On the Stability of Software Clones: A Genealogy-Based Empirical Study

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

By
Manishankar Mondal

Permission to Use

Abstract

Clones are a matter of great concern to the software engineering community because of their dual but contradictory impact on software maintenance. While there is strong empirical evidence of the harmful impact of clones on maintenance, a number of studies have also identified positive sides of code cloning during maintenance. Recently, to help determine if clones are beneficial or not during software maintenance, software researchers have been conducting studies that measure source code stability (the likelihood that code will be modified) of cloned code compared to non-cloned code. If the presence of clones in program artifacts (files, classes, methods, variables) causes the artifacts to be more frequently changed (i.e., cloned code is more unstable than non-cloned code), clones are considered harmful. Unfortunately, existing stability studies have resulted in contradictory results and even now there is no concrete answer to the research question "Is cloned or non-cloned code more stable during software maintenance?"

The possible reasons behind the contradictory results of the existing studies are that they were conducted on different sets of subject systems with different experimental setups involving different clone detection tools investigating different stability metrics. Also, there are four major types of clones (Type 1: exact; Type 2: syntactically similar; Type 3: with some added, deleted or modified lines; and, Type 4: semantically similar) and none of these studies compared the instability of different types of clones. Focusing on these issues we perform an empirical study implementing seven methodologies that calculate eight stability-related metrics on the same experimental setup to compare the instability of cloned and non-cloned code in the maintenance phase. We investigated the instability of three major types of clones (Type 1, Type 2, and Type 3) from different dimensions. We excluded Type 4 clones from our investigation, because the existing clone detection tools cannot detect Type 4 clones well. According to our in-depth investigation on hundreds of revisions of 16 subject systems covering four different programming languages (Java, C, C#, and Python) using two clone detection tools (NiCad and CCFinder) we found that clones generally exhibit higher instability in the maintenance phase compared to non-cloned code. Specifically, Type 1 and Type 3 clones are more unstable as well as more harmful compared to Type 2 clones. However, although clones are generally more unstable sometimes they exhibit higher stability than non-cloned code. We further investigated the effect of clones on another important aspect of stability: method co-changeability (the degree methods change together). Intuitively, higher method co-changeability is an indication of higher instability of software systems. We found that clones do not have any negative effect on method co-changeability; rather, cloning can be a possible way of minimizing method co-changeability when clones are likely to evolve independently. Thus, clones have both positive and negative effects on software stability. Our empirical studies demonstrate how we can effectively use the positive sides of clones by minimizing their negative impacts.

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. Christopher Dutchyn, Dr. Derek Eager, and Dr. Abdullah Mamun 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 Md. Saidur Rahman, Khalid Billah, Masudur Rahman, Farouq Al. Omari, Shamima Yeasmin, Minhaz Fahim Zibran, Muhammad Asaduzzaman, Md. Sharif Uddin, and Mohammad Asif Ashraf Khan.

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 Janice Thompson, Gwen Lancaster, Maureen Desjardins, and Heather Webb.

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 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.

I dedicate this thesis to my mother, Gita Rani Mondal, whose inspiration helps me to accomplish every step of my life.

Contents

List of Tables

List of Figures

List of Abbreviations

AA	Average Age
ALCD	Average Last Change Date
CCMS	Co-changeability of Methods
CD	Change Dispersion
COMS	Connectivity of Co-changed Method Groups
LOC	Lines of Code
LS	Life Span
MF	Modification Frequency
MI	Method Instability
MMCG	Methods appearing in Multiple Commit Groups
MOR	Modification Occurrence Rate
MP	Modification Probability
MWW	Mann Whitney Wilcoxon
OICR	Overall Instability of Code Region
PCRM	Proportion of Code Region Modified
UL	Unstable Length
UP	Unstable Proportion
UPHL	Unstable Proportion per Hundred LOC

Chapter 1

Introduction

1.1 Motivation

Software maintenance is one of the most important phases of the software development life cycle. The maintenance phase consists of those changes that are made to a software system after it has been deployed to the client upon client acceptance. According to some recent statistics [95] between 40 to 80 percent of the annual software expenditure is being spent for maintaining existing software systems. During evolution and maintenance, changes to a software system are unavoidable but are sometimes very risky. A particular change without proper awareness of its consequences might cause a software system to enter into an inconsistent state. Frequent changes to a program artifact (e.g. file, class, method) which is related to (logically coupled with) several other program entities have the potential to introduce bugs and temporarily hidden inconsistencies in the related entities [85]. Code clones have been said to be responsible for introducing additional change challenges [74, 75].

Code cloning is a common practice during software development and maintenance. Reuse of code fragments with or without modifications by copying and pasting from one location to another is frequently done by software developers. This results in the existence of same or similar code blocks in different components of a software system. Code fragments that are exactly the same or very similar to each other are known as clones. There are four types of clones: Type 1, Type 2, Type 3, and Type 4 by increasing degree of difference among the code fragments. Although each of these will be precisely defined in the next chapter, the precise distinction do not concern us here. Generally, there are a number of reasons behind code cloning. In addition to copy-paste activity, some other issues including programmers' behavior like laziness and tendency to repeat common solutions, technology limitations, code understandability and external business forces have influences on code cloning [55]. Whatever may be the causes behind cloned code, the impact of clones is of great concern from the maintenance programmer's point of view.

1.1.1 Problem Statement

Code cloning is a frequently-voiced controversial issue. Due to the contradictory claims of some recent empirical studies [5, 7, 8, 11, 40, 46, 47, 49, 52, 55, 59, 63–65, 70, 73–75, 94, 104, 106, 114] cloning has become one of the prime concerns to the software engineering community. Some studies [7, 8, 46, 70, 73, 75, 87] have strongly argued against clones by showing concrete empirical evidences of their harmful impacts (such as propagation of hidden bugs, unintentional inconsistent changes, higher instability) on software maintenance and evolution. On the other hand, a number of studies [5, 11, 47, 49, 59, 63, 94, 104] have revealed several positive impacts of code cloning including faster development and reduction of maintenance effort and costs. Recently, software researchers are measuring the stability of cloned code and comparing it with that of non-cloned code to determine which one (cloned code or non-cloned code) is more stable in the maintenance phase. The underlying idea is that if cloned code appears to get more frequent modifications (exhibits higher instability) than non-cloned code in the maintenance phase, clones can be considered to have negative impacts on software evolution. Otherwise, clones should not be considered as harmful. However, the stability related studies [40, 47, 65, 74, 75] could not come to a consensus and thus, the long-lived research question 'Is cloned or non-cloned code more stable during software maintenance?' does not have any concrete answer yet.

A possible reason behind these contradictory outcomes is that different studies were performed on different experimental setups investigating different sets of subject systems considering limited aspects of stability. Also, none of these studies investigated the variability of impacts of different types of clones.

1.1.2 Our contributions

Focusing on the above explanations of contradictory outcomes, we performed a series of empirical studies.

First study

In our first study we investigated eight stability measurement metrics (six metrics were proposed by five pre-existing studies [40, 47, 65, 74, 75], and the remaining two metrics are our proposed new ones) on the same experimental setup using two clone detection tools (CCFinder [19] and NiCad [96]). We investigated the instabilities of three major types of clones (Type 1, Type 2 and Type 3) in the maintenance phase with promising outcomes that have the potential to assist in better maintenance of software systems. Our empirical studies involve the automatic inspection of hundreds of revisions of sixteen diverse subject systems written in three different programming languages (Java, C, C#). According to our experimental results on eight metrics, clones are generally more unstable than non-cloned code in the maintenance phase. Our results also suggest that Type 1 and Type 3 clones are potential threats to the stability of software systems because they exhibit higher instability in the maintenance phase compared to the Type 2 clones. We also observe that clones in the subject systems written in Java and C exhibit higher instability compared to clones in C# systems. Therefore, programmers should be more careful while working with Type 1 and Type 3 clones. More specifically, the programmers should be conscious that:

(1) the code fragments to be copied do not contain any bugs and

(2) changes to these two more unstable types of clones (Type 1 and Type 3) have been properly propagated to other similar fragments.

Finally, limiting Type 1 clones can limit many of Type 3 clones, because many Type 3 clones are created from Type 1 clones.

Second study

From our study on eight metrics, we found that although clones are more unstable than non-cloned code in general, sometimes clones exhibit higher stability compared to non-cloned code in the maintenance phase. To further investigate this matter we performed another in-depth empirical study on one of our proposed metrics, change dispersion, involving manual analysis on all of the changes occurred to the clones of our candidate subject system CTAGS [33] during its evolution. According to this study, the presence of clones in the methods of the subject systems written in Java and C has the potential to increase method instability. Also, half of the changes occurring to the clones are made so that the clone fragments in a particular clone class remain consistent. Consistency ensuring changes mainly occur to the Type 3 clones.

Third study

As from the previous studies we found that clones can potentially increase system instability, we further investigated the effect of clones on the co-changeability of program artifacts (such as: files, methods, classes). Co-changeability is the degree to which program artifacts change together. Co-changeability is another aspect of stability. Intuitively, higher co-changeability among program entities is an indication of higher instability of source code. We limited our investigation on the effect of clones on method (in Object-Oriented programming languages) or function (in procedural programming languages) co-changeability only. In the rest of the thesis we use the term *method* to represent: (i) methods in object-oriented programming languages, and (ii) functions or procedures in procedural programming languages. According to our investigation, clones do not increase method co-changeability. Moreover, method cloning can be a possible way of minimizing method co-changeability only when the cloned methods are likely to evolve independently.

We see that clones have both positive and negative effects of software stability. However, our empirical studies clearly demonstrate how we can use the positive sides of clones (e.g. minimization of method co-changeability using method cloning) by minimizing their negative effects (minimization of highly unstable clone types: Type 1 and Type 3). Thus, our thesis statement is that *we can minimize the negative effects of clones by minimizing Type 1 and Type 3 clones because these two types of clones show higher instability compared to Type 2 clones, also method cloning can be a possible way of minimizing method co-changeability provided that cloned methods will evolve independently.*

From our empirical studies we believe that proper tool support is mandatory for managing clones. During clone refactoring, Type 1 and Type 3 clones should be given higher priority. As Type 2 clones do not appear to be as unstable as the other two types, we do not require extensive refactoring of Type 2 clones. Thus, our findings have the potential to eliminate a considerable amount of refactoring effort being spent for Type 2 clones.

1.2 Related Publications

Several parts of this thesis have been published previously. The list of publications (one journal paper and four conference papers) is given below. The last paper in the list is submitted to a journal. In each of the publications, I am the primary author and the associated empirical studies were conducted by me under the supervision of Dr. Chanchal K. Roy and Dr. Kevin A. Schneider. The other co-authors (in the second and fifth paper in the list) took part in editing and reviewing.

- M. Mondal, C. Roy, and K. Schneider, "An Empirical Study on Clone Stability". *ACM SIGAPP Applied Computing Review (ACR)*, 2012, Volume 12, Issue 3, pp. 20-36. [84]
- 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". The 27th Annual ACM Symposium on Applied Computing (*SAC*), 2012, pp. 1227-1234 (**Best Paper Award**). [83]
- M. Mondal, C. K. Roy, and K. A. Schneider, "Dispersion of Changes in Cloned and Non-cloned Code". ICSE 6th International Workshop on Software Clones (*IWSC*), 2012, pp. 29-35. [87]
- M. Mondal, C. K. Roy, and K. A. Schneider, "Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems". The 22nd annual international conference hosted by the Centre for Advanced Studies Research, IBM Canada Software Laboratory (*CASCON*), 2012, pp. 205 – 219. [85]
- 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". The 19th International Conference on Program Comprehension (*ICPC*), 2011, pp. 242 – 245. [86]
- 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*, 2012, 42 pp. (Submitted)

The published journal article (the first one in the above list) is an extension of the second paper, published in *ACM SAC 2012*, mentioned in the above list. We extended the journal article by empirically studying four new stability measurement metrics that were not investigated by any previous studies.

In Chapter 2 we discuss some background related to code clones, their impacts and detection techniques. Chapter 3 describes method genealogy and our proposed concurrent framework for extracting method genealogy. Chapter 4 elaborates on our empirical study involving the existing and proposed metrics and investigating the instability of different types of clones in different programming languages. Chapter 5 presents our extended empirical study (submitted to the Science of Computer Programming Journal) on the dispersion of changes in cloned and non-cloned code. Chapter 6 describes our investigation on the co-changeability of methods and finally, in Chapter 7 we conclude our thesis by mentioning our future work.

Chapter 2
Background

2.1 Code Clone

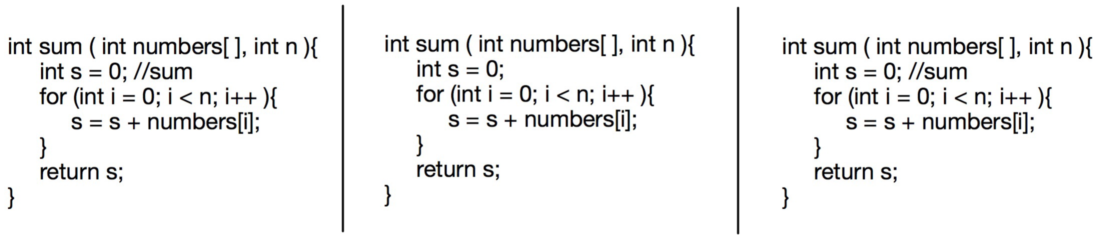
Code clones are exactly or nearly similar code fragments in a code-base. These are often created as a result of the copy-paste activities performed by the programmers during the evolution of a software system. A group of clone fragments that are similar to one another form a *clone class.* There might be different other causes for clones including: programmer's behaviour like laziness and tendency to repeat common solutions, technology limitations [9, 92] (e.g. lack of reusing mechanism in programming languages), code evolvability and code understandability [86]. External business forces might also have influences on code cloning [55]. Whatever may be the causes behind cloned code, the impacts of clones are of great concern from the maintenance point of view. In the following sections we describe the types, detection techniques and impacts of clones in brief.

2.2 Types of Clones

According to the literature there are four types of clones: Type 1, Type 2, Type 3 and Type 4. These are defined as follows.

Type 1 clone: Type 1 clone fragments are exactly similar code fragments with only differences in comments and/or code formatting. An example of a Type 1 clone class with three clone fragments is shown in Fig. 2.1.

Type 2 clone: Type 2 clone fragments retain their syntactic similarity. Type 2 clones are created by



```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    int s = 0;
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

Figure 2.1: A Type 1 clone class with three clone fragments

```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int doSum ( int num[ ], int n ){
    int sum = 0;
    for (int i = 0; i < n; i++ ){
        sum = sum + num[i];
    }
    return sum;
}
```

```
int add ( int a[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + a[i];
    }
    return s;
}
```

Figure 2.2: A Type 2 clone class with three clone fragments

```
int sum ( int numbers[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int doSum ( int num[ ], int n ){
    int sum = 0;
    for (int i = 0; i < n; i++ ){
        sum += num[i];
    }
    return sum;
}
```

```
int add ( int a[ ], int n ){
    int s = 0; //sum
    for (int i = 0; i < n;  ){
        s = s + a[i];
        i++;
    }
    return s;
}
```

Figure 2.3: A Type 3 clone class with three clone fragments

pasting a copied code fragment and changing in identifier names and types. In Fig. 2.2 we present an example of a Type 2 clone class with three clone fragments.

Type 3 clone: Type 3 clone fragments are created because of the additions and deletions of lines in Type 1 or Type 2 clone fragments. Fig. 2.3 shows a Type 3 clone class containing three clone fragments.

Type 4 clone: Semantically similar code fragments are termed as Type 4 clones. In general, Type 4 clone fragments perform the same task but with different implementations. An example of a Type 4 clone class with two semantically similar code fragments is shown in Fig. 2.4.

Existing studies [10, 50, 56, 67, 71, 81] show that the proportion of cloned code in both open-source and industrial software systems can vary from 5% to 20%. However, one study [32] shows that the percentage of cloning in a COBOL system (investigated in the study) was about 50% of the entire codebase.

```
int sum ( int numbers[ ], int n ){
    int s = 0;
    for (int i = 0; i < n; i++ ){
        s = s + numbers[i];
    }
    return s;
}
```

