these research questions, and focus on the possible reasons behind these findings.

Finding 1: *Type 3 clones have the highest possibility of experiencing late propagations among the three clone-types: Type 1, Type 2, and Type 3.*

From our investigations in RQ 2 we found that Type 3 clones exhibit the highest intensity of late propagations among the three clone-types (Type 1, Type 2, and Type 3). A possible reason behind why Type 3 clones exhibit a higher tendency of late propagations is that Type 3 clones are gapped clones. Because of the existence of these gaps (i.e., non-cloned lines) consistent changing of Type 3 clones is not always as straight forward as of the other two clone-types. We suspect that higher percentage of inconsistencies as well as late propagations in Type 3 clones are caused by the gaps. However, we do not have enough supporting evidence for this. In future, we would like to investigate whether different levels of dissimilarities (i.e., different dissimilarity thresholds) in Type 3 clones have effects on the intensity of late propagations in such clones.

Finding 2: *Late propagations in Type 3 clones have the highest possibility of introducing bugs and inconsistencies to the code-base compared to the late propagations in the other two clone-types: Type 1 and Type 2.*

From our answers to the research questions RQ 3 and RQ 4 we understand that the late propagations in Type 3 clones have a higher tendency of introducing bugs and inconsistencies to a software system's code-base compared to the late propagations in each of the other two clone-types. We again suspect that the main reason behind such a scenario is the existence of gaps in Type 3 clone fragments. However, we do not have supporting evidence for this. We would like to investigate this in future.

Finding 3: *Creating block clones is more risky than creating method clones because late propagations mostly occur in block clones.*

According to our analysis in RQ 6, the clone fragments that experience late propagations are mostly block clones rather than method clones. Such a finding implies that creating block clones is more risky than creating method clones. The reason behind why a significantly higher proportion of block clones experience late propagations compared to method clones is that block clones do not have well defined boundaries as of method clones. Thus, tracking as well as consistent updating of block clones without proper tool support is intuitively much more difficult compared to method clones. Here, we should again note that an existing tool called *CloneTracker* [61] provides support for tracking and simultaneous editing of clone fragments. However, this tool tracks only the programmer selected clone fragments. Currently there is no tool for automatic tracking of all clone fragments in a software system. Such a tool could help us minimize late propagations in code clones considerably.

Finding 4: *Managing SPCP clones can help us minimize the occurrences of late propagations considerably.*

From our investigation regarding RQ 7 we observe that late propagations mainly occur to SPCP clones (i.e., the clones that evolve following a similarity preserving change pattern called SPCP). A previous study [163] suggests us to mainly consider SPCP clones for refactoring and tracking. As late propagation clones are mostly SPCP clones, we believe that managing of SPCP clones will help us minimize late propagations considerably. Moreover, from our findings we confirm that SPCP clones are the important candidates from a clone management perspective. We also believe that a clone tracker with the capability of automatically detecting and tracking of SPCP clones could help programmers efficiently manage code clones by minimizing late propagations.

7.5 Related Work

A number of studies have already been done on clone evolution and late propagation in clones during evolution. Kim et al. [119] studied clone evolution by defining and extracting clone genealogies from two Java systems using CCFinder[5] as the clone detector. Krinke [131] studied the consistent and inconsistent changes to the Type 1 clones considering the evolutions of five open source subject systems using Simian[6] clone detector. He also studied the stability of clones [132] in comparison with non-cloned code. Göde et al [75, 78] analyzed clone evolution and its effect on software maintenance by enhancing Krinke's study [132].

In a recent study, Barbour et al. [30] investigated eight different patterns of late propagation by studying three open-source subject systems written in Java and identified two patterns that have higher likelihood of introducing inconsistencies to a code-base. They used three clone detection tools NiCad, CCFinder, and Simian in their study. However, Type 3 clones were not considered in this study. Aversano et al. [19] investigated clone evolution on two subject systems to determine how clones are maintained. According to their observation 18% of the clones experienced late propagation. They show that late propagation in clones can directly be related to bugs and thus late propagation is risky. In another study Thummalapenta et al. [241] investigate late propagation in clones considering four subject systems and reported that late propagation is often related to faults and inconsistencies.

Bazrafshan [36] conducted an empirical study to investigate how differently the near-miss clones evolve compared to the identical clones. He investigated the conversion of near-miss clones to identical clones, and also, identical clones to near-miss clones. According to his findings, near-miss clones should be given a higher priority than the identical clones when taking clone management decisions. Our study is different in the sense that we investigate the intensity and harmfulness of late-propagation in three clone-types (Type 1, Type 2, and Type 3) separately.

In a previous study [168] we investigated and compared the bug-proneness of code clones in different clone-types. We did not investigate late propagation in that study. However, in our study presented in this chapter we detect late propagation in different types of code clones, and investigate whether late propagation in code clones are related to bugs. Thus, our contributions in this study are different than in our previous study [168].

We see that while there are a number of great studies, none of these focus on the intensity and harmfulness

[5]CCFinder. `http://www.ccfinder.net/ccfinderxos.html`

[6]Simian. `http://www.harukizaemon.com/simian/index.html`.

of late propagation separately in different types of clones. Also, the existing studies did not investigate the tendency of late propagation in SPCP clones (i.e., the clone fragments that evolve following a Similarity Preserving Change Pattern). In this study, we investigate these issues by answering seven research questions. We believe that our findings are important and have the potential to help us in better clone maintenance.

7.6 Threats to Validity

The number as well as the percentage of clone genealogies that experienced late propagation may vary because of the variation of the detection parameters of the clone detection tool (NiCad in our study). Wang et al. [254] defined this problem as the *confounding configuration choice* problem and conducted an extensive study considering six clone detectors to ameliorate the effects of the problem. However, the settings that we have used for NiCad are considered standard and with these settings NiCad can detect clones with higher precision and recall [198, 201]. Thus, the experimental results reported in this chapter are of significant importance.

NiCad can detect three types of clones: Type 1 (identical), Type 2 (near-miss), and Type 3 (near-miss). Any other near-miss clone detectors [231] could provide us with different experimental results as well as different scenarios. However, Svajlenko and Roy [231] showed that NiCad is a very good clone detector for detecting all three types of code clones in comparison with the other modern clone detectors. Also, NiCad can report three types of clones (Type 1, Type 2, Type 3) separately. Thus, it helped us to investigate the intensity of late propagation on these clone-types separately.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique followed by Barbour et al. [30]. Such a technique proposed by Mocus and Votta [153] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. Barbour et al. [30] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits. We also perform manual investigations on the bug-fix commits detected from our subject systems. As mentioned in Section 7.4.3, we confirmed that around 84% of these commits are true positives. Thus, we believe that our reported results regarding the bug-proneness of different types of late propagations in code clones are considerable.

In case of a Type 3 clone pair it might happen that one of the two clone fragments experienced particular changes, however, the two fragments were still considered as clones. The particular changes experienced by one fragment might later be propagated to the other fragment. Our late propagation detection mechanism cannot identify such type of late propagation where the participating clone fragments always preserve their similarity during the whole period of propagation.

Two clone fragments of a particular clone type might be regarded as clone fragments of another type after experiencing the diverging period. Bazrafshan [36] previously investigated on such conversions of clone types. Xie et al. [263] called it clone mutation and performed an in-depth investigation regarding this. According to their investigation on three subject systems, up to 60% of the clone genealogies can experience mutation. In our research we were concerned about the late propagations in each of the three clone-types (Type 1, Type 2, and Type 3) separately. We ignored type conversions (i.e., mutations) of code clones. However, if a clone-pair is considered of different clone-types during different periods of evolution, we find the late propagations experienced by the clone-pair considering each duration separately. Thus, our experimental results regarding late propagation considering individual clone-type are not affected by the type-conversion of code clones.

The difference between the number of clone genealogies in different clone-types might be a confounding factor behind our finding regarding the comparative scenario of experiencing late propagations by different clone-types. We wanted to investigate whether this is true. Our finding is that *Type 3 clones have a higher possibility of experiencing late propagations than Type 1 and Type 2 clones.* We investigate whether this finding has been affected by the number of clone genealogies in different clone-types. We first consider the two clone-types: Type 1 and Type 3. From Table 7.4 we see that for five subject systems (Ctags, BRL-CAD, Carol, Jabref, and Java-ML) the number of Type 3 clone genealogies is higher than the number of Type 1 clone genealogies. For the remaining three systems (Camellia, jEdit, and Freecol), the number of Type 1 clone genealogies is higher. However, from Table 7.5 we see that for each of these eight systems, the percentage of clone genealogies that experienced late propagation is much higher in Type 3 case than in Type 1 case. Thus, it seems that the number of clone genealogies in Type 1 and Type 3 case does not impact the comparative scenario of experiencing late propagations by the code clones of these two types. Now, we make a comparison between the clone-types: Type 2 and Type 3. For each of our eight subject systems, the number of Type 2 clone genealogies is much lower compared to Type 3 (c.f., Table 7.4). However, for three systems (Camellia, Freecol, and Java-ML) the proportion of late propagation clone genealogies is higher in Type 2 case compared to Type 3 (c.f., Table 7.5). Thus, we again see that the total numbers of clone genealogies in the two clone-types (Type 2, and Type 3) do not affect the comparative scenario of experiencing late propagations by the code clones of these two types. Finally, we believe that our findings are not affected by the number of clone-genealogies in different clone-types.

A clone pair which was created just before the last revision of our investigated evolution history of a candidate system, and diverged at the last revision can converge in near future (i.e., shortly after the last revision) which is unknown to us. The earliest possible revision of convergence can be the one which will be created just after the last revision. However, as the future is unknown to us we consider this pair as a non-SPCP clone pair in our experiment. We should also note that this pair has not yet completed experiencing a late propagation according to the known evolution history. Thus, we believe that our decision about considering this pair as a non-SPCP clone pair is reasonable and such a consideration has not affected our findings.

The number of subject systems that we have used in our experiment is not sufficient to take a concrete decision regarding the possible causes of late propagation. However, we selected our subject systems focusing on the diversity of sizes (from small to large) and application domains (five different application domains) to generalize our findings. Thus, we believe that our findings are important and have the potential to minimize late propagation in clones.

7.7 Summary

In this study, we investigate late propagation in three types of clones (Type 1, Type 2, and Type 3) separately. Through our experiment we tried to answer seven important research questions (mentioned in the Introduction) regarding the intensity, and bug-proneness of late propagation. According to our study on thousands of revisions of eight diverse subject systems written in two programming languages,

- The percentage of late propagations in Type 3 clones occurred only because of the changes in the non-matched (i.e., non-cloned) portions of the clone fragments is very low (less than one) for most of our candidate systems. However, this proportion can be considerable for some subject systems (for example our subject system jEdit). Such late propagations should be ignored when making clone management decisions. Our implemented system can automatically detect such ignorable late propagations so that we can discard these from considerations while taking clone management decisions.
- The intensity of late propagation in Type 3 clones is higher compared to the other two clone-types (Type 1, and Type 2).
- More importantly, Type 3 clones have higher possibilities of experiencing buggy late propagations than the clone fragments in the other two types.
- Most of the clone fragments that experience late propagations are block clones. It seems that the creation of block clones is more risky than the creation of method clones.
- Refactoring and tracking of SPCP-clones can possibly help us minimize the future occurrences of late propagations considerably.

As a future work, we plan to investigate whether programming languages as well as application domains of the subject systems can bias the intensity of late propagation. Considering Type 3 clones, we plan to investigate different late propagation patterns, their frequencies and effects on software maintenance.

Chapter 8
Automatic Mining of Important Clones (AMIC)

8.1 Introduction

In the last five chapters we described our studies on identifying and ranking code clones as well as prioritizing clone types for the purpose of management such as refactoring and tracking. By accumulating all the techniques and technologies in all of our studies we have developed an automatic system, AMIC (Automatic Mining of Important Clones), for identifying and ranking important clones for refactoring and tracking. We believe that AMIC has the potential to support clone management by pinpointing and prioritizing important code clones for management from different perspectives such as evolution pattern, change-proneness, bug-proneness, and late propagation tendencies of code clones. We develop AMIC using Java with MySQL as the back-end database server. Chapter 8 presents a detailed description of AMIC.

The rest of the chapter is organized as follows. Section 8.2 discusses an example use case of AMIC, Section 8.3 describes the concept behind implementing AMIC, Section 8.4 elaborates on the working procedure of AMIC, Section 8.5 focuses on the implementation, Section 8.6 describes the output generated by AMIC, and Section 8.7 concludes this chapter by mentioning possible future work.

8.2 An Example Use Case of AMIC

Let us assume that a software system has been in the maintenance phase for a long time without clone management. Recently, the client reported a bug saying that certain functionality in a certain module of the software system is not working in the expected way. A developer was appointed to fix the bug. After investigating the bug in the reported module she fixes it by modifying a piece of code in that module.

However, after some days the client again reports a similar bug in another module. The project manager asks the developer to fix it and investigate why the same bug is being reported even after fixing. After investigating, the programmer reports that there was a similar piece of buggy code in the newly reported module. She also fixed it in the similar way. Then, the project manager suspects that the similar piece of buggy code might even exist in some other places not yet reported. So, he asks the developer to search for those in the whole code-base. The developer investigates and finds that some other similar pieces of code with the same bug really exist in the code-base and these code fragments also need to be fixed.

From this the project manager realizes that the code-base might have many other groups of similar code fragments. He feels the necessity of refactoring each of these groups so that in order to fix such a bug in future a developer does not need to change in too many places. He asks the developer to first detect code clones in the software system using a clone detection tool and then, to refactor those clones.

The developer downloads the clone detection and refactoring tools. She applies a clone detector and detects a large amount of clones grouped into different clone groups from the system. Then, she attempts to apply a refactoring tool on the detected clones. However, she experiences that there are many situations where the refactoring tools cannot refactor the clone fragments. She can manually refactor in some of these situations. Also, during manual checking she understands that many clones might not be important to be refactored and also many clones are not even refactorable. Moreover, while deciding to refactor some clones she feels the necessity of understanding how they evolved in the past. She realizes that she should primarily focus on those clones that are more change-prone and have a tendency of co-evolving preserving their similarity. The clones that never changed in the past might be kept in the system as they are because they have very low probability of getting changed in future. She also understands that some clone fragments are eligible to be refactored on the basis of their syntactic structure however, they are closely related to their surrounding non-clone fragments, and thus removal of these clone fragments through refactoring might not be a wise decision. Removal of such clone fragments might negatively affect the future evolution of the related non-clone fragments. Such clone fragments might be important for tracking using a clone tracker. However, to consider all these things she needs to analyze the clone evolution history. She understands that deciding important refactoring as well as tracking candidates by analyzing the evolution histories of a large set of clones might even take several months to complete. She feels helpless and realizes the necessity of a tool that can automatically analyze the clone evolution history and identify the important clones to be refactored or tracked. We believe that our tool AMIC can be her best friend in this situation. AMIC automatically mines and analyzes the evolution histories of the clone fragments and reports the important clones to be refactored or tracked by ranking them according to the necessity of refactoring or tracking.

8.3 Concept behind AMIC

AMIC automatically analyzes the past evolution history of the clone fragments in a code-base and identifies which clone fragments are SPCP clones (i.e., which clone fragments evolved by following a *Similarity Preserving Change Pattern* called SPCP). A *Similarity Preserving Change Pattern* consists of only *Similarity Preserving Change* and/or *Re-synchronizing Change.* We have discussed these terms in Chapter 3. Fig. 8.1 shows examples of similarity preserving change and re-synchronizing change. A re-synchronizing change consists of a diverging change and a converging change.

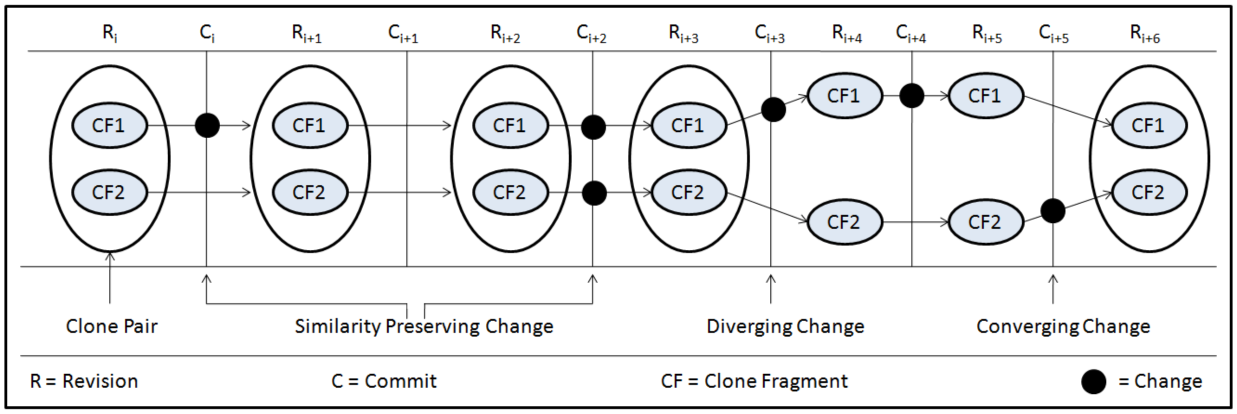


Figure 8.1: A similarity preserving change pattern followed by two clone fragments *CF1* and *CF2* is presented in this figure. We see that *CF1* and *CF2* received similarity preserving changes in commits: C_i and C_{i+1}. They received diverging change at commit C_{i+3}. However, they again converged after the changes in commit C_{i+5}. Thus, the figure also shows a re-synchronizing change consisting of the diverging change and converging change.

8.3.1 Separating the SPCP Clones into Two Subsets

AMIC analyzes the evolutionary coupling of the SPCP clones, and separates these SPCP clones into two disjoint subsets: (1) *cross-boundary SPCP clones*, and (2) *non-cross-boundary SPCP clones* on the basis of this analysis. The SPCP clones in the first subset have evolutionary couplings (i.e., change couplings) with other code fragments (non-clone fragments or clone fragments from other clone classes) beyond their class boundaries. Thus, removal of such an SPCP clone fragment through refactoring might negatively affect the future evolution of the related code fragments beyond its class boundary [162]. Our empirical study [162] shows that *cross-boundary SPCP clones* are the most suitable ones for tracking. The details of how we detect *cross-boundary SPCP clones* using *association rules* and constrained *support* and *confidence* values have been presented in Chapter 5. Non-cross-boundary SPCP clones are the most suitable ones for refactoring. *AMIC* also makes groups of the non-cross-boundary SPCP clones. We conducted an in-depth empirical study [162] on the cross-boundary and non-cross-boundary SPCP clones. AMIC ranks the cross-boundary as well as non-cross-boundary SPCP clones on the basis of the strengths of their evolutionary coupling and bug-proneness. The ranking mechanisms have been elaborated in Chapters 3 and 5.

8.4 Description of AMIC

AMIC works on the output of a clone detector. Given the SVN repository URL of a subject system, it first automatically extracts all the revisions of the system from the repository using *export* command of SVN, and then applies the clone detector to detect clones from each of the revisions. The user only specifies the SVN repository URL. The rest of the task is automatically done by AMIC. After detecting clones from each

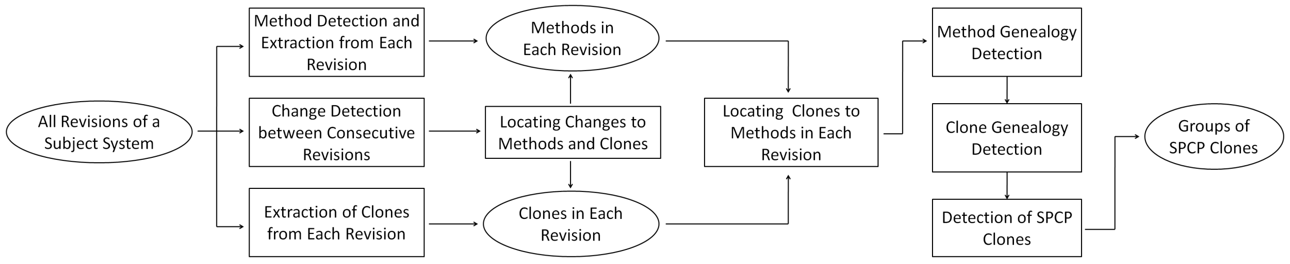

All Revisions of a Subject System
Method Detection and Extraction from Each Revision
Change Detection between Consecutive Revisions
Extraction of Clones from Each Revision
Methods in Each Revision
Locating Changes to Methods and Clones
Clones in Each Revision
Locating Clones to Methods in Each Revision
Method Genealogy Detection
Clone Genealogy Detection
Detection of SPCP Clones
Groups of SPCP Clones

Figure 8.2: The steps in detecting SPCP clones. There are eight processing steps in detecting SPCP clones. The rectangles in this figure indicate these steps.

of the extracted revisions AMIC performs eight sequential steps (c.f., Fig. 8.2) in order to detect the SPCP clone fragments. These steps are:

- Method detection and extraction from each of the revisions using CTAGS[1],
- Extraction of code clones for each revision from the clone detection results of the clone detector.
- Detection of changes between every two consecutive revisions using *diff*,
- Locating these changes to the already detected methods as well as clones of the corresponding revisions,
- Locating the code clones detected from each revision to the methods of that revision,
- Detection of method genealogies considering all revisions using the technique proposed by Lozano and Wermelinger [145],
- Detection of clone genealogies by identifying the propagation of each clone fragment through a method genealogy, and
- Detection of SPCP clone fragments by analyzing clone change patterns as described in Chapter 3.

AMIC considers clone fragments residing within methods. Before detecting SPCP clone fragments, AMIC detects method genealogies and clone genealogies considering all the revisions of the subject system. Detecting the genealogy for a particular method involves identifying each instance of that method in each of the revisions where the method was alive. By detecting the genealogy of a method, we can determine how it changed during evolution. We detect clone genealogies by locating the clones detected from each revision to the already detected methods of that revision. The genealogy of a particular clone fragment also helps us determine how it evolved through the commits. We assign unique IDs to the method genealogies and clone genealogies to recognize them across revisions. As we detect changes between revisions and reflect these changes to the methods as well as clones, we can examine how two clone fragments from a particular clone

[1]CTAGS: `http://ctags.sourceforge.net/`

class changed during evolution by examining their genealogies. If these two clone fragments always received similarity preserving changes and/or re-synchronizing changes during evolution, then these are considered as a pair of SPCP clone fragments.