```
int sum ( int numbers[ ], int n ){
    if(n == 1)
        return numbers[n-1];
    else
        return numbers[n-1]+sum(numbers,n-1);
}
```

Figure 2.4: A Type 4 clone class with two clone fragments

2.3 Impacts of Clones

Clones have become controversial in the realm of software engineering research because of their dual, and contradictory impacts during software maintenance. While there are several in-depth studies [5, 11, 47, 49, 55, 59, 63, 94, 104] showing the positive impacts of clones in terms of faster development and reduction of maintenance cost and effort, other studies [7, 8, 46, 52, 70, 73–75, 87] have shown strong empirical evidences of potential negative impacts of clones on software maintenance in terms of hidden bug propagation and unintentional inconsistent changes.

To see the first negative impact, we consider a code fragment containing a bug. This code fragment is copied and pasted to several other places without the awareness of this bug, resulting in increased modifications (more modifications) to the source code after the discovery of the bug in any one of the clone fragments. Repair of the bug should take place in each fragment. In the second case, if a clone fragment is updated with some modifications, the same update might need to be propagated to other clone fragments to ensure consistency. If this update is not propagated because of programmer unawareness, inconsistencies occur during code evolution. Elimination of these inconsistencies might require the propagation of the update to the other fragments causing an increased amount of modifications.

Thus we see that the two main negative impacts of clones cause increased modifications to the source code. In other words, each of these two negative impacts decreases the stability of cloned code as well as the whole software system.

2.4 Stability

The term *stability* [40, 47, 64, 65, 74, 83] has recently been introduced to quantify the change-proneness of a software system. In general, higher *change-proneness* indicates lower *stability* (or higher instability). *Stability* and its aspects are discussed in the following section.

Definition: According to Pan [91], *stability* of a code region refers to the resistance to the amplification of changes in that code region. In other words, the *stability* of a particular code region of a software system quantifies the extent to which that code region remains stable during evolution and maintenance of the software system.

Different studies have proposed different measurement metrics for stability. By comparing the stability of two kinds of code (e.g. cloned and non-cloned regions) during evolution and maintenance we can determine which code region requires more effort. Generally, a code region with higher instability (or higher change-proneness) requires greater amount of maintenance effort compared to the other [40].

Recently, software researchers are measuring the stability of cloned and non-cloned regions of a software system separately and then comparing them to determine which code region (cloned or non-cloned) generally exhibits higher instability during evolution and maintenance [40, 47, 65, 74]. On one hand, if cloned code remains more unstable compared to non-cloned code in the maintenance phase then it can be suggested that cloned code requires more effort and cost to be maintained than non-cloned code. In this case, cloned code can be considered as harmful. On the other hand, if cloned code does not appear more unstable than non-cloned code then, clones do not require more maintenance effort compared to non-cloned code. In this case, clones can not necessarily be considered as harmful.

The existing and our proposed stability related metrics are discussed below in brief by grouping them into two broad categories: modification probabilities and age. These metrics will be discussed in detail in Chapter 4.

2.4.1 Metrics quantifying the modification probability of source code

Some studies [40, 47, 74, 84, 87] have measured stability by quantifying the changes to a code region using three general approaches:

(i) determination of the ratio of the number of lines added, modified and deleted to the total number of lines in a code region (cloned or non-cloned) [40, 74, 84] and

(ii) determination of the frequency of modifications to the cloned and non-cloned code [47] with the hypothesis that the higher the modification frequency of a code region is the less stable it is.

(iii) measuring the dispersion of changes in a code region (cloned or non-cloned) [87]. Change dispersion in a code region quantifies how much scattered the changes in that code region are.

2.4.2 Metrics related to the age of LOCs

There are mainly two approaches related to the measurement of the ages of LOCs.

(i) The first one [65] determines the *average last change date* of LOCs (elaborated in Chapter 4 Section 4.3.3) in a particular code region (cloned or non-cloned code). The hypothesis is that the older the average last change date of a code region is, the more stable it is.

(ii) The second one [83], a variant of the previous one, determines the *average ages* of LOCs in cloned and non-cloned regions.

2.5 Clone Detection Techniques

In a previous section (Section 2.3) we discussed some negative impacts of clones on software maintenance. Thus, it is important to detect clones in the software systems so that we can manage them properly. Different techniques for detecting clones already exist. These are discussed below in brief.

- In the textual approaches [22, 51, 69, 77, 80, 96, 115, 116] the source code is considered as a sequence of characters. Thus, these approaches are independent of programming languages and work even in those cases where the source code is not compilable. However, most of these approaches [51, 69, 77, 80, 116] do not apply any source code transformation step before comparing two separate code fragments. As a result these do not work well when the same syntactical structure is represented differently in different places. To identify high level concept clones Marcus and Maletic [80] applied latent semantic indexing (LSI) technique to source text. However, they considered only comments and identifiers disregarding the entire source code.

NiCad [96] is a recently introduced hybrid clone detection tool that detects clones through the following two steps.

(1) Tree-based structural analysis based on lightweight parsing to implement flexible pretty-printing, code normalization, source transformation and code filtering.

(2) Text based comparison of the code blocks obtained from the preprocessed codebase that we get after the first step.

Application of the first step helps NiCad to eliminate the already discussed drawbacks of the previous textual approach.

- In case of lexical approaches, the source code is considered as a sequence of tokens. Such approaches are best suited when changes between clone fragments are small such as identifier renaming. Dup [6], CCFinder [54], iClones [41] are some of the examples of token-based clone detectors.

- Tree-based approaches [10, 20, 36, 50, 60, 62, 68, 113] at first convert the source code into parse trees or abstract syntax trees and then use different tree-matching algorithms to detect clones of similar subtrees. The time complexity of tree-based approaches is higher compared to the time complexity of text based and token based techniques.

- In the graph-based approaches [34, 43, 45, 66], the source code is represented as a program dependency graph (PDG). These approaches are robust for simple modifications (e.g. reordering of lines) of code clones. However, these approaches are programming language dependent, require syntactically correct programs, and have high time complexity.

- In the metrics-based approaches [24, 37, 81, 107] a number of metrics (e.g. names, layouts, expressions, and control flow of functions) are calculated for code fragments at a particular level of granularity such as functions/methods, classes, or any syntactic unit. Such approaches have also been used to find duplicate web pages or finding clones in web documents [15, 76].

Clone detection technique is not limited to detecting source code clones only. It has also been used for detecting clones in binary executables [112], intermediate representation of programming languages for cross language clone detection [3], Java Bytecode [57, 58, 103], assembly language instructions [25, 26], large models used in model-based development of control systems [27, 28], MATLAB/Simulink models [2, 88, 93], software requirement specification [29, 72] and UML sequence diagrams [110].

In this thesis we have used two clone detection tools: NiCad and CCFinderX to detect clones in source code. There are a number of reasons behind using these tools. NiCad can detect three types of clones: Type 1, Type 2, and Type 3 separately. No other existing techniques and tools can provide us these three types of clones individually. The other tools can only provide us clone results by combining different types of clones. NiCad helped us to investigate the comparative stability of three types of clones. Here, we should mention that no existing clone detection tool can detect Type 4 clones.

CCFinderX is a widely-used token-based clone detector. While tree-based and graph-based techniques require the source code to be syntactically correct, CCFinderX can detect clones even in presence of syntax errors. CCFinderX is faster than any tree-based and graph-based clone detector.

2.6 Clone Management

As there is empirical evidence of the negative impact of clones, it important to refactor the clone classes existing in a code base. However, refactoring all clone classes is not feasible because the clone fragments of Type 3 clone classes often evolve independently. So, we have to manage clones properly to facilitate software maintenance activities. A number of clone management techniques have already been proposed by different researchers [31, 44, 53, 67, 82, 119–121]. We briefly describe these techniques in the following subsections.

- Preventive clone management techniques primarily focus on how to minimize clone creation rather than detecting and removing them. Lagüe et al. [67] proposed two ways of using a clone detection tool to avoid clones during software development process. These are: (1) *preventive control* and (2) *problem mining*. While the first approach ensures that each of the new functions that is going to be added to a system is a non-cloned snippet, the second way involves monitoring of all source code changes submitted to the central source code repository.

- Corrective clone management techniques [44, 53] involve the refactoring of suspicious clone fragments to reduce potential source of errors and to increase system understandability. Such techniques are effective for those software systems where clones were not managed from the beginning.

- Compensative clone management technique focuses on facilitating the evolution of those clone classes that can not be eradicated from the system. *Simultaneous editing* [82] is one of the first approaches regarding compensative clone management. This approach helps the developers to make the same change to occur to all fragments of a given clone class at the same time to ensure consistency. Duala-Ekoko and Robillard [31] developed a tool called *Clone Tracker* with two facilities: (1) notification of the developers when they attempt to change a clone fragment of a particular clone class and (2) simultaneous editing.

2.7 Conclusion

This chapter discusses on different types of clones, clone impact (both positive and negative), clone detection tools and techniques, and clone management approaches. We have seen that clone impact is a controversial issue. Measuring the stability of cloned and non-cloned code is a recently introduced approach for assessing the impact of clones on software maintenance. We have seen that the existing studies regarding clone stability could not come to a consensus and there is no concrete answer to the long lived research question 'Is cloned or non-cloned code more stable in the maintenance phase?'. We performed a series of empirical studies to find the answer to this question. These studies have been elaborated in the following chapters.

Chapter 3
Method Genealogy

A number of clone-related studies [74,75,87] have investigated the impacts of clones on methods. Extraction of method genealogies is one of the most important and time-consuming steps in each of these studies. In this chapter we discuss method genealogy and an existing approach [75] that we have used for detecting and extracting method genealogies. However, the existing approach takes much time if it is executed sequentially. In order to facilitate faster extraction of method genealogies we have proposed a parallel framework on which we apply the existing approach [75]. This chapter elaborates on our proposed parallel framework too. The empirical studies in the following chapters begin with our method-genealogy extraction step.

Here, we should mention that *method* is an object-oriented term. However, in this thesis the term *method genealogy* is not only limited to the object-oriented programming languages. The process of extracting method genealogies described in this chapter also extracts the genealogies of functions or procedures (in case of procedural programming languages). In the rest of the thesis, the term *method* also represents *function* or *procedure*.

3.1 Method Genealogy

During the evolution of a software system a particular method might be added to a particular code revision and can remain alive in multiple future revisions. Each of these revisions has separate instance of this method. *Method genealogy* identifies all of these instances as belonging to the same method. Inspecting the genealogy of a particular method we can determine whether the method has received any change during maintenance. It is possible that a particular method will remain unchanged during the maintenance phase. In that case, all the instances of this method (in different revisions) will be the same.

In Fig. 3.1 we show five examples of method genealogies. We see that there are four revisions in total. A commit operation on a particular revision creates the immediate next revision. An example method genealogy (the top most one) in this figure consists of the method instances: m11, m21, m31, and m41. These are the instances of the same method and each instance belongs to a particular revision. A commit operation applied on a revision might not change all the method instances in that revision. If the commit operation on Revision-1 makes changes to the method instance m11, m21 will be different from m11. Otherwise, m11 and m21 will be the same.

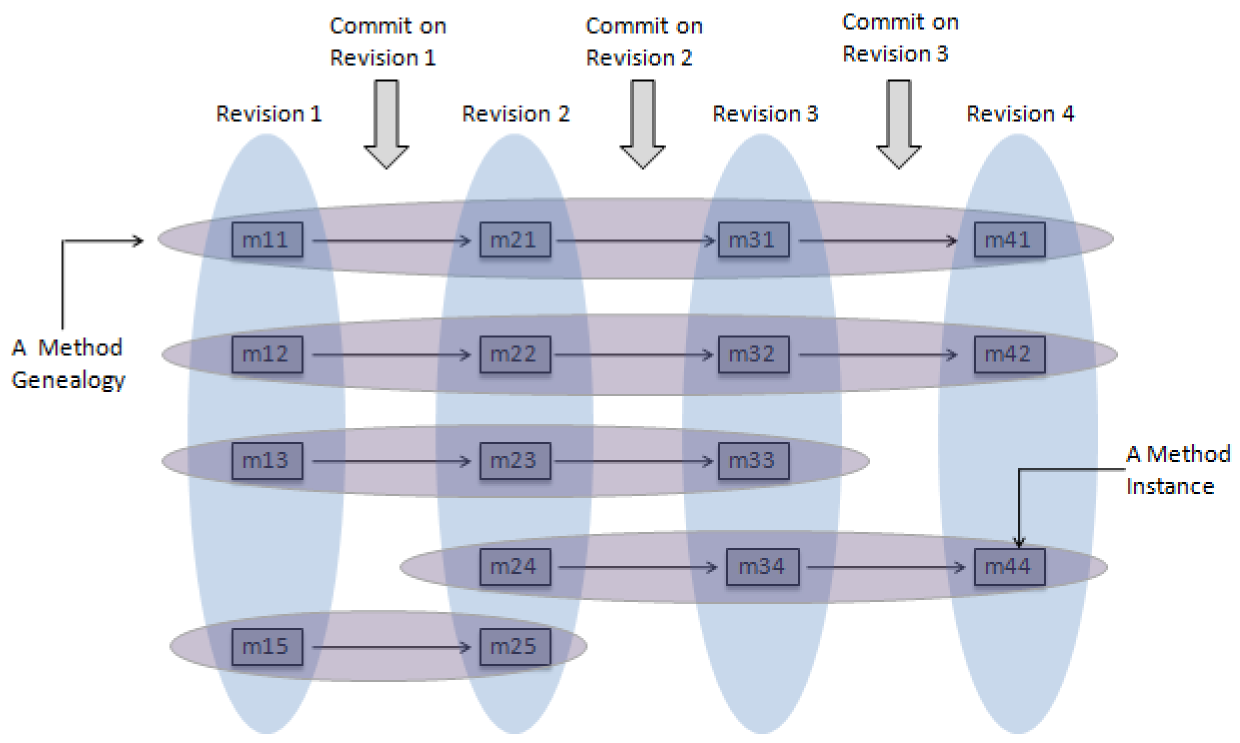


Figure 3.1: Example of Method Genealogies

3.2 Detection of Method Genealogies

Lozano and Wermelinger [75] proposed an approach for detecting method genealogies. We have followed this approach for detecting method genealogies in our studies. As proposed by Lozano and Wermelinger, there are mainly two steps in detecting method genealogies from a given set of revisions of a subject system. These are:

- **Method Detection:** Detection of methods from each of the given revisions and
- **Method Mapping:** Making a one-to-one correspondence between the methods in every two consecutive revisions.

For detecting methods we used Ctags [33]. Methods are detected along with their signatures and location information. The location information consists of the file, package (in case of Java), and class (in case of Java and C#) in which the method exists. After detecting the methods in all revisions we perform method mapping. Method mapping means finding a one-to-one correspondence between the methods of every two consecutive revisions. Method mapping is performed in the following way.

Suppose, m_i is a method in revision R_i. For finding the corresponding instance of this method in revision R_{i+1} we consider the following two cases.

- Case 1: If a method m_{i+1} in R_{i+1} have the same signature and location information as of m_i, m_{i+1} is an instance of m_i. The contents of these two methods might be the same or different. If the commit operation applied on R_i makes some changes to the method instance m_i, the contents of m_i and m_{i+1} will be different.

- Case 2: For m_i in R_i we might not locate any method in R_{i+1} with the same signature and location information. In that case, we detect two sets of methods in R_{i+1}. The first set S_{s-l} contains those methods that have the same signature but different location, and the second set S_{l-s} contains those methods that have the same location but different signatures. We call the methods in these two sets as the candidate methods. We then compute the similarity between m_i and each of the candidate methods in the first set using the Strike A Match algorithm [111] (An algorithm for determining text similarity) and record the best similarity value and the corresponding candidate method. If this value is above 70% we consider the associated candidate method as the instance of m_i in R_{i+1}. If no candidate method in the first set has a similarity value greater than 70%, we go through the same process with the second set. If no method in the second set has a similarity value greater than 70% then, m_i is considered deleted in $revision_{i+1}$.

After performing the mapping operation between every two consecutive revisions we get the method genealogies. A particular method genealogy, as illustrated in the Fig.3.1, consists of the corresponding method instances in two or more consecutive revisions.

3.3 Our proposed concurrent framework for method genealogy extraction

Extraction of method genealogies is a complex and time-consuming task which can be divided into multiple smaller and independent tasks that can be executed in parallel to reduce the total execution time. We implemented a parallel and distributed framework for this method genealogy extraction algorithm.

We need to emphasize on the following standard parallel execution objectives during the task-division.

(1) The tasks should not be too small so that the processes can spend more time on task completion rather than inter-process communication.

(2) The processes must synchronize among themselves to ensure the consistency of execution.

(3) Task distribution should ensure load balancing.

Focusing on these objectives, we construct our genealogy extraction model as a ‘Manager-Workers’ system [4] where there is a single manager who manages or coordinates the tasks of several workers. At the very beginning of execution, manager divides the whole range of revisions into a number of sub-ranges of equal length. Each sub-range contains multiple consecutive revisions and the count of sub-ranges is equal to the number of workers. The manager then assigns each of the sub-ranges to a particular worker. Each worker is

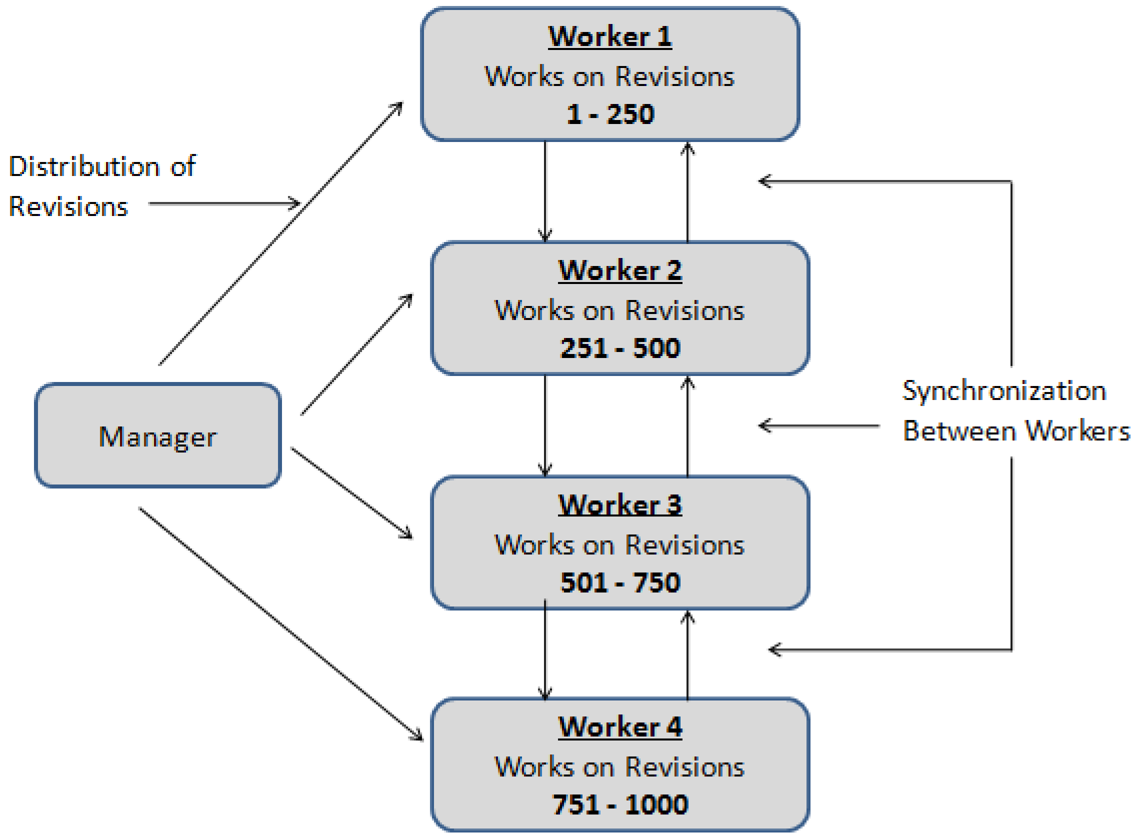


Figure 3.2: Parallel framework for extracting method genealogies

responsible for the extraction and mapping of the methods of the revisions it has been assigned to. To get the final method mapping for the whole range of revisions, the workers need to synchronize among themselves. In the following paragraphs we describe task distribution and synchronization among workers.

The figure 3.2 demonstrates how a manager distributes revisions to the workers. We see that the total number of revisions to be distributed is 1000 and the number of workers is four. Thus, the manager assigns 250 consecutive revisions to each of the workers. However, the number of revisions might not be an exact multiple of the number of available workers. In that case the last worker gets some extra revisions. As an example, if 701 revisions need to be distributed among four workers, each of the first three workers gets 175 consecutive revisions. The fourth worker gets 176 revisions.

The synchronization process between two consecutive workers is described below with an example.

According to Fig. 3.2, two workers Worker 1 and Worker 2 are responsible for revisions with ranges 1 to 250 and 251 to 500 respectively. Each worker will have to complete the extraction and mapping of methods of its respective revisions. Each worker is disciplined in such a way that it at first extracts the methods of i^{th} revision in its range, stores the methods with associated information into a file and then maps the methods remaining in the files resulted from i^{th} and $(i-1)^{th}$ revisions. Then, the worker proceeds with the $(i+1)^{th}$

revision. With this discipline, Worker2 will be able to extract the methods of revision 251 but will not be able to complete the mapping between revisions 250 and 251, because Worker2 does not know whether the methods from revision 250 have been extracted and stored by Worker1. Worker1 and Worker2 are executing in parallel by starting their execution at around the same time and are processing their respective range of revisions beginning with the very first revisions of their ranges. With proper load balancing, it is likely that Worker1 and Worker2 will be processing respectively the revisions 250 and 500 at around the same time. In this case, Worker1 needs to send a message to Worker2 after it has extracted and stored the methods of revision 250. Worker2, in this situation, extracts methods of revisions 251 to 500 and performs mapping on the revisions 252 to 500 and then waits for the message from Worker1. After getting the message, Worker2 performs mapping between revisions 250 and 251. We implemented the concurrent framework in Java programming language using the Actor Architecture platform [1] that provides the asynchronous message passing facility to the manager and the workers.

3.4 Conclusion

This chapter elaborates on method genealogy, an existing approach [75] that we have used for extracting method genealogies, and our proposed parallel framework that facilitates the faster extraction of method genealogies. Each of our empirical studies described in the following chapters includes the method genealogy extraction step.

Chapter 4
Stability Measurement Metrics: Their calculation and Interpretation

4.1 Introduction

Code cloning is a controversial software engineering practice. There is empirical evidence of both positive [5,11,47,49,55,59,63,94,104] and negative [7,8,46,52,70,73–75,87] impact of clones on maintenance. Different studies proposed and investigated different impact assessment metrics. A widely used term to assess the impact of clones on software maintenance is stability [40, 47, 64, 65, 74, 83]. According to the literature, stability of a particular code region measures the extent to which that code region remains stable (less frequently changed) during software evolution. Seven studies defined eight different stability related metrics. Five [40,47,65,74,75] of these studies are pre-existing and the remaining two [83,87] were conducted as part of this research. The eight stability metrics are listed below.

(1) *modification frequency (MF)* of cloned or non-cloned code [47]

(2) *modification probability (MP)* of cloned or non-cloned code [40]. This metric is originally termed as *overall instability* by Göde and Harder [40].

(3) *average last change dates (ALCD)* of cloned or non-cloned LOC [65].

(4) *average age (AvgAge)* of cloned or non-cloned LOC (Our proposed methodology [83])

(5) *impact* of changes of cloned and non-cloned code [75]

(6) *likelihood* of changes of cloned and non-cloned code [75]

(7) *average instability per cloned method (AICM)* due to cloned or non-cloned code [74]. This metric is a composite one which incorporates two proportions: (i) average proportion of cloning in cloned methods (*EPCM*) and (ii) average percentage of changes to the clones in cloned methods (*CPCM*).

(8) *change dispersion* in cloned and non-cloned code (Our proposed methodology [87])

Interpretations of these metrics are briefly discussed below.

- *Modification frequency* is the measurement of how frequently a code region (cloned or non-cloned) gets modified. It focuses on the count of changes ignoring the quantity (or amount) of lines affected by a change.

- *Modification probability* focuses on the count of affected lines.

- *Average last change date* determines how lately a code region (cloned or non-cloned) gets modified.

- *Average age* (slightly different from *average last change date*) calculates how long a cloned or non-cloned LOC remains unchanged on an average.

- *Impact of changes* determines the average number of co-changed methods because of the change of a particular method.

- *Likelihood of changes* quantifies the change probability of a particular cloned or non-cloned method.

- *Average instability per cloned method* determines the average proportion of the number of changes happening in the cloned portions of cloned methods to the total number of changes in the cloned methods. In other words, this is the probability by which changes take place in the cloned portions of the cloned methods.

- Finally, *dispersion of changes* quantifies the extent to which the changes in the cloned or non-cloned regions are scattered over the respective region.

We see that each of these eight existing metrics is directly related to differential changes in cloned and non-cloned regions. This supports the contention that the negative effects of clones in maintenance are directly related to the increased changes in source code. As noted before, there are two main causes for the negative impacts of clones: (i) hidden bug propagation and (ii) unintentional inconsistent changes. Let us at first consider bug propagation. Suppose a code fragment contains a bug which is temporarily hidden and this code fragment is copied by cloning process to several other places without the awareness of the existence of the bug. If any instance of this propagated bug is discovered at a certain stage of evolution, its repair should take place in all code segments where it has been propagated. Thus, bug propagation by cloning causes increased modifications to the respective clones during evolution. Secondly, a new change made in a clone fragment might need to be propagated to other clones falling in the same clone family to maintain consistency. Whether such changes propagate consistently or inconsistently, there is no doubt that they increase efforts during software evolution. Thus, we see that the negative impacts of clones are directly related to the higher changes in the cloned code. In other words, negative impacts of clones increase software instability.

Hence, if we can identify the changes occurring in the cloned and non-cloned regions of a software system and can make a comparative analysis of these changes, we will be able to understand the real impact of clones on maintenance for that software system. From this assumption we limited our target list of metrics (as well as studies [40,47,65,74,75,83,87]) to only those that represent different aspects of stability. We did not consider those studies that aim to identify whether clones introduce bugs or are maintained consistently, or not [5,11,52,55,63,114].

Table 4.1: Research Questions (RQ)

Research Questions Corresponding to Eight Metrics		
	Research Questions	**Metrics and Studies**
RQ1	Which code changes more frequently, cloned or non-cloned?	Modification Frequency (MF), Hotta et al. [47]
RQ2	Which code exhibits higher modification probability, cloned or non-cloned?	Modification Probability (MP), Göde and Harder [40]
RQ3	Which code changed more recently, cloned or non-cloned?	Average Last Change Date (ALCD), Krinke [65]
RQ4	Which code remains unchanged for greater lengths of time, cloned or non-cloned?	Average Age (AA), Our study [83]
RQ5	Which method exhibits higher impact (elaborated in Section 4.3.5) of changes in it, cloned or non-cloned?	Impact, Lozano and Wermelinger [75]
RQ6	Which method is more likely to change, cloned or non-cloned?	Likelihood, Lozano and Wermelinger [75]
RQ7	Which code in partially cloned methods exhibits higher average instability, cloned or non-cloned?	Average Instability per Cloned Method (AICM), Lozano and Wermelinger [74]
RQ8	Which code gets more scattered changes, cloned or non-cloned?	Change Dispersion (CD), Our study [87]
Research Questions Corresponding to Drawbacks of Existing Studies		
	Research Questions	**Drawbacks**
RQ9	Do different types of clones exhibit different stability?	DES 3 (Section 4.1.1)
RQ10	Do clones of different programming languages show different stability?	DES 4 (Section 4.1.1)
DES = Drawback of Existing Studies		

4.1.1 Problem Identification

Many stability studies [40, 47, 64, 65, 74, 75, 83] tried to identify, analyze, and compare the changes happening in the cloned and non-cloned code of different software systems. However, these studies did not agree about the comparative stability of cloned and non-cloned code. As a result, there is no concrete answer to the long lived research question: 'Is cloned code really stable in the maintenance phase?'. To illuminate this question further, we investigated each of the prior stability-related studies. We identified the following drawbacks in the existing studies.

(1) Lack of a common framework: Different studies were conducted on different experimental setups, more specifically

- on different sets of subject systems
- using different clone-detection tools with different parameters
- on releases or different sets of revisions (of subject systems) taken at different intervals. Considering revisions at particular time-intervals has the potential to disregard a significant portion of changes occurred to the code base during those intervals.
- inconsistent preprocessing of subject systems.

Thus, different studies may have different outcomes.

(2) Investigation on insufficient metrics: Different studies investigated different subset of metrics. However, a complete assessment of impacts requires the assessment of all of the existing metrics on the same experimental settings.

(3) Lack of investigation on different types of clones: None of these studies except [83] could draw a clear comparison among the impacts of different clone types because the clone detection tools used in these studies cannot detect different types of clones separately. Such a comparison is very important, because this can suggest us to concentrate on more vulnerable clone-types leaving others alone and can thus reduce a significant amount of refactoring efforts being spent for non-vulnerable clone types.

(4) Lack of programming-language centered investigation: None of the existing studies investigated whether the same clone types in different programming languages behave in different ways and show different impacts as well. This information can help software developers to be more careful while developing projects with more vulnerable (in terms of clone instability) programming languages.

(5) Lack of system diversity: Most of the studies have drawn conclusions without investigating a wide variety of subject systems.

4.1.2 Our Contribution

Focusing on the above issues we perform an in-depth empirical study where we evaluate all known (eight in total) stability measurement metrics (proposed in the studies [40,47,65,74,75,83,87]) on the same experimental setup which we term as a uniform framework. The stability metrics and the corresponding studies have already been listed in the introduction of this chapter. Different metrics were calculated following different techniques. We term these techniques as methodologies in the rest of the thesis. We implement these methodologies (seven methodologies in total from seven studies [40, 47, 65, 74, 75, 83, 87]) on our uniform framework and apply them on twelve subject systems of diversified sizes, application domains, purposes and implementation languages. We mentioned that five studies [40, 47, 65, 74, 75] are pre-existing. Four [40, 65, 74, 75] of these investigated only a small number of Java systems (two systems in [40], three systems in [65], five systems in each of [75] and [74]). The remaining study [47] investigated fifteen subject systems covering four programming languages. However, this study does not investigate which programming languages have more unstable clones.

We implemented the candidate methodologies for calculating the metrics using two clone detection tools: NiCad [96], CCFinderX [19]. We analyzed our experimental results from four different dimensions: (1) implementation language, (2) clone-types, (3) subject systems, and (4) clone detection tools to find the answer to the central research question 'Is cloned or non-cloned code more stable during software maintenance?'. However, we decompose this central question into eight questions corresponding to the eight metrics. These questions are mentioned in Table 4.1. The last two questions in this table address the third and fourth drawbacks.

For answering the last two questions we defined two null hypothesis as stated below.

Null hypothesis 1 (Corresponds to RQ9): *There is no significant difference among the stabilities of different types of clones.*

Null hypothesis 2 (Corresponds to RQ10): *There is no significant difference among the stabilities of clones of different programming languages.*

We performed two-tailed Fisher's exact tests using the implementation at [35] on our observed data for inspecting the acceptance or rejection of these two hypotheses. We also answered each of the other eight questions with many interesting outcomes in the corresponding subsections of the result section. In general, we can summarize the results that we got for eight metrics in the following way.

- Cloned code is generally (not always) more unstable than non-cloned code in the maintenance phase.
- Both Type 1 and Type 3 clones appear to be more unstable compared to Type 2 clones in the maintenance phase. Thus, Type 1 and Type 3 clones can be regarded as potential threats to the maintenance phase according to our analysis.
- Clones in Java and C languages are more harmful than the clones in C#.
- It seems that object-oriented programming languages promote more cloning compared to procedural programming language. However, changes to the clones in procedural language are more scattered compared to the changes to the clones in object-oriented programming languages.

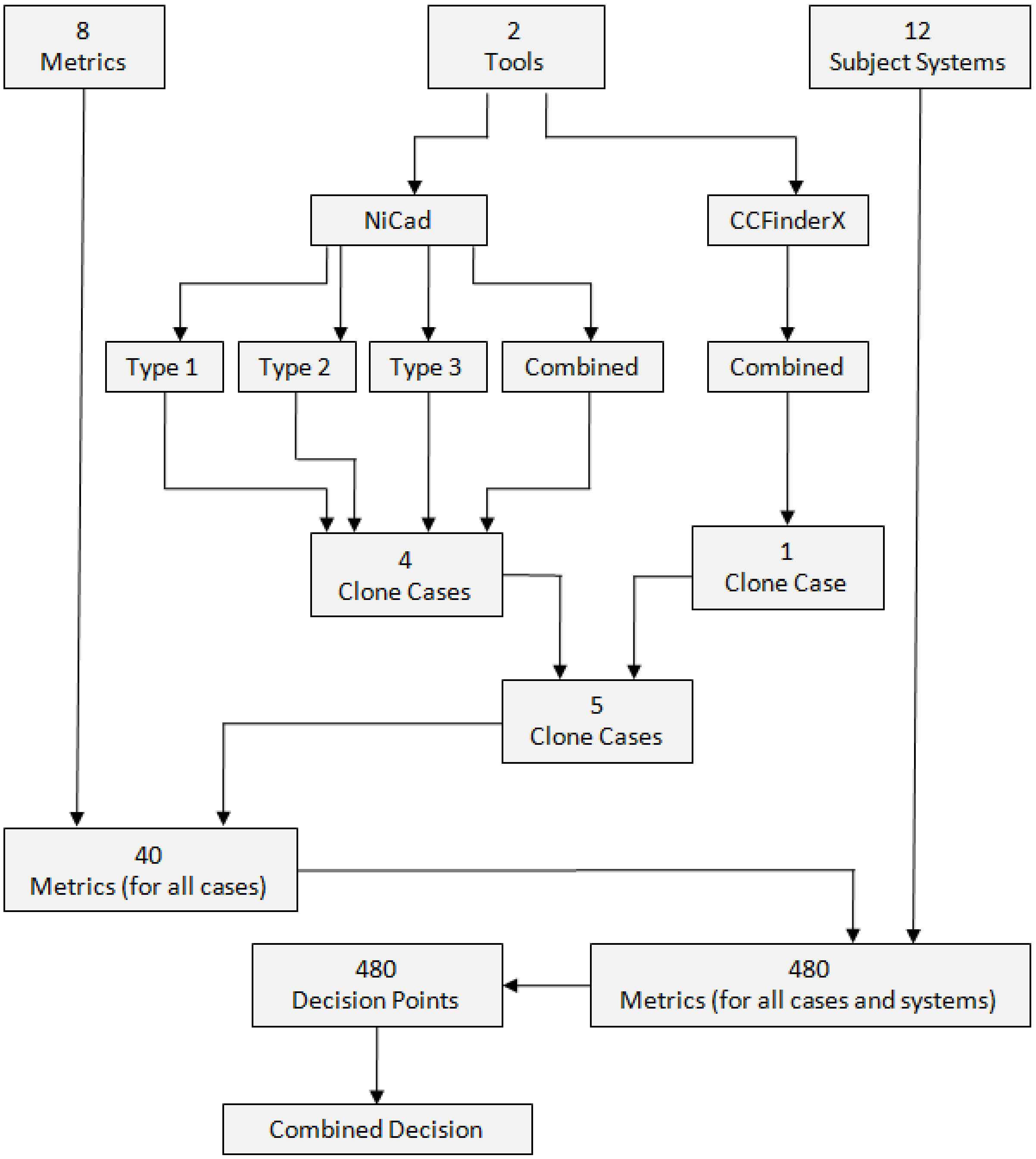


Figure 4.1: The decision making procedure

4.2 Uniform Framework

We mentioned that different studies analyzed clone stabilities using different experimental setups which might be a potential cause to the different outcomes. Focusing on this point, we implemented all the candidate methodologies (seven in total) that calculate eight metrics (one methodology [75] calculates two metrics) on a common framework written in Java programming language using MySQL as the back-end database server. Implementation in a common framework supports time efficient sharing of intermediate results among the methodologies. We emphasized on the commonality of the data storage structure in the MySQL back-end so that once the preprocessing of files, identification of changes between files of consecutive revisions and detection of clones from each revision for a single subject system are done and outputs are stored into the database, these stored results can be used by each of the candidate methodologies to work on that subject system. Figure 4.1 describes our decision-making strategy based on the common framework.

The figure shows that we evaluated eight stability measurement metrics (in total). We implemented the methodologies and calculated the metrics using two clone detection tools: CCFinder [19] and NiCad [96]. From these two clone detection tools we can obtain clone results for the following five cases.

(1) Type 1 clone results from NiCad

(2) Type 2 clone results from NiCad

(3) Type 3 clone results from NiCad

(4) Combined clone results (clone results combining above three clone types) from NiCad

(5) Combined clone results (clone results combining Type 1 and Type 2 clones) from CCFinderX.

We will explain these cases in detail in Section 4.6. For these five cases we have five different implementations of each of the methodologies for calculating each of the eight metrics.

Thus, we have 40 stability-related metrics (5 clone cases $\times$ 8 metrics) in total. We calculate these 40 metrics from each of the 12 candidate subject systems. Thus, for all subject systems, we calculated 480 (40 metrics $\times$ 12 subject systems) results in total. However, for calculating these 480 results we conducted 420 separate experiments (7 methodologies $\times$ 5 clone cases $\times$ 12 subject systems). One methodology proposed by Lozano and Wermelinger [75] calculates two metrics (Impact and Likelihood). We define each of the 480 results as a decision point from which we decide about whether cloned code is more harmful than non-cloned code or not. After analyzing all of these 480 decision points from different perspectives we take a combined decision on the comparative stability of cloned and non-cloned code.

4.3 Stability Measuring Methodologies and Metrics

We discuss the candidate stability measurement methodologies (seven methodologies) and related metrics (eight metrics) in the following subsections. The first five methodologies [40,47,65,74,75] are already existing and the remaining two are our proposed new ones. However, our proposed methodologies, the related metrics, and associated empirical studies have been published in international conferences [83,87] and journal [84].

4.3.1 Modification Frequency (MF) Proposed by Hotta et al.

Hotta et al. [47] calculated: (i) MF_c (Modification Frequencies of Cloned Code or Duplicate code) and (ii) MF_n (Modification Frequencies of Non-Duplicate code) considering all the revisions of a particular codebase extracted from subversion repository. Their metric calculation strategy involves several sequential steps including: (1) identification and checking out of relevant revisions of a subject system, (2) normalization of source files by removing blank lines, comments and indents, (3) detection and storing of each line of duplicate code into the database. The differences between consecutive revisions were also identified and stored in the database. Then, MC_c (Modification Count in Duplicate code region) and MC_n (Modification Count in Non-Duplicate code region) were determined exploiting the information saved in the database and finally MF_c and MF_n were calculated using the following equations [47]:

$$MF_c = \frac{\sum_{r \epsilon R} MC_c(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_c(r)} \tag{4.1}$$

$$MF_n = \frac{\sum_{r \epsilon R} MC_n(r)}{|R|} * \frac{\sum_{r \epsilon R} LOC(r)}{\sum_{r \epsilon R} LOC_n(r)} \tag{4.2}$$

Here, R is the number of revisions of the candidate subject system. $MC_c(r)$ and $MC_n(r)$ are the number of modifications (defined in the next paragraph) in the cloned and non-cloned code regions respectively between revisions r and $(r+1)$. MF_c and MF_n are the modification frequencies of the cloned and non-cloned code regions of the system. $LOC(r)$ is the number of LOC in revision r. $LOC_c(r)$ and $LOC_n(r)$ are respectively the numbers of cloned and non-cloned LOCs in revision r.