We determine all the SPCP clone pairs by examining all the clone genealogies. We merge these pairs to determine SPCP clone groups. If two different SPCP clone pairs have a common SPCP clone fragment, then we can say that the three clone fragments in these two pairs together followed a similarity preserving change pattern and thus, we can merge these pairs to make a group of three SPCP clone fragments. This group can also be merged with another pair or group if they share common SPCP clone fragments.

After detecting the SPCP clone groups, we analyze the evolutionary coupling of each of the SPCP clone fragments in each of these groups. If one or more SPCP clone fragments in a group have cross-boundary evolutionary couplings, then we consider this group for tracking. Otherwise, we consider it for refactoring.

8.5 Implementation and Data Storage

We implement AMIC using Java programming language. We use MySQL as the back-end database server. While examining each revision of a subject system, AMIC extracts the methods and clones in that revision and stores these in the database. The changes between every two consecutive revisions are also stored. For a particular method we store the method name, signature, source file path, starting and ending line numbers, class name (if any), and package name (if any). For each clone fragment we store the file path, starting and ending line numbers, and the clone class ID. After detecting the method and clone genealogies we provide unique IDs to the methods and clones. Here, we should note that we neither store the whole method body nor the actual clone fragment in the files. As we have just described we store enough information for a method or clone fragment so that we can get the corresponding code using the information. Currently AMIC supports four programming languages: C, Java, C#, and Python.

8.6 Output of AMIC

By working on all revisions of a subject system AMIC generates: **(1)** An XML file containing the groups of all SPCP clones, **(2)** An XML file containing the groups of non-cross-boundary SPCP clones, and **(3)** An XML file containing the cross-boundary SPCP clones. The groups of non-cross-boundary SPCP clones are important for refactoring. The cross-boundary SPCP clones are important for tracking.

8.7 Summary

In this chapter we present our automatic clone ranking system, AMIC, which is capable of automatically identifying SPCP clones (i.e., the important clones from the perspectives of clone management) by examining the clone evolution history of a software system. AMIC also analyzes the evolutionary couplings of the SPCP clones and separates these SPCP clones into two disjoint sets on the basis of these evolutionary couplings. The clone fragments in one set are important for refactoring, and the clones in the other set are important for tracking. AMIC also ranks the clones in these two sets on the basis of the necessity of refactoring and tracking. AMIC is the pioneer in detecting important clones for refactoring and tracking. We believe that AMIC can complement the existing clone detection, refactoring, and tracking tools and thus, can help us in better management of code clones. AMIC is available on-line [174] for download. In Appendix B we present a user manual for installing and using AMIC. We have also developed a website [173] for AMIC. The website supports finding and ranking important clones of a given software system.

Chapter 9

Conclusion

Code clones have both positive and negative impacts on the evolution and maintenance of software systems. Focusing on the issues related to code clones software researchers suggest managing code clones through refactoring and tracking. However, a software system may contain a huge number of code clones, and it is impractical to consider all these code clones for refactoring or tracking. Clone refactoring is time consuming, and it often requires interactions from expert programmers. Moreover, code clones might not always be refactorable. We can track code clones where refactoring is impossible. However, clone tracking is resource intensive. In such a situation, it is essential to identify code clones that are important for refactoring or tracking from the huge number of code clones in a software system. Our research focuses on identifying the important clones for refactoring and tracking.

The rest of this chapter is organized as follows. Section 9.1 contains the summary of our five studies presented in the last five chapters, Section 9.2 describes our contribution to the state of the art in clone management, and Section 9.3 discusses future research possibilities in clone refactoring and tracking.

9.1 Summary

In our first study (Chapter 3) we discover and investigate a particular clone change pattern called Similarity Preserving Change Pattern (SPCP) such that code clones that evolve by following this pattern can be considered important for refactoring. We rank SPCP clones (code clones that evolve by following an SPCP) for refactoring by analyzing evolutionary coupling among them. The non-SPCP clones (code clones that do not evolve by following an SPCP) either evolve independently or are rarely changed during system evolution. Thus, non-SPCP clones should not be considered important for management.

Our second study (Chapter 4) focuses on clone tracking. The primary purpose of clone tracking is to make programmers aware about the co-change candidates of clone fragments. When a programmer attempts to make changes to a particular clone fragment in a clone class, the other clone fragments in that class are considered the co-change candidates of that particular clone fragment. These other clone fragments (i.e., these co-change candidates) might need to be changed together (i.e., co-changed) consistently with the particular clone fragment that the programmer is going to change. However, each of these co-change candidates might not be equally important to be co-changed. We mine evolutionary couplings among the clone fragments in a clone class, and analyze these couplings for ranking co-change candidates so that the candidates with high possibilities of getting co-changed are given high ranks.

In our third study (Chapter 5) we we discover the cross-boundary evolutionary couplings (i.e., the evolutionary couplings beyond the class boundary) of the SPCP clones. We analyze such couplings and find that the SPCP clones having such couplings should not be considered for removal through refactoring. Removal of cross-boundary SPCP clones might leave the beyond boundary code fragments that are related to it in an inconsistent state. Thus, cross-boundary SPCP clones should be considered important for tracking. The non-cross-boundary SPCP clones should be considered for refactoring. We ranked the cross-boundary SPCP clones on the basis of the strength of their cross-boundary evolutionary coupling. For the non-cross-boundary SPCP clones we suggest a ranking which is similar to the ranking technique that we used in our first study.

Although in our previous studies we have analyzed the evolutionary coupling of code clones for ranking them for management (refactoring and tracking), we realize that bug-proneness of code clones should also be considered for the purpose of ranking. Code clones with high bug-proneness should be given high priorities for refactoring and tracking. Focusing on this we analyze and compare the bug-proneness of three major types of code clones (Type 1, Type 2, and Type 3) in our fourth study (Chapter 6). We found that Type 3 clones have the highest bug-proneness among the three clone-types. We also found that bug-prone clones of Type 3 have the highest possibility of evolving following a Similarity Preserving Change Pattern. Thus, Type 3 clones should be given the highest priority for management.

In our last study (Chapter 7) we investigated late propagation in code clones. Late propagation is commonly suspected to be the primary cause of bug-proneness in code clones. Studies [29] show that late propagation is directly related to inconsistencies in the code-base. We should also consider late propagation when prioritizing code clones for management. Clone-types with higher tendencies of experiencing late propagations should be given a higher priority for management. We compared the intensities of late propagation in different clone-types, and found that Type 3 clones have the highest intensity. We also found that late propagation in code clones can be significantly minimized by refactoring or tracking the SPCP clones.

9.2 Contribution

Our research towards addressing the research problem of finding the important clones for management contributes to the state of the art in clone management in the following way.

- **Identifying code clones that are important for refactoring** We discover a particular clone change pattern called **Similarity Preserving Change Pattern** (**SPCP**) such that the clone fragments that evolved following this pattern can be considered important for refactoring. Our work has been published in an international software engineering conference **CSMR-WCRE'2014** [163].

- **Ranking co-change candidates of code clones** When a programmer attempts to change a particular clone fragment in a clone class, the other clone fragments in the class might also need to be co-changed

(changed together) consistently with that particular clone fragment. We perform a study on ranking these co-change candidates so that the candidates with high possibilities of getting co-changed get high ranks. Our study has been published in a major software engineering conference **MSR'2014** [167].

- **Identifying code clones that are important for tracking** We analyze the evolutionary coupling of **SPCP clones** (i.e., Code clones that evolved following a particular change pattern called SPCP) and determine whether such clones have cross-boundary relationships (i.e., relationships beyond class boundaries). We consider the cross-boundary SPCP clones to be the important candidates for tracking, and the non-cross-boundary ones to be the important candidates for refactoring. Our study was published in a software engineering conference **SCAM'2014** [162].

- **Comparing bug-proneness of different types of code clones** We compare the bug-proneness of the three major clone-types: Type 1, Type 2, and Type 3 and find that Type 3 clones have the highest possibility of containing bugs. We also find that the bug-fix clones of Type 3 have the highest possibility of evolving following a similarity preserving change pattern (SPCP). We reported that Type 3 clones should be given a higher priority for management compared to the other two clone-types. This work was published in an international software engineering conference **ICSME'2015** [168].

- **Investigating late propagation in different types of code clones** We investigated and compared the intensities of late propagation in the major clone-types: Type 1, Type 2, and Type 3. We also analyzed late propagations in which type of code clones have high possibilities of being related with bugs. We finally investigate whether we can minimize the occurrences of late propagation by managing (refactoring or tracking) SPCP clones. This study was published in the journal called **Software Quality Journal** in January, 2016 [171].

Beside these major contributions (described above) we have a number of related publications [92, 159–161, 164–166, 169, 170, 172] in major software engineering venues from our research.

Automatic support for identifying the important clones for management

On the basis of our studies and findings we develop an automatic system, AMIC (Automatic Mining of Important Clones), for identifying and ranking the important code clones for refactoring and tracking. Given the repository link of a software system, AMIC can download the revisions of the system, detect code clones in each of these revisions by applying the NiCad [48] clone detector, identify the SPCP clones, detect the cross-boundary relationships of the SPCP clones, and rank the cross-boundary and non-cross-boundary SPCP clones on the basis of their evolutionary couplings, bug-proneness, and tendencies of experiencing late propagations. AMIC is available on-line [174]. Appendix B contains a user manual regarding installing and using AMIC for identifying and ranking important clones. We have also developed a website [173] for AMIC. A part of AMIC was published as a tool called SPCP-Miner in the international software engineering conference **SANER'2015** [170].

9.3 Future Research Directions

From our analysis on the existing clone refactoring and tracking research we feel the necessity of further research in the following directions:

Post-refactoring analysis on the effects of clone refactoring on system performance

Analyzing the effect of clone refactoring on system performance is important. Rajapakse and Jarzabek [187] showed that clone refactoring negatively affects the performance of web applications written in PHP and significantly increases the testing effort. Such studies should also be performed considering software systems developed in other programming languages such as: Java, C, and C#. It is also important to investigate whether clone refactoring affects energy consumptions of software systems. A recent study [143] shows that small changes in the code-base can cause significant difference in the energy consumption of a software system. Mahmoud and Niu [148] discovered that removal of code clones through refactoring can negatively affect requirements to code traceability. We believe that clone refactoring should be investigated with a focus on software requirements engineering. The particular types of refactoring that are likely to be harmful for code traceability need to be identified so that software engineers can avoid such types of refactoring.

Increasing language support of the clone refactoring tools

Most of the existing clone refactoring tools support refactoring code clones from only one programming language. However, some of these tools apply clone detectors that can detect clones from multiple languages. For example, we consider the clone refactoring tool DCRA [67] that applies NiCad [48] clone detector for detecting clones. While NiCad [48] supports Java, C, C#, and Python programming languages, DCRA only supports clone refactoring in Java systems. We understand that different programming languages have different constructs, designs, and coding styles. Thus, refactoring patterns should be different for different programming languages. However, the same refactoring pattern might be applicable to multiple languages that support similar coding style. Future research on which refactoring patterns can be commonly applicable to which programming languages, and which are the language specific patterns can be much important for clone management.

Refactoring Type 4 clones

By the definition [192, 203], Type 4 clones (i.e., semantically similar code fragments) do not have syntactic similarity. Thus, the traditional refactoring tools cannot be used for refactoring semantic clones. However, if two code fragments in two places of a code-base are detected as semantic clones, then their refactoring might involve discarding one clone fragment and using the other one in both places possibly through method calls. Deciding which clone fragment to remove and which one to use should depend on the run-time complexity, coding standards, and code comprehensibility of the candidate Type 4 clone fragments. Intuitively, the clone fragment with lower run-time complexity should be more promising compared to the more complex one. We believe that future investigations on automatically comparing the run-time complexity as well as the comprehensibility of two semantically similar code fragments can add much to clone refactoring research.

Inter-project clone refactoring

The existing clone refactoring studies and techniques only deal with intra-project clone refactoring (i.e., refactoring of code clones in the same software system). However, different software systems that are written in the same programming language may have common code fragments. These code fragments are known as inter-project clones. It is important to detect and refactor inter-project code clones. A code fragment (for example, a method) that has been used for implementing more than one software systems should be given importance, because this code code fragment may again be used for implementing another project in future. Thus, such code fragments, that is the inter-project clones, should be managed with equal importance. Refactoring of inter-project clones can be done by developing a global library that will contain these clones and calling appropriate methods in this library in place of the corresponding code clones. Inter-project clone refactoring has not yet been emphasized by software researchers. Future investigations in this area can add much to clone maintenance as well as software maintenance research.

Big-data clone refactoring

Inter-project clone detection and refactoring should be facilitated in a big-data environment empowered by Hadoop-MapReduce framework. Let us consider a particular software company where programmers are working on a number of projects. Also, a number of projects have already been developed in the company. Programmers can get coding help for their on-going projects from these already developed projects through inter-project clone detection and refactoring. Inter-connecting all the already developed as well as on-going projects, parallel detection and refactoring of inter-project code clones from these projects can only be facilitated in a big-data environment. Thus, future research on big-data clone refactoring has much potential to advance the state-of-the-art of clone detection and refactoring.

Increasing language support of the clone tracking tools

Most of the clone tracking tools only support tracking of code clones in Java systems. The tool Simultaneous Editing [152] also supports HTML. However, this tool cannot track code clones through evolution. The tool gCad [210] supports Java, C, and C#. However, it cannot support simultaneous editing, and programmer notification. Future research on enhancing clone trackers so that they can deal with code clones from different programming languages such as: C, C++, C#, and Python can make an important contribution towards clone management.

Clone tracking in a big-data environment

An alternative of automatic tracking of all important code clones in a code-base is instant detection of code clones in a time efficient manner. Let us assume a programmer is working on a piece of code. If we can detect all the duplicate copies of this piece of code instantly, then it might obviate the necessity of clone tracking. Clone tracking requires maintaining a clone database. Also, the evolution of each of the clone fragments need to be tracked through different revisions. For this purpose we need a clone genealogy analyzer. However, instant detection of code clones can possibly eliminate these necessities. In order to facilitate instant clone detection, we possibly need a big-data environment empowered by Hadoop-MapReduce framework. In the parallel computing environment of Hadoop we might be able to detect code clones instantly. Investigations in this direction can be much important.

Comparing the benefits of clone tracking and refactoring

The clone fragments in a particular clone class can be refactored or tracked. Refactoring removes all instances of the clone fragments by a single instance, whereas tracking does not remove any instance of clone fragments but ensures consistent updates of the fragments. Refactoring is beneficial because it obviates the necessity of implementing the same change to multiple clone fragments. Refactoring also reduces the size of the code-base. Moreover, refactoring of a clone class might not always be possible, however, tracking of the class is always possible. In such a situation it is important to perform a trade-off analysis of the benefits gained from refactoring and tracking. We should perform this trade-off analysis in a way that is similar to the investigation of Rajapakse et al. [187]. We should have two copies of the same software system. We should refactor a number of clone classes in one copy, and those clone fragments in the other copy should tracked for a certain period of evolution. Then we should analyze the evolution history considering the following points: (1) time and effort required for refactoring, (2) time and effort for updating code fragments after refactoring, (3) time and effort for updating clone fragments under tracking, (4) system performance after refactoring, (5) system performance while tracking. We believe that investigations on comparing refactoring and tracking benefits through evolution analysis can be much important for efficient software maintenance.

References

[1] Effect size interpretation. https://en.wikipedia.org/wiki/Effect_size.

[2] Exuberant CTAGS. https://sourceforge.net/projects/ctags/.

[3] Mann-Whitney-Wilcoxon Test. https://en.wikipedia.org/wiki/Mann

[4] Mann-Whitney-Wilcoxon Test Online. http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx.

[5] SourceForge. https://sourceforge.net/.

[6] S. Abd-El-Hafiz. A metrics-based data mining approach for software clone detection. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC'12)*, pages 35 – 41, 2012.

[7] R. Agrawal, T. Imieliski, and A. Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD International Conference on Management of Data (ACM SIGMOD'93)*, 22(2):207 – 216, 1993.

[8] S. N. Ahsan and F. Wotawa. Fault prediction capability of program file's logical-coupling metrics. In *Proceedings of 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Software Measurement (IWSM-MENSURA'11)*, pages 257 – 262, 2011.

[9] F. Al-omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'12)*, pages 405 – 414, 2012.

[10] M.H. Alalfi, E.P. Antony, and J.R. Cordy. An approach to clone detection in sequence diagrams and its application to security analysis. *Software and Systems Modelling*, pages 1–35, 2016.

[11] M.H. Alalfi, J.R. Cordy, and T. Dean. Analysis and clustering of model clones: An automotive industrial experience. In *Proceedings of the IEEE CSMR-WCRE Software Evolution Week (CSMR-WCRE'14)*, pages 375 – 378, 2014.

[12] M.H. Alalfi, J.R. Cordy, T.R. Dean, M. Stephan, and A. Stevenson. Models are code too: Near-miss clone detection for simulink models. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM'12)*, pages 295 – 304, 2012.

[13] M.H. Alalfi, J.R. Cordy, T.R. Dean, M. Stephan, and A. Stevenson. Near-miss model clone detection for simulink models. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 78 – 79, 2012.

[14] A. Alali, B. Bartman, C. D. Newman, and J. I. Maletic. A preliminary investigation of using age and distance measures in the detection of evolutionary couplings. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 169 – 172, 2013.

[15] N. Ali, F. Jaafar, and A.E. Hassan. Leveraging historical co-change information for requirements traceability. In *Proceedings of the 2013 Working Conference on Reverse Engineering (WCRE'13)*, pages 361 – 370, 2013.

[16] M. Asaduzzaman. Visualization and analysis of software clones. *MSc Thesis, University of Saskatchewan*, pages 1 – 106, 2011.

[17] M. Asaduzzaman, C. K. Roy, and K. A. Schneider. VisCad: Flexible code clone analysis support for NiCad. In *Proceedings of the Tool Demo Track of the ICSE 5th International Workshop on Software Clones (IWSC'11)*, pages 77 – 78, 2011.

[18] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou. CSCC: Simple, efficient, context sensitive code completion. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 71 – 80. IEEE, 2014.

[19] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 81 – 90, 2007.

[20] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49 – 57, 1992.

[21] B. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95)*, pages 86 – 95, 1995.

[22] B. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28 – 42, 1996.

[23] B. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATEC'98)*, pages 15 – 15, 1998.

[24] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM'07)*, pages 24 – 33, 2007.

[25] M. Balazinska, E. Merlo, M. Dagenais, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 98 – 107, 2000.

[26] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the 6th International Symposium on Software Metrics (METRICS'99)*, pages 292 – 303, 1999.

[27] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 326 – 336, 1999.

[28] F. Bantelay, M. B. Zanjani, and H. Kagdi. Comparing and combining evolutionary couplings from interactions and commits. In *Proceedings of the 2013 Working Conference on Reverse Engineering (WCRE'13)*, pages 311 – 320, 2013.

[29] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 273 – 282, 2011.

[30] L. Barbour, F. Khomh, and Y. Zou. An empirical study of faults in late propagation clone genealogies. *Journal of Software: Evolution and Process*, 25(11):1139 – 1165, 2013.

[31] H. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Software Engineering Notes*, 30: 156 – 165, 2005.

[32] H. Basit and S. Jarzabek. *Towards Structural Clones: Analysis and semi-automated detection of design-level similarities in software.* 2010.

[33] H. Basit, S. Puglisi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the Joint Meeting on European software engineering conference (ESEC) and the ACM SIGSOFT symposium on the foundations of software engineering (FSE): companion*, pages 513 – 516, 2007.

[34] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. An empirical study on the developers' perception of software coupling. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 692 – 701, 2013.

[35] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–378, 1998.

[36] S. Bazrafshan. Evolution of near-miss clones. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, pages 74 – 83, 2012.

[37] Y. Bian, X. Su, and P. Ma. Identifying accurate refactoring opportunities using metrics. In *Proceedings of the International Conference on Soft Computing Techniques and Engineering Application (ICSCTEA'14)*, pages 141 – 146, 2014.

[38] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, pages 1885 – 1892, 2006.

[39] C. Brown and S. Thompson. Clone detection and elimination for Haskell. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, pages 111 – 120, 2010.

[40] D. Cai and M. Kim. An empirical study of long-lived code clones. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: part of the joint European Conferences on Theory and Practice of Software (FASE'11/ETAPS'11)*, pages 432 – 449, 2011.

[41] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'10)*, pages 1 – 10, 2010.

[42] G. Canfora, L. Cerulo, and M. D. Penta. On the use of line co-change for identifying crosscutting concern code. In *Proceeding of the International Conference on Software Maintenance (ICSM'06)*, pages 213 – 222, 2006.

[43] M. Ceccarelli, L. Cerulo, G. Canfora, and M. D. Penta. An eclectic approach for change impact analysis. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*, pages 163 – 166, 2010.

[44] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement (ESEM'11)*, pages 20 – 29, 2011.

[45] E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored? : A case study of three OSS projects. In *Proceedings of the 5th Workshop on Refactoring Tools (WRT'12)*, pages 1 – 7, 2012.

[46] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11)*, pages 7 – 13, 2011.

[47] J. R. Cordy and C. K. Roy. Debcheck: Efficient checking for open source clones in software systems. In *Proceedings of the Tool Demo Track of the 19th International Conference on Program Comprehension (ICPC'11)*, pages 217 – 218, 2011.

[48] J. R. Cordy and C. K. Roy. The NiCad clone detector. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC' 11)*, pages 219 – 220, 2011.

[49] J. R. Cordy and C. K. Roy. Tuning research tools for scalability and performance: The NICAD experience. *Science of Computer Programming Journal*, 79(1):158 – 171, 2012.

[50] A. Cuomo, A. Santone, and U. Villano. A novel approach based on formal methods for clone detection. In *Proceedings of the International Workshop on Software Clones (IWSC'12)*, pages 8 – 14, 2012.

[51] N. Göde D. Steidl. Feature-based detection of bugs in clones. In *Proceedings of the 7th International Workshop on Software Clones (IWSC'13)*, pages 76 – 82, 2013.

[52] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 189 – 198, 2006.

[53] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5): 720 – 735, 2009.

[54] I. Davis and M. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the International Workshop on Software Clones (IWSC'10)*, pages 77 – 78, 2010.

[55] I. Davis and M. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'10)*, pages 242 – 246, 2010.

[56] M. Deepika and S. Sarala. Implication of clone detection and refactoring techniques using delayed duplicate detection refactoring. *International Journal of Computer Applications*, 93(6): 5 – 10, 2014.

[57] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proceedings of the International Workshop on Software Clones (IWSC'10)*, pages 57 – 64, 2010.

[58] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 603 – 612, 2008.

[59] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 166 – 177, 2000.