According to the definition of Hotta et al. [47], a modification can affect multiple consecutive lines. Suppose, n lines of a method (or any other program entity) were modified through additions, deletions or changes. If these n lines are consecutive then, the count of modification is one. If these n lines are not consecutive then, the count of modifications equals to the number of unchanged portions within these n lines plus one.

They performed their empirical study on 15 open source subject systems incorporating the clone detectors: CCFinderX [19], Simian [108], and Scorpio [105] with a general conclusion that cloned code is modified less frequently than non-cloned code. Thus according to this study cloned code is more stable than non-cloned code.

4.3.2 Modification Probability (MP) proposed by Göde and Harder

Göde and Harder [40] replicated, extended and validated Krinke's study [64] to determine whether clones are responsible for increasing maintenance effort and if does so, to what extent. In this study, they used an incremental token-based clone detection tool. They calculated the modification probabilities of cloned and non-cloned code with respect to addition, deletion and modification according to the following equations.

Modification probability of cloned code (MP_c) was calculated using the following equation.

$$MP_c = \frac{\sum_{r \epsilon R} A_c(r) + D_c(r) + C_c(r)}{\sum_{r \epsilon R} Tok_c(r)} \tag{4.3}$$

Modification probability of non-cloned code (MP_n) was determined using the equation given below.

$$MP_n = \frac{\sum_{r \epsilon R} A_n(r) + D_n(r) + C_n(r)}{\sum_{r \epsilon R} Tok_n(r)} \tag{4.4}$$

In the above equations, R is the set of all revisions of the candidate subject system. $A_c(r)$, $D_c(r)$, and $C_c(r)$ are respectively the total number of tokens added, deleted and changed (or modified) in the cloned regions of revision r of the subject system. In the same way, $A_n(r)$, $D_n(r)$, and $C_n(r)$ are the total number of tokens added, deleted and modified in the non-cloned regions of revision r. $Tok_c(r)$ and $Tok_n(r)$ are the total number of tokens in respectively the cloned and non-cloned regions of the subject system.

However, the modification probabilities MP_c and MP_n were termed as overall instability of cloned and non-cloned code in the original study [40]. We named these as modification probabilities considering the equations Eq. 4.3 and Eq. 4.4. The right sides of these equations determine the ratios of the modified tokens to the total number of tokens in cloned and non-cloned regions respectively. The greatest possible value of the ratio of a particular region is '1' (if all the tokens in this region are changed). The lowest possible value '0' indicates that no tokens have changed.

Göde and Harder performed this study [40] on two open source Java systems with a general conclusion that cloned code is more stable than the non-cloned code.

4.3.3 Average Last Change Date (ALCD) Proposed by Krinke

Krinke [65] introduced a new concept of code stability measurement by calculating the average last change dates of cloned and non-cloned regions of a codebase based on the *blame* annotation in the SVN repository. He considers only a single revision (generally the last revision) (Hotta et al. [47] considers all the revisions up to the last one as is already described). He calculates the average last change dates of cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code from the file level and system level granularities in the following way.

File level metrics

- Percentage of files where the average last change date of cloned code is older than that of non-cloned code (cloned code is older than non-cloned code) in the last revision of a subject system.
- Percentage of files where the average last change date of cloned code is newer than that of non-cloned code (cloned code is younger than non-cloned code) in the last revision of a subject system.

System level metrics

- Average last change date of cloned code ($ALCD_c$) for the last revision of a candidate subject system.
- Average last change date of non-cloned code ($ALCD_n$) for the last revision of a candidate subject system.

Calculation of average last change date (ALCD): Krinke calculated the average last change dates in the following way. Suppose five lines in a file correspond to five revision dates 01-Jan-11, 05-Jan-11, 08-Jan-11, 12-Jan-11, 20-Jan-11. The average of these dates was calculated by determining the average distance (in days) of all other dates from the oldest date 01-Jan-11. This average distance is (4+7+11+19)/4 = 10.25 and thus the average date is 10.25 days later to 01-Jan-11 yielding 11-Jan-11. We see that this process of calculating ALCD has the possibility of introducing rounding error. In the given example, the fraction '0.25' cannot be reflected to the calculated ALCD. This might force the ALCDs of cloned and non-cloned code to be equal [83].

According to the average last change date calculation process described above, we calculate the average last change date of cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code in the following ways.

$$ALCD_c = ODCL + \frac{\sum_{l \epsilon CL} DATE(l) - ODCL}{|CL|} \tag{4.5}$$

$$ALCD_n = ODNL + \frac{\sum_{l \epsilon NL} DATE(l) - ODNL}{|NL|} \tag{4.6}$$

In the above equations (Eq. 4.5 and Eq. 4.6), $ODCL$ and $ODNL$ are respectively the oldest change dates of cloned and non-cloned lines in the last revision of the candidate subject system. CL and NL are the sets of cloned and non-cloned lines in the last revision. $DATE(l)$ is the change date corresponding to the source code line l.

The two ratios in file level metrics were calculated considering only the analyzable files in the last revision of the subject systems. The set of analyzable files consists of those files which contain both cloned and non-cloned code. The files containing no cloned code and the fully cloned files were excluded from consideration while determining file level metrics. As these files contain only cloned or only non-cloned code, none of these files can determine whether cloned or non-cloned code is older. However, as different files can be of different sizes, percentage of files cannot be a good metric for comparing the stability of cloned and non-cloned code. Also, some files might be discarded from calculation because they contain only a single type of code (cloned or non-cloned). For these reasons, in our experiment we calculated the system level metrics only. System level metrics are calculated considering all source files in a codebase. The intuition behind this methodology is that the older the code is the more stable it is. That means, if a code region (cloned or non-cloned) remains unchanged for longer duration compared to the other (non-cloned or cloned) the former code region can be regarded as more stable.

Krinke performed this study on three open source Java systems using Simian [108] clone detector considering only Type 1 clones with the conclusion that cloned code is more stable than non-cloned code.

4.3.4 Average Age (AA)

We have just described Krinke's methodology [65] for calculating the average last change date of cloned and non-cloned lines of a codebase. The outputs of this methodology are dates. We propose a variant [83] of this methodology to analyze the longevity (stability) of cloned and non-cloned code by calculating their average ages (in days). This methodology also uses the *blame* command of SVN (as was used by Krinke [65]) to calculate the age for each of the cloned and non-cloned lines in a subject system.

Average Age measurement technique

Suppose we have several subject systems. For a specific subject system this methodology works on the last revision r_o. By applying a clone detector on revision r_o, the lines of each source file can be separated into two disjoint sets: (i) one containing all cloned lines and (ii) the other containing all non-cloned lines. Different lines of a file contained in r_o can belong to different previous revisions. If the *blame* command on a file assigns the revision r to a line x, then it is understood that line x was produced in revision r and has not been changed up to last revision r_o. The creation date of r is denoted as $DATE(r)$. In the current revision R, the age (in days) of this line is calculated by the following equation:

$$AGE(x) = DATE(r_o) - DATE(r) \tag{4.7}$$

Two average ages of cloned and non-cloned code were calculated from system level granularity. These are as follows.

- Average age of cloned code (AA_c) in the last revision of a subject system. This is calculated by considering all cloned lines of all source files of the system. The equation for calculating AA_c is given below.

$$AA_c = \frac{\sum_{l \epsilon CL} LRD - DATE(l)}{|CL|} \tag{4.8}$$

  In the above equation, CL is the set of all cloned lines in the last revision of the candidate subject system. LRD is the creation date of the last revision of the subject system. $DATE(l)$ is the last change date of source code line l.

- Average age of non-cloned code (AA_n) in the last revision of a subject system. AA_n is calculated by considering all non-cloned lines of all source files of the system. The equation for calculating AA_n is given below.

$$AA_n = \frac{\sum_{l \epsilon NL} LRD - DATE(l)}{|NL|} \tag{4.9}$$

  In the above equation, NL is the set of all non-cloned lines in the last revision of the candidate subject system.

According to this methodology, a higher average age is the implication of higher stability. We introduced this variant to address the following issues in Krinke's methodology [65].

(1) *blame* command of SVN gives the revisions as well as revision dates of all lines of a source file including its comments and blank lines. Krinke's methodology does not exclude blank lines and comments from consideration. This might play a significant role on skewing the real stability scenario.

(2) As indicated in the average last change date calculation process, Krinke's methodology often introduces some rounding errors in its results.

(3) Krinke's methodology [65] is dependent on file level metrics and this dependability sometimes alters the real stability scenario [83].

Our technique overcomes these issues while calculating stability results. It does not calculate any file-level metrics because its system level metrics are adequate in decision making. It should also be mentioned that Hotta et al.'s methodology [47] also ensures the exclusion of blank lines and comments from consideration through some preprocessing steps prior to clone detection.

4.3.5 Likelihood and Impact of Methods proposed by Lozano and Wermelinger

Lozano and Wermelinger performed an in-depth study [75] to assess the impacts of clones on maintenance by examining all the revisions of candidate subject systems. Their calculations were based on method level granularity using CCFinderX [19] clone detection tool. According to their definition, a cloned method can be fully or partially cloned (only a portion of the method is cloned). They detected the methods in different versions using CTAGS [33]. They also performed the origin analysis of methods in consecutive revisions to see how a method gets changed as it passes through multiple revisions. Using origin analysis they separated the methods into the following three subsets.

Always cloned methods: An always cloned method contains a cloned portion in it in all revisions in which it remains alive.

Never cloned methods: A never cloned method contains no cloned portions in its total lifetime.

Sometimes cloned methods: A sometimes cloned method contains cloned portions for a limited period of its total lifetime.

They calculated the following stability metrics.

Likelihood: The likelihood of change of a method m during its cloned period (or non-cloned period) is the ratio between the number of changes to m and the total number of changes to the system (all methods) during cloned period (or non-cloned period).

Impact: The impact of a method m is denoted by the average percentage of the system that gets changed whenever m changes during its cloned period or non-cloned period. This is calculated by the average number of methods changed on those commit transactions where method m gets changed.

We calculate the average impact of always cloned methods and the cloned periods of sometimes cloned methods. We term this impact as the impact of cloned code (ICC). We calculate *ICC* according to the following equations.

$$ICC = \frac{\sum_{m \epsilon M_c} Impact_{cloned}(m) + \sum_{m \epsilon M_{sc}} Impact_{cloned}(m)}{|M_c| + |M_{sc}|} \tag{4.10}$$

$$Impact_{cloned}(m) = \frac{\sum_{c \epsilon CCP(m)} CCM(c)}{|CCP(m)|} \tag{4.11}$$

In the above equations (Eq. 4.10 and Eq. 4.11), M_c, M_{sc}, and M_n are the sets of always cloned, sometimes cloned and never cloned methods. $Impact_{cloned}(m)$ is the impact of the cloned period of method m. $CCP(m)$ is the set of all commits where m was changed during its cloned period. $CCM(c)$ is the count of methods changed in commit c.

We also calculate the average impact of never cloned methods and the non-cloned periods of sometimes cloned methods. We term this impact as the impact of non-cloned code (INC). INC is calculated according to the following equations.

$$INC = \frac{\sum_{m \epsilon M_n} Impact_{non-cloned}(m) + \sum_{m \epsilon M_{sc}} Impact_{non-cloned}(m)}{|M_n| + |M_{sc}|} \tag{4.12}$$

$$Impact_{non-cloned}(m) = \frac{\sum_{c \epsilon CNP(m)} CCM(c)}{|CNP(m)|} \tag{4.13}$$

In the above equations, $Impact_{non-cloned}(m)$ is the impact of the non-cloned period of method m. $CNP(m)$ is the set of all commits where m was changed during its non-cloned period. Other terms are already described for the equations: Eq. 4.10 and Eq. 4.11. We also calculate the likelihood of cloned (LCC) and non-cloned code (LNC). LCC is the average likelihood considering always cloned methods and the cloned periods of the sometimes cloned methods. We calculate LCC according to the following equations.

$$LCC = \frac{\sum_{m \epsilon M_c} Likelihood_{cloned}(m) + \sum_{m \epsilon M_{sc}} Likelihood_{cloned}(m)}{|M_c| + |M_{sc}|} \tag{4.14}$$

$$Likelihood_{cloned}(m) = \frac{\sum_{c \epsilon CCP(m)} NCM(m, c)}{\sum_{c \epsilon CCP(m)} \sum_{m_i \epsilon M} NCM(m_i, c)} \tag{4.15}$$

In the above equations (Eq. 4.14 and Eq. 4.15), $Likelihood_{cloned}(m)$ is the likelihood of the method m during its cloned period. $NCM(m, c)$ is the number of changes to the method m in commit c. M is the set of all methods. The remaining terms in these equations have already been defined. We also calculate LNC, the average likelihood considering never cloned methods and the non-cloned periods of the sometimes cloned methods, according to the following equations.

$$LNC = \frac{\sum_{m \epsilon M_n} Likelihood_{non-cloned}(m) + \sum_{m \epsilon M_{sc}} Likelihood_{non-cloned}(m)}{|M_n| + |M_{sc}|} \tag{4.16}$$

$$Likelihood_{non-cloned}(m) = \frac{\sum_{c \epsilon CNP(m)} NCM(m, c)}{\sum_{c \epsilon CNP(m)} \sum_{m_i \epsilon M} NCM(m_i, c)} \tag{4.17}$$

In the above equation (Eq. 4.17), $Likelihood_{non-cloned}(m)$ is the likelihood of the method m during its non-cloned period.

Lozano and Wermelinger performed this study [75] on five open source Java systems with a conclusion that is contradictory to the outcomes of the previous studies. According to this study, cloned methods are more likely to be modified compared to the non-cloned methods. Also, cloned methods seem to increase maintenance efforts significantly.

4.3.6 Average instability per cloned method (AICM) proposed by Lozano and Wermelinger

The methodology [74] is an improvement of the previous work [75] and gives more sophisticated analysis of the impacts of clones. In the previous study [75] several issues such as clone families, inclusion and exclusion of clones in families as well as back propagation of changes were not considered. But, in this study [74] all these matters were taken into account and many more measurements were made using the same tools. The implementation and analysis are based on method-level granularity where the definition of cloned method remains the same as described in the previous methodology [75]. We measured the following two metrics from this methodology, because these are related to code stability. These two metrics together can help us to determine whether cloned or non-cloned code is more stable. We term these two metrics together as *average instability per cloned method (AICM)* (as is mentioned before).

Extension per cloned method (EPCM): This metric was calculated by determining the average proportion of cloned tokens in the cloned methods at a particular commit transaction.

Proportion of changes to the clones of cloned methods (CPCM): We calculated this metric by the average ratio between the number of changes in the cloned tokens and the total number of changes in the cloned methods up to a particular commit transaction. This metric is originally termed as *Stability per method* by Lozano and Wermelinger [74].

For a particular commit operation we determine $EPCM$ and $CPCM$ according to the following equations.

$$EPCM = \frac{\sum_{m \epsilon M} \frac{CTok(m) \times 100}{Tok(m)}}{|M|} \tag{4.18}$$

$$CPCM = \frac{\sum_{m \epsilon M} \frac{CTok_{changed}(m) \times 100}{Tok_{changed}(m)}}{|M|} \tag{4.19}$$

In the above equations (Eq. 4.18 and Eq. 4.19), M is the set of all cloned methods in a particular commit operation. $CTok(m)$ is the number of cloned tokens in a cloned method m and $Tok(m)$ is the number of total tokens in m. $CTok_{changed}(m)$ is the number of cloned tokens changed in the cloned method m. $Tok_{changed}(m)$ is the number of total tokens changed in m.

Lozano and Wermelinger performed this study [74] on five open source subject system written in Java. According to their experimental results, cloned methods have a higher density changes compared to the non-cloned methods. This outcome also contradicts with the conclusions drawn by Krinke [65] and Hotta et al. [47]. According to the explanation of Lozano and Wermelinger, such a contradiction occurred because of different setups of the different clone detection tools incorporated in these studies.

4.3.7 Dispersion of Changes (CD)

We proposed and implemented a methodology for measuring the dispersions of changes [87] in the cloned and non-cloned regions considering method level granularity. Our motivation behind introducing change dispersion is described below.

Existing stability measurement approaches [47,65,74,75] fail to investigate an important aspect regarding change: are the changes occurring in the same regions or in different regions? This information is very important for analyzing code stability as well as its impact. Repeated changes to the same code region (e.g., method) are more manageable compared to the scattered changes in different regions. If a change takes place to a method for the first time, programmers need to spend a considerable amount of time understanding the method context and to determine the possible impacts of the changes to other related code regions (or methods). This might be necessary so that relevant changes are propagated to other similar code fragments (clones) to maintain consistency. Even a small change might have a great impact to the whole software system. According to a recent study [52], every second unintentional inconsistent change to a clone leads to a fault. However, further changes to the same method do not require as much effort because the impact analysis of changes to this method has already been done. Thus, several changes to different code regions generally are more difficult to tackle than those changes in the same code region. We can thus say that if during the evolution of a software system, C changes take place to n_1 different regions of cloned code and the same number of changes (C) take place to n_2 different regions of non-cloned code for the same number of consecutive revisions where $n_1 > n_2$, then, for this software system non-cloned code is more stable than cloned code because modifications in the cloned code require more effort to be managed than those of its non-cloned counterpart. To incorporate this information in the stability measurement process we introduced *change dispersion*. We define change dispersion in the following way.