[60] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 158 – 167, 2007.

[61] E. Duala-Ekoko and M. P. Robillard. CloneTracker: Tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 843 – 846, 2008.

[62] E. Duala-Ekoko and M. P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20(1):1 – 31, 2010.

[63] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 109 – 118, 1999.

[64] R. Elva and G. Leavens. Semantic clone detection using method IOE behavior. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, 2012.

[65] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'07)*, pages 150 – 159, 2007.

[66] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering Journal*, 13:601 – 643, 2008.

[67] F. A. Fontana, M. Zanoni, and F. Zanoni. A duplicated code refactoring advisor. *Agile Processes, in Software Engineering, and Extreme Programming*, LNBIP(212): 3 – 14, 2015.

[68] Martin Fowler. Refactoring techniques. http://www.refactoring.com/catalog/.

[69] C. O. Fritz, P. E. Morris, and J. J. Richler. Effect size estimates: Current use, calculations, and interpretation. *Journal of Experimental Psychology: General*, 141(1): 2 – 18, 2012.

[70] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 321 – 330, 2008.

[71] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 190 – 198, 1998.

[72] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 13 – 23, 2003.

[73] P. Geesaman, J.R. Cordy, and A. Zouaq. Ontology alignment using best-match clone detection. In *Proceedings of the 7th International Workshop on Software Clones (IWSC'13)*, pages 1–7, 2013.

[74] N. Göde. Clone removal: Fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*, pages 33 – 40, 2010.

[75] N. Göde and J. Harder. Clone stability. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, pages 65 – 74, 2011.

[76] N. Göde and R. Koschke. Incremental clone detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 219 – 228, 2009.

[77] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, 25(2):165 – 192, 2013.

[78] N. Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 311 – 320, 2011.

[79] N. Hanakawa. Visualization for software evolution based on logical coupling and module coupling. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC'07)*, pages 214 – 221, 2007.

[80] J. Harder and N. Göde. Efficiently handling clone data: RCF and cyclone. In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11)*, pages 81 – 82, 2011.

[81] B. Hauptmann, E. Juergens, and V. Woinke. Generating refactoring proposals to remove clones from automated system tests. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC'15)*, pages 115 – 124, 2015.

[82] Y. Higo, K. Hotta, and S. Kusumoto. Enhancement of CRD-based clone tracking. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE'13)*, pages 28 – 37, 2013.

[83] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. *Product Focused Software Process Improvement*, (LNCS 3009):220 – 233, 2004.

[84] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: refactoring support tool for code clone. In *Proceedings of the 3rd Workshop on Software Quality (3-WoSQ'05)*, pages 1 – 4, 2005.

[85] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'09)*, pages 315 – 316, 2009.

[86] Y. Higo and S. Kusumoto. Identifying clone removal opportunities based on co-evolution analysis. In *Proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE'13)*, pages 63 – 67, 2013.

[87] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, (20): 435 – 461, 2008.

[88] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'11)*, pages 3 – 12, 2011.

[89] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE'10)*, pages 73 – 82, 2010.

[90] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *Proceedings of the International Conference on Program Comprehension (ICPC'09)*, pages 238 – 242, 2009.

[91] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 94 – 95, 2012.

[92] J. F. Islam, M. Mondal, and C. K. Roy. Bug replication in code clones: An empirical study. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 68 – 78, 2016.

[93] J. Itkonen, M. Hillebrand, and V. Lappalainen. Application of relation analysis to a small java software. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 233 – 239, 2004.

[94] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol. An exploratory study of macro co-changes. In *Proceedings of the 2011 Working Conference on Reverse Engineering (WCRE'11)*, pages 325 – 334, 2011.

[95] P. Jablonski and D. Hou. CReN: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology Exchange (OOPSLA'07)*, pages 16 – 20, 2007.

[96] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 96 – 105, 2007.

[97] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'07)*, pages 55 – 64, 2007.

[98] J. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering (CASCON'93)*, pages 171 – 183, 1993.

[99] J. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM'94)*, pages 120 – 126, 1994.

[100] J.R.Cordy. SIMONE: Architecture-sensitive near-miss clone detection for simulink models. In *Proceedings of the 1st International Workshop on Automotive Software Architecture (WASA'15)*, pages 1–2, 2015.

[101] E. Juergens. Research in cloning beyond code: a first roadmap. In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11)*, pages 67 – 68, 2011.

[102] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 79 – 88, 2010.

[103] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective - a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 603 – 606, 2009.

[104] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 485 – 495, 2009.

[105] N. Juillerat and B. Hirsbrunner. An algorithm for detecting and removing clones in Java code. *Electronic Communications OF EASST*, (3):1 – 12, 2006.

[106] H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933 – 969, 2013.

[107] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Proceedings of the 17th IEEE Working Conference on Reverse Engineering (WCRE'10)*, pages 119 – 128, 2010.

[108] T. Kamiya. Agec: An execution-semantic clone detection tool. In *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, pages 227 – 229, 2013.

[109] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654 – 670, 2002.

[110] C. Kapser. Toward an understanding of software code cloning as a development practice. *PhD thesis, University of Waterloo, Canada*, 2009.

[111] C. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering Journal*, 13(6): 645 – 692, 2008.

[112] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: a tool for automatic code clone detection in the IDE. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, pages 313 – 314, 2009.

[113] I. Keivanloo, C. K. Roy, and Y. Chen. Near-miss software clones in open source games: An empirical study. In *Proceedings of the 27th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'14)*, pages 1 – 7, 2014.

[114] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 36 – 42. IEEE Press, 2012.

[115] I. Keivanloo, C. K. Roy, and J. Rilling. SeByte: A semantic clone detection tool for intermediate languages. In *Proceedings of the IEEE 20th International Conference on Program Comprehension (ICPC'12)*, pages 247 – 249, 2012.

[116] I. Keivanloo, C. K. Roy, and J. Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Science of Computer Programming*, 95:426 – 444, 2014.

[117] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland. Shuffling and randomization for scalable source code clone detection. In *Proceedings of the ICSE 6th International Workshop on Software Clones (IWSC'12)*, pages 83 – 84, 2012.

[118] M. Khan, C. K. Roy, and K. Schneider. Active clones: Source code clones at runtime. *Electronic Communications of the EASST*, 63: 1 - 19, 2014.

[119] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'13)*, pages 187 – 196, 2005.

[120] G. Kitnab, G. McCalla, and C. K. Roy. Recommending software experts using code similarity and social heuristics. In *Proceedings of the 2014 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'14)*, pages 4 – 18, 2014.

[121] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*, pages 40 – 56, 2001.

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

[123] O. Kononenko, C. Zhang, and M. W. Godfrey. Compiling clones: What happens? In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 481 – 485, 2014.

[124] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'95)*, pages 96 – 103, 1995.

[125] K. Kontogiannis, R. Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1/2):77 – 108, 1996.

[126] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'12)*, pages 309 – 318, 2012.

[127] R. Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747 – 769, 2014.

[128] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'06)*, pages 253 – 262, 2006.

[129] S. Kotsiantis and D. Kanellopoulos. Association rules mining: A recent overview. *GESTS International Transactions on Computer Science and Engineering*, 32(1): 71 – 82, 2006.

[130] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'01)*, pages 301 – 309, 2001.

[131] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*, pages 170 – 178, 2007.

[132] J. Krinke. Is cloned code more stable than non-cloned code? In *Proceedings of th 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 57 – 66, 2008.

[133] J. Krinke. Is cloned code older than non-cloned code? In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11)*, pages 28 – 33, 2011.

[134] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Proceedings of IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'14), Software Evolution Week*, pages 104 – 113, 2014.

[135] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance (ICSM'97)*, pages 314 – 321, 1997.

[136] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'03)*, pages 379 – 386, 2003.

[137] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou. Inferring repository file structure modifications using nearest-neighbor clone detection. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*, pages 325 – 334, 2012.

[138] S. Lee, G. Bae, H. S. Chae, D. Bae, and Y. R. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Software – Practice and Experience*, 41:521 – 550, 2011.

[139] H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, pages 169 – 177, 2009.

[140] H. Li and S. Thompson. Similar code detection and elimination for erlang programs. *Practical Aspects of Declarative Languages*, 5937:104 – 118, 2010.

[141] J. Li and M. D. Ernst. CBCD: cloned buggy code detector. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, pages 310 – 320, 2012.

[142] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference and Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 20 – 33, 2004.

[143] L. G. Lima, G. Melfe, F. Soares-Neto, P. Lieuthier, J. P. Fernandes, and F. Castor. Haskell in green land: Analyzing the energy behavior of a purely functional language. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 517 – 528, 2016.

[144] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proceedings of the 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pages 269 – 276, 2006.

[145] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'08)*, pages 227 – 236, 2008.

[146] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*, pages 65 – 72, 2010.

[147] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC'02)*, pages 481 – 486, 2002.

[148] A. Mahmoud and N. Niu. Supporting requirements to code traceability through refactoring. *Requirements Engineering*, 19 (2014):309 – 329, 2013.

[149] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 107 – 114, 2001.

[150] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 244 – 253, 1996.

[151] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 392 – 402, 2015.

[152] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference (USENIX'01)*, pages 161 – 174, 2001.

[153] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 120 – 130, 2000.

[154] M. Mondal. On the stability of software clones: A genealogy-based empirical study,. *MSc. Thesis, University of Saskatchewan, Canada*, pages 1 – 152, 2013.

[155] M. Mondal, M. S. Rahman, R. K. Saha, C. K. Roy, J. Krinke, and K. A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*, pages 242–245. IEEE, 2011.

[156] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, pages 1227 – 1234, 2012.

[157] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of co-changed method groups: A case study on open source systems. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'12)*, pages 205 – 219, 2012.

[158] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on clone stability. *ACM SIGAPP Applied Computing Review*, 12(3): 20 – 36, 2012.

[159] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling. In *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC'13)*, pages 223 – 226, 2013.

[160] M. Mondal, C. K. Roy, and K. A. Schneider. Insight into a method co-change pattern to identify highly coupled methods: An empirical study. In *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC'13)*, pages 103 – 112, 2013.

[161] M. Mondal, C. K. Roy, and K. A. Schneider. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. *Science of Computer Programming Journal*, 95(4):445 – 468, 2013.

[162] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic identification of important clones for refactoring and tracking. In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM'14)*, pages 11 – 20, 2014.

[163] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'14), Software Evolution Week*, pages 114 – 123, 2014.

[164] M. Mondal, C. K. Roy, and K. A. Schneider. A fine-grained analysis on the evolutionary coupling of cloned code. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 51 – 60, 2014.

[165] M. Mondal, C. K. Roy, and K. A. Schneider. Improving the detection accuracy of evolutionary coupling by measuring change correspondence. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'14), Software Evolution Week*, pages 358 – 362, 2014.

[166] M. Mondal, C. K. Roy, and K. A. Schneider. Late propagation in near-miss clones: An empirical study. In *Proceedings of the 8th International Workshop on Software Clones (IWSC'14)*, pages 1 – 15, 2014.

[167] M. Mondal, C. K. Roy, and K. A. Schneider. Prediction and ranking of co-change candidates for clones. In *Proceedings of the 11th Working Conference on Mining Software (MSR'14)*, pages 32 – 41, 2014.

[168] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the bug-proneness of different types of code clones. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, pages 91 – 100, 2015.

[169] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on change recommendation. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering (CASCON'15)*, pages 141 – 150, 2015.

[170] M. Mondal, C. K. Roy, and K. A. Schneider. SPCP-Miner: A tool for mining code clones that are important for refactoring or tracking. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, pages 482 – 486, 2015.

[171] M. Mondal, C. K. Roy, and K. A. Schneider. A comparative study on the intensity and harmfulness of late propagation in near-miss code clones. *Software Quality Journal*, 24(4):883 – 915, 2016.

[172] M. Mondal, C. K. Roy, and K. A. Schneider. An empirical study on ranking change recommendations retrieved using code similarity. In *Proceedings of the 10th International Workshop on Software Clones (IWSC'16)*, pages 44 – 50, 2016.

[173] Manishankar Mondal. AMIC: automatic mining of important clones. http://sr-p2irc-big2.usask.ca/amic/index.php.

[174] Manishankar Mondal. Automatic mining of important clones. https://homepage.usask.ca/m̃am815/amic/AMIC.zip.

[175] Manishankar Mondal. Mining association rules among clones. https://homepage.usask.ca/m̃am815/tools.php.

[176] T. Muhammad, M. F. Zibran, Y. Yamamoto, and C. K. Roy. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proceedings of the 2013 Canadian Conference on Electrical and Computer Engineering (CCECE 2013)*, pages 1 – 6, 2013.

[177] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, pages 93 – 102, 2013.

[178] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Transactions on Software Engineering*, 38(5):1008 – 1026, 2011.

[179] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 491 – 494, 2009.

[180] G. A. Oliva and M. A. Gerosa. On the interplay between structural and logical dependencies in open-source software. In *Proceedings of the 25th Brazilian Symposium on Software Engineering (SBES'11)*, pages 144 – 153, 2011.

[181] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 276 – 286, 2009.

[182] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the International Conference on Software Maintenance (ICSM'06)*, pages 469 – 478, 2006.

[183] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, pages 72 – 81, 2010.

[184] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a context-aware IDE-based meta search engine for recommendation about programming errors and exceptions. In *Proceedings of the 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'14)*, pages 194 – 203. IEEE, 2014.

[185] M. S. Rahman, A. Aryani, C. K. Roy, and F. Perin. On the relationships between domain-based coupling and code clones: An exploratory study. In *Proceedings of the New Ideas and Emerging Results Track of the 35th International Conference on Software Engineering (ICSE 2013),*, pages 1265 – 1268, 2013.

[186] M. S. Rahman and C. K. Roy. A change-type based empirical study on the stability of cloned code. In *Proceedings of the 14th IEEE International Working Conference on Software Code Analysis and Manipulation (SCAM'14)*, pages 31 – 40, 2014.

[187] D. C. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 116 – 126, 2007.

[188] E.J. Rapos, A. Stevenson, M. Alalfi, and J.R. Cordy. Simnav: Simulink navigation of model clone classes. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, pages 241 – 246, 2015.

[189] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology Journal*, 55(7): 1165 – 1199, 2013.

[190] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In *Proceedings of the 2008 Working Conference on Reverse Engineering (WCRE'08)*, pages 42 – 46, 2008.

[191] T. Rolfsnes, S. D. Alesio, R. Behjati, L. Moonen, and D. W. Binkley. Generalizing the analysis of evolutionary coupling for software change impact analysis. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, pages 201 – 212, 2016.

[192] C. K. Roy. Detection and analysis of near-miss software clones. In *Proceedings of the Doctoral Symposium Track of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 447 – 450, 2009.

[193] C. K. Roy. Detection and analysis of near-miss software clones. *Ph.D. Thesis, Queen's University, Canada*, pages 1 – 247, 2009.

[194] C. K. Roy and J. R. Cordy. A survey on software clone detection research. In *Tech Report TR 2007-541, School of Computing, Queens University, Canada*, pages 1 – 115, 2007.

[195] C. K. Roy and J. R. Cordy. An empirical evaluation of function clones in open source software. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 81–90, 2008.

[196] C. K. Roy and J. R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 172 – 181, 2008.

[197] C. K. Roy and J. R. Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings of the 2008 C3S2E Conference (C3S2E'08)*, pages 137 – 140. ACM, 2008.

[198] C. K. Roy and J. R. Cordy. A mutation / injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (Mutation'09)*, pages 157 – 166, 2009.

[199] C. K. Roy and J. R. Cordy. Are scripting languages really different? In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*, pages 17 – 24. ACM, 2010.

[200] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *Journal of Software: Evolution and Process*, 22(3):165 – 189, 2010.

[201] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming Journal*, 74 (2009): 470 – 495, 2009.

[202] C. K. Roy, M. G. Uddin, B. Roy, and T. R. Dean. Evaluating aspect mining techniques: A case study. In *Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 167 – 176. IEEE, 2007.

[203] C. K. Roy, M. F. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE'14), Software Evolution Week*, pages 18 – 33, 2014.

[204] C.K. Roy and J.R. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pages 153 – 162, 2008.

[205] V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 336 – 339, 2004.

[206] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the 18th international symposium on Software testing and analysis (ISSTA'09)*, pages 117 – 128, 2009.

[207] R. Saha. Detection and analysis of near-miss clone genealogies. *M.Sc. thesis, University of Saskatchewan*, 2011.

[208] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, pages 87 – 96, 2010.

[209] R. K. Saha, C. K. Roy, and K. A. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 293 – 302, 2011.

[210] R. K. Saha, C. K. Roy, and K. A. Schneider. gCad: A near-miss clone genealogy extractor to support clone evolution analysis. In *Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM'13)*, pages 488 – 491, 2013.

[211] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the evolution of Type-3 clones: an exploratory study. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR'13)*, pages 139 – 148. IEEE, 2013.

[212] R. K. Saha, C. K. Roy, and K.A. Schneider. Visualizing the evolution of code clones. In *Proceedings of ICSE 5th International Workshop on Software Clones (IWSC'11)*, pages 71 – 72, 2011.

[213] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale clone detection. In *Proceedings of the International Workshop on Software Clones (IWSC'13)*, pages 46 – 52, 2013.

[214] H. Sajnani, J. Ossher, and C. Lopes. Parallel code clone detection using mapreduce. In *Proceedings of the International Conference on Program Comprehension (ICPC'12)*, pages 261 – 262, 2012.

[215] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and Cristina V. Lopes. SourcererCC: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 1157 – 1168, 2016.

[216] A. Santone. Clone detection through process algebras and java bytecode. In *Proceedings of the International Workshop on Software Clones (IWSC'11)*, pages 73 – 74, 2011.

[217] S. Sarala and M. Deepika. Unifying clone analysis and refactoring activity advancement towards C# applications. In *Proceedings of the 4th International Conference on Computing, Communications and Networking Technologies (ICCCNT'13)*, pages 1 – 5, 2013.

[218] S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *Proceedings of the Workshop of GI-Fachgruppe Software Reengineering (SRE)*, pages 1 – 2, 2009.

[219] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *Proceedings of the 2nd Workshop on Refactoring Tools (WRT'08)*, pages 1 – 4, 2008.

[220] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *17th Working Conference on Reverse Engineering (WCRE'10)*, pages 13 – 21, 2010.

[221] M. J. Shepperd and S. G. MacDonell. Evaluating prediction systems in software project estimation. *Information & Software Technology*, 54(8): 820 – 827, 2012.

[222] M. Stephan, M.H. Alalfi, J.R. Cordy, and A. Stevenson. Evolution of model clones in simulink. In *Proceedings of Models and Evolution Workshop (MODELS'13)*, pages 38 – 47, 2013.

[223] M. Stephan, M.H. Alalfi, A. Stevenson, and J.R. Cordy. Towards qualitative comparison of simulink model clone detection approaches. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 84 – 85, 2012.

[224] M. Stephan, M.H. Alalfi, A. Stevenson, and J.R. Cordy. Using mutation analysis for a model clone detector comparison framework. In *Proceedings of the New Ideas and Emerging Results track of the International Conference on Software Engineering (ICSE'13)*, pages 1261 – 1264, 2013.

[225] M. Stephan and J.R. Cordy. Identification of simulink model antipattern instances using model clone detection. In *Proceedings of the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 276 – 285, 2015.

[226] M. Stephan and J.R. Cordy. Identifying instances of model design patterns and antipatterns using model clone detection. In *Proceedings of the 7th International Workshop on Modelling in Software Engineering (MiSE'15)*, pages 48 – 53, 2015.

[227] M. Stephan and J.R. Cordy. Model-driven evaluation of software architecture quality using model clone detection. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'16)*, pages 92 – 99, 2016.

[228] H. Störrle. Towards clone detection in UML domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (ECSA'10)*, pages 285 – 293, 2010.

[229] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 476 – 480. IEEE, 2014.

[230] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using the classical detectors: An exploratory study. *Journal of Software Evolution and Process*, 27(6):430 – 464, 2015.

[231] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 321 – 330, 2014.

[232] J. Svajlenko and C. K. Roy. Evaluating clone detection tools with bigclonebench. In *Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME'15)*, pages 131 – 140, 2015.

[233] J. Svajlenko, C. K. Roy, and J. R. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the 7th International Workshop on Software Clones (IWSC'13)*, pages 8 – 9. IEEE, 2013.

[234] J. Svajlenko, C. K. Roy, and S. Duszynski. Forksim: Generating software forks for evaluating cross-project similarity analysis tools. In *Proceedings of the Tool Paper track of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*, pages 37 – 42, 2013.

[235] J. T. Svajlenko, I. Keivanloo, and C. K. Roy. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Proceedings of the ICSE 7th International Workshop on Software Clones (IWSC'13)*, pages 16 – 22, 2013.

[236] R. Tairas. Clone detection and refactoring. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pages 780 – 781, 2006.

[237] R. Tairas. Clone maintenance through analysis and refactoring. In *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium (FSEDS'08)*, pages 29 – 32, 2008.

[238] R. Tairas and J. Gray. Sub-clone refactoring in open source software artifacts. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*, pages 2373 – 2374, 2010.

[239] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Journal of Information and Software Technology*, 54 (2012): 1297 – 1307, 2012.

[240] M. Thomsen and F. Henglein. Clone detection using rolling hashing, suffix trees and dagification: A case study. In *Proceedings of the International Workshop on Software Clones (IWSC'12)*, pages 22 – 28, 2012.

[241] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering Journal*, 15(1): 1 – 34, 2009.

[242] M. Tokunaga, N. Yoshida, K. Yoshioka, M. Matsushita, and K. Inoue. Towards a collection of refactoring patterns based on code clone categorization. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs (AsianPLoP'11)*, pages 1 – 6, 2011.

[243] W. Toomey. CTcompare: Code clone detection using hashed token sequences. In *Proceedings of the International Workshop on Software Clones (IWSC'12)*, pages 92 – 93, 2012.

[244] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 173 – 180, 2004.

[245] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11): 1055 – 1090, 2015.

[246] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting nearmiss clones in large scale software systems. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE'11)*, pages 13 – 22, 2011.

[247] M. S. Uddin, V. Gaur, C. Gutwin, and C. K. Roy. On the comprehension of code clone visualizations: A controlled study using eye tracking. In *Proceedings of the 15th IEEE International Working Conference on Software Code Analysis and Manipulation (SCAM'15)*, pages 161–170, 2015.

[248] S. Uddin, C. K. Roy, and K. Schneider. Towards convenient management of software clone codes in practice: An integrated approach. In *Proceedings of the 2015 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'14)*, pages 211 – 220, 2014.

[249] S. Uddin, C. K. Roy, and K. A. Schneider. SimCad : An extensible and faster clone detection tool for large scale software systems. In *Proceedings of the IEEE 21st International Conference on Program Comprehension (ICPC'13)*, pages 236 – 238, 2013.

[250] A. Vanya, R. Premraj, and H. V. Vliet. Interactive exploration of co-evolving software entities. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*, pages 260 – 263, 2010.

[251] R. Venkatasubramanyam, S. Gupta, and H. K. Singh. Prioritizing code clone detection results for clone management. In *Proceedings of the 7th International Workshop on Software Clones (IWSC'13)*, pages 30 – 36, 2013.

[252] N. Volanschi. Safe clone-based refactoring through stereotype identification and iso-generation. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 50 – 56, 2012.

[253] V. Wahler, D. Seipel, J. Wol, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM'04)*, pages 128 – 135, 2004.

[254] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 455 – 465, 2013.

[255] W. Wang and M. W. Godfrey. A study of cloning in the linux SCSI drivers. In *Proceedings of the 11th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, pages 95 – 104, 2011.

[256] W. Wang and M. W. Godfrey. We have all of the clones, now what? toward integrating clone analysis into qa. In *Proceedings of the 6th International Workshop on Software Clones (IWSC'12)*, pages 88 – 89, 2012.

[257] W. Wang and M. W. Godfrey. Investigating intentional clone refactoring. *Electronic Communications of the EASST*, 63:1 – 7, 2014.

[258] W. Wang and M. W. Godfrey. Recommending clones for refactoring using design, context, and history. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*, pages 331 – 340, 2014.

[259] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I clone this piece of code here? In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pages 170 – 179, 2012.

[260] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. *Master's thesis, Freie Universitat Berlin, Germany*, pages 1 – 5, 2008.

[261] M. D. Wit, A. Zaidman, and A. V. Deursen. Managing clones using dynamic change tracking and resolution. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance (ICSM'09)*, pages 169 – 178, 2009.

[262] S. Wong and Y. Cai. Generalizing evolutionary coupling with stochastic dependencies. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*, pages 293 – 302, 2011.

[263] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 149 – 158, 2013.

[264] W. Yang. Identifying syntactic differences between two programs. *Software Practice Experience*, 21:739 – 755, 1991.

[265] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574 – 586, 2004.

[266] N. Yoshida, E. Choi, and K. Inoue. Active support for clone refactoring: A perspective. In *Proceedings of the 2013 ACM workshop on Workshop on Refactoring Tools (WRT'13)*, pages 13 – 16, 2013.

[267] S. Yoshioka, N. Yoshida, K. Fushida, and H. Iida. Scalable detection of semantic clones based on two-stage clustering. In *ISSRE*, pages 1–2, 2011.

[268] L. Yu and S. Ramaswamy. Improving modularity by refactoring code clones: A feasibility study on linux. *ACM SIGSOFT Software Engineering Notes*, 33(2): 1 – 5, 2008.

[269] Y. Yuan and Y. Guo. CMCD: Count matrix based code clone detection. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC'11)*, pages 250 – 257, 2011.

[270] Y. Yuan and Y. Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the International Conference on Automated Software Engineering (ASE'12)*, pages 286 – 289, 2012.

[271] M. F. Zibran. Management aspects of software clone detection and analysis. *Ph.D. Thesis, Department of Computer Science, University of Saskatchewan, Canada*, 2014.

[272] M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of code clone refactoring: a constraint programming approach. In *Proceedings of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*, pages 266 – 269. IEEE, 2011.

[273] M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*, pages 105 – 114, 2011.

[274] M. F. Zibran and C. K. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *Proceedings of the 5th International Workshop on Software Clones (IWSC'11)*, pages 75 – 76, 2011.

[275] M. F. Zibran and C. K. Roy. IDE-based real-time focused search for near-miss clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, pages 1235 – 1242. ACM, 2012.

[276] M. F. Zibran and C. K. Roy. The road to software clone management: A survey. *Tech. Report 2012-03, Department of Computer Science, University of Saskatchewan, Canada*, pages 1 – 62, 2012.

[277] M. F. Zibran and C. K. Roy. Conflict-aware optimal scheduling of prioritised code clone refactoring. *IET Software*, 7(3): 167 – 186, 2013.

[278] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'11)*, pages 295 – 304, 2011.

[279] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proceedings of the Software Engineering track of the 28th ACM Symposium On Applied Computing (ACM SAC'13)*, pages 1223 – 1230, 2013.

[280] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Genealogical insights into the facts and fictions of clone removal. *Applied Computing Review*, 13(4): 30 – 42, 2013.

[281] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 563 – 572, 2004.

Appendix A

Publications From This Thesis Research

This section contains the list of publications out of this thesis research. I am the primary author in each of these publications. The research behind each publication was fully conducted by me under the supervision of Dr. Chanchal K. Roy and Dr. Kevin A. Schneider.

Refereed Journal Contributions

- **Manishankar Mondal**, Chanchal K. Roy, and Kevin A. Schneider, "A Comparative Study on the Intensity and Harmfulness of Late Propagation in Near-Miss Code Clones", Software Quality Journal, January 2016, pp. 1 - 33.
- **Manishankar Mondal**, Chanchal K. Roy, and Kevin A. Schneider, "An Insight into the Dispersion of Changes in Cloned and Non-cloned Code: A Genealogy Based Empirical Study", Science of Computer Programming Journal, December 2014, 95(4):445 468.

Refereed Conference and Workshop Contributions

- **Manishankar Mondal**, Chanchal K. Roy, Kevin A. Schneider, "An Exploratory Study on Change Suggestions for Methods Using Clone Detection", in the Proceedings of the 26th Annual International Conference hosted by the Centre for Advanced Studies Research, IBM Canada Software Laboratory (CASCON 2016), Markham, Ontario, Canada, November 2016. ACM
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "An Empirical Study on Ranking Change Recommendations Retrieved using Code Similarity", in the Proceedings of the 10th International Workshop on Software Clones (IWSC 2016), Osaka, Japan, March 2016.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "An Empirical Study on Change Recommendation", in the Proceedings of the 25th Annual International Conference hosted by the Centre for Advanced Studies Research, IBM Canada Software Laboratory (CASCON 2015), Markham, Ontario, Canada, November 2015, pp. 141 - 150. ACM.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "A Comparative Study on the Bug-Proneness of Different Types of Code Clones", in the Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME 2015), Bremen, Germany, October 2015, pp. 91 100. IEEE Computer Society.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "SPCP-Miner: A Tool for Mining Code Clones that are Important for Refactoring or Tracking", in the Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), Montreal, Canada, March 2015, pp. 484 - 488. IEEE.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking", in the Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014), Victoria, Canada, September 2014, pp. 11 - 20. IEEE.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, "A Fine-Grained Analysis on the Evolutionary Coupling of Cloned Code", in the Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), Victoria, Canada, September 2014, pp. 51 - 60. IEEE.

- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, “Prediction and Ranking of Co-change Candidates for Clones”, in the Proceedings of the 11th International Working Conference on Mining Software Repositories (MSR 2014), Hyderabad, India, June 2014, pp. 32 - 41. ACM.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, “Automatic Ranking of Clones for Refactoring through Mining Association Rules”, in the Proceedings of the IEEE International Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 2014), Antwerp, Belgium, February 2014, pp. 114 - 123. IEEE.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, “Improving the Detection Accuracy of Evolutionary Coupling by Measuring Change Correspondence”, in the Proceedings of the IEEE International Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE 2014), Antwerp, Belgium, February 2014, pp. 358 - 362. IEEE.
- **Manishankar Mondal**, Chanchal K. Roy and Kevin Schneider, “Late Propagation in Near-Miss Clones: An Empirical Study”, in the Proceedings of the 8th International Workshop on Software Clones (IWSC 2014), Antwerp, Belgium, February 2014, 17pp.
- **Manishankar Mondal**, Chanchal K. Roy, Kevin A. Schneider. “Insight into a Method Co-change Pattern to Identify Highly Coupled Methods: An Empirical Study”, in the Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013), San Francisco, California, USA, May 2013, pp. 103 - 112. IEEE Computer Society.
- **Manishankar Mondal**, Chanchal K. Roy, and Kevin A. Schneider, “Improving the Detection Accuracy of Evolutionary Coupling”, in the Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013), San Francisco, California, USA, May 2013, pp. 223 - 226. IEEE Computer Society.

Appendix B

User Manual for AMIC

Chapter 8 provides a conceptual description of AMIC (Automatic Mining of Important Clones). In Appendix B, we describe how to install and use AMIC for identifying and ranking the important clones in the code-base of a software system. AMIC can be used in two ways: (1) through its desktop implementation, and (2) through its web-based implementation (AMIC website is now on-line [173]). We will describe these two ways in the following sections.

B.1 Using AMIC through its Desktop Implementation

B.1.1 Installation

We have implemented AMIC using Java programming language. The JAR file for AMIC is available on-line [174]. For executing this JAR file, we need to have the followings pre-installed:

- JDK 1.7.0_71 or later.
- MySQL 5.6.17 or later.
- Subversion (SlikSVN 1.8 or later for Windows platform).

We also need Java and MySQL connector (as a JAR file) version 5.1.15 or later for running AMIC. In order to apply AMIC to a particular software system we need to create a database in MySQL server with certain tables. The table creation script is available on-line [174]. We need to create a separate database for each system we would like to investigate.

B.1.2 Executing AMIC

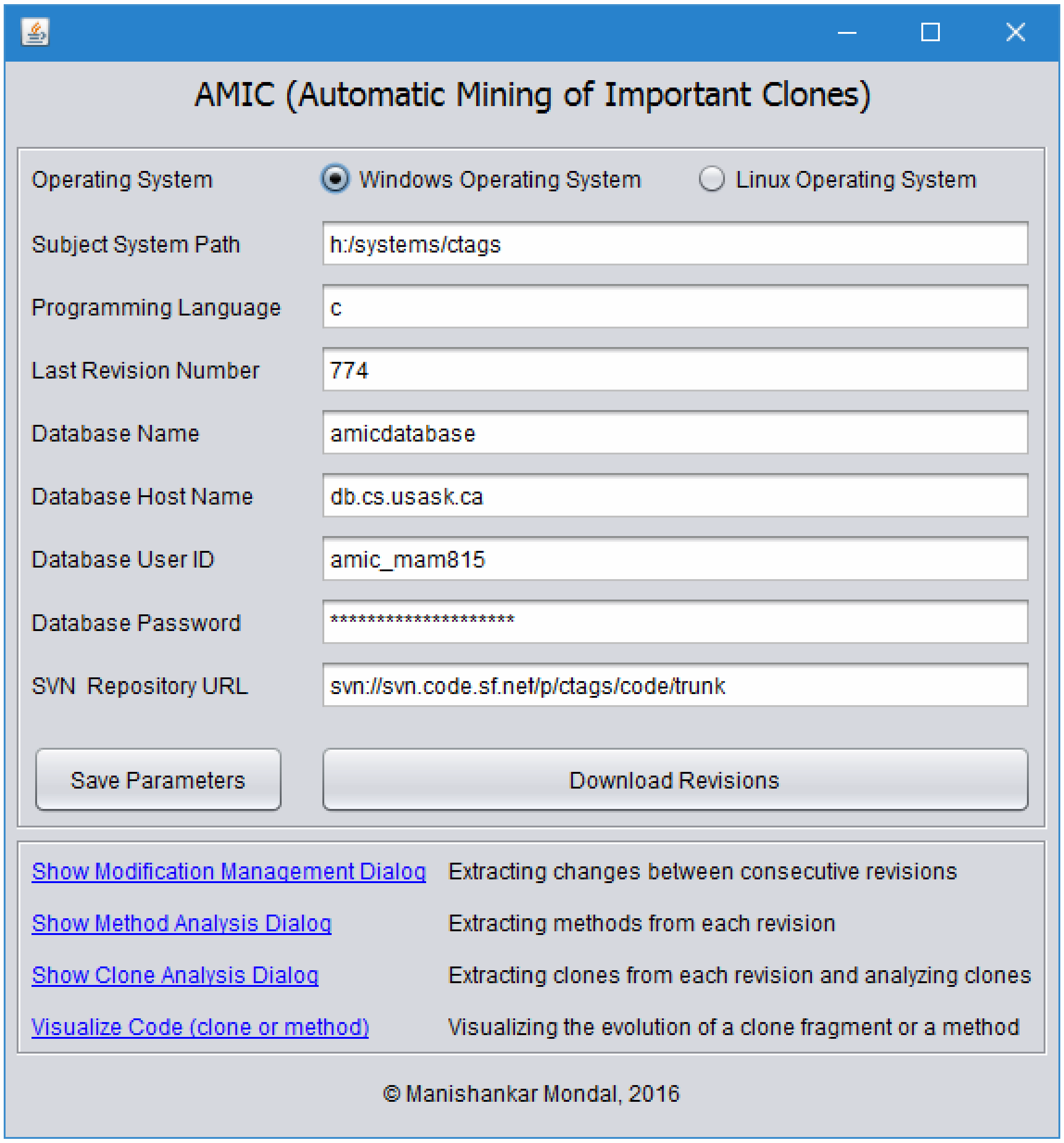
For executing AMIC we can perform the followings sequentially.

Providing the Input Parameters

After double clicking the JAR file of AMIC we get the dialog in Fig. B.1. We then input nine parameters as follows: (1) operating system type, (2) subject system path, (3) programming language, (4) last revision number of the subject system, (5) the name of the database that you have created for your subject system, (6) the host name of the MySQL server, (7) database user id, (8) password for the database, and (9) the SVN repository URL from where AMIC can download the revisions of the subject system. Fig. B.1 shows the input parameters for a subject system called Ctags. After providing all the parameters we click the button named '**Save Parameters**'. Once we have saved the parameters for a particular subject system, we can use those for any execution of AMIC afterwards. The programming language field takes the following values: (1) c for subject systems written in C, (2) java for systems written in Java, (3) cs for systems code in C#, and (4) py for systems developed using Python.

Downloading Revisions of the Subject System

After storing the parameters we need to download the revisions of the subject system for analysis. For this purpose we click the button called '**Download Revisions**'. We then get the dialog as shown in Fig. B.2. This dialog prepopulates the followings: SVN repository link, starting revision number (i.e., from which we need to begin download), and the ending revision number. We then click the '**Download**' button for downloading the revisions. If AMIC finds that the revisions were previously downloaded, it then notifies the user about it as shown in Fig. B.3.



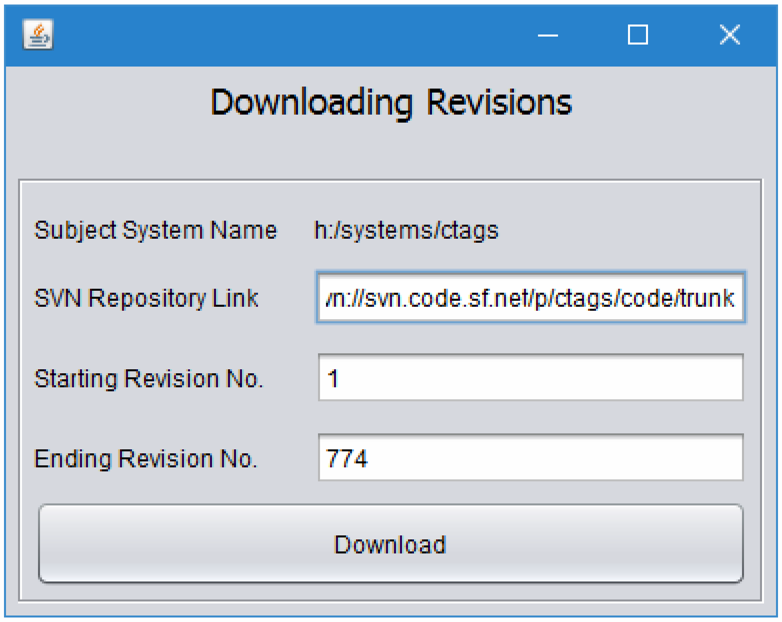
AMIC (Automatic Mining of Important Clones)

Operating System	[✓ Windows Operating System] Linux Operating System
Subject System Path	h:/systems/ctags
Programming Language	c
Last Revision Number	774
Database Name	amicdatabase
Database Host Name	db.cs.usask.ca
Database User ID	amic_mam815
Database Password	********************
SVN Repository URL	svn://svn.code.sf.net/p/ctags/code/trunk

Show Modification Management Dialog	Extracting changes between consecutive revisions
Show Method Analysis Dialog	Extracting methods from each revision
Show Clone Analysis Dialog	Extracting clones from each revision and analyzing clones
Visualize Code (clone or method)	Visualizing the evolution of a clone fragment or a method

© Manishankar Mondal, 2016

Figure B.1: The first dialog of AMIC



Downloading Revisions

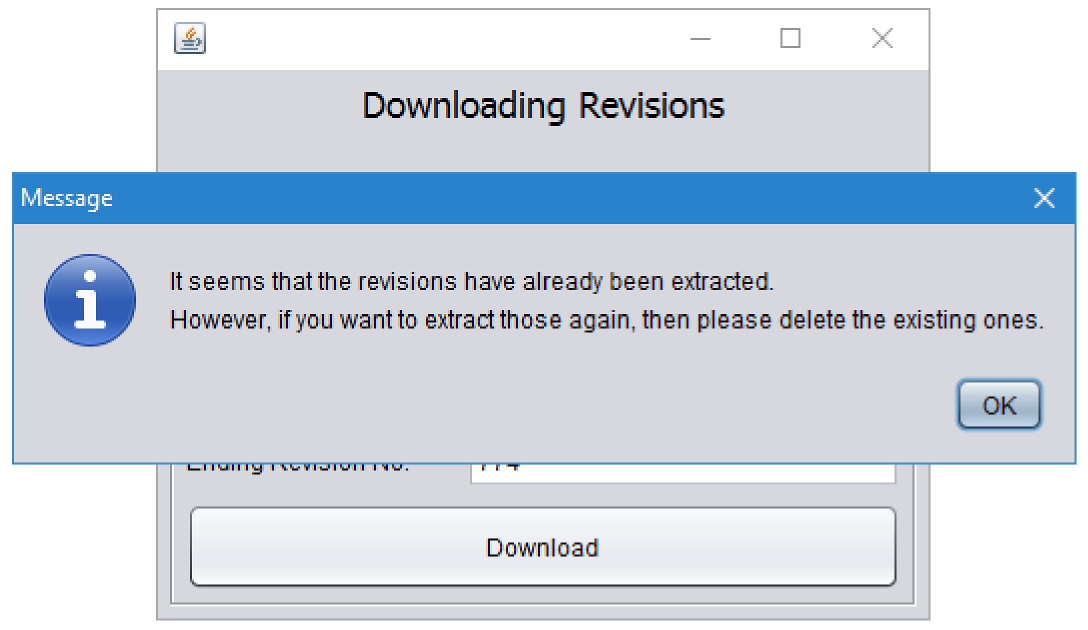
Subject System Name h:/systems/ctags

SVN Repository Link /n://svn.code.sf.net/p/ctags/code/trunk

Starting Revision No. 1

Ending Revision No. 774

Figure B.2: The dialog for downloading revisions of a subject system



Downloading Revisions

Message

It seems that the revisions have already been extracted.
However, if you want to extract those again, then please delete the existing ones.

Figure B.3: Notifying users about the pre-existing revisions of the subject system

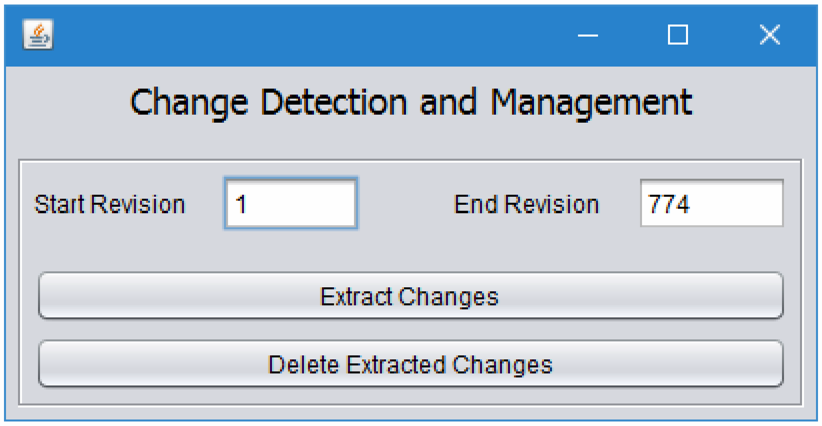


Figure B.4: Extracting changes between revisions of the subject system

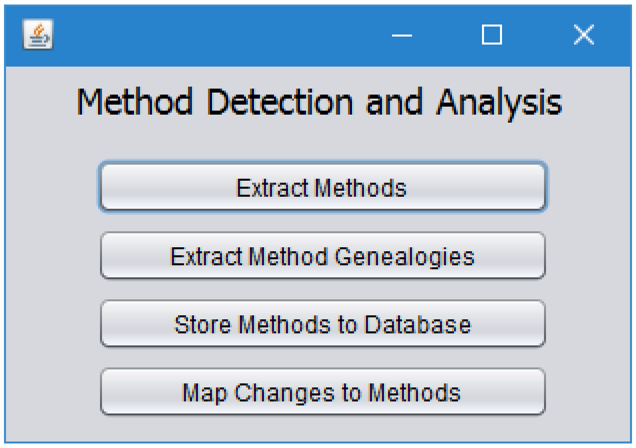

Method Detection and Analysis
Extract Methods
Extract Method Genealogies
Store Methods to Database
Map Changes to Methods

Figure B.5: Extracting and analyzing methods from each revision

Extracting Changes between Consecutive Revisions

After downloading the revisions we extract and store the changes between every two consecutive revisions. For this purpose we click the link called 'Show Modification Management Dialog' at the bottom of the first dialog as shown Fig. B.1. This is the first link among the four links (underlined with blue color) in Fig. B.1. After clicking this link we get the dialog in Fig. B.4. This dialog prepopulates the starting and ending revision numbers of the candidate system. It shows two buttons for extracting changes, and also, for deleting the already extracted changes.

Extracting Methods from Each Revision

After extracting and storing changes between consecutive revisions of the subject system, we extract methods from each revision. For this purpose we click the second link called 'Show Method Analysis Dialog' in Fig. B.1. After clicking this link we get a dialog as shown in Fig. B.5. This dialog contains four buttons for extracting methods, detecting method genealogies, storing methods to database, and mapping the already extracted changes (the changes extracted in the previous step) to the methods. Each of these four buttons work on all the revisions of the candidate system.