Definition: *Change dispersion* in a particular code region (cloned or non-cloned) is the percentage of method genealogies affected by changes in that region during a particular period of evolution. As the evolution period we considered the total duration up to the creation of the last revision (mentioned in Table 4.3) of a particular software system. Each of the commit operations in the evolution period involving some modifications to the source code was taken into account. We calculate *change dispersion* in the following way.

A method (or function in case of C language) is defined as a cloned method when it contains some cloned lines in it. According to this methodology [87] there are two types of cloned methods: (i) fully cloned methods (all of the lines contained in these methods are cloned lines) and (ii) partially cloned methods (these methods contain some non-cloned portions in it). For calculating the dispersion of cloned code, the changes in the cloned portions of the cloned (fully or partially) methods are considered. Partially cloned methods are also considered while calculating the dispersion of non-cloned code because, changes might occur in the non-cloned portions of the partially cloned methods. Moreover, while determining method genealogies it might be seen that a partially cloned method has become fully cloned or fully non-cloned after receiving a change. These methods are considered in calculating the dispersions of both cloned and non-cloned code.

Suppose, for a subject system, the sets of cloned and non-cloned method genealogies are C and N respectively. C_c is the set of cloned method genealogies which received some changes in their cloned portions during the evolution. The number of changes received by the genealogies in the set C_c is generally greater than $|C_c|$, because a particular method genealogy generally gets many changes during evolution. In the same way, N_c is the set of non-cloned method genealogies that received some changes in their non-cloned portions. The dispersion of changes in cloned code (CD_c) and non-cloned code (CD_n) were expressed by the following equations. We multiplied '100' only for determining the percentages.

$$CD_c = \frac{|C_c| \times 100}{|C|} \tag{4.20}$$

$$CD_n = \frac{|N_c| \times 100}{|N|} \tag{4.21}$$

Implementation of this methodology requires the extraction of method genealogies. We extracted method genealogies using the algorithm proposed by Lozano and Wermelinger [75].

4.4 Experimental Steps

We implemented seven stability measurement methodologies using two clone detection tools for conducting this experiment. While three of these methodologies [74, 75, 87] calculate the respective stability metrics considering method level granularity, the remaining four methodologies [40, 47, 65, 83] calculate using block granularity. In the following subsections we describe the sequential steps for calculating the metrics.

4.4.1 Extraction of Repositories

All of the subject systems on which we applied our candidate methodologies were downloaded from open-source SVN repositories. For a subject system, we extracted only those revisions which were created because of some source code modification (addition, deletion or change) rather than just tagging and branching operations. To determine whether a revision should be extracted or not, we checked the extensions of the files which were modified to create the revision. If some of these modified files are source files, we considered the revision as our target revision and extracted it.

4.4.2 Preprocessing

We applied the following two preprocessing steps to all the revisions of the subject systems before applying the methodologies on them except for Krinke's methodology [65] and average age calculation process [83].

(1) Rearrangements of lines so that an isolated left or right brace (if a left or right brace remains in a line associated with no other character) was deleted and added at the end of the previous line.

(2) Deletion of blank lines and comments.

These preprocessing steps were done to avoid the effects of changes to the comments and indentations on the calculated metrics.

It was not possible to apply the mentioned preprocessing steps in case of Krinke's methodology [65] and its variant because these methodologies work on the output of SVN *blame* command for a specific file, not on the original file. For these methodologies (Krinke's methodology [65], and the methodology for calculating average age), we just ignored the blank lines and comments from *blame* command output.

4.4.3 Detection of method

In order to detect the methods of a specific revision we applied Ctags [33] on the source files of that revision. For each method we determined its

(1) file name,

(2) class name (Java and C# systems),

(3) package name (Java),

(4) method name,

(5) signature,

(6) starting and ending line numbers, and

(7) revision number

In case of the subject systems written in C, we determined five properties from the above list excluding class name and package name. We detected the methods for all target revisions. The method detection process for a particular revision can be made faster by reusing the methods stored for the immediate previous revision. If we have completed the detection and storage activities for revision r_i, we do not need to apply Ctags [33] for the source files which remain unchanged in revision r_{i+1}. Methods of these unchanged files can be retrieved from the database and forwarded to the r_{i+1}. We apply Ctags [33] to only those source files which received some changes while being forwarded from r_i to r_{i+1}. However, we do not store the methods in the database at this stage because we need the information about which methods have clones and which methods have got changed before being forwarded to the next revision.

Table 4.2: NiCad Settings

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

4.4.4 Clone Detection

As noted previously, we applied NiCad [96] and CCFinderX [19] clone detection tools to each target revision to detect clone blocks. These clone blocks were then mapped to the already detected methods of this revision by comparing the beginning and ending line numbers of clone blocks and methods. So, for each method we collect the begining and ending cloned line numbers (if exist). CCFinderX currently outputs the beginning and ending token numbers of clone blocks. We automatically retrieve the corresponding line numbers from the generated preprocessed files. We store the clones as well as clone families detected from a revision in the database. We used the following setups for the clone detection tools.

Setup for CCFinderX: CCFinderX [19] is a token based clone detection tool that currently detects block clones of Type-1 and Type-2. We set CCFinderX to detect clone blocks of minimum 30 tokens with TKS (minimum number of distinct types of tokens) set to 12 (as default).

Setup for NiCad: NiCad can detect both exact and near-miss clones at the function or block level of granularity. We detected block clones with a minimum size of 5 LOC in the pretty-printed format that removes comments and formatting differences. We used the NiCad settings in Table 4.2 for detecting three types of clones. The dissimilarity threshold means that the clone fragments in a particular clone class may have dissimilarities up to that particular threshold value between the pretty-printed and/or normalized code fragments. We set the dissimilarity threshold to 20% with blind renaming of identifiers for detecting Type 3 clones. For all these settings NiCad was shown to have high precision and recall [98, 102].

4.4.5 Detection and Reflection of Changes

We identified the changes between corresponding files of consecutive revisions using UNIX *diff* command. *diff* outputs three types of changes:

(1) addition,

(2) deletion, and

(3) modification

with corresponding line numbers. We mapped these changes to methods using line information. So, for each method we gathered two more pieces information: (1) the count of lines changed in cloned portions and (2) the count of changed lines in non-cloned portions. We not only map the changes to method but also store the changes in the database with corresponding line information.

4.4.6 Storage of Methods

At this stage, we have all necessary pieces of information of all methods belonging to a particular revision. We store these methods in database with individual entry for each method. The attributes that we store for a particular method have already been listed in Section 4.4.3. We do not store the statements inside a method. For each of the revisions of a particular software system we detected the methods along with cloning and change information and stored the methods in the database. So, our database contains different method sets for different revisions: one set for each revision.

4.4.7 Method Genealogy Detection

After completing method detection and storage for all revisions of a subject system, we detected method genealogies following the technique described previously [75] to identify the propagation of methods across revisions. Suppose a method was created in revision r_i and was alive and propagated to the next two revisions r_{i+1} and r_{i+2} with or without some changes. So, this method has corresponding entries in all of these three revisions. By detecting method genealogies we can identify these entries as belonging to the same method. For the purpose of genealogy detection we assign a single unique ID to all of the entries of a particular method residing in different revisions.

4.4.8 Metrics Calculation

After completing all the steps described above for a particular subject we calculated the metrics. We calculate the following four metrics using the method information stored in the database.

(1) impact,

(2) likelihood,

(3) AICM (Average Instability per Cloned Method), and

(4) CD (Change Dispersion)

The remaining metrics listed below are calculated using the information stored for clones and changes.

(5) MF (Modification Frequency),

(6) MP (Modification Probability),

(7) ALCD (Average Last Change Date), and

(8) AA (Average)

4.5 Subject Systems

Table 4.3 lists the subject systems that were included in our study along with their associated attributes. We downloaded the subject systems from SourceForge [109]. The subject systems are of diverse variety in terms of the followings.

Table 4.3: Subject Systems

	Systems	**Domains**	**LLR**	**Revisions**
Java	DNSJava	DNS protocol	23,373	1635
	Ant-Contrib	Web Server	12,621	176
	Carol	Game	25,092	1699
	jabref	Reference Management	59,648	3000
C	Ctags	Code Definition Generator	33,270	774
	Camellia	Multimedia	100,891	608
	QMail Admin	Mail Management	4,054	317
	GNUMakeUniproc	Auto-build system for C/C++ projects	75,745	863
C#	GreenShot	Multimedia	37,628	999
	ImgSeqScan	Multimedia	12,393	73
	Capital Resource	Database Management	75,434	122
	MonoOSC	Formats and Protocols	18,991	355

LLR = LOC in Last Revision

(1) **Application domains:** The candidate systems span 10 different application domains as mentioned in Table 4.3.

(2) **Implementation language:** The systems cover three programming languages: Java, C, and C#.

(3) **System size:** The systems are of different sizes, from very small (4 KLOC) to large (100 KLOC).

4.6 Experimental Results and Analysis

In this section, we presented the results obtained for eight stability metrics in sixteen tables. The corresponding analysis for each of these metrics (as well as table data) is also given in this section.

Normalization of metric values: The values corresponding to five candidate metrics: *modification probability*, *impact*, *likelihood*, *change dispersion*, and *average instability per cloned method* (*EPCM*, *CPCM*) are normalized within the range zero to one. The equations for calculating *modification probabilities* (Eq. 4.3, Eq. 4.4), *impacts* (Eq. 4.10, Eq. 4.12), and *likelihoods* (Eq. 4.14, Eq. 4.16) provide us values which are normalized within zero to one. However, the equations for calculating *change dispersions* (Eq. 4.20, Eq. 4.21), *EPCMs* (Eq. 4.18), and *CPCMs* (Eq. 4.19) give us percentages. We normalize these percentages within zero to one by dividing them by 100.

It was not possible to normalize the values of the remaining three metrics: *modification frequency*, *average last change date*, and *average age* in a particular range (e.g. zero to one) because of the following reasons.

(1) None of these metrics has a fixed upper bound.

(2) The values corresponding to the metrics *average last change date* and *average age* are dates and ages (in days) respectively.

The experimental results corresponding to each metric can be broadly divided into two categories based on the discrimination power of the clone detectors:

(1) Individual type results: These results assist us to analyze the influence of each type of clones on the stability measurement metrics individually. Individual type results were obtained by applying NiCad clone detector, because it not only detects the three major types (Type 1, Type 2 and Type 3) of clones combinedly but also facilitates the separation of three types clones from one another. NiCad detects clones by separating them into individual classes. Generally, Type 2 clone detection results of NiCad include Type 1 clone classes. In the same way, the results obtained by detecting Type 3 clones include both Type 1 and Type 2 clone classes. To get the exact Type 2 clone classes we excluded Type 1 classes from Type 2 results. In the same way, exact Type 3 clone classes were obtained by excluding Type 2 and Type 1 classes from Type 3 results. However, separation of individual clone types is not possible with CCFinderX.

(2) Combined type results: These results help us to analyze the combined effect of three types of clones on the stability measurement metrics. We used both NiCad and CCFinderX to get these results. For determining the combined impacts of three types of clones using NiCad, we used the Type 3 clone detection results without excluding Type 1 and Type 2 classes. We also used CCFinderX to get combined type results. However, CCFinderX detects only Type 1 and Type 2 clones.

For each metric there are two tables. While one table contains the individual type results, the other contains the combined type results. For the purpose of analysis we interpreted the tables as consisting of *decision points*.

Decision point: A particular decision point consists of the followings.

- Metric value for cloned code,
- Metric value for non-cloned code, and
- A remark indicating whether cloned code is more stable than non-cloned code or not considering that metric value.

The decision points in the tables containing the individual type results reflect the stability scenario of three types (Type 1, Type 2, Type 3) of clones. The overall stability of cloned code combining three clone-types is reflected by the decision points in the tables containing the combined type results.

For the purpose of analysis we categorize the decision points into the following three categories on the basis of an *eligibility value*. We at first describe the categories and then we present the mathematical definition and explanation for *eligibility value*.

- **Category 1 (CLONES MORE STABLE):** For each of the points belonging to Category 1, two condition holds: (1) the eligibility value is greater than or equal to a threshold value, and (2) cloned code appears to be more stable than non-cloned code. These points are marked with $\oplus$.

- **Category 2 (CLONES LESS STABLE):** For each of the points belonging to this category, two condition holds: (1) the eligibility value is greater than or equal to a threshold value, and (2) cloned code appears to be less stable than non-cloned code. These points are marked with $\ominus$.
- **Category 3 (NEUTRAL):** For each of the points belonging to this category, the eligibility value is less than a threshold value. Such a point is regarded as an insignificant decision point. We mark these points with $\bigcirc$.

We analyze the comparative stability of cloned and non-cloned code based on the significant decision points (belonging to Category 1 and Category 2). We do not consider the insignificant points (belonging to Category 3) for comparative stability analysis because of the following two reasons.

(1) The amount of insignificant decision points (belonging to Category 3) is very low (only 46 points are insignificant among 480 points in total) compared to the significant points. Considering all the eight metrics, we have 480 decision points in total. While only 46 of these points belong to Category 3, respectively 196 and 238 points belong to Category 1 and Category 2.

(2) From the insignificant points we cannot take any decision whether cloned code is more stable than non-cloned code or not, because the values of the stability metric corresponding to cloned and non-cloned code for such a point are not significantly different.

Eligibility Value: We calculate an *eligibility value* for each decision point to determine whether the metric values corresponding to cloned and non-cloned code for a particular point are significantly different. The following equation calculates eligibility value for each decision point except for the decision points corresponding to the metrics: *average last change date (ALCD)*, *average age (AA)*, and *average instability per cloned method (AICM)*. We cannot apply the following equation for these exceptions, because: (1) dates are ordinal, not cardinal, so any difference is significant, (2) average age > 1 is significant, and (3) AICM requires detailed analysis (Section 4.6.6).

$$Eligibility\ Value = \frac{(HMV - LMV) * 100}{LMV} \tag{4.22}$$

Here, HMV is the higher value of the metric values corresponding to cloned and non-cloned code for a particular decision point. LMV is the smaller one of the two metric values corresponding to cloned and non-cloned code for that decision point. If this *eligibility value* corresponding to a particular decision point is greater than or equal to a threshold value 10, as was selected by our previous study [87], we say that the decision point is significant and falls in Category 1 or Category 2. We select the threshold magnitude of *eligibility value* in such a way that, it will force a decision point having larger but very near metric-values (such as 41 and 40. *Eligibility Value* = (41-40)*100/40 = 2.5) to be selected as insignificant while a decision point with smaller but near metric-values (such as 3 and 4. *Eligibility Value* = 33.33) to be selected as significant which is expected. An example of categorization using the *Eligibility Value* is given below.

The Table 4.4 contains the combined type results obtained by applying Hotta et al.'s [47] methodology on