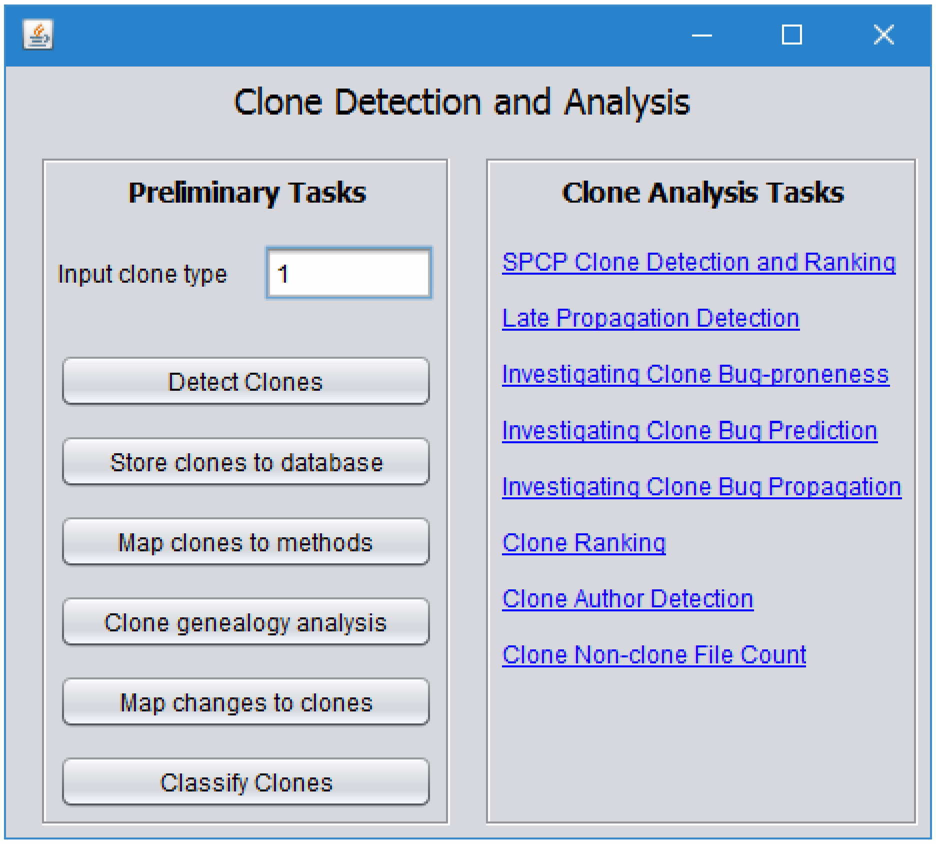

Clone Detection and Analysis
Preliminary Tasks
Input clone type
1
Detect Clones
Store clones to database
Map clones to methods
Clone genealogy analysis
Map changes to clones
Classify Clones
Clone Analysis Tasks
SPCP Clone Detection and Ranking
Late Propagation Detection
Investigating Clone Bug-proneness
Investigating Clone Bug Prediction
Investigating Clone Bug Propagation
Clone Ranking
Clone Author Detection
Clone Non-clone File Count

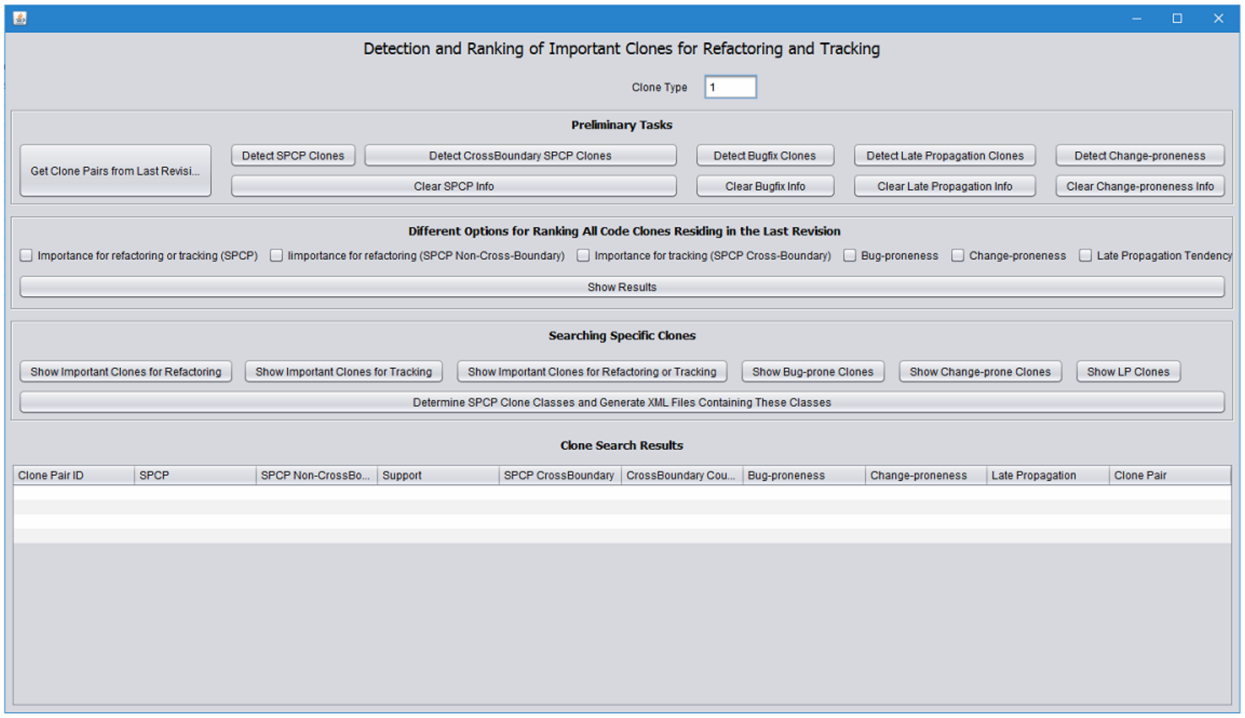
Figure B.6: Extracting and analyzing code clones from each revision

Extracting and Analyzing Clones

After working on the methods, we can extract code clones from each of the revisions of the subject system and can analyze those. For clone extraction and analysis we click the third link named 'Show Clone Analysis Dialog' as shown in Fig. B.1. After clicking this link we a dialog as shown in Fig. B.6. We see that the dialog in Fig. B.6 contains two sections named 'Preliminary Tasks' and 'Clone Analysis Tasks'. We first do the preliminary tasks before doing the analysis tasks.

Preliminary Tasks

There are six buttons in the section titled 'Preliminary Tasks'. These buttons are used for detecting clones, storing clones to the database, mapping clones to the already detected methods, detecting clone genealogies, mapping the already detected changes to clones, and classifying clones (i.e., identifying the clone classes). All these six tasks should be done sequentially. Before doing these tasks we specify the clone-type in the text-field. The preliminary tasks are done for a particular clone-type. We can specify three numbers (1, 2, or 3) for three clone-types (Type 1, Type 2, or Type 3). For detecting clones, AMIC applies the NiCad clone detector [48]. The users might think of detecting clones by themselves using NiCad (i.e., not by using AMIC). In that case, they do not need to click the 'Detect Clones' button in Fig. B.6. After doing the preliminary tasks we can perform the clone analysis tasks.


Detection and Ranking of Important Clones for Refactoring and Tracking
Clone Type
1
Preliminary Tasks
Get Clone Pairs from Last Revisi...
Detect SPCP Clones
Detect CrossBoundary SPCP Clones
Detect Bugfix Clones
Detect Late Propagation Clones
Detect Change-proneness
Clear SPCP Info
Clear Bugfix Info
Clear Late Propagation Info
Clear Change-proneness Info
Different Options for Ranking All Code Clones Residing in the Last Revision
Importance for refactoring or tracking (SPCP)
Iimportance for refactoring (SPCP Non-Cross-Boundary)
Importance for tracking (SPCP Cross-Boundary)
Bug-proneness
Change-proneness
Late Propagation Tendency
Show Results
Searching Specific Clones
Show Important Clones for Refactoring
Show Important Clones for Tracking
Show Important Clones for Refactoring or Tracking
Show Bug-prone Clones
Show Change-prone Clones
Show LP Clones
Determine SPCP Clone Classes and Generate XML Files Containing These Classes
Clone Search Results

Clone Pair ID	SPCP	SPCP Non-CrossBo...	Support	SPCP CrossBoundary	CrossBoundary Cou...	Bug-proneness	Change-proneness	Late Propagation	Clone Pair

Figure B.7: Identifying and ranking important code clones for management

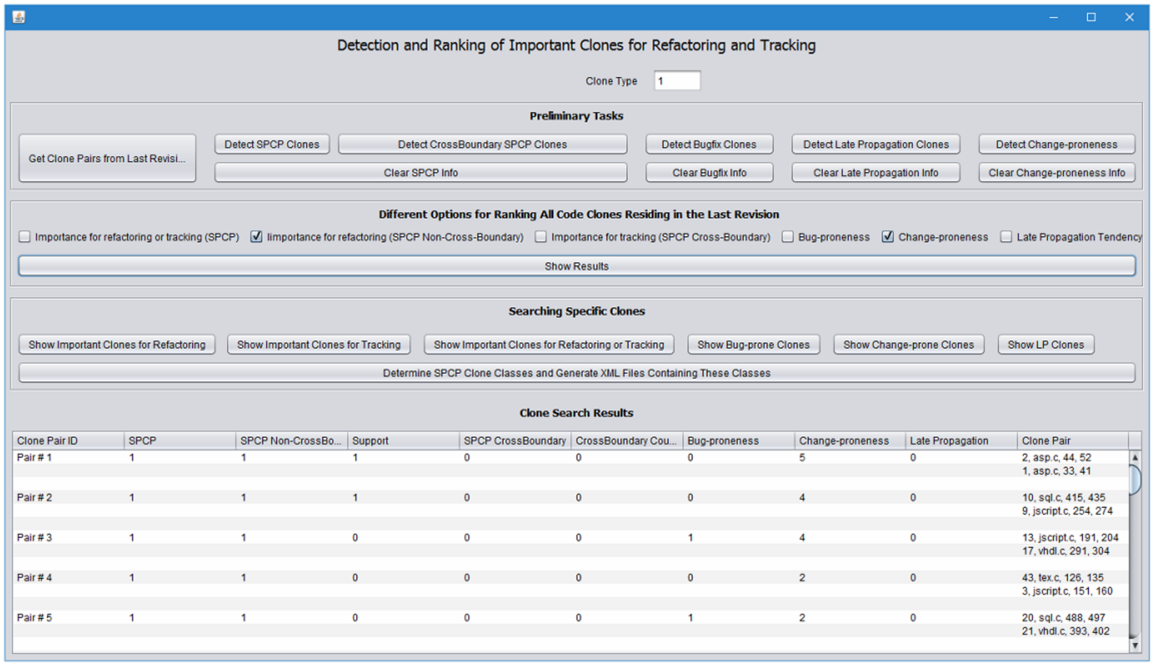
Clone Analysis Tasks

Clone analysis tasks mainly involves identifying and ranking important clones for refactoring and tracking. In our previous chapters we described that SPCP clones (code clones that evolved by following a Similarity Preserving Change Pattern called SPCP) are the important ones for refactoring or tracking. AMIC identifies the SPCP clones and ranks them on the basis of their evolutionary coupling, bug-proneness, change-proneness, and late propagation tendencies. AMIC also shows the non-SPCP clones residing in the system. For the purpose of detecting and ranking important code clones we click on the link named 'SPCP Clone Detection and Ranking' as shown in Fig. B.6, and we get the dialog in Fig. B.7.

Identifying and Ranking Important Clones

In Fig. B.7 we see that there is a section called 'Preliminary Tasks'. The tasks in this section must be done for identifying the important clones. These tasks need to be done before doing the ranking tasks. The preliminary tasks should be done sequentially from left to right. After getting the clone pairs from the last revision we detect the SPCP as well as cross-boundary SPCP clones by clicking the appropriate buttons. Then we detect which clones experienced bugs by clicking the button called 'Detect Bug-fix Clones'. By clicking the button called 'Detect Late Propagation Clones' we identify which code clones experienced late propagations during evolution. We finally determine the change-proneness of code clones using the button called 'Detect Change-proneness'. We can now perform the ranking tasks.

The dialog in Fig. B.7 contains a section called 'Different Options for Ranking All Code Clones Residing in the Last Revision'. The check boxes and button in this section allow us to rank code clones in different ways. From the names of the check boxes it is easy to realize that we can rank code clones considering their importance for refactoring and/or tracking, bug-proneness, change-proneness, and tendencies of experiencing late propagation. We can choose any number of the given criteria for the purpose of ranking. For example, Fig. B.8 shows that we have selected two check boxes for ranking. After selecting the check boxes we need to press the 'Show Results' button for seeing the ranked results. The results appear in the table with caption


Detection and Ranking of Important Clones for Refactoring and Tracking
Preliminary Tasks
Get Clone Pairs from Last Revisi...
Detect SPCP Clones
Detect CrossBoundary SPCP Clones
Detect Bugfix Clones
Detect Late Propagation Clones
Detect Change-proneness
Clear SPCP Info
Clear Bugfix Info
Clear Late Propagation Info
Clear Change-proneness Info
Show Results
Searching Specific Clones
Show Important Clones for Refactoring
Show Important Clones for Tracking
Show Important Clones for Refactoring or Tracking
Show Bug-prone Clones
Show Change-prone Clones
Show LP Clones
Determine SPCP Clone Classes and Generate XML Files Containing These Classes

Clone Type: 1

Different Options for Ranking All Code Clones Residing in the Last Revision

☐ Importance for refactoring or tracking (SPCP) ☑ Iimportance for refactoring (SPCP Non-Cross-Boundary) ☐ Importance for tracking (SPCP Cross-Boundary) ☐ Bug-proneness ☑ Change-proneness ☐ Late Propagation Tendency

Clone Search Results

Clone Pair ID	SPCP	SPCP Non-CrossBo...	Support	SPCP CrossBoundary	CrossBoundary Cou...	Bug-proneness	Change-proneness	Late Propagation	Clone Pair
Pair # 1	1	1	1	0	0	0	5	0	2, asp.c, 44, 52 1, asp.c, 33, 41
Pair # 2	1	1	1	0	0	0	4	0	10, sql.c, 415, 435 9, jscript.c, 254, 274
Pair # 3	1	1	0	0	0	1	4	0	13, jscript.c, 191, 204 17, vhdl.c, 291, 304
Pair # 4	1	1	0	0	0	0	2	0	43, tex.c, 126, 135 3, jscript.c, 151, 160
Pair # 5	1	1	0	0	0	1	2	0	20, sql.c, 488, 497 21, vhdl.c, 393, 402

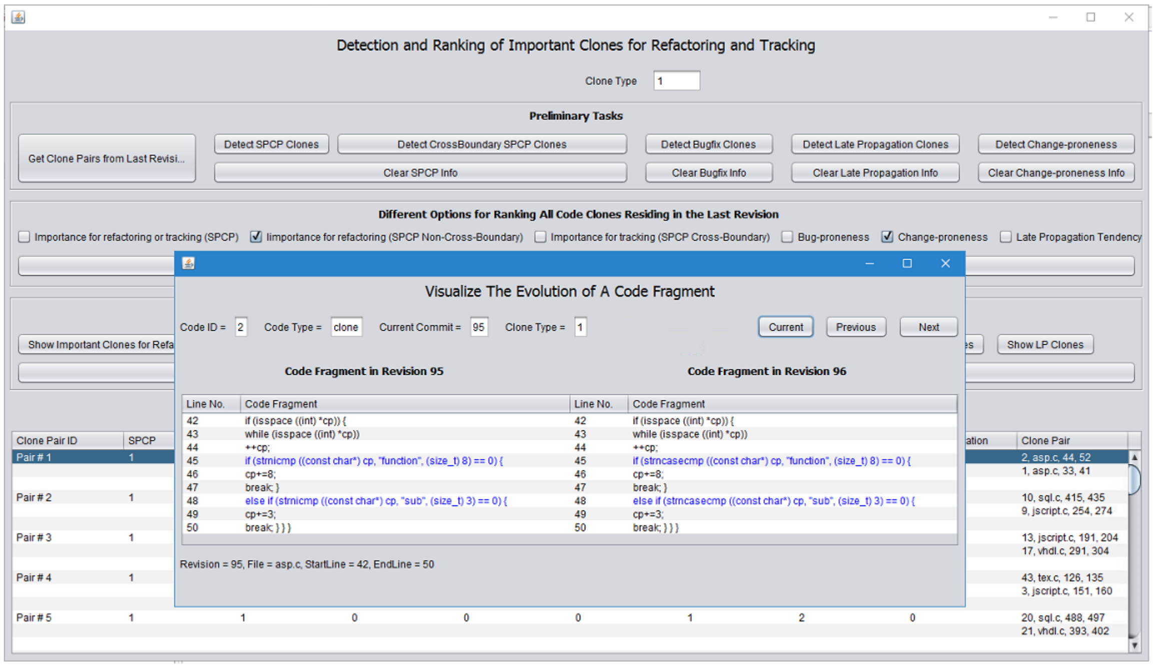
Figure B.8: Ranking code clones using more than one criterion

'Clone Search Results'. Each time we press the 'Show Results' button, AMIC also generates an XML file containing the ranked clone-pairs. The XML file is named as 'clonepairs.xml' and it gets generated in the same path where the JAR file of AMIC exists.

In Fig. B.8 we see that there is another section named 'Searching Specific Clones'. This section contains several buttons for getting clones with specific criteria. For example, the button called 'Show Important Clones for Refactoring' will let us retrieve and see only the non-cross-boundary SPCP clones, because these clones are considered important for refactoring [162]. In the same way, the button 'Show Important Clones for Tracking' can show us only the cross-boundary SPCP clones, because such clones are important for tracking [162]. We can see other buttons for retrieving the bug-prone clones, change-prone clones, and clones having tendencies of late propagation. The section 'Searching Specific Clones' also contains a big button called 'Determine SPCP Clone Classes and Generate XML Files Containing These Classes'. If we click this button, AMIC will first determine classes considering the SPCP clone pairs, and then, will generate three XML files. The first file contains all the SPCP clone classes, the second one contains only those SPCP clone classes that are suitable for refactoring, and the third one contains SPCP clone classes that are suitable for tracking. These files are 'spcpgroups_all.xml', 'spcpgroups_noncrossboundary.xml', and 'spcpgroups_crossboundary.xml' respectively. We get these files in the folder where the JAR file of AMIC exists.

Analyzing Clone Evolution

AMIC also helps us analyze the evolution of a clone fragment. Whenever the table 'Clone Search Results' in Fig. B.8 gets populated with clone pairs, we can see the evolution of any clone fragment by clicking the corresponding row in the table. For example, if we click the first row in the table in Fig. B.8 we can see another dialog as shown in Fig. B.9. The dialog has the title 'Visualize the Evolution of a Clone Fragment'. It helps us roam through a clone genealogy through three buttons called 'Current', 'Previous', and 'Next'. From Fig. B.9 we can see that the dialog contains four fields: code id, code type, current commit, and clone type. Code id field shows the id of the clone fragment, and code type field makes us realize that we are watching the evolution of a clone fragment. This dialog can also be used for watching the evolution of a


Detection and Ranking of Important Clones for Refactoring and Tracking
Preliminary Tasks
Get Clone Pairs from Last Revisi...
Detect SPCP Clones
Detect CrossBoundary SPCP Clones
Detect Bugfix Clones
Detect Late Propagation Clones
Detect Change-proneness
Clear SPCP Info
Clear Bugfix Info
Clear Late Propagation Info
Clear Change-proneness Info
Different Options for Ranking All Code Clones Residing in the Last Revision
Show Important Clones for Refa
Show LP Clones
Visualize The Evolution of A Code Fragment
Current
Previous
Next
Code Fragment in Revision 95
Code Fragment in Revision 96

Clone Type 1

[ ] Importance for refactoring or tracking (SPCP) [✓] Iimportance for refactoring (SPCP Non-Cross-Boundary) [ ] Importance for tracking (SPCP Cross-Boundary) [ ] Bug-proneness [✓] Change-proneness [ ] Late Propagation Tendency

Code ID = 2 Code Type = clone Current Commit = 95 Clone Type = 1

Line No.	Code Fragment	Line No.	Code Fragment
42	if (isspace ((int) *cp)) {	42	if (isspace ((int) *cp)) {
43	while (isspace ((int) *cp))	43	while (isspace ((int) *cp))
44	++cp;	44	++cp;
45	if (strnicmp ((const char*) cp, "function", (size_t) 8) == 0) {	45	if (strncasecmp ((const char*) cp, "function", (size_t) 8) == 0) {
46	cp+=8;	46	cp+=8;
47	break; }	47	break; }
48	else if (strnicmp ((const char*) cp, "sub", (size_t) 3) == 0) {	48	else if (strncasecmp ((const char*) cp, "sub", (size_t) 3) == 0) {
49	cp+=3;	49	cp+=3;
50	break; } } }	50	break; } } }

Revision = 95, File = asp.c, StartLine = 42, EndLine = 50

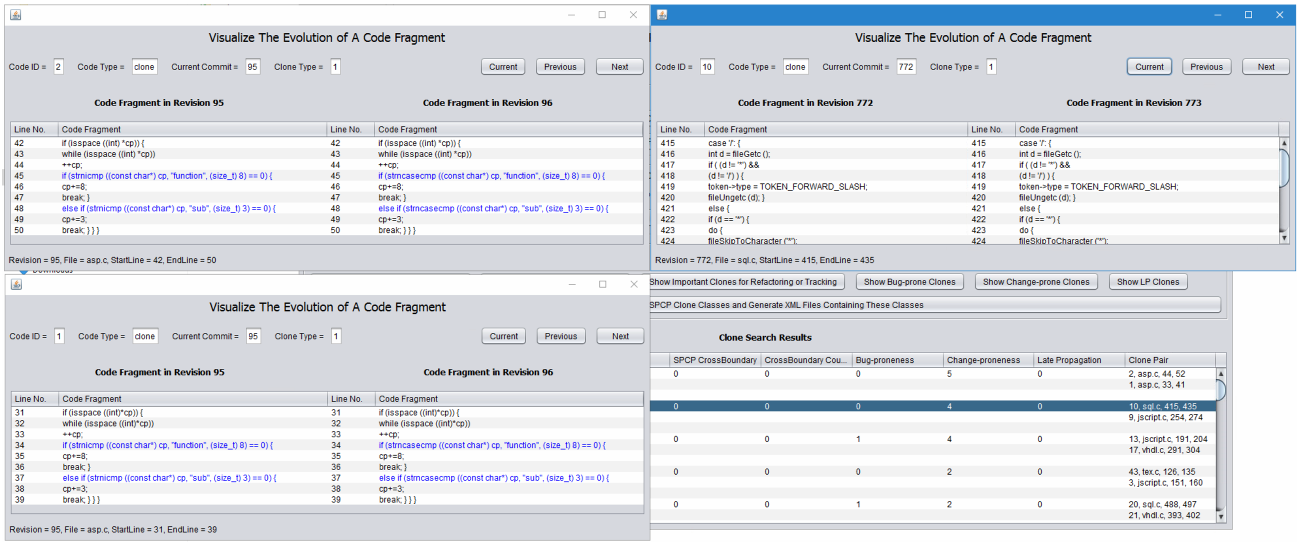
Clone Pair ID	SPCP							ation	Clone Pair
Pair # 1	1								2, asp.c, 44, 52 1, asp.c, 33, 41
Pair # 2	1								10, sql.c, 415, 435 9, jscript.c, 254, 274
Pair # 3	1								13, jscript.c, 191, 204 17, vhdl.c, 291, 304
Pair # 4	1								43, tex.c, 126, 135 3, jscript.c, 151, 160
Pair # 5	1	1	0	0	0	1	2	0	20, sql.c, 488, 497 21, vhdl.c, 393, 402

Figure B.9: Visualizing clone evolution

method. In that case the code type will be 'Method'. The field named 'Current Commit' shows the changes occurred to a code fragment in a particular commit operation. In Fig. B.9, the current commit field contains 95. If we can see below the code fragments, we realize that these code fragments are the snapshots of the same clone fragment in two different revisions: 95 and 96. The dialog also shows the differences between the snapshots. In such a state of the dialog if we click the button called 'Previous', then we can visualize whether there were any changes to the clone fragment in commit operation 94. In other words, we will see the snapshots of the clone fragment in revisions 94 and 95. In the same way, the button called 'Next' lets us approach through newer commit operations. We also see that the four fields: code id, code type, current commit, and clone type can by modified by the user. We can choose to watch the evolution of a different clone fragment in this dialog. In that case, we specify the clone fragment id, any commit number, and the clone-type of the clone fragment in the corresponding fields, and then click the button called 'Current'. The dialog will show us the state of the specified clone fragment in the specified commit operation. We can even want to watch the evolution of a method through this dialog. For watching method evolution, we specify a particular method id in the code id field, the code type field should be changed to 'Method', and we specify a particular commit number in the current commit field. The clone type field will be ignored in case of watching the evolution of a method. After populating the three fields (except clone type filed), we click the button named 'Current'. We can then see the state of the method in the specified commit operation. By clicking the 'Previous' and 'Next' buttons we can approach through the evolution history of the method.