12 subject systems. This table has 24 decision points in total. Each decision point contains the modification frequencies of cloned and non-cloned code and a remark ($\oplus$, $\ominus$ or $\bigcirc$) indicating whether the point falls in Category 1 (CLONES MORE STABLE), Category 2 (CLONES LESS STABLE), or Category 3 (NEUTRAL). Among these points, 20 points fall in Category 1 or Category 2. These decision points are *significant decision points* because the differences between the modification frequencies for these points are significant according to the *eligibility value* calculated using Eq. 4.22. As an example, we calculate the *eligibility value* for the decision point corresponding to DNSJava and CCFinder in this table. For this point the *modification frequency* of cloned code is 16.94 and that of non-cloned code is 7.66. Here, HMV = 16.94 and LMV = 7.66. Thus, *eligibility value* for this point = 121.15. As 121.15 >10, the point is significant. Also, as the modification frequency of cloned code is greater than that of non-cloned code, cloned code appears to be less stable for this point. Thus, the point belongs to Category 2 ($\ominus$).

We interpreted the results obtained for each metric individually from the following four perspectives.

(1) Overall analysis: This analysis is based only on the combined type results. For each table (containing the combined type results) corresponding to a particular metric, we calculated three proportions:

(i) the proportion of decision points belonging to Category 1 (CLONES MORE STABLE)

(ii) the proportion of the decision points belonging to Category 2 (CLONES LESS STABLE).

(iii) the proportion of the decision points belonging to Category 3 (NEUTRAL).

The following example will explain this.

Example: Among 24 decision points in Table 4.4, 20 points are significant (belong to Category 1 or Category 2) and 4 points are insignificant (belong to Category 3). According to 45.8% (11 points) of all the decision points, cloned code is modified less frequently than the non-cloned code. The 37.5% points suggests the opposite. The remaining 16.7% points fall in Category 3 (NEUTRAL). Now, if we consider the significant decision points we can say that *cloned code is more stable in general than non-cloned code for this table*, because the percentage of decision points (belonging to Category 1) agreeing with lower modification frequency of cloned code is greater than the percentage of decision points (belonging to Category 2) agreeing with higher modification frequency of cloned code.

(2) Programming-language centric analysis: We compare the stabilities of cloned and non-cloned code with respect to the three programming languages in language centric analysis. Here, we also identify which language exhibits higher instability of clones according to which stability metric. This analysis is based on combined type results. In each table containing the combined type results for a particular metric, a particular programming language contributes eight decision points. From these eight decision points, we determine the proportions of decision points belonging to three categories: Category 1, Category 2, and Category 3. For each metric we present a graph showing the language-wise proportions of significant decision points.

Example: As an example, among the eight points belonging to Java in Table 4.4, 5 points are significant. Among these significant points, four points belong to Category 2 (CLONES LESS STABLE) and 1 point belongs to Category 1 (CLONES MORE STABLE). The graph in Fig. 4.2 shows the language centric analysis result for this table.

(3) Type-centric analysis: Our type-centric analysis for a particular metric is based on the table containing the individual-type results for that metric. Each table for a particular metric contains 12 decision points belonging to a particular clone-type. We calculate the proportions of decision points belonging to three categories: Category 1, Category 2, and Category 3 considering these points. For each metric, we present a graph showing the individual type-wise proportions of the decision points based on the table containing the individual type results for that metric. The following example will explain this.

Example: Table 4.5 contains the individual type results obtained by applying Hotta et al.'s methodology on twelve subject systems. Each clone type contributes one decision point for each subject system. If we consider the decision points belonging to Type 1, we see that 8 points belong to Category 1 (CLONES MORE STABLE), 3 points belong to Category 2 (CLONES LESS STABLE), and the remaining one point belongs to Category 3 (NEUTRAL). We see that while 66.7% (8 x 100 / (8+3 + 1) = 66.7) of these points belong to Category 1, only 25% and 8.3% points belong to Category 2 and 3 respectively.

(4) Type-centric analysis for each language: This analysis for a particular metric is also based on the table containing the individual type results for that metric. In such a table, four points belong to a particular clone type of a particular programming language. From these four points we determine the proportions of decision points belonging to Category 1, 2, and 3. For each metric we draw a graph showing the type-centric analysis result for each language. The following example will explain this.

Example: Among the four points belonging to Type 3 case of Java in Table 4.5: (i) two points belong to Category 1, (ii) one point belongs to Category 2, and (iii) one point belongs to Category 3. So, the percentages of the points belonging to Category 1, 2, and 3 are 50% (2 x 100 / 4) = 50), 25%, and 25% respectively. The graph in Fig. 4.4 shows the percentages for every combination of type and language for this table.

However, we see that for a single metric, our type-centric analysis for a particular programming language depends only on four decision points (for each combination of language and clone type). We presented a cumulative analysis in Section 4.7.3 considering eight metrics. In this analysis, we take language-wise type-centric decisions considering 32 decision points obtained from all combinations of languages and clone-types.

We also presented cumulative analyses from different dimensions in Section 4.7 considering the results obtained for all the metrics. In the following subsections we present our four-dimensional analysis of the experimental results obtained for each of the eight metrics.

4.6.1 Analysis of the Experimental Results Regarding Modification Frequency

Hotta et al. [47] calculated the modification frequencies of cloned (MF_c) and non-cloned (MF_n) code according to Eq. 4.1 and Eq. 4.2 and argued that cloned code changes less frequently than non-cloned code in general. Using our implementations of Hotta et al.'s methodology (using CCFinder and NiCad) we calculated

Table 4.4: Modification Frequencies by Hotta et al.'s methodology (Combined Type Result)

Lang	Systems	NiCad Combined			CCFinder Combined		
		MF_c	MF_n	Rem	MF_c	MF_n	Rem
Java	DNSJava	10.40	10.66	○	16.94	7.66	⊖
	Ant-Contrib	4.12	4.4	○	2.58	4.91	⊕
	Carol	26.06	15.55	⊖	21.89	16.32	⊖
	jabref	20.23	20.86	○	62.06	7.59	⊖
C	Ctags	10.16	7.74	⊖	8.932	7.86	⊖
	Camellia	34.59	17.31	⊖	47.02	14.20	⊖
	QMail Admin	50.95	51.36	○	39.15	22.07	⊖
	Gnumakeuniproc	35.04	38.82	⊕	35.69	40.33	⊕
C#	GreenShot	8.54	9.63	⊕	8.35	9.85	⊕
	ImgSeqScan	0	25.67	⊕	11.82	28.68	⊕
	Capital Resource	9.87	31.57	⊕	22.61	34.25	⊕
	MonoOSC	13.03	22.84	⊕	15.46	23.59	⊕

MF_c= Modification Frequency of Cloned Code

MF_n= Modification Frequency of Non-Cloned Code *Rem = Remark*

⊕= MF_c <MF_n (Category 1, CLONES MORE STABLE)

⊖= MF_c >MF_n (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 11 *Count of (⊖) = 9* *Count of (○) = 4*

Table 4.5: Modification Frequencies by Hotta et al.'s methodology (Individual Type Result)

Lang	**Systems**	**Type 1**			**Type 2**			**Type 3**		
		MF_c	MF_n	Rem	MF_c	MF_n	Rem	MF_c	MF_n	Rem
Java	DNSJava	27.91	8.69	⊖	11.93	10.48	○	10.62	10.61	○
	Ant-Contrib	9.96	4.09	⊖	3.18	4.27	⊕	3.76	4.41	⊕
	Carol	10.11	18.05	⊕	11.70	11.70	○	19.52	17.21	⊖
	jabref	17.67	22.46	⊕	16.38	22.53	⊕	19.83	22.68	⊕
C	Ctags	11.13	7.83	⊖	13.48	7.78	⊖	9.37	7.81	⊖
	Camellia	18.50	18.04	○	42.37	17.73	⊖	30.02	17.53	⊖
	QMail Admin	45.99	51.50	⊕	56.41	50.75	⊖	56.14	50.69	⊖
	Gnumakeuniproc	32.25	39.74	⊕	74.06	37.70	⊖	78.64	36.50	⊖
C#	GreenShot	4.13	9.62	⊕	7.62	9.59	⊕	8.65	9.60	⊕
	ImgSeqScan	0	24.97	⊕	0	25.28	⊕	0	25.43	⊕
	Capital Resource	0	31.69	⊕	0	31.37	⊕	12.18	31.48	⊕
	MonoOSC	7.09	22.71	⊕	15.18	22.53	⊕	13.09	22.78	⊕

MF_c= Modification Frequency of Cloned Code

MF_n= Modification Frequency of Non-Cloned Code   Rem = Remark

⊕= MF_c <MF_n (Category 1, CLONES MORE STABLE)

⊖= MF_c >MF_n (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 20   Count of (⊖) = 12   Count of (○) = 4

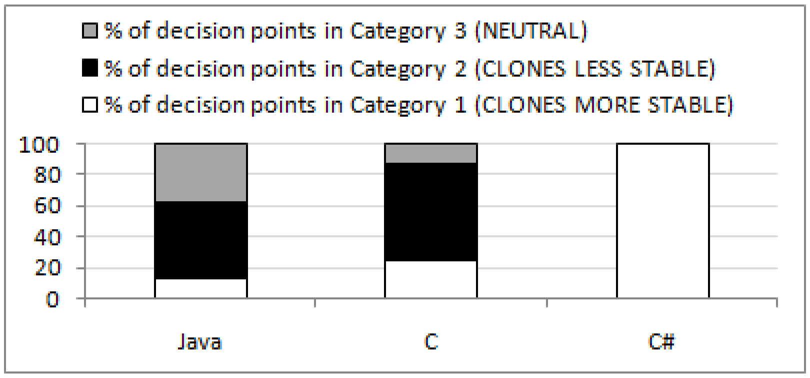


Figure 4.2: Language centric statistics for modification frequency (MF)

the modification frequencies of cloned and non-cloned code of each of the subject systems and populated the Tables 4.4 and 4.5. Table 4.4 and Table 4.5 contain the combined type and individual type results respectively. For a particular significant decision point in these tables,

(i) if $MF_c < MF_n$, then changes to the cloned code are less frequent compared to the changes to non-cloned code and this point falls in Category 1 (CLONES MORE STABLE)

(ii) if $MF_c > MF_n$, then changes to the cloned code are more frequent compared to the changes to non-cloned code and this point falls in Category 2 (CLONES LESS STABLE).

The following four sections of analysis answer the first research question RQ1 from four directions.

Overall analysis: Considering the 24 decision points of Table 4.4 we see that: (i) 45.8% points (11 points) belong to Category 1 (CLONES MORE STABLE), because these points agree with lower modification frequency of cloned code compared to non-cloned code, (ii) 37.5% points (9 points) belong to Category 2 (CLONES LESS STABLE), because these points agree with higher modification frequency of cloned code, and (iii) the remaining 16.7% points (4 points) belong to Category 3 (NEUTRAL). So, considering the percentages of decision points belonging to Category 1 and Category 2, we can say that *in general modification frequency of cloned code is smaller than that of non-cloned code.* This confirms Hotta et al.'s result.

Language Centric Analysis: For language centric analysis on Table 4.4, we present the graph in Fig. 4.2. According to this graph, *the clones in the subject systems written in both Java and C have a very high probability of getting more frequent changes compared to the non-cloned code in general. The opposite is true for C# systems.*

Type Centric Analysis: From the type centric statistics in Fig. 4.3 constructed from Table 4.5 we see that for each of the three clone-types (Type 1, Type 2, and Type 3), the highest proportion of decision points belong to Category 1 (CLONES MORE STABLE) and agree with lower modification frequency of cloned code. Thus, we can say that *each of the three types of clones are likely to receive less frequent changes compared to the non-cloned code in general.*

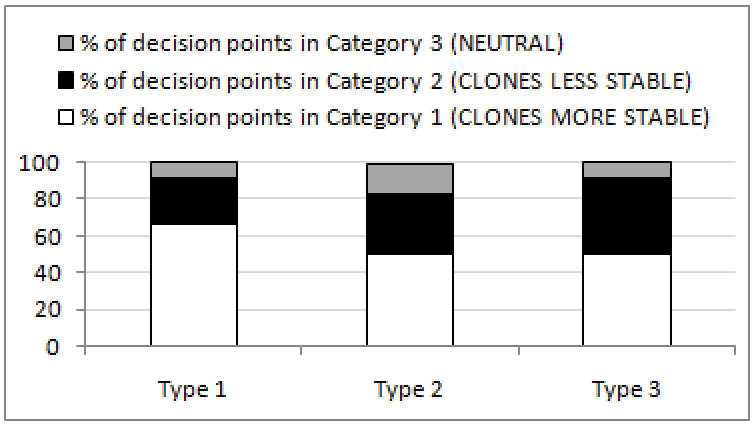


Figure 4.3: Type centric statistics for modification frequency (MF)

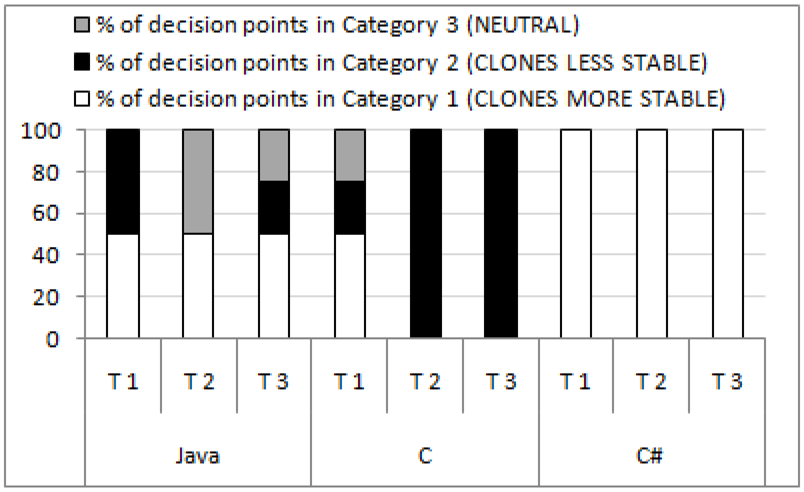


Figure 4.4: Language-wise type centric statistics for modification frequency (MF)

Table 4.6: Modification Probability by Göde et al.'s methodology (Combined Type Result)

Lang	Systems	NiCad Combined MP_c	NiCad Combined MP_n	NiCad Combined Rem	CCFinder Combined MP_c	CCFinder Combined MP_n	CCFinder Combined Rem
Java	DNSJava	0.00194282	0.00162636	⊖	0.002786	0.001116	⊖
	Ant-Contrib	0.0014545	0.000723646	⊖	0.000844	0.000827	○
	Carol	0.00113242	0.000682822	⊖	0.001050	0.000716	⊖
	jabref	0.000256	0.000246	○	0.001555	0.000099	⊖
C	Ctags	0.000790214	0.000974922	⊕	0.000943	0.000962	○
	Camellia	0.00166593	0.000536073	⊖	0.001712	0.000425	⊖
	QMail Admin	0.00626616	0.00304	⊖	0.002419	0.001444	⊖
	Gnumakeuniproc	0.000367565	0.000319181	⊖	0.000309	0.000543	⊕
C#	GreenShot	0.00113053	0.000929968	⊖	0.001087	0.000920	⊖
	ImgSeqScan	0	0.0255783	⊕	0.014523	0.027462	⊕
	Capital Resource	0.00016982	0.000986181	⊕	0.001175	0.000912	⊖
	MonoOSC	0.00128906	0.00391499	⊕	0.001618	0.004126	⊕

MP_c= Modification Probability of Cloned Code

MP_n= Modification Probability of Non-Cloned Code   Rem = Remark

⊕= $MP_c < MP_n$ (Category 1, CLONES MORE STABLE)

⊖= $MP_c > MP_n$ (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 7   Count of (⊖) = 14   Count of (○) = 3

Type Centric Analysis for Each Language: For this analysis we draw the graph in Fig. 4.4 from Table 4.5. According to this graph, *both Type 2 and Type 3 clones of programming language C have a very high probability of getting more frequent changes compared to non-cloned code.* From the graphs in Fig. 4.3, and 4.4 we can say that *although each of the three types of clones (Type 1, Type 2, Type 3) appear to receive less frequent changes in general, Type 2 and Type 3 clones are likely to receive more frequent changes than non-cloned code in case of the subject systems written in C.*

4.6.2 Analysis of the Experimental Results Regarding Modification Probability

Using our implementation of Göde et al.'s [40] methodology we calculated the modification probabilities of cloned and non-cloned code for each of the 12 subject systems using the equations: Eq. 4.3 and Eq. 4.4. The combined type and individual type results are shown in Tables 4.6 and 4.7 respectively. For a particular significant decision point in these tables,

(i) if $MP_c < MP_n$, then clone has lower probability of getting changes compared to the probability of non-cloned code and this point falls in Category 1 (CLONES MORE STABLE)

Table 4.7: Modification Probability by Göde et al.'s methodology (Individual Type Result)

Lang	Systems	Type 1			Type 2			Type 3		
		MP_c	MP_n	Rem	MP_c	MP_n	Rem	MP_c	MP_n	Rem
Java	DNSJava	0.004926	0.000641	$\ominus$	0.002260	0.001636	$\ominus$	0.001980	0.001628	$\ominus$
	Ant-Contrib	0.005800	0.000728	$\ominus$	0.000359	0.000855	$\oplus$	0.001079	0.000796	$\ominus$
	Carol	0.000599	0.000818	$\oplus$	0.000521	0.000826	$\oplus$	0.000928	0.000780	$\ominus$
	jabref	0.000390	0.000466	$\oplus$	0.000412	0.000467	$\oplus$	0.000405	0.000469	$\oplus$
C	Ctags	0.001048	0.000958	$\ominus$	0.001157	0.000955	$\ominus$	0.000689	0.000981	$\oplus$
	Camellia	0.001655	0.000572	$\ominus$	0.002405	0.000567	$\ominus$	0.001539	0.000551	$\ominus$
	QMail Admin	0.005822	0.003326	$\ominus$	0.006150	0.003326	$\ominus$	0.007129	0.003052	$\ominus$
	Gnumakeuniproc	0.000304	0.000328	○	0.000590	0.000319	$\ominus$	0.000660	0.000308	$\ominus$
C#	GreenShot	0.001360	0.000934	$\ominus$	0.000778	0.000944	$\oplus$	0.001000	0.000938	$\ominus$
	ImgSeqScan	0	0.024880	$\oplus$	0	0.025192	$\oplus$	0	0.025340	$\oplus$
	Capital Resource	0	0.000988	$\oplus$	0	0.000978	$\oplus$	0.000209	0.000983	$\oplus$
	MonoOSC	0.001018	0.003862	$\oplus$	0.001246	0.003833	$\oplus$	0.001177	0.003902	$\oplus$

MP_c= *Modification Probability of Cloned Code*

MP_n= *Modification Probability of Non-Cloned Code* *Rem = Remark*

$\oplus$= $MP_c < MP_n$ *(Category 1, CLONES MORE STABLE)*

$\ominus$= $MP_c > MP_n$ *(Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of ($\oplus$) = *17* *Count of* ($\ominus$) = *18* *Count of* (○) = *1*

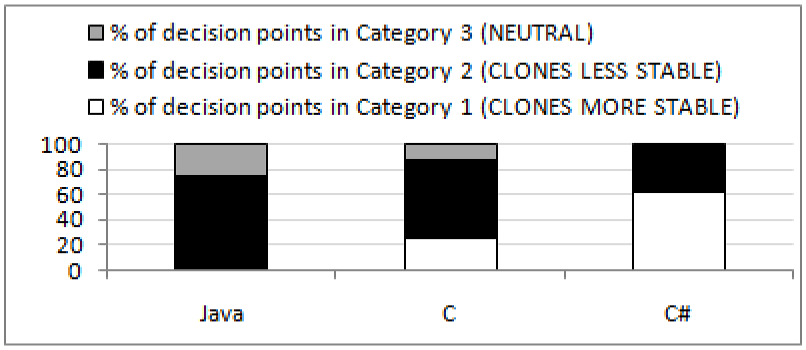


Figure 4.5: Language centric statistics for modification probability (MP)

(ii) if $MP_c > MP_n$, then clone has higher probability of getting changes compared to the probability of non-cloned code and this point falls in Category 2 (CLONES LESS STABLE)

We answer the second research question RQ 2 from four dimensions as follows.

Overall analysis: Among 24 decision points in Table 4.6: (i) 29.2% points (seven points) agree with lower probability of modifications to the cloned code (Category 1, CLONES MORE STABLE), (ii) 58.3% points suggest higher modification probability of cloned code compared to non-cloned code (Category 2, CLONES LESS STABLE), and (iii) the remaining 12.5% points (three points) are insignificant. This indicates that *the modification probability of cloned code is comparatively higher than that of non-cloned code in general.* In other words, *the proportion of source code lines affected in the cloned regions in a single commit operation is generally greater than the proportion of lines affected in the non-cloned regions.*

Language centric analysis: Considering the decision points of Table 4.6 we draw a graph in Fig. 4.5 for this analysis. We observe that (Fig. 4.5), 75% of the decision points belonging to Java programming language show higher modification probability of cloned code (Category 2, CLONES LESS STABLE). The remaining 25% points are insignificant (Category 3, NEUTRAL). In case of C, 62.5% of the decision points suggest cloned code to be more unstable (Category 2, CLONES LESS STABLE). So, *cloned code in both C and Java programming languages shows higher modification probability (higher instability) than non-cloned code.* But, an opposite scenario has been exhibited by the subject systems written in C#. Only 37.5% of the decision points belonging to this language shows higher probability of modifications in cloned code.

Type Centric Analysis: We constructed the graph in Fig. 4.6 from Table 4.7 for observing the type centric statistics. According to this graph, *both Type 1 and Type 3 clones are likely to exhibit higher modification probability compared to the Type 2 clones.* In other words, *both Type 1 and Type 3 clones have higher probability of getting modified compared to the probability of Type 2 clones.*

Type centric analysis for each language: We draw the graph in Fig. 4.7 from Table 4.7 for this analysis. According to the graph, *each of the three clone-types (Type 1, Type 2, and Type 3) of C exhibits higher modification probability compared to non-cloned code.* However, *in case of Java, only Type 1 and Type 3 are unstable* because, respectively 50% and 75% of the points belonging to these two types show higher modification probability of cloned code.

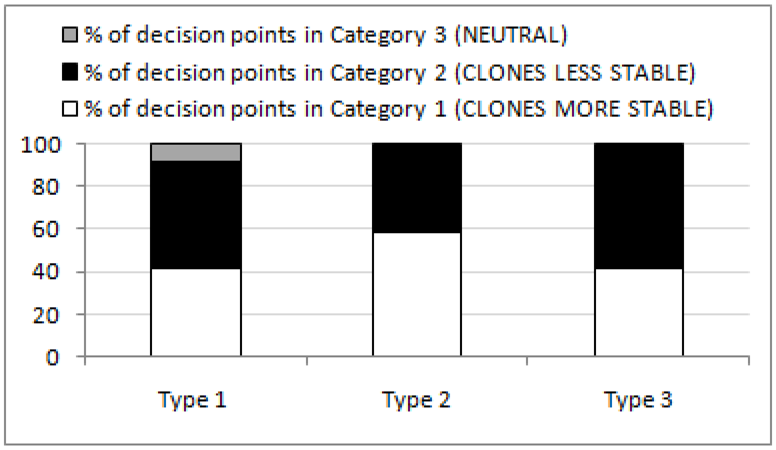


Figure 4.6: Type centric statistics for modification probability (MP)

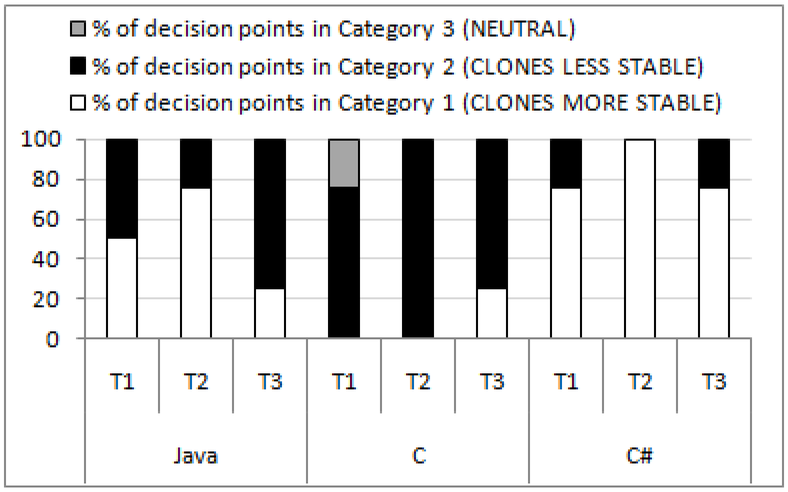


Figure 4.7: Language-wise type centric statistics for modification probability (MP)

Table 4.8: Average Last Change Dates by Krinke's methodology (Combined Type Result)

Lang	Systems	NiCad Combined $ALCD_c$	NiCad Combined $ALCD_n$	NiCad Combined Rem	CCFinder Combined $ALCD_c$	CCFinder Combined $ALCD_n$	CCFinder Combined Rem
Java	DNSJava	18-Nov-04	1-Jul-04	$\ominus$	17-Aug-04	2-Jul-04	$\ominus$
	Ant-Contrib	26-Aug-06	10-Aug-06	$\ominus$	16-Dec-06	1-Aug-06	$\ominus$
	Carol	14-Aug-07	23-Aug-07	$\oplus$	14-Aug-07	23-Aug-07	$\oplus$
	jabref	03-May-06	05-Jun-06	$\oplus$	21-Feb-06	8-Jun-06	$\oplus$
C	Ctags	24-Oct-06	27-Mar-07	$\oplus$	15-Nov-05	04-Jan-07	$\oplus$
	Camellia	5-Oct-08	10-Nov-07	$\ominus$	26-Jan-08	13-Nov-07	$\ominus$
	QMail Admin	2-Dec-03	27-Oct-03	$\ominus$	13-Oct-03	28-Oct-03	$\oplus$
	Gnumakeuniproc	10-Jul-09	18-Oct-09	$\oplus$	22-Jul-09	04-Oct-09	$\oplus$
C#	GreenShot	23-Jun-10	18-Jun-10	$\ominus$	13-Jun-10	19-Jun-10	$\oplus$
	ImgSeqScan	19-Jan-11	13-Jan-11	$\ominus$	18-Jan-11	12-Jan-11	$\ominus$
	Capital Resource	11-Dec-08	12-Dec-08	$\oplus$	13-Dec-08	12-Dec-08	$\ominus$
	MonoOSC	14-Mar-09	31-Mar-09	$\oplus$	22-Apr-09	01-Mar-09	$\ominus$

$ALCD_c$= Average Last Change Date of Cloned Code

$ALCD_n$= Average Last Change Date of Non-Cloned Code *Rem = Remark*

$\oplus$= *$ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)*

$\ominus$= *$ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)*

$\bigcirc$= *The decision point falls in Category 3*

Count of ($\oplus$) = 12 *Count of ($\ominus$) = 12* *Count of ($\bigcirc$) = 0*

Table 4.9: Average Last Change Dates by Krinke's methodology (Individual Type Result)

Lang	Clone Types / Systems	Type 1 $ALCD_c$	Type 1 $ALCD_n$	Type 1 Rem	Type 2 $ALCD_c$	Type 2 $ALCD_n$	Type 2 Rem	Type 3 $ALCD_c$	Type 3 $ALCD_n$	Type 3 Rem
Java	DNSJava	26-Aug-04	4-Jul-04	⊖	16-Nov-04	2-Jul-04	⊖	8-Dec-04	1-Jul-04	⊖