Analyzing the Co-evolution of More than One Code Fragment

AMIC can also be used for visually analyzing the co-evolution of more than one code fragment (clone fragment or method). Fig. B.10 demonstrates an example where we can see the evolution of three clone fragments (the first three clone fragments in the clone search result). For analyzing the co-evolution of multiple clone fragments we click the corresponding rows in the clone search results. We will see a separate dialog titled 'Visualizing the Evolution of a Code Fragment' for each of the clicked clone fragment.


Current
Previous
Next
Show Important Clones for Refactoring or Tracking
Show Bug-prone Clones
Show Change-prone Clones
Show LP Clones
SPCP Clone Classes and Generate XML Files Containing These Classes

Visualize The Evolution of A Code Fragment

Code ID = 2 Code Type = clone Current Commit = 95 Clone Type = 1

Code Fragment in Revision 95 **Code Fragment in Revision 96**

Line No.	Code Fragment	Line No.	Code Fragment
42	if (isspace ((int) *cp)) {	42	if (isspace ((int) *cp)) {
43	while (isspace ((int) *cp))	43	while (isspace ((int) *cp))
44	++cp;	44	++cp;
45	if (strnicmp ((const char*) cp, "function", (size_t) 8) == 0) {	45	if (strncasecmp ((const char*) cp, "function", (size_t) 8) == 0) {
46	cp+=8;	46	cp+=8;
47	break; }	47	break; }
48	else if (strnicmp ((const char*) cp, "sub", (size_t) 3) == 0) {	48	else if (strncasecmp ((const char*) cp, "sub", (size_t) 3) == 0) {
49	cp+=3;	49	cp+=3;
50	break; } } }	50	break; } } }

Revision = 95, File = asp.c, StartLine = 42, EndLine = 50

Visualize The Evolution of A Code Fragment

Code ID = 10 Code Type = clone Current Commit = 772 Clone Type = 1

Code Fragment in Revision 772 **Code Fragment in Revision 773**

Line No.	Code Fragment	Line No.	Code Fragment
415	case '/': {	415	case '/': {
416	int d = fileGetc ();	416	int d = fileGetc ();
417	if ((d != '*') &&	417	if ((d != '*') &&
418	(d != '/')) {	418	(d != '/')) {
419	token->type = TOKEN_FORWARD_SLASH;	419	token->type = TOKEN_FORWARD_SLASH;
420	fileUngetc (d); }	420	fileUngetc (d); }
421	else {	421	else {
422	if (d == '*') {	422	if (d == '*') {
423	do {	423	do {
424	fileSkipToCharacter ('*');	424	fileSkipToCharacter ('*');

Revision = 772, File = sql.c, StartLine = 415, EndLine = 435

Visualize The Evolution of A Code Fragment

Code ID = 1 Code Type = clone Current Commit = 95 Clone Type = 1

Code Fragment in Revision 95 **Code Fragment in Revision 96**

Line No.	Code Fragment	Line No.	Code Fragment
31	if (isspace ((int)*cp)) {	31	if (isspace ((int)*cp)) {
32	while (isspace ((int)*cp))	32	while (isspace ((int)*cp))
33	++cp;	33	++cp;
34	if (strnicmp ((const char*) cp, "function", (size_t) 8) == 0) {	34	if (strncasecmp ((const char*) cp, "function", (size_t) 8) == 0) {
35	cp+=8;	35	cp+=8;
36	break; }	36	break; }
37	else if (strnicmp ((const char*) cp, "sub", (size_t) 3) == 0) {	37	else if (strncasecmp ((const char*) cp, "sub", (size_t) 3) == 0) {
38	cp+=3;	38	cp+=3;
39	break; } } }	39	break; } } }

Revision = 95, File = asp.c, StartLine = 31, EndLine = 39

Clone Search Results

SPCP CrossBoundary	CrossBoundary Cou...	Bug-proneness	Change-proneness	Late Propagation	Clone Pair
0	0	0	5	0	2, asp.c, 44, 52 1, asp.c, 33, 41
0	0	0	4	0	10, sql.c, 415, 435 9, jscript.c, 254, 274
0	0	1	4	0	13, jscript.c, 191, 204 17, vhdl.c, 291, 304
0	0	0	2	0	43, tex.c, 126, 135 3, jscript.c, 151, 160
0	0	1	2	0	20, sql.c, 488, 497 21, vhdl.c, 393, 402

Figure B.10: Visualizing co-evolution of clone fragments

B.2 Using AMIC through its Web-based Implementation

AMIC can be used through its website [173]. The web-based implementation does not require the user to install anything. AMIC website supports all the functionalities that we have already described in Section B.1. A user needs to register before accessing the functionalities. Fig. B.11 shows a snap-shot of the website. In the following subsections we describe how to access different functionalities available in AMIC.

B.2.1 User Registration

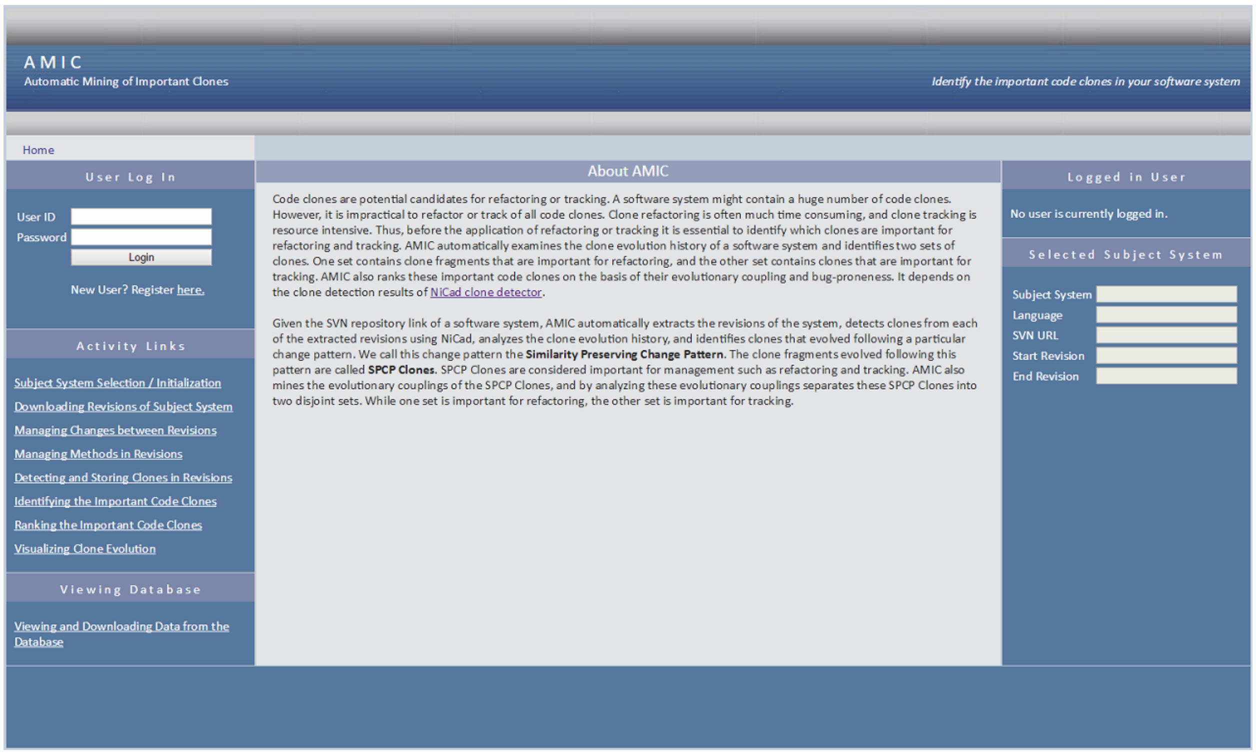
A new user can perform registration by clicking the link demonstrated in Fig. B.12. After clicking this link, the user registration form (as shown in Fig. B.12) appears. After performing registration through this form, a user can log in using the 'User Log In' section.

B.2.2 Subject System Initialization / Selection

After logging in, a user can work on a subject system by initializing it through the subject system initialization form as shown in Fig. B.13. We can get this form by clicking the link in the red box at the left hand side of the figure. Once a subject system is initialized, a folder dedicated for it is created in the server machine. A separate database for the subject system is also created in the MySQL database management system. After initializing a subject system, it appears in a table below the subject system initialization form as shown in Fig. B.14. A user needs to select a subject system by clicking its underlined name in the first column of the table. At the top right corner of this figure (Fig. B.14) we see that a user is currently logged in. It is impossible to initialize a subject system or view the already initialized subject systems without logging in. After clicking the name of a subject system, it appears in the section named 'Selected Subject System' as shown in Fig. B.15. After selecting a subject system a user can work on it.

B.2.3 Downloading the Revisions of a Selected Subject System

After selecting a subject system, a user can download its revisions using a form as shown in Fig. B.16. We get this form by clicking the link demonstrated at the left hand side.

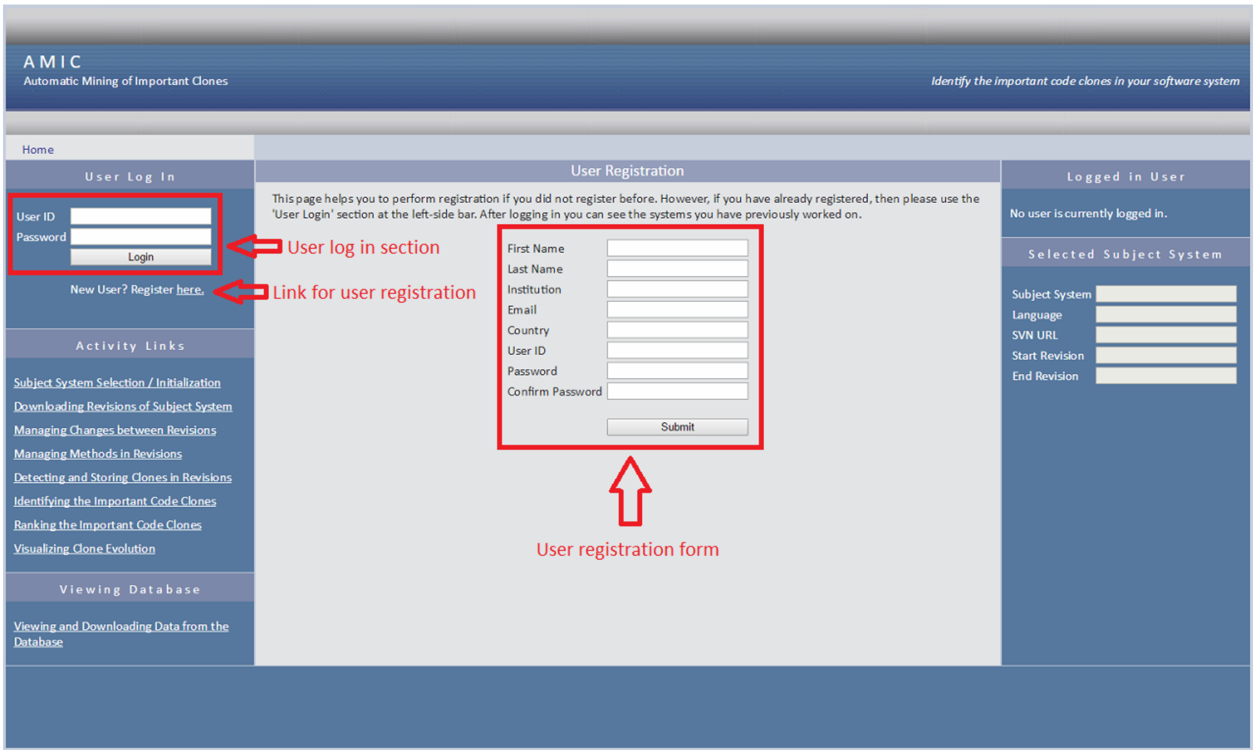

AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
User ID
Password
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
About AMIC
Logged in User
Selected Subject System
Subject System
Language
SVN URL
Start Revision
End Revision

Code clones are potential candidates for refactoring or tracking. A software system might contain a huge number of code clones. However, it is impractical to refactor or track of all code clones. Clone refactoring is often much time consuming, and clone tracking is resource intensive. Thus, before the application of refactoring or tracking it is essential to identify which clones are important for refactoring and tracking. AMIC automatically examines the clone evolution history of a software system and identifies two sets of clones. One set contains clone fragments that are important for refactoring, and the other set contains clones that are important for tracking. AMIC also ranks these important code clones on the basis of their evolutionary coupling and bug-proneness. It depends on the clone detection results of NiCad clone detector.

Given the SVN repository link of a software system, AMIC automatically extracts the revisions of the system, detects clones from each of the extracted revisions using NiCad, analyzes the clone evolution history, and identifies clones that evolved following a particular change pattern. We call this change pattern the **Similarity Preserving Change Pattern**. The clone fragments evolved following this pattern are called **SPCP Clones**. SPCP Clones are considered important for management such as refactoring and tracking. AMIC also mines the evolutionary couplings of the SPCP Clones, and by analyzing these evolutionary couplings separates these SPCP Clones into two disjoint sets. While one set is important for refactoring, the other set is important for tracking.

No user is currently logged in.

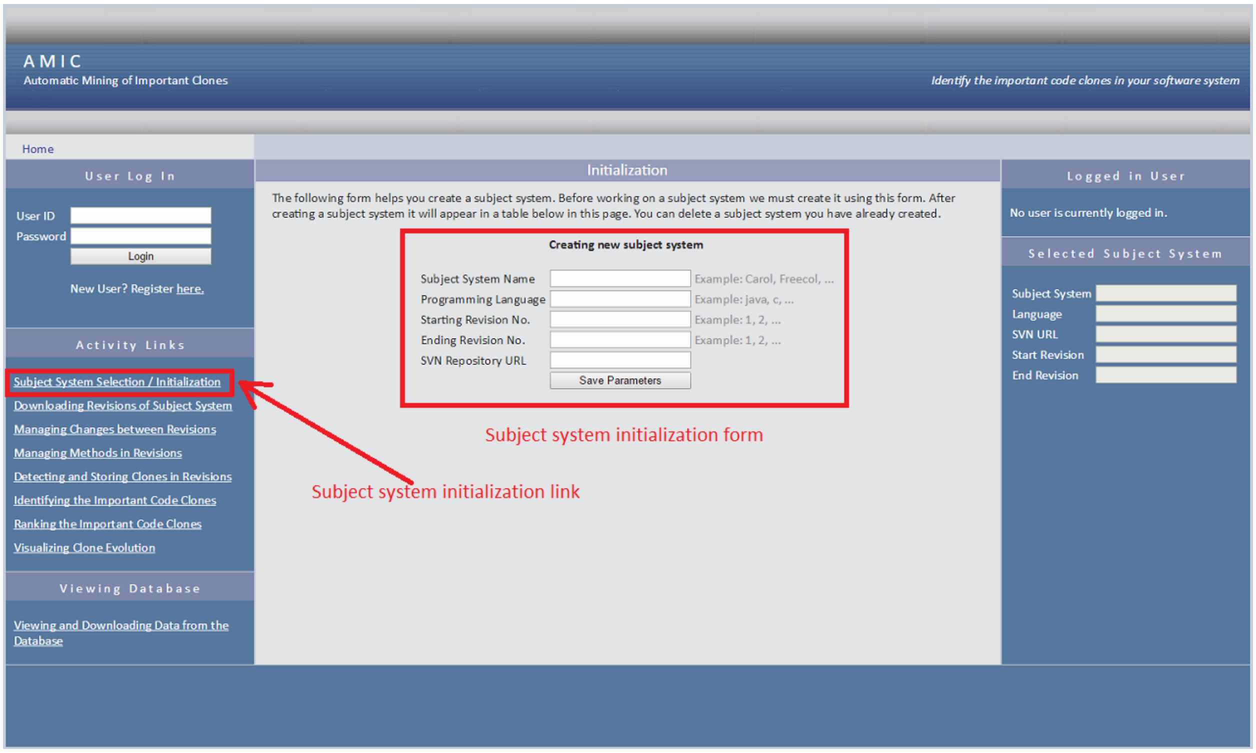
Figure B.11: Snap-shot of AMIC website


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
User ID
Password
Login
New User? Register here.
User log in section
Link for user registration
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
User Registration
First Name
Last Name
Institution
Email
Country
User ID
Password
Confirm Password
Submit
User registration form
Logged in User
Selected Subject System
Subject System
Language
SVN URL
Start Revision
End Revision

This page helps you to perform registration if you did not register before. However, if you have already registered, then please use the 'User Login' section at the left-side bar. After logging in you can see the systems you have previously worked on.

No user is currently logged in.

Figure B.12: User registration


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system

Home

User Log In

User ID
Password
Login

New User? Register here.

Activity Links

Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution

Viewing Database

Viewing and Downloading Data from the Database

Initialization

The following form helps you create a subject system. Before working on a subject system we must create it using this form. After creating a subject system it will appear in a table below in this page. You can delete a subject system you have already created.

Creating new subject system

Subject System Name	Example: Carol, Freecol, ...
Programming Language	Example: java, c, ...
Starting Revision No.	Example: 1, 2, ...
Ending Revision No.	Example: 1, 2, ...
SVN Repository URL	

Save Parameters

Logged in User

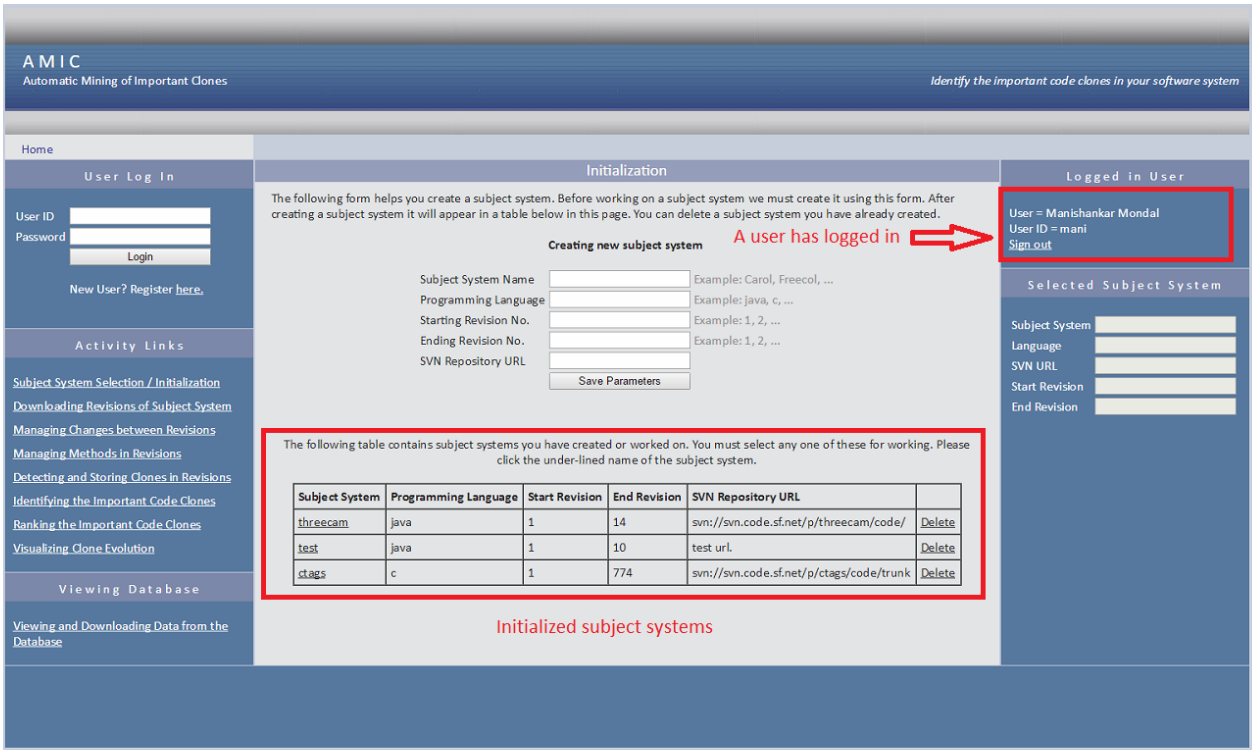
No user is currently logged in.

Selected Subject System

Subject System
Language
SVN URL
Start Revision
End Revision

Subject system initialization form
Subject system initialization link

Figure B.13: Initializing subject system


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system

Home

User Log In

User ID
Password
Login

New User? Register here.

Activity Links

Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution

Viewing Database

Viewing and Downloading Data from the Database

Initialization

The following form helps you create a subject system. Before working on a subject system we must create it using this form. After creating a subject system it will appear in a table below in this page. You can delete a subject system you have already created.

Creating new subject system

Subject System Name	Example: Carol, Freecol, ...
Programming Language	Example: java, c, ...
Starting Revision No.	Example: 1, 2, ...
Ending Revision No.	Example: 1, 2, ...
SVN Repository URL	

Save Parameters

The following table contains subject systems you have created or worked on. You must select any one of these for working. Please click the under-lined name of the subject system.

Subject System	Programming Language	Start Revision	End Revision	SVN Repository URL	
threecam	java	1	14	svn://svn.code.sf.net/p/threecam/code/	Delete
test	java	1	10	test url.	Delete
ctags	c	1	774	svn://svn.code.sf.net/p/ctags/code/trunk	Delete

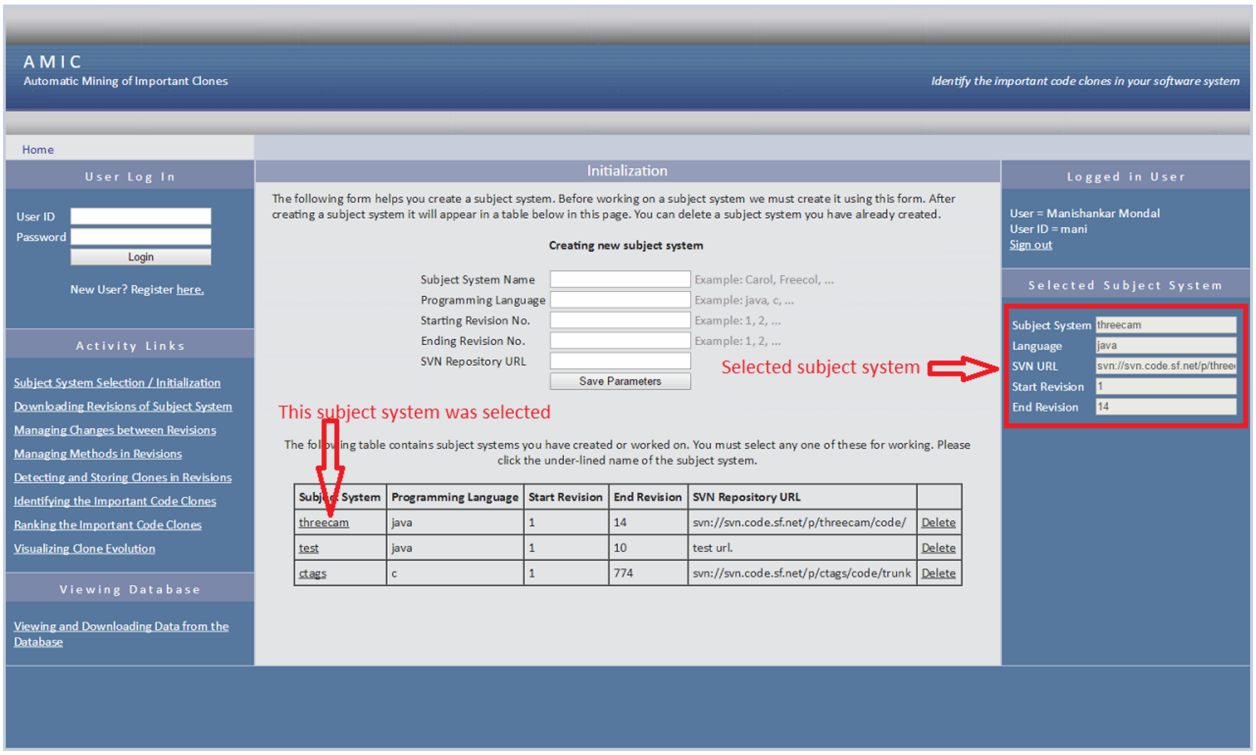
Logged in User

User = Manishankar Mondal
User ID = mani
Sign out

Selected Subject System

Subject System
Language
SVN URL
Start Revision
End Revision

Figure B.14: Viewing initialized subject systems


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system

Home

User Log In

User ID	
Password	

Login

New User? Register here.

Activity Links

- Subject System Selection / Initialization
- Downloading Revisions of Subject System
- Managing Changes between Revisions
- Managing Methods in Revisions
- Detecting and Storing Clones in Revisions
- Identifying the Important Code Clones
- Ranking the Important Code Clones
- Visualizing Clone Evolution

Viewing Database

- Viewing and Downloading Data from the Database

Initialization

The following form helps you create a subject system. Before working on a subject system we must create it using this form. After creating a subject system it will appear in a table below in this page. You can delete a subject system you have already created.

Creating new subject system

Subject System Name		Example: Carol, Freecol, ...
Programming Language		Example: java, c, ...
Starting Revision No.		Example: 1, 2, ...
Ending Revision No.		Example: 1, 2, ...
SVN Repository URL		

Save Parameters

Selected subject system

This subject system was selected

The following table contains subject systems you have created or worked on. You must select any one of these for working. Please click the under-lined name of the subject system.

Subject System	Programming Language	Start Revision	End Revision	SVN Repository URL	
threecam	java	1	14	svn://svn.code.sf.net/p/threecam/code/	Delete
test	java	1	10	test url.	Delete
ctags	c	1	774	svn://svn.code.sf.net/p/ctags/code/trunk	Delete

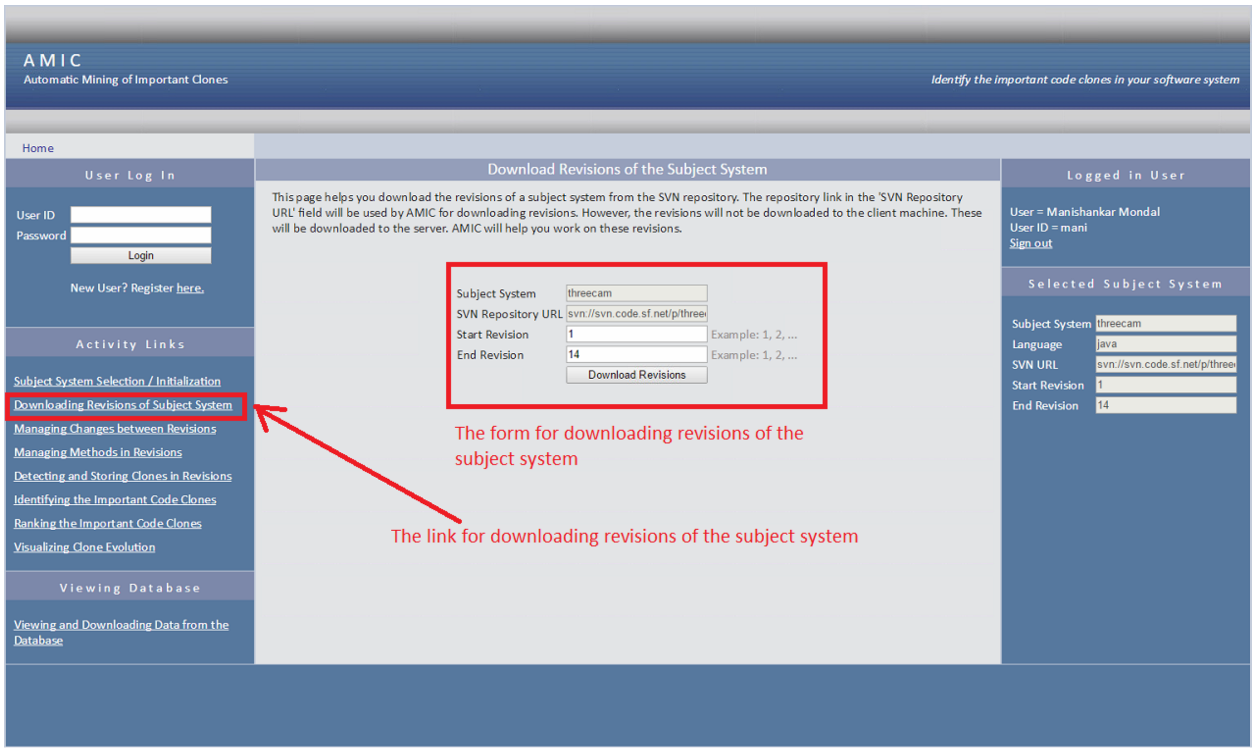
Logged in User

User = Manishankar Mondal
User ID = mani
Sign out

Selected Subject System

Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.15: Subject system selection


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system

Home

User Log In

User ID	
Password	

Login

New User? Register here.

Activity Links

- Subject System Selection / Initialization
- Downloading Revisions of Subject System
- Managing Changes between Revisions
- Managing Methods in Revisions
- Detecting and Storing Clones in Revisions
- Identifying the Important Code Clones
- Ranking the Important Code Clones
- Visualizing Clone Evolution

Viewing Database

- Viewing and Downloading Data from the Database

Download Revisions of the Subject System

This page helps you download the revisions of a subject system from the SVN repository. The repository link in the 'SVN Repository URL' field will be used by AMIC for downloading revisions. However, the revisions will not be downloaded to the client machine. These will be downloaded to the server. AMIC will help you work on these revisions.

Subject System	threecam	
SVN Repository URL	svn://svn.code.sf.net/p/three	
Start Revision	1	Example: 1, 2, ...
End Revision	14	Example: 1, 2, ...

Download Revisions

The form for downloading revisions of the subject system

The link for downloading revisions of the subject system

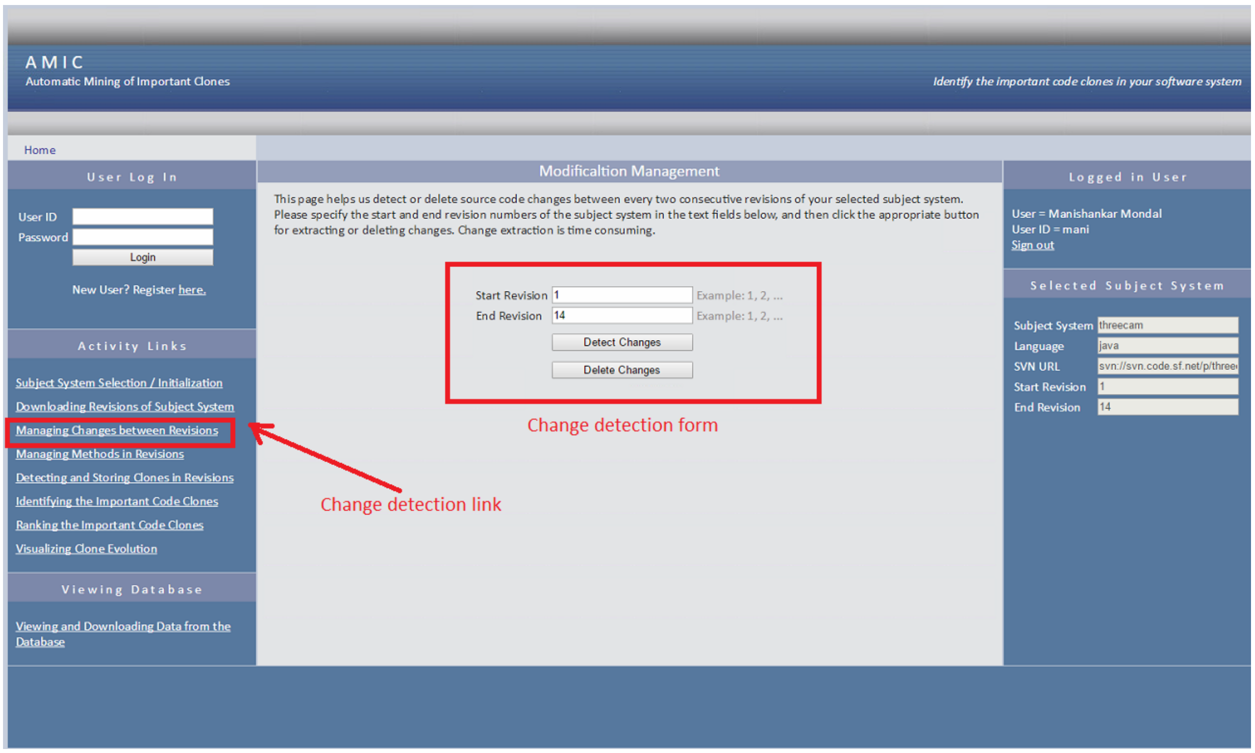
Logged in User

User = Manishankar Mondal
User ID = mani
Sign out

Selected Subject System

Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.16: Downloading revisions of a subject system


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system

Home

Modificaltion Management

This page helps us detect or delete source code changes between every two consecutive revisions of your selected subject system. Please specify the start and end revision numbers of the subject system in the text fields below, and then click the appropriate button for extracting or deleting changes. Change extraction is time consuming.

Start Revision: 1 (Example: 1, 2, ...)
End Revision: 14 (Example: 1, 2, ...)

User = Manishankar Mondal
User ID = mani

Subject System: threecam
Language: java
SVN URL: svn://svn.code.sf.net/p/three
Start Revision: 1
End Revision: 14

User Log In
User ID
Password
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
Detect Changes
Delete Changes
Change detection form
Change detection link
Logged in User
Sign out
Selected Subject System

Figure B.17: Detecting changes between revisions of a subject system

B.2.4 Detecting Changes between Revisions

We can detect changes between revisions using the form as shown in fig. B.17. This form can be obtained by clicking the demonstrated link at the left hand side.

B.2.5 Detecting and Managing Methods in Revisions

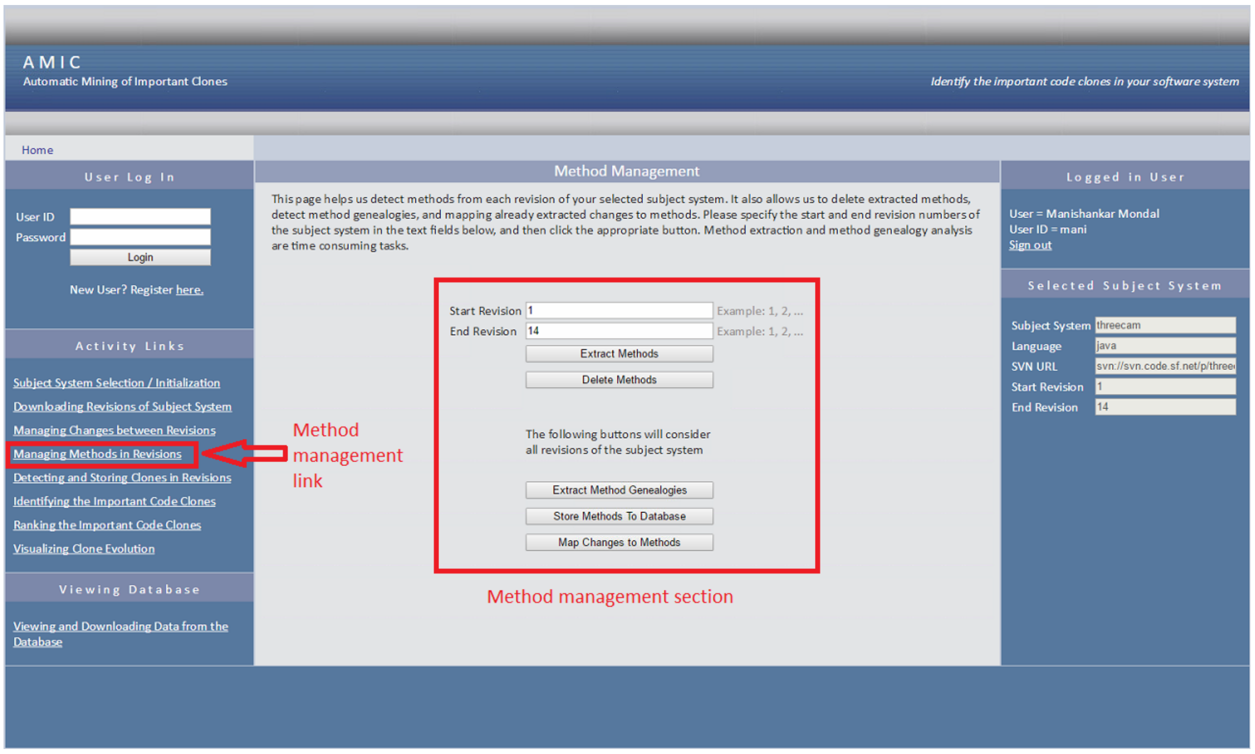
The web-page shown in Fig. B.18 helps us detect methods from each revision, extract method genealogies, store methods to database, and map changes to methods. In the figure we see that there are separate buttons for these tasks. The tasks should be done sequentially. We click the button demonstrated at the left hand side for getting the method management page.

B.2.6 Detecting and Managing Clones in Revisions

Fig. B.19 shows a web-page that supports detecting and managing code clones in each revision of a subject system. From the figure we can see that there are buttons for detecting and storing different types of clones, mapping clones to methods, detecting clone genealogies, and mapping changes to clones. After performing the clone management tasks a user can detect the important clones from different perspectives.

B.2.7 Detecting the Important Code Clones

Detection of important clones is facilitated by the web-page shown in Fig. B.20. This page contains separate buttons for detecting SPCP clones (i.e., code clones that evolved by following a similarity preserving change pattern), detecting cross-boundary SPCP clones (SPCP clones that have relationships beyond their class boundaries), detecting bug-prone clones, detecting code clones that experienced late propagations, and


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
Method Management
Extract Methods
Delete Methods
Extract Method Genealogies
Store Methods To Database
Map Changes to Methods
Method management link
Method management section
Logged in User
Sign out
Selected Subject System

This page helps us detect methods from each revision of your selected subject system. It also allows us to delete extracted methods, detect method genealogies, and mapping already extracted changes to methods. Please specify the start and end revision numbers of the subject system in the text fields below, and then click the appropriate button. Method extraction and method genealogy analysis are time consuming tasks.

User ID
Password

Start Revision: 1 (Example: 1, 2, ...)
End Revision: 14 (Example: 1, 2, ...)

The following buttons will consider all revisions of the subject system

User = Manishankar Mondal
User ID = mani

Subject System: threecam
Language: java
SVN URL: svn://svn.code.sf.net/p/three
Start Revision: 1
End Revision: 14

Figure B.18: Managing methods in revisions of a subject system

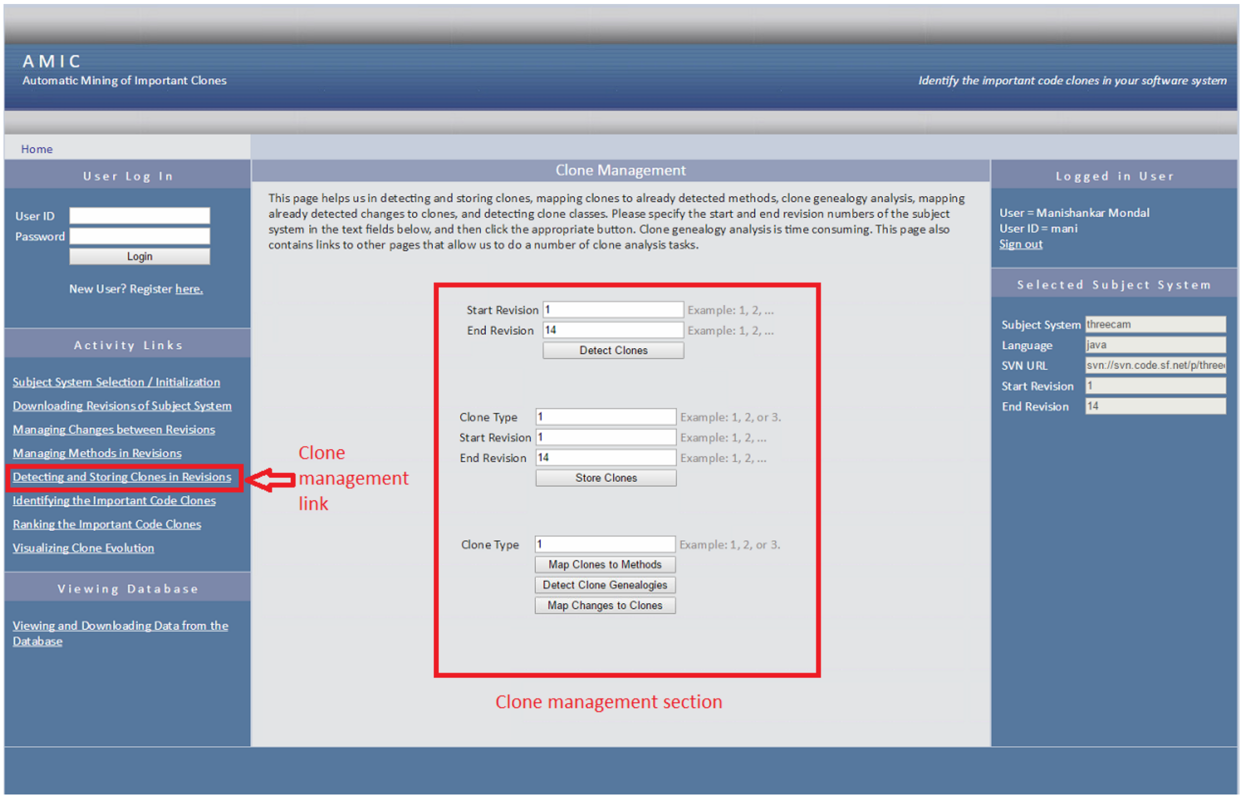
detecting code clones that exhibited change-proneness during evolution. These tasks should be done sequentially. After performing all the tasks in this page, we can go to the clone ranking page for ranking clones from different perspectives.