	Ant-Contrib	14-Sep-06	11-Aug-06	⊖	17-Jul-06	11-Aug-06	⊕	1-Sep-06	10-Aug-06	⊖
	Carol	8-Sep-07	22-Aug-07	⊖	7-Sep-07	22-Aug-07	⊖	6-Aug-07	23-Aug-07	⊕
	jabref	13-Jun-06	4-Jun-06	⊖	7-Feb-06	5-Jun-06	⊕	13-May-06	5-Jun-06	⊕
C	Ctags	18-Mar-08	30-Dec-06	⊖	29-Dec-06	31-Dec-06	⊕	05-Jan-07	31-Dec-06	⊖
	Camellia	4-Nov-07	14-Nov-07	⊕	17-Jul-08	14-Nov-07	⊖	8-Feb-09	9-Nov-07	⊖
	QMail Admin	7-Nov-03	27-Oct-03	⊖	13-Nov-03	27-Oct-03	⊖	11-Nov-03	27-Oct-03	⊖
	Gnumake Uniproc	27-Aug-09	28-Sep-09	⊕	18-Oct-09	27-Sep-09	⊖	19-Oct-09	27-Sep-09	⊖
C#	GreenShot	8-Jun-10	18-Jun-10	⊕	19-Jun-10	18-Jun-10	⊖	23-Jun-10	18-Jun-10	⊖
	ImgSeqScan	19-Jan-11	13-Jan-11	⊖	14-Jan-11	13-Jan-11	⊖	19-Jan-11	13-Jan-11	⊖
	Capital Resource	13-Dec-08	12-Dec-08	⊖	10-Dec-08	12-Dec-08	⊕	11-Dec-08	12-Dec-08	⊕
	MonoOSC	25-Jun-09	21-Mar-09	⊖	14-Mar-09	21-Mar-09	⊕	26-Dec-08	22-Mar-09	⊕

$ALCD_c$= Average Last Change Date of Cloned Code

$ALCD_n$= Average Last Change Date of Non-Cloned Code *Rem = Remark*

⊕= $ALCD_c$ is older than $ALCD_n$ (Category 1, CLONES MORE STABLE)

⊖= $ALCD_c$ is newer than $ALCD_n$ (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 12 *Count of (⊖) = 24* *Count of (○) = 0*

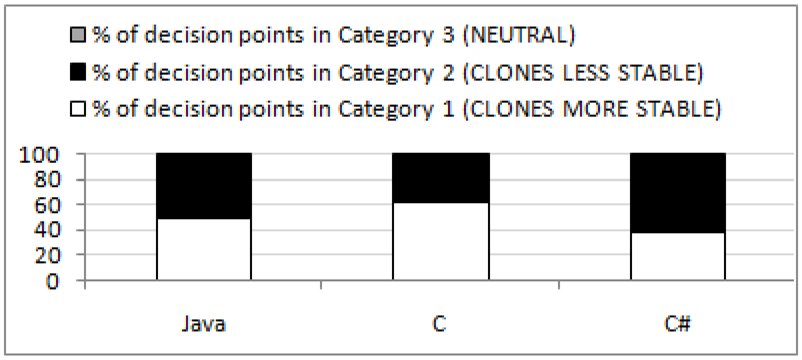


Figure 4.8: Language centric statistics for average last change date (ALCD)

4.6.3 Analysis of the Experimental Results for Average Last Change Date

We calculated the average last change dates of the cloned ($ALCD_c$) and non-cloned ($ALCD_n$) code of the selected last revisions (Table 4.3) of the candidate subject systems by applying our implementation of Krinke's Methodology [65]. $ALCD_c$ and $ALCD_n$ were calculated using the equations Eq. 4.5 and Eq. 4.6 respectively. Tables 4.8 and 4.9 contain the combined type and individual type results respectively. We mentioned (with explanation) that for categorizing the decision points in the tables for this metrics we did not use Eq. 4.22. If two dates corresponding to a particular decision point are different, we considered the decision point as a significant one. For a particular significant decision point in these tables,

(i) if $ALCD_c$ is older than $ALCD_n$, then this point falls in Category 1 (CLONES MORE STABLE), because for this point, changes to non-cloned code are more recent on an average compared to the changes to the cloned code.

(ii) if $ALCD_n$ is older than $ALCD_c$, then this point falls in Category 2 (CLONES LESS STABLE), because for this point, changes to the cloned code are more recent on an average compared to the changes to non-cloned code.

The following four sections of analysis answer the third research question RQ 3 from four dimensions.

Overall analysis: All the decision points (24 in total) of Table 4.8 are significant. According to 50% of these points, average last change date of cloned code is younger than that of non-cloned code. The remaining 50% points suggest that non-cloned code is more lately changed. Although the proportions are same, the observed scenario indicates that *cloned code can be changed more lately as compared to non-cloned code during the evolution of a software system.*

Language centric analysis: Fig. 4.8 constructed from Table 4.8 helps us to take language centric decision for this metric. According to this graph, *the cloned code in the subject systems written in C# has higher probability of being changed more lately compared to non-cloned code. In other words, the clones in C# appear to be more unstable compared to non-cloned code. An opposite scenario has been shown by the subject systems written in C. In case of Java, the same proportion of decision points belong to both of the two categories: Category 1 and Category 2.*

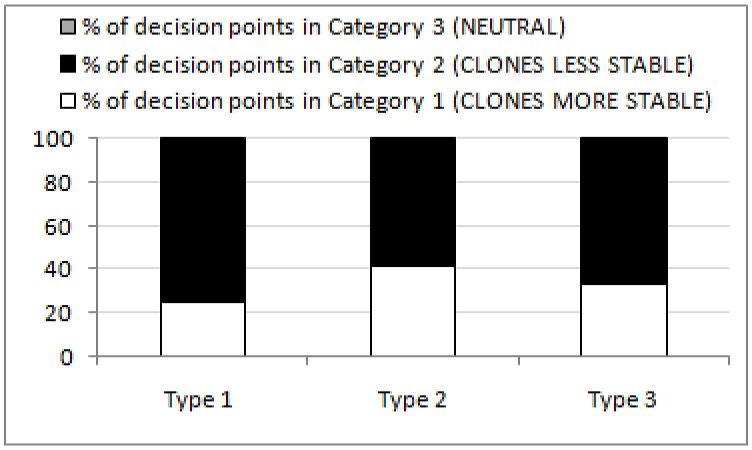


Figure 4.9: Type centric statistics for average last change date (ALCD)

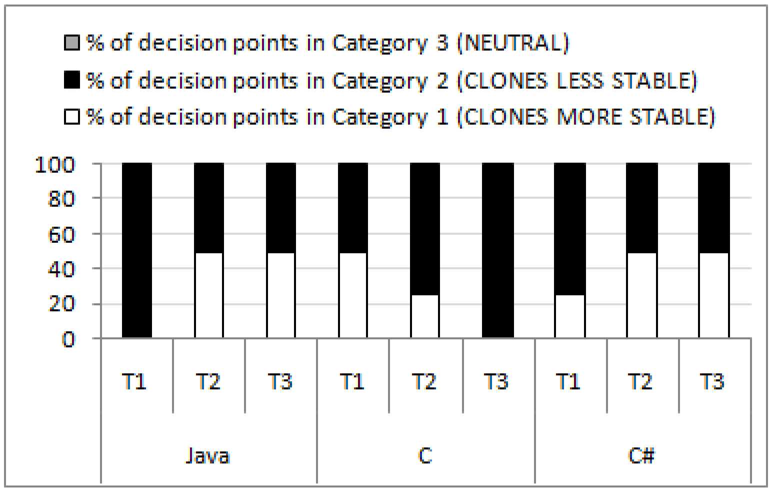


Figure 4.10: Language-wise type centric statistics for average last change date (ALCD)

Type centric analysis: According to the type centric statistics of the graph in Fig. 4.9, *each of the three types of clones are likely to be modified more lately as compared to non-cloned code.* The graph also indicates that *both Type 1 and Type 3 clones have higher probability of getting more recent changes in comparison with the Type 2 clones.*

Type Centric Analysis for Each Language: According to the graph in Fig. 4.10, for all of the significant points belonging to Type 1 case of Java, average last change date of cloned code is younger than that of non-cloned code. The same is true for Type 3 clones of C. Type 2 clones of C and Type 1 clones of C# also show higher probabilities of being changed more lately. For all other cases, both cloned and non-cloned code have equal possibilities of being more lately changed. So, according to this metric, *each of the three clone-types of the selected programming languages is a threat to software stability during the maintenance phase with Type 1 clones of Java and Type 3 clones of C being the most vulnerable ones.*

Table 4.10: Average Ages of Cloned and Non-cloned Code (Combined Type Result)

Lang	Systems	NiCad Combined			CCFinder Combined		
		AA_c	AA_n	Rem	AA_c	AA_n	Rem
Java	DNSJava	2303.49	2443.15	⊖	2395.84	2442.42	⊖
	Ant-Contrib	887.52	903.65	⊖	776.27	912.83	⊖
	Carol	218.55	210.24	⊕	218.89	209.82	⊕
	jabref	1104.97	1071.88	⊕	1175.59	1068.58	⊕
C	Ctags	1498.10	1344.02	⊕	1798.81	1339.45	⊕
	Camellia	730.55	1060.50	⊖	984.28	1057.77	⊖
	QMail Admin	2638.87	2674.94	⊖	2689.10	2674.36	⊕
	Gnumakeuniproc	301.02	200.93	⊕	288.06	215.16	⊕
C#	GreenShot	278.35	283.30	⊖	287.56	281.98	⊕
	ImgSeqScan	14.39	20.37	⊖	14.55	20.65	⊖
	Capital Resource	88.03	86.56	⊕	85.65	86.64	○
	MonoOSC	330.32	313.41	⊕	276.45	341.93	⊖

AA_c= Average Age of Cloned Code.

AA_n= Average Age of Non-cloned Code. *Rem = Remark*

⊕= $AA_c > AA_n$ (Category 1, CLONES MORE STABLE)

⊖= $AA_c < AA_n$ (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 12 *Count of (⊖) = 11* *Count of (○) = 1*

Table 4.11: Average Ages of Cloned and Non-cloned Code (Individual Type Result)

Lang	Systems	Type 1			Type 2			Type 3		
		AA_c	AA_n	Rem	AA_c	AA_n	Rem	AA_c	AA_n	Rem
Java	DNSJava	2386.51	2440.31	⊖	2304.85	2442	⊖	2282.94	2443.23	⊖
	Ant-Contrib	869.21	903.39	⊖	927.56	902.93	⊕	881.61	903.73	⊖
	Carol	193.61	210.75	⊖	194.65	210.85	⊖	226.65	210.04	⊕
	jabref	1063.87	1072.57	⊖	1189.77	1071.64	⊕	1094.87	1072.13	⊕
C	Ctags	1050.05	1345.77	⊖	1425.36	1344.75	⊕	1433.15	1344.46	⊕
	Camellia	1066.84	1056.77	⊕	810.96	1057.39	⊖	604.95	1062.48	⊖
	QMail Admin	2664.22	2674.61	⊖	2658.24	2674.62	⊖	2660.30	2674.63	⊖
	Gnumakeuniproc	255.37	221.64	⊕	213.91	222.74	⊖	212.97	222.74	⊖
C#	GreenShot	292.88	282.61	⊕	282.24	283.23	⊖	278.35	283.30	⊖
	ImgSeqScan	14.0	20.26	⊖	18.66	20.25	⊖	14.39	20.37	⊖
	Capital Resource	86.35	86.59	○	89.09	86.57	⊕	88.03	86.56	⊕
	MonoOSC	224.43	314.06	⊖	325.65	313.54	⊕	389.37	312.49	⊕

AA_c= Average Age of Cloned Code.

AA_n= Average Age of Non-cloned Code. *Rem = Remark*

⊕= *$AA_c > AA_n$ (Category 1, CLONES MORE STABLE)*

⊖= *$AA_c < AA_n$ (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 13 *Count of (⊖) = 22* *Count of (○) = 1*

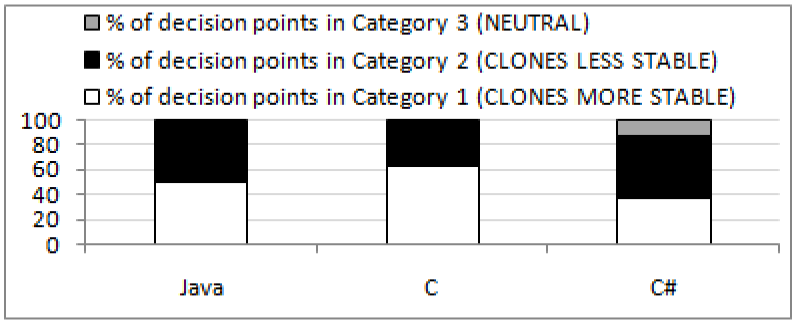


Figure 4.11: Language centric statistics for average age (AA)

4.6.4 Analysis of the Experimental Results Regarding Average Age

By applying our variant [83] of Krinke's methodology [65] we calculated the arithmetic average ages (in days) of cloned (AA_c) and non-cloned (AA_n) code of a subject system considering its last revision as mentioned in the Table 4.3. AA_c and AA_n were calculated according to the equations: Eq. 4.8 and Eq. 4.9 respectively. We present our obtained data for this metric in Tables 4.10 (combined type results) and 4.11 (individual type results). For a particular significant decision point in these tables,

(i) if $AA_c > AA_n$, then cloned code is more stable than non-cloned code for this point and this point belongs to Category 1 (CLONES MORE STABLE).

(ii) if $AA_c < AA_n$, then cloned code is less stable than non-cloned code for this point and this point belongs to Category 2 (CLONES LESS STABLE).

We answer the fourth research question RQ-4 regarding average age of cloned and non-cloned code in the following four sections.

Overall analysis: The Table 4.10 contains 24 decision points in total. 50% (twelve points) of these points suggest that cloned code remains unchanged for longer time compared to non-cloned code. These points fall in Category 1 (CLONES MORE STABLE). 45.8% points (eleven points) suggest the opposite and these points belong to Category 2 (CLONES LESS STABLE). The remaining point is not a significant one. We see that *the highest proportion of decision points agree with higher longevity of clones compared to non-cloned code*. However, *although clones appear to be more stable than non-cloned code in general, clones often exhibit higher instability compared to non-cloned code according to this metric.*

Language centric analysis: The language centric statistics (Fig. 4.11 constructed from Table 4.10) of this metric are almost the same as those of average last change date. The reason behind this is that each of these metrics are computed considering the last revision of a candidate subject system. We see that *the clones in C# appear to be more unstable compared to non-cloned code. An opposite scenario is exhibited by the decision points in C. In case of Java, the same proportion of decision points (50%) belong to both Category 1 (CLONES MORE STABLE) and Category 2 (CLONES LESS STABLE).*

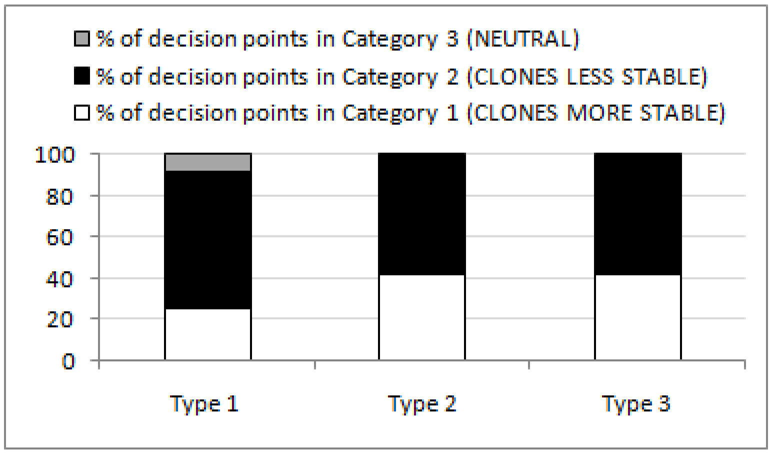


Figure 4.12: Type centric statistics for average age (AA)

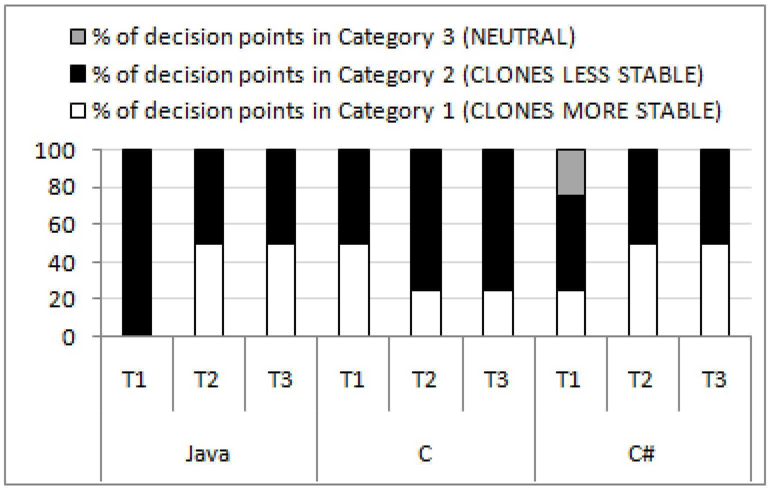


Figure 4.13: Language-wise type centric statistics for average age (AA)

Type centric analysis: The type centric statistics (Fig. 4.12) regarding *average age* follows the type centric statistics of *average last change date.* The reason behind this has already been described in the language centric analysis regarding average age (AA). We see that *each of the three types of clones has higher probability of getting changed more lately compared to non-cloned code.* In other words, *each clone-type appears to be more unstable than non-cloned code according to this metric.*

Type Centric Analysis for Each Language: The language-wise type centric analysis result is shown in the graph of Fig. 4.13 constructed from Table 4.11. This graph is almost the same as that we obtained for average last change date. All of the decision points belonging to Type 1 case of Java indicate smaller longevity of cloned code. For Type 2 and Type 3 cases of C and Type 1 case of C#, major portions of the decision points suggest that non-cloned code lives longer than cloned code. For the remaining 5 cases, both cloned and non-cloned code exhibit equal probability of being unstable. So, according to this metric, *each* *of the clone types of each candidate programming language exhibits high probability of being more unstable compared to non-cloned code.*

We see that the language centric statistics (Fig. 4.11) is not agreeing with both type centric statistics (Fig. 4.12) and language-wise type centric statistics (Fig. 4.13). While language centric statistics suggest cloned code to be more stable than non-cloned code for two languages Java and C, type centric statistics for these two languages (Fig. 4.13) suggest cloned code to be less stable in general. The fact is that Fig. 4.11 and Fig. 4.13 have been constructed from two different tables: Table 4.10 and Table 4.11 respectively. While Table 4.10 contains the combined type results of both NiCad and CCFinderX, Table 4.11 contains the individual type results of NiCad only. The graph in Fig. 4.11 is influenced by CCFinderX results. If we consider only the NiCad results in Table 4.10, we see that the same proportion (50%) of points belong to both Category 1 (CLONES MORE STABLE) and Category 2 (CLONES LESS STABLE). However, if we consider only the CCFinderX results in Table 4.10, we see that the proportions of points belonging to Category 1 and Category 2 are 50% and 41.7% respectively. In other words, the highest proportion of decision points belonging to CCFinderX results in Table 4.10 suggest cloned code to be more stable. Thus, because of the influence of CCFinderX results on Fig. 4.11 this figure is not agreeing with the other two figures: Fig. 4.12 and Fig. 4.13.

4.6.5 Analysis of the Experimental Results Regarding Impact and Likelihood

We calculate the followings using our implementation of the methodology proposed by Lozano and Wermelinger [75].

(1) Impact of cloned code (ICC) using Eq. 4.10.

(2) Impact of non-cloned code (INC) using Eq. 4.12.

(3) Lilkelihood of cloned code (LCC) using Eq. 4.14.

(4) Likelihood of non-cloned code (LNC) using Eq. 4.16.

The combined type and individual type results for impact and likelihood of cloned and non-cloned code are shown in Tables 4.12, 4.13, 4.14 and 4.15 respectively. For a particular significant decision point contained in Tables 4.12 and 4.13,

(i) if $ICC < INC$, then this point falls in Category 1 (CLONES MORE STABLE)

(ii) if $ICC > INC$, then this point falls in Category 2 (CLONES LESS STABLE)

Also, for a particular significant decision point contained in Table 4.14 and 4.15,

(i) if $LCC < LNC$, then this point falls in Category 1 (CLONES MORE STABLE), because for this point, cloned code is less likely to be changed compared to non-cloned code.

(ii) if $LCC > LNC$, then this point falls in Category 2 (CLONES LESS STABLE), because for this point, cloned code is more likely to be changed than non-cloned code.

We answer the fifth and sixth research questions (RQ 5 and RQ 6) regarding impact and likelihood in the following eight sections.

Table 4.12: Impact of cloned and non-cloned code by the methodology of Lozano and Wermelinger (Combined Type Result)

Lang	**Systems**	**NiCad Combined**			**CCFinder Combined**		
		ICC	*INC*	Rem	*ICC*	*INC*	Rem
Java	DNSJava	0.003330	0.002450	⊖	0.003093	0.002309	⊖
	Ant-Contrib	0.015564	0.015398	○	0.014623	0.014612	○
	Carol	0.002528	0.001489	⊖	0.002269	0.001651	⊖
	jabref	0.004282	0.003048	⊖	0.003453	0.002901	⊖
C	Ctags	0.002770	0.002547	○	0.002832	0.002869	○
	Camellia	0.013323	0.013170	○	0.013722	0.012873	○
	QMail Admin	0.030669	0.040555	⊕	0.011889	0.042659	⊕
	Gnumakeuniproc	0.062269	0.026350	⊖	0.026147	0.023723	⊖
C#	GreenShot	0.003607	0.003220	⊖	0.003030	0.002906	○
	ImgSeqScan	0	0.051490	⊕	0.035525	0.067790	⊕
	Capital Resource	0.012300	0.011517	○	0.011333	0.011554	○
	MonoOSC	0.005366	0.007618	⊕	0.009784	0.007366	⊖

ICC= Impact of Cloned Code

INC= Impact of Non-Cloned Code *Rem = Remark*

⊕*= ICC <INC (Category 1, CLONES MORE STABLE)*

⊖*= ICC >INC (Category 2, CLONES LESS STABLE)*

○*= The decision point falls in Category 3*

Count of (⊕) = 5 *Count of (⊖) = 10* *Count of (○) = 9*

Table 4.13: Impact of cloned and non-cloned code by the methodology of Lozano and Wermelinger (Individual Type Result)

Lang	**Systems**	**Type 1**			**Type 2**			**Type 3**		
		ICC	*INC*	Rem	*ICC*	*INC*	Rem	*ICC*	*INC*	Rem
Java	DNSJava	0.004306	0.001593	⊖	0.003644	0.002289	⊖	0.003378	0.002441	⊖
	Ant-Contrib	0.035714	0.014431	⊖	0.017205	0.014818	⊖	0.0161872	0.015604	○
	Carol	0.002214	0.001302	⊖	0.002166	0.001318	⊖	0.002670	0.001446	⊖
	jabref	0.003630	0.003038	⊖	0.003237	0.003097	○	0.003648	0.003116	⊖
C	Ctags	0.0034611	0.002582	⊖	0.002533	0.002818	⊕	0.002510	0.002524	○
	Camellia	0.014529	0.013044	⊖	0.013855	0.013020	○	0.014032	0.013155	○
	QMail Admin	0.037046	0.027128	⊖	0.120279	0.013481	⊖	0.031811	0.020772	⊖
	Gnumake Uniproc	0.030799	0.027795	⊖	0.025762	0.025433	○	0.061090	0.025809	⊖
C#	GreenShot	0.003980	0.003020	⊖	0.003310	0.003074	○	0.003917	0.003133	⊖
	ImgSeqScan	0	0.056784	⊕	0	0.051490	⊕	0	0.051490	⊕
	Capital Resource	0	0.011515	⊕	0	0.011515	⊕	0.012300	0.011517	○
	MonoOSC	0.004739	0.007185	⊕	0.055743	0.007391	⊖	0.005750	0.007577	⊕

ICC= Impact of Cloned Code

INC= Impact of Non-Cloned Code *Rem = Remark*

⊕= *ICC <INC (Category 1, CLONES MORE STABLE)*

⊖= *ICC >INC (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 8 *Count of (⊖) = 20* *Count of (○) = 8*

Table 4.14: Likelihood of changes of cloned and non-cloned methods by the methodology of Lozano and Wermelinger (Combined Type Result)

Lang	**Systems**	**NiCad Combined**			**CCFinder Combined**		
		LCC	*LNC*	Rem	*LCC*	*LNC*	Rem
Java	DNSJava	0.010594	0.006741	⊖	0.011812	0.005109	⊖
	Ant-Contrib	0.152325	0.105432	⊖	0.064301	0.091879	⊕
	Carol	0.020661	0.010073	⊖	0.023418	0.011425	⊖
	jabref	0.008857	0.002179	⊖	0.004541	0.002599	⊖
C	Ctags	0.033359	0.013643	⊖	0.010325	0.007249	⊖
	Camellia	0.046392	0.019575	⊖	0.020016	0.035980	⊕
	QMail Admin	0.038644	0.016072	⊖	0.028739	0.018757	⊖
	Gnumakeuniproc	0.100809	0.080028	⊖	0.073717	0.099099	⊕
C#	GreenShot	0.022600	0.017880	⊖	0.061601	0.010974	⊖
	ImgSeqScan	0	0.279315	⊕	0.531635	0.231955	⊖
	Capital Resource	0.012811	0.035023	⊕	0.058653	0.029643	⊖
	MonoOSC	0.032967	0.030469	○	0.074932	0.030858	⊖

LCC= Likelihood of Cloned Code

LNC= Likelihood of Non-Cloned Code *Rem = Remark*

⊕= *LCC <LNC (Category 1, CLONES MORE STABLE)*

⊖= *LCC >LNC (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 5 *Count of (⊖) = 18* *Count of (○) = 1*

Table 4.15: Likelihood of changes of cloned and non-cloned methods by the methodology of Lozano and Wermelinger (Individual Type Result)

Lang	Systems	Type 1			Type 2			Type 3		
		LCC	*LNC*	Rem	*LCC*	*LNC*	Rem	*LCC*	*LNC*	Rem
Java	DNSJava	0.010465	0.006079	⊖	0.010300	0.004838	⊖	0.011197	0.006453	⊖
	Ant-Contrib	0.75	0.050016	⊖	0.053453	0.080826	⊕	0.168700	0.098941	⊖
	Carol	0.024513	0.007073	⊖	0.025205	0.007664	⊖	0.023643	0.009888	⊖
	jabref	0.012831	0.001861	⊖	0.014853	0.002014	⊖	0.009401	0.003026	⊖
C	Ctags	0.065153	0.008283	⊖	0.024657	0.008239	⊖	0.069739	0.013531	⊖
	Camellia	0.023832	0.020232	○	0.040466	0.019938	⊖	0.039178	0.019385	⊖
	QMail Admin	0.030872	0.022933	⊖	0.047893	0.018799	⊖	0.041239	0.021687	⊖
	Gnumakeuniproc	0.256817	0.140758	⊖	0.040816	0.081776	⊕	0.094228	0.068551	⊖
C#	GreenShot	0.125183	0.014503	⊖	0.022490	0.016716	⊖	0.035680	0.016386	⊖
	ImgSeqScan	0	0.304426	⊕	0	0.279315	⊕	0	0.279315	⊕
	Capital Resource	0	0.032684	⊕	0	0.032684	⊕	0.012811	0.035023	⊕
	MonoOSC	0.061955	0.032516	⊖	0.119999	0.027509	⊖	0.018143	0.030559	⊕

LCC = Likelihood of Cloned Code

LNC = Likelihood of Non-Cloned Code   *Rem = Remark*

⊕= *LCC <LNC (Category 1, CLONES MORE STABLE)*

⊖= *LCC >LNC (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 9   *Count of (⊖) = 26*   *Count of (○) = 1*

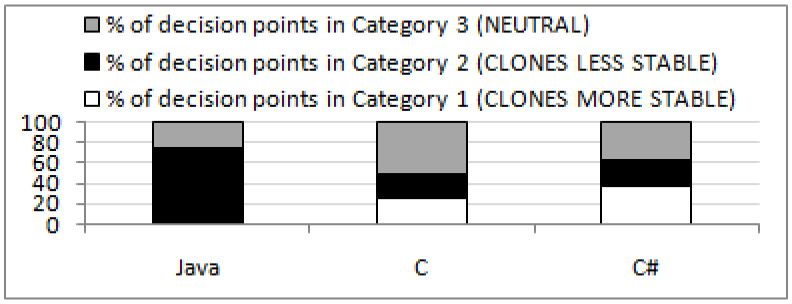


Figure 4.14: Language centric statistics for impact

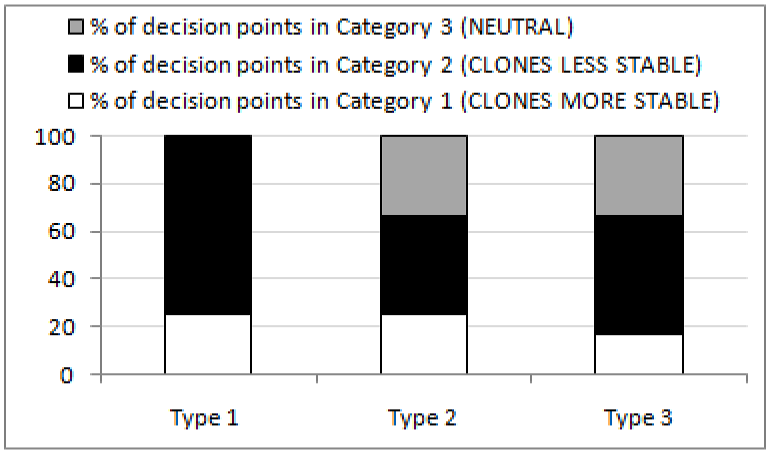


Figure 4.15: Type centric statistics for impact

Overall analysis for impact: Table 4.12 contains 24 decision points in total. Fifteen points are significant and the remaining nine points are insignificant. For 41.7% of the points, cloned code has higher impact than the non-cloned code (Category 2, CLONES LESS STABLE). Only 20.8% points suggest that cloned code has comparatively lower impact (Category 1, CLONES MORE STABLE). The remaining 37.5% points are insignificant. The strong difference between the percentages of decision points belonging to Category 1 and Category 2 indicate that *cloned code has higher impact than non-cloned code in the maintenance phase.* In other words, *the average number of co-changed methods for a change in a cloned method is higher than the average number of co-changed methods for a change in a non-cloned method.*

Language centric analysis for impact: The language centric analysis for this metric is shown in the graph of Fig. 4.14. We see that all of the significant points belonging to Java programming language (Table 4.12) suggest that cloned code has higher impact than non-cloned code (Category 2, CLONES LESS STABLE). However, in case of C#, higher proportion of significant decision points suggest lower impact of cloned code (Category 1, CLONES MORE STABLE). For C, the same proportion (25%) of decision points belong to both Category 1 and Category 2. According to this metric, *clones in Java programming language show higher instability compared to non-cloned code in the maintenance phase.*