B.2.8 Ranking Code Clones

The web-page in Fig. B.21 facilitates ranking code clones from different perspectives. From the figure we see that the top most red box contains different buttons and check boxes for ranking clones and showing the ranking results. There are buttons for ranking code clones on the basis of their importance for refactoring, tracking, their bug-proneness, change-proneness, and late propagation tendencies. The clone rank results are shown in a table indicated by the biggest red box. We also see that the clone rank results can also be downloaded. AMIC generates and shows XML files if we click the download link just above the clone rank results table. At the bottom of the top most red box we see that there is a button for determining SPCP clone classes and generating XML files. If we click this button, three XML files will be generated. One file contains SPCP clone classes considering all the SPCP clones, the second one contains SPCP clone classes considering all the non-cross-boundary SPCP clones (clones that are important for refactoring), and the third file contains classes formed from all the cross-boundary SPCP clones (clones that are important for tracking). We can download these XML files by clicking the three download links below the button.

B.2.9 Viewing Clone Evolution

AMIC website also facilitates visualizing clone evolution. The web-page in Fig. B.22 supports this facility. In the figure we can see the changes to a clone fragment in commit operation 3. In the text fields we specify the clone ID, clone type, and commit number and then click the button called ‘Current’ to see the changes to the clone fragment (having the specified clone ID) in the specified commit operation. We can then click the previous and next buttons to see the changes to the clone fragment in the previous or next commit



AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
User ID
Password
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
Clone management link
Clone Management
Detect Clones
Store Clones
Map Clones to Methods
Detect Clone Genealogies
Map Changes to Clones
Clone management section
Logged in User
Sign out
Selected Subject System

This page helps us in detecting and storing clones, mapping clones to already detected methods, clone genealogy analysis, mapping already detected changes to clones, and detecting clone classes. Please specify the start and end revision numbers of the subject system in the text fields below, and then click the appropriate button. Clone genealogy analysis is time consuming. This page also contains links to other pages that allow us to do a number of clone analysis tasks.

Start Revision: 1 (Example: 1, 2, ...)
End Revision: 14 (Example: 1, 2, ...)

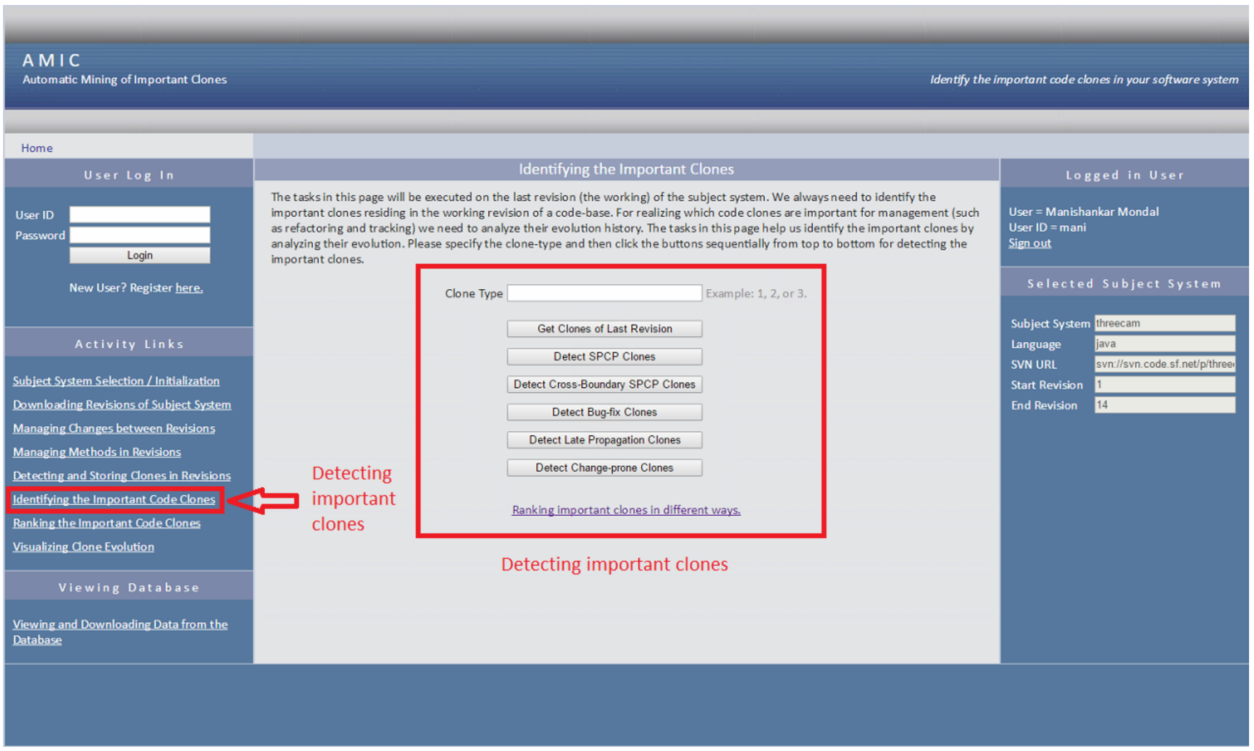
Clone Type: 1 (Example: 1, 2, or 3.)
Start Revision: 1 (Example: 1, 2, ...)
End Revision: 14 (Example: 1, 2, ...)

Clone Type: 1 (Example: 1, 2, or 3.)

User = Manishankar Mondal
User ID = mani

Subject System: threecam
Language: java
SVN URL: svn://svn.code.sf.net/p/three
Start Revision: 1
End Revision: 14

Figure B.19: Managing clones in revisions of a subject system


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
User ID
Password
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
Identifying the Important Clones
Clone Type
Example: 1, 2, or 3.
Get Clones of Last Revision
Detect SPCP Clones
Detect Cross-Boundary SPCP Clones
Detect Bug-fix Clones
Detect Late Propagation Clones
Detect Change-prone Clones
Ranking important clones in different ways.
Detecting important clones
Detecting important clones
Logged in User
Selected Subject System

The tasks in this page will be executed on the last revision (the working) of the subject system. We always need to identify the important clones residing in the working revision of a code-base. For realizing which code clones are important for management (such as refactoring and tracking) we need to analyze their evolution history. The tasks in this page help us identify the important clones by analyzing their evolution. Please specify the clone-type and then click the buttons sequentially from top to bottom for detecting the important clones.

User = Manishankar Mondal
User ID = mani
Sign out

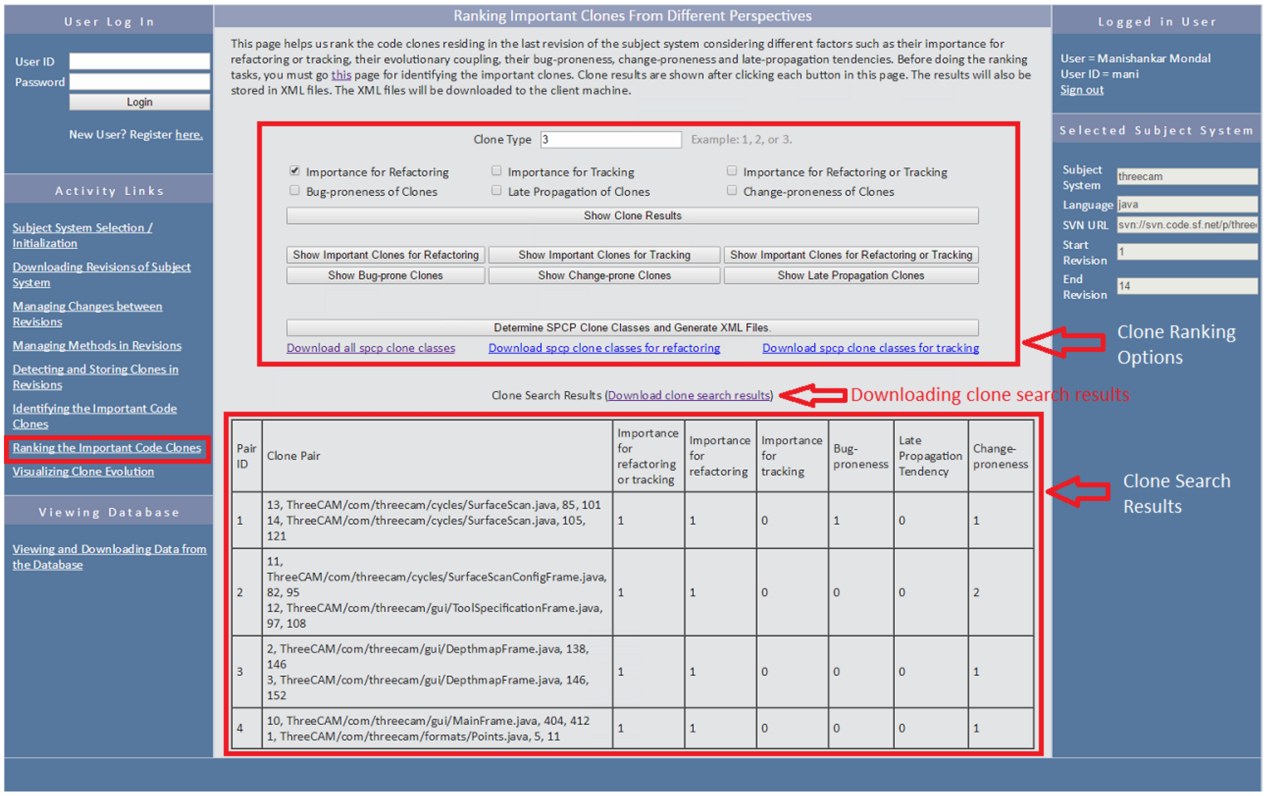
Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.20: Identifying the important clones of a subject system

operations. We can get the clone IDs from the clone rank results. If we look at Fig. B.21 we see that the table containing the clone rank results has a field with caption 'Clone Pair'. This field in each row contains information about two clone fragments in a pair. The information for each clone fragment contains four pieces of information: clone ID, clone file path, start line, and end line sequentially. We see that these four pieces of information have been given by separating them by commas. We can get clone IDs from this 'Clone Pair' field of the clone rank results table. If we further look at this table (Fig. B.21), we see that the fourth row (Pair ID = 4) contains a clone pair. One clone fragment has a clone ID of 10, and the other fragment has a ID of 1. Also, from the last column in this row we see that the change-proneness value for this pair is 1. That means, any one of the two clone fragments in this clone pair changed once during evolution. We checked both fragments using AMIC and found that the clone fragment with ID = 1 changed in the commit operation applied on revision 3. This change has been demonstrated in Fig. B.22.

B.2.10 Viewing and Downloading Table Data from Database

AMIC facilitates viewing and downloading table data from the database. Fig. B.23 shows the web-page responsible for this. In the top most red box we see that there are underlined table names. These tables exist in the database. If we click each of these table names, the data from that table will be shown in the table captioned 'Table Data' in this page. At the right side of the caption, there is a download link. By clicking this link we can download table data.


User Log In
User ID
Password
Login
New User? Register here.
Activity Links
Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution
Viewing Database
Viewing and Downloading Data from the Database
Ranking Important Clones From Different Perspectives
Show Clone Results
Show Important Clones for Refactoring
Show Important Clones for Tracking
Show Important Clones for Refactoring or Tracking
Show Bug-prone Clones
Show Change-prone Clones
Show Late Propagation Clones
Determine SPCP Clone Classes and Generate XML Files.
Download all spcp clone classes
Download spcp clone classes for refactoring
Download spcp clone classes for tracking
Clone Ranking Options
Downloading clone search results
Clone Search Results
Logged in User
Sign out
Selected Subject System

This page helps us rank the code clones residing in the last revision of the subject system considering different factors such as their importance for refactoring or tracking, their evolutionary coupling, their bug-proneness, change-proneness and late-propagation tendencies. Before doing the ranking tasks, you must go this page for identifying the important clones. Clone results are shown after clicking each button in this page. The results will also be stored in XML files. The XML files will be downloaded to the client machine.

Clone Type 3 Example: 1, 2, or 3.

[✓ Importance for Refactoring] Importance for Tracking Importance for Refactoring or Tracking
Bug-proneness of Clones Late Propagation of Clones Change-proneness of Clones

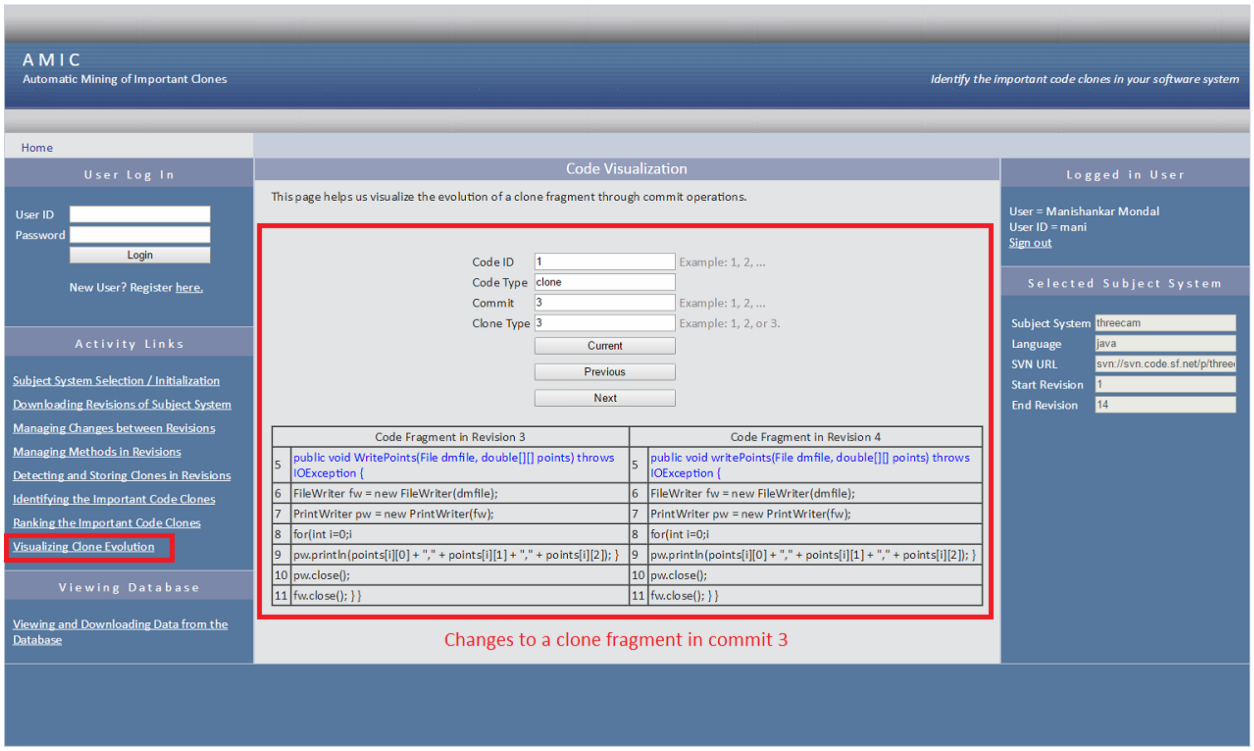
Clone Search Results (Download clone search results)

Pair ID	Clone Pair	Importance for refactoring or tracking	Importance for refactoring	Importance for tracking	Bug-proneness	Late Propagation Tendency	Change-proneness
1	13, ThreeCAM/com/threecam/cycles/SurfaceScan.java, 85, 101 14, ThreeCAM/com/threecam/cycles/SurfaceScan.java, 105, 121	1	1	0	1	0	1
2	11, ThreeCAM/com/threecam/cycles/SurfaceScanConfigFrame.java, 82, 95 12, ThreeCAM/com/threecam/gui/ToolSpecificationFrame.java, 97, 108	1	1	0	0	0	2
3	2, ThreeCAM/com/threecam/gui/DepthmapFrame.java, 138, 146 3, ThreeCAM/com/threecam/gui/DepthmapFrame.java, 146, 152	1	1	0	0	0	1
4	10, ThreeCAM/com/threecam/gui/MainFrame.java, 404, 412 1, ThreeCAM/com/threecam/formats/Points.java, 5, 11	1	1	0	0	0	1

User = Manishankar Mondal
User ID = mani

Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.21: Ranking code clones of a subject system


AMIC
Automatic Mining of Important Clones
Identify the important code clones in your software system
Home
User Log In
Code Visualization
Logged in User
Activity Links
Viewing Database
Selected Subject System
Login
Current
Previous
Next
Changes to a clone fragment in commit 3

User ID
Password

New User? Register here.

Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution

Viewing and Downloading Data from the Database

This page helps us visualize the evolution of a clone fragment through commit operations.

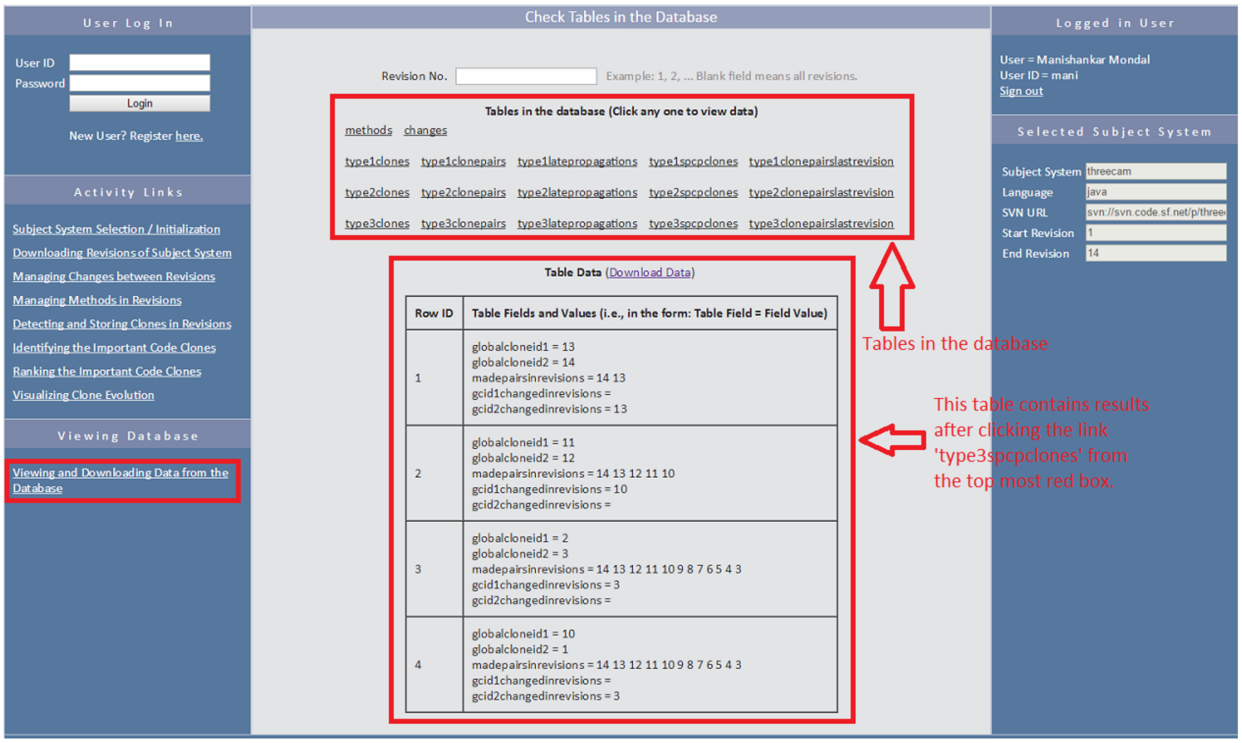
Code ID	1	Example: 1, 2, ...
Code Type	clone	
Commit	3	Example: 1, 2, ...
Clone Type	3	Example: 1, 2, or 3.

Code Fragment in Revision 3		Code Fragment in Revision 4	
5	public void WritePoints(File dmfile, double[][] points) throws IOException {	5	public void writePoints(File dmfile, double[][] points) throws IOException {
6	FileWriter fw = new FileWriter(dmfile);	6	FileWriter fw = new FileWriter(dmfile);
7	PrintWriter pw = new PrintWriter(fw);	7	PrintWriter pw = new PrintWriter(fw);
8	for(int i=0;i	8	for(int i=0;i
9	pw.println(points[i][0] + "," + points[i][1] + "," + points[i][2]); }	9	pw.println(points[i][0] + "," + points[i][1] + "," + points[i][2]); }
10	pw.close();	10	pw.close();
11	fw.close(); } }	11	fw.close(); } }

User = Manishankar Mondal
User ID = mani
Sign out

Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.22: Visualizing clone evolution


User Log In
Check Tables in the Database
Logged in User
Activity Links
Viewing Database
Selected Subject System
Login
Tables in the database
This table contains results after clicking the link 'type3spcpclones' from the top most red box.

User ID
Password

New User? Register here.

Subject System Selection / Initialization
Downloading Revisions of Subject System
Managing Changes between Revisions
Managing Methods in Revisions
Detecting and Storing Clones in Revisions
Identifying the Important Code Clones
Ranking the Important Code Clones
Visualizing Clone Evolution

Viewing and Downloading Data from the Database

Revision No. Example: 1, 2, ... Blank field means all revisions.

Tables in the database (Click any one to view data)

methods changes

type1clones type1clonepairs type1latepropagations type1spcpclones type1clonepairslastrevision

type2clones type2clonepairs type2latepropagations type2spcpclones type2clonepairslastrevision

type3clones type3clonepairs type3latepropagations type3spcpclones type3clonepairslastrevision

Table Data (Download Data)

Row ID	Table Fields and Values (i.e., in the form: Table Field = Field Value)

1	globalcloneid1 = 13
	globalcloneid2 = 14
	madepairsinrevisions = 14 13
	gcid1changedinrevisions =
	gcid2changedinrevisions = 13

2	globalcloneid1 = 11
	globalcloneid2 = 12
	madepairsinrevisions = 14 13 12 11 10
	gcid1changedinrevisions = 10
	gcid2changedinrevisions =

3	globalcloneid1 = 2
	globalcloneid2 = 3
	madepairsinrevisions = 14 13 12 11 10 9 8 7 6 5 4 3
	gcid1changedinrevisions = 3
	gcid2changedinrevisions =

4	globalcloneid1 = 10
	globalcloneid2 = 1
	madepairsinrevisions = 14 13 12 11 10 9 8 7 6 5 4 3
	gcid1changedinrevisions =
	gcid2changedinrevisions = 3

User = Manishankar Mondal
User ID = mani
Sign out

Subject System	threecam
Language	java
SVN URL	svn://svn.code.sf.net/p/three
Start Revision	1
End Revision	14

Figure B.23: Viewing and downloading table data from database