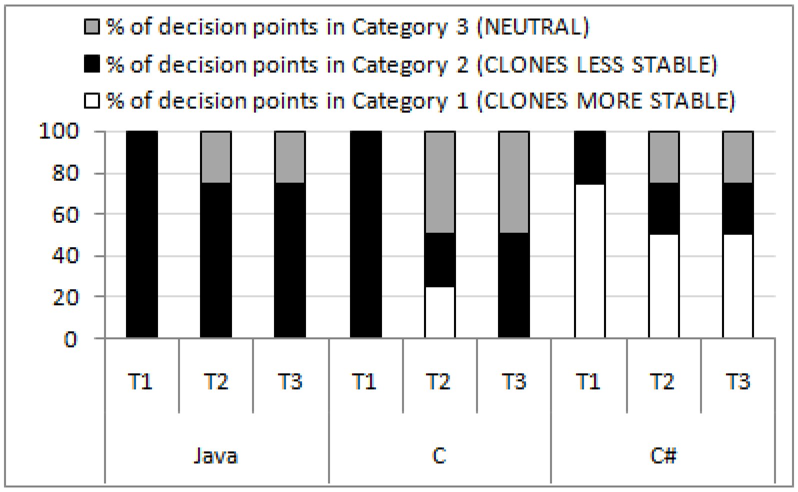


Figure 4.16: Language-wise type centric statistics for impact

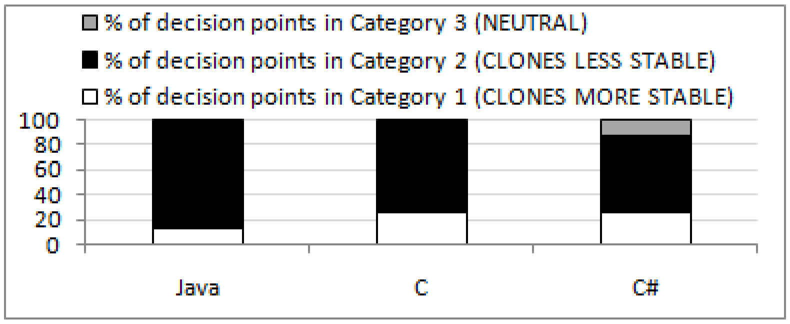


Figure 4.17: Language centric statistics for likelihood

Type centric analysis for impact According to the type centric analysis regarding impact (Fig. 4.15), *for each of the three clone types, the impact of changing cloned methods is generally higher than the impact of changing non-cloned methods.* In other words, *according to this metric (impact), each type of clones is more unstable compared to the non-cloned code.*

Type Centric Analysis for Each Language Regarding Impact According to the type centric statistics shown in the graph of Fig. 4.16, *all of the three clone types of Java are suggested to be highly unstable in the maintenance phase. The same is true for the Type 1 and Type 3 clones of C. In case of C#, each of the three types of clones exhibits lower impact than non-cloned code during maintenance phase.*

Overall analysis for likelihood: Amond 24 decision points in Table 4.14, (i) 20.8% points (five points) suggest cloned code to be more stable (Category 1), (ii) 75% points (eighteen points) suggest cloned code to be more unstable than non-cloned code (Category 2), and (iii) the remaining 4.2% points are insignificant (Category 3). We see that the percentage of points suggesting cloned code to be more unstable is much higher than the percentage of points suggesting cloned code to be more stable. From this we can say that *cloned code is more likely to be changed than the non-cloned code in the maintenance phase.*

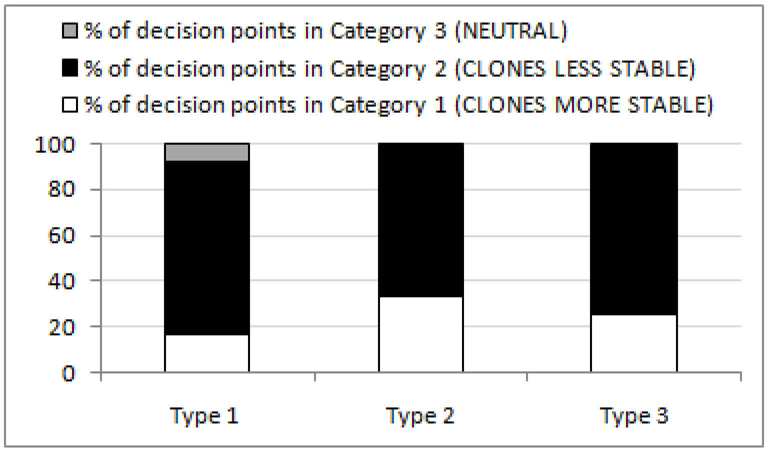


Figure 4.18: Type centric statistics for likelihood

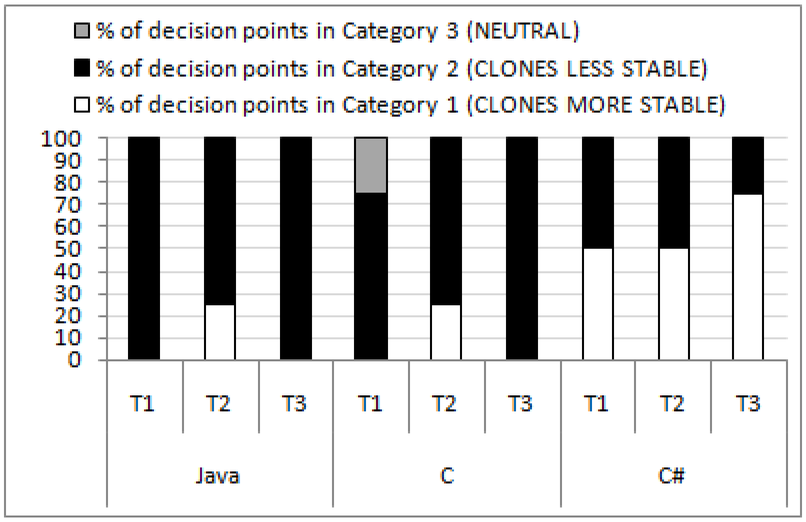


Figure 4.19: Language-wise type centric statistics for likelihood

Language centric analysis for likelihood: According to the language centric statistics of the graph in Fig. 4.17, *clones of each of the three programming languages have much higher likelihood of changes compared to non-cloned code.* So, *according to this metric, clones are generally more unstable than non-cloned code.*

Type centric analysis for likelihood According to the type centric statistics exhibited by the graph in Fig. 4.18, *for each of the three clone types, cloned code is more likely to get modified compared to non-cloned code.*

Type Centric Analysis for Each Language Regarding Likelihood: The language-wise type centric statistics of the graph in Fig. 4.19 strongly suggests that *each of the three clone-types of Java and C are highly unstable in the maintenance phase because they exhibit much higher likelihood of changes compared to non-cloned code. Also, 50% of the points belonging to both Type 1 and Type 2 case of C# suggest cloned code to be more unstable. However, Type 3 clones of this language (C#) are more stable than non-cloned code.*

Table 4.16: Average Instability per Cloned Method by the methodology of Lozano and Wermelinger (Combined Type Result)

Lang	**Systems**	**NiCad Combined**			**CCFinder Combined**		
		CPCM	*EPCM*	R	*CPCM*	*EPCM*	R
Java	DNSJava	0.3473	0.4967	⊕	0.1939	0.2748	⊕
	Ant-Contrib	0.2236	0.5346	⊕	0.0769	0.3355	⊕
	Carol	0.6383	0.6725	○	0.2216	0.3258	⊕
	jabref	0.2123	0.4322	⊕	0.1034	0.2178	⊕
C	Ctags	0.2704	0.3519	⊕	0.2021	0.3230	⊕
	Camellia	0.1606	0.3422	⊕	0.1562	0.1695	○
	QMail Admin	0.1820	0.1620	⊖	0.11	0.1057	○
	Gnumakeuniproc	0.10	0.6739	⊕	0.1935	0.1904	○
C#	GreenShot	0.5254	0.5851	⊕	0.1932	0.8054	⊕
	ImgSeqScan	0	0.6591	⊕	0.0667	0.1866	⊕
	Capital Resource	0.80	0.4635	⊖	0.3225	0.3091	○
	MonoOSC	0.8101	0.8578	○	0.2274	0.3253	⊕

CPCM= Average percentage of changes taking place to the cloned portions of cloned methods.

EPCM= Average percentage of cloning per method. *R = Remark*

⊕= *CPCM <EPCM (Category 1, CLONES MORE STABLE)*

⊖= *CPCM >EPCM (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 16 *Count of (⊖) = 2* *Count of (○) = 6*

Table 4.17: Average Instability per Cloned Method by Lozano and Wermelinger's methodology (Individual Type Result)

Lang	Systems	Type 1			Type 2			Type 3		
		CPCM	*EPCM*	Rem	*CPCM*	*EPCM*	Rem	*CPCM*	*EPCM*	Rem
Java	DNSJava	0.8547	0.7639	⊖	0.4187	0.5721	⊕	0.3829	0.5102	⊕
	Ant-Contrib	0.3888	0.3967	○	0.2307	0.3913	⊕	0.1666	0.5484	⊕
	Carol	0.3895	0.5778	⊕	0.5181	0.7023	⊕	0.6431	0.6340	○
	jabref	0.1177	0.5142	⊕	0.1413	0.4164	⊕	0.1983	0.4734	⊕
C	Ctags	0.1875	0.2887	⊕	0.2965	0.3147	○	0.2347	0.3235	⊕
	Camellia	0.0152	0.0812	⊕	0.0745	0.0852	⊕	0.1588	0.3668	⊕
	QMail Admin	0.1234	0.128	○	0.0890	0.0755	⊖	0.1844	0.1674	⊖
	Gnumakeuniproc	0.4044	0.7947	⊕	0	0.0444	⊕	0.0145	0.2918	⊕
C#	GreenShot	0.5172	0.8223	⊖	0.4418	0.9004	⊕	0.4795	0.8407	⊕
	ImgSeqScan	0	0.4705	⊕	0	0.8082	⊕	0	0.4876	⊕
	Capital Resource	0	0.8940	⊕	0	0.7332	⊕	0.80	0.4348	⊖
	MonoOSC	0.3333	0.2751	⊖	0.60	0.8502	⊕	0.90	0.9352	○

CPCM= Average percentage of changes taking place to the cloned portions of cloned methods.

EPCM= Average percentage of cloning per method. *R = Remark*

⊕= *CPCM <EPCM (Category 1, CLONES MORE STABLE)*

⊖= *CPCM >EPCM (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 25 *Count of (⊖) = 6* *Count of (○) = 5*

4.6.6 Analysis of the Experimental Results Regarding Average Instability per Cloned Method

We calculated *EPCM* and *CPCM* according to the equations Eq. 4.18 and 4.19 using our implementation of Lozano and Wermelinger's study [74]. However, the values of *EPCM* and *CPCM* that we get using these equations are percentages. We normalize these values within zero to one by dividing them by 100.

These two metrics (*EPCM* and *CPCM*) together can help us to take decision about the instabilities of cloned methods due to cloned and non-cloned code (Cloned methods might also contain non-cloned portions). The combined type and individual type results (normalized between zero to one) for these two metrics are presented in Tables 4.16 and 4.17 respectively. Our decision making procedure using these two metrics is explained below.

We have already mentioned that $EPCM$ is the average proportion of cloning in the cloned methods. Also, $CPCM$ is the average proportion of changes to the clones in the cloned methods. We take stability decisions comparing these two proportions in the following way.

(1) For a particular decision point, if the difference between $EPCM$ and $CPCM$ is not significant (the eligibility value calculated by Eq. 4.23 is less than the threshold value), then we can say that for this point cloned code is getting about that proportion of changes which it should get considering its proportion in the method. This is the ideal case (indicated by ○ in the tables 4.16, 4.17) which does not indicate any positive or negative impact of cloned code. Such a point falls in Category 3 (NEUTRAL).

(2) If $CPCM>EPCM$ with an eligibility value greater than or equal to the threshold value, we understand that cloned portions of the cloned methods are getting more changes than they would get in the ideal situation. In other words, the instability of cloned methods due to cloned portions is higher than the instability of the cloned methods due to non-cloned portions. Such decision points (marked with ⊖) belong to Category 2 (CLONES LESS STABLE).

(3) The decision points (indicated by ⊕) where $CPCM<EPCM$ with an eligibility value greater than or equal to the threshold value belong to Category 1 (CLONES MORE STABLE).

The following equation is used to calculate the eligibility value.

$$EligibilityValue = \frac{(HVal - LVal) * 100}{LVal} \tag{4.23}$$

Here, HVal stands for Higher value between $EPCM$ and $CPCM$ where LVal is elaborated as Lower value between these two. An eligibility value of at least 10 is treated as a significant one as was done for some previous cases. In the following four sections we answer the seventh research question RQ 7.

Overall analysis: Table 4.16 contains 24 decision points in total. Among these points, (1) 66.7% points (16 points) suggest higher stability of cloned code, (2) 8.3% points suggest higher instability of cloned code, and (3) the remaining 25% points belong to Category 3. We see that the percentage of points in favor of clones is much higher than the percentage of points against clones. Considering the percentages of the significant decision points we can say that *the instability caused by the cloned portions of the cloned methods is smaller than the instability caused by the non-cloned portions of the cloned methods.* In other words, *cloned code is generally more stable than non-cloned code according to this metric.*

Language centric analysis: The language centric analysis presented in the graph of Fig. 4.20 constructed from Table 4.16 suggests that *for each of the programming languages, cloned code introduces less instability to a software system than the instability introduced by non-cloned code.*

Type centric analysis According to the type centric statistics in the graph of Fig. 4.21, *the instability of the cloned methods due to each clone-type is less than the instability of the cloned methods due to non-cloned code.*

Type Centric Analysis for Each Language: Also, in case of type centric statistics for each candidate programming language (Fig. 4.22 constructed from Table 4.17) we see that *no clone types are notably unstable*

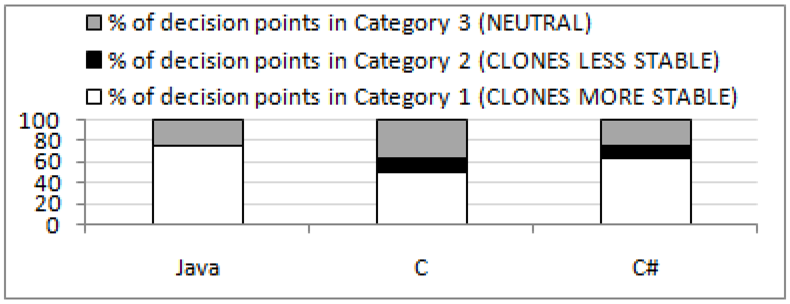


Figure 4.20: Language centric statistics for average instabilities of cloned methods (AICM)

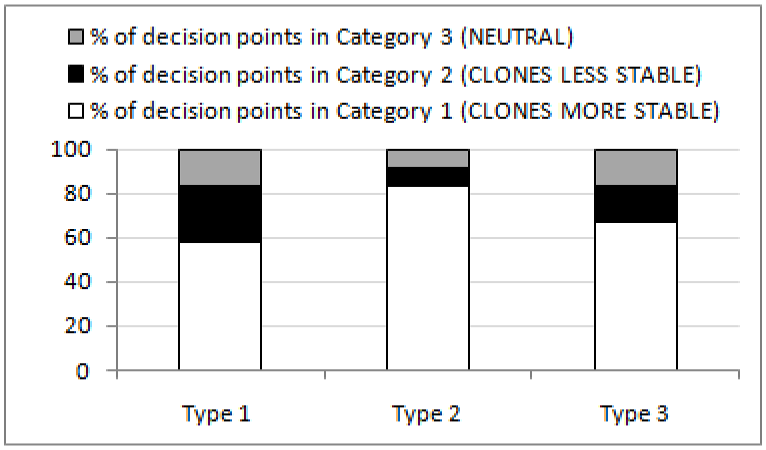


Figure 4.21: Type centric statistics for average instabilities of cloned methods (AICM)

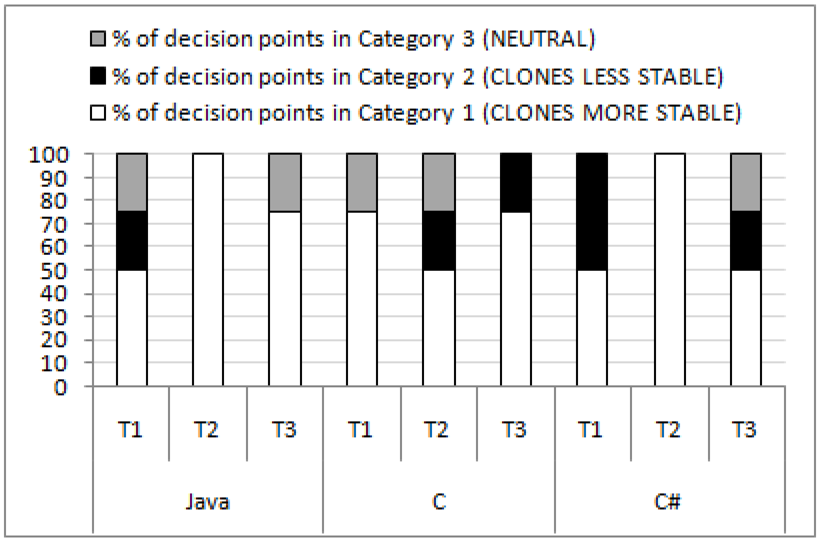


Figure 4.22: Language-wise type centric statistics for average instabilities of cloned methods (AICM)

Table 4.18: Change Dispersion in Cloned and Non-cloned Methods (Combined Type Result)

Lang	Systems	NiCad Combined			CCFinder Combined		
		CD_c	CD_n	Rem	CD_c	CD_n	Rem
Java	DNSJava	0.17	0.06	⊖	0.18	0.07	⊖
	Ant-Contrib	0.0526	0.0176	⊖	0.05	0	⊖
	Carol	0.2056	0.0912	⊖	0.23	0.07	⊖
	jabref	0.1234	0.1985	⊕	0.31	0.08	⊖
C	Ctags	0.13	0.09	⊖	0.21	0.10	⊖
	Camellia	0.31	0.08	⊖	0.30	0.09	⊖
	QMail Admin	0.53	0.07	⊖	0.50	0.12	⊖
	Gnumakeuniproc	0.04	0.06	⊕	0.0126	0.0273	⊕
C#	GreenShot	0.1088	0.0388	⊖	0.14	0.05	⊖
	ImgSeqScan	0.0	0.0372	⊕	0.25	0.0055	⊖
	Capital Resource	0.0395	0.0447	⊕	0.0395	0.0458	⊕
	MonoOSC	0.34	0.10	⊖	0.3944	0.12	⊖

CD_c= Change Dispersion in Cloned Code.

CD_n= Change Dispersion in Non-cloned Code. *Rem = Remark*

⊕= CD_c <CD_n (Category 1, CLONES MORE STABLE)

⊖= CD_c >CD_n (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 6 *Count of (⊖) = 18* *Count of (○) = 0*

for the maintenance phase compared to non-cloned code. Thus, this analysis agrees with the previous analyses regarding this metric.

4.6.7 Analysis of the Experimental Results Regarding Change Dispersion

Using our proposed methodology [87], we calculated the change dispersions in cloned (CD_c) and non-cloned code (CD_n) according to the equations: Eq. 4.20 and Eq. 4.21 respectively. However, these equations provide us percentages. We normalized these values within zero to one by dividing them by 100. The normalized dispersions are shown in Tables 4.18 and 4.19. In the following four paragraphs we answer the eighth research question RQ 8 regarding change dispersion.

Overall analysis: According to the results in Table 4.18 (24 points in total, 6 points in Category 1, 18 points in Category 2, no points in Category 3), for 25% of the decision points, dispersion of changes in cloned code is less than the dispersion of changes in the non-cloned code These points fall in Category 1 (CLONES

Table 4.19: Change Dispersion (CD) in Cloned and Non-cloned Methods (Individual Type Result)

Lang	Systems	Type 1			Type 2			Type 3		
		CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem
Java	DNSJava	0.2453	0.0591	⊖	0.1517	0.0782	⊖	0.1816	0.0755	⊖
	Ant-Contrib	0.1764	0.0163	⊖	0.0222	0.0195	⊖	0.05	0.0197	⊖
	Carol	0.0587	0.1950	⊕	0.0935	0.1933	⊕	0.1892	0.1972	○
	jabref	0.1123	0.2043	⊕	0.0895	0.2178	⊕	0.1305	0.1844	⊕
C	Ctags	0	0.1004	⊕	0.20	0.0966	⊖	0.1453	0.0978	⊖
	Camellia	0	0.0985	⊕	0.125	0.0955	⊖	0.35	0.0876	⊖
	QMail Admin	0.50	0.0729	⊖	0.4285	0.0803	⊖	0.60	0.0815	⊖
	Gnumakeuniproc	0.125	0.0042	⊖	0	0.0050	⊕	0.0138	0.0051	⊖
C#	GreenShot	0.0888	0.2932	⊕	0.2296	0.2949	⊕	0.3037	0.3047	○
	ImgSeqScan	0.0	0.0376	⊕	0.0	0.0373	⊕	0.0	0.0372	⊕
	Capital Resource	0.0	0.0492	⊕	0.0	0.0479	⊕	0.0395	0.0447	⊕
	MonoOSC	0.0317	0.1048	⊕	0.0526	0.1042	⊕	0.3913	0.1003	⊖

CD_c= Change Dispersion in Cloned Code.

CD_n= Change Dispersion in Non-cloned Code. *Rem = Remark*

⊕= *CD_c <CD_n (Category 1, CLONES MORE STABLE)*

⊖= *CD_c >CD_n (Category 2, CLONES LESS STABLE)*

○= *The decision point falls in Category 3*

Count of (⊕) = 18 *Count of (⊖) = 16* *Count of (○) = 2*

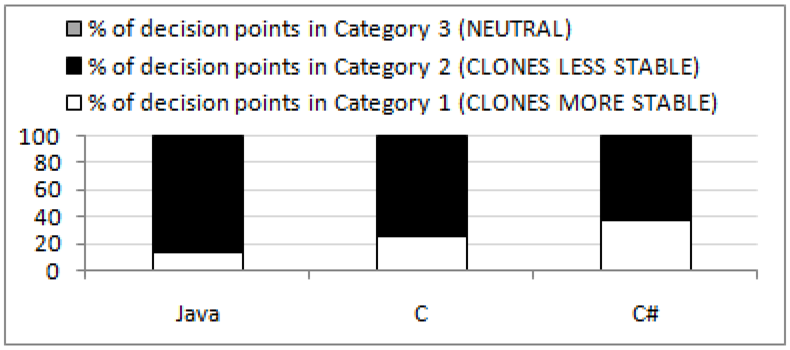


Figure 4.23: Language centric statistics for change dispersion (CD)

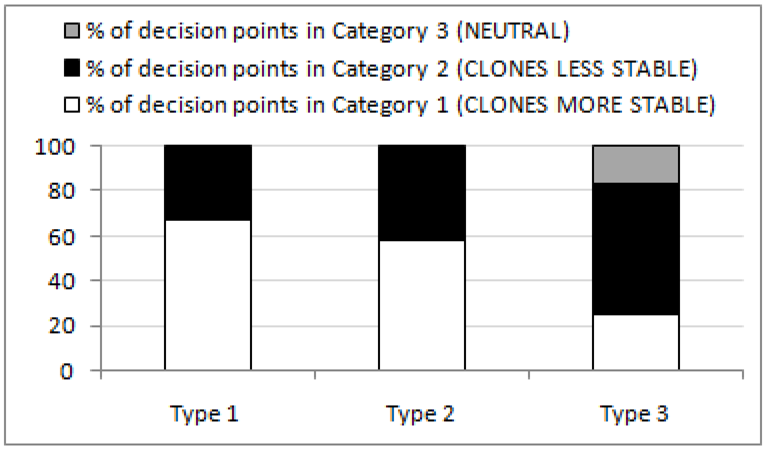


Figure 4.24: Type centric statistics for change dispersion (CD)

MORE STABLE). The opposite is true for the remaining 75% decision points (Category 2, CLONES LESS STABLE). The strong difference between these two percentages indicates that *the changes in the cloned portions of a subject system are more scattered than the changes in the non-cloned portions.* In other words, *the proportion of methods affected by the changes in cloned code is generally greater than the proportion of methods affected by the changes in the non-cloned code.*

Language centric analysis: From the graph in Fig. 4.23 we see that in case of Java programming language, 87.5% of the significant decision points suggest higher dispersion of changes in the cloned code. This percentage for the other two languages is 75% (c) and 62.5% (C#). So, according to this metric, *clones in each of the three programming languages are more unstable for the maintenance phase compared to non-cloned code.*

Type centric analysis: According to the type centric statistics shown in the graph of Fig. 4.24 (constructed from Table 4.19), *Type 3 clones are more likely to get highly dispersed changes compared to the other two types (Type 1, Type 2) of clones. While Type 3 clones appear to be more unstable than non-cloned code, the other two types appear to be more stable.*

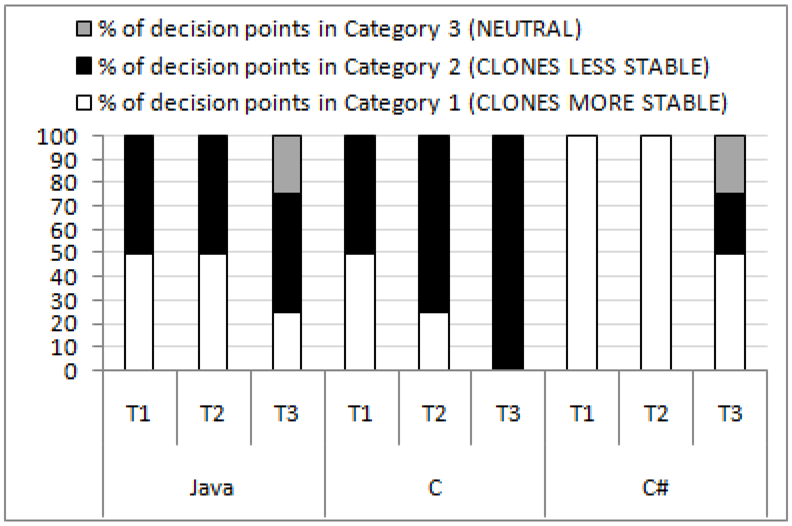


Figure 4.25: Language-wise type centric statistics for change dispersion (CD)

Type Centric Analysis for Each Language: According to the language-wise type centric statistics shown in graph of Fig. 4.25 we see that Type 3 clones in Java and both of the Type 2 and Type 3 clones in C appear to be more unstable than non-cloned code during software maintenance. However, each of the three clone-types of C# seems to be more stable compared to non-cloned code.

4.7 Cumulative Statistics and Analysis of Metrics

So far we have presented our three dimensional analysis for the results obtained for individual metrics. This section presents our analysis of experimental results from different perspectives by aggregating all the eight metrics (of seven methodologies).

4.7.1 Overall analysis

We performed overall analysis in the following four ways considering both of the combined type and individual type results.

Analysis on combined type results:

For this analysis we draw a graph in Fig. 4.26 showing the proportions of the decision points belonging to Category 1 (CLONES MORE STABLE), Category 2 (CLONES LESS STABLE), Category 3 (NEUTRAL) for each of the eight tables (corresponding to each of the eight metrics) containing the combined type results. This graph is the aggregation of the overall analysis of the individual metrics.

According to this graph, four metrics (Modification Probability, Impact, Likelihood, Change Dispersion) suggest that cloned code exhibits more instability than non-cloned code in the maintenance phase. Three

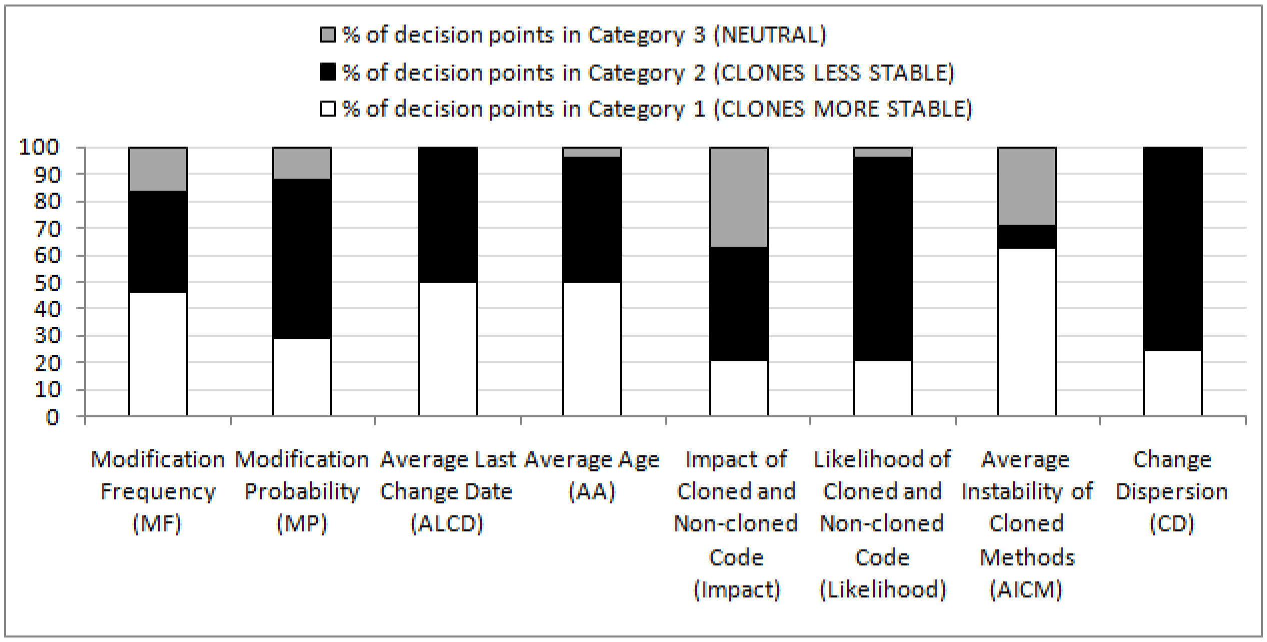


Figure 4.26: Proportions of decision points (Category 1 and Category 2) for 8 metrics considering combined type results

metrics (Modification Frequency, Average Age, Average Instability of Cloned Method) suggest the opposite. The remaining one (Average Last Change Date) shows equal proportions of instability of cloned and non-cloned code. We see that *majority of the metrics agree with cloned code to be more unstable than non-cloned code during maintenance.*

The graph shows that cloned code is less frequently modified than non-cloned code in general. As the *modification frequency* of cloned code is comparatively smaller, its average age should be higher as compared to non-cloned code. This is reflected in the overall scenario of *average age*. Also, *average instability of cloned method* suggests that the proportion of changes received by the cloned portions of the cloned methods is less than the extension of the cloned portions in the cloned methods for most of the cases.

However, the statistics regarding modification probability implies that the proportion of lines affected in cloned regions in a single commit operation is generally greater than the proportion of lines affected in non-cloned regions. Also, the statistics regarding likelihood indicates that methods are more likely to be changed while cloned than while not cloned. According to the statistics regarding change dispersion, the changes in cloned code are more scattered than the changes in the non-cloned code and thus, the changes in cloned code are more difficult to manage than those of non-cloned code. This scenario is also partially reflected by the statistics regarding *impact*. According to our observation regarding *impact*, changes in a cloned method causes higher proportion of other methods to be changed as compared to the changes in a non-cloned method.

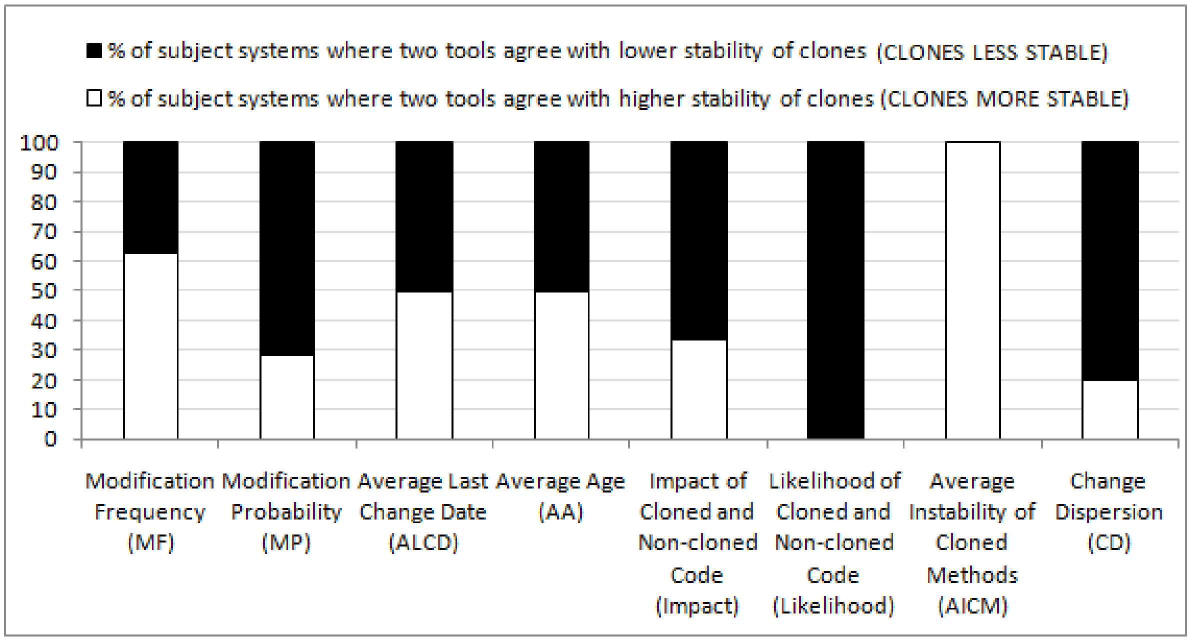


Figure 4.27: Proportions of decision points (Category 1 and Category 2) for 8 metrics considering the agreement of two tools in combined type results

Considering the agreement of two tools regarding the significant decision points of combined type results:

Our second analysis is similar to the first one. The difference is that for this case we considered the agreement of two clone detection tools for getting more precise information. The analysis process is described below.

For each metric, we at first determined the number of subject systems where the decisions (regarding higher instability of cloned or non-cloned code) corresponding to the two tools are the same and then determined the following two proportions.

- The proportion of subject systems for which the decisions corresponding to both of the two tools were against cloned code (cloned code has higher instability than non-cloned code, indicated by $\ominus$)
- The proportion of subject systems for which the decisions corresponding to both of the two tools were in favor of cloned code (cloned code has lower instability than non-cloned code, indicated by $\oplus$)

We plotted these proportions for each metric in the graph of Fig. 4.27. From this graph we can also see that the same four metrics: Modification Probability, Impact, Likelihood, and Change Dispersion suggest higher instability of cloned code. Two metrics: Modification Frequency and Average Instability of Cloned Method suggest the opposite. For each of the remaining two metrics: Average Age and Average Last Change Date, the calculated proportions were equal (50%). Ignoring these two metrics we again see that *majority of the metrics agree with cloned code to be more changeable (or harmful) than non-cloned code during maintenance.*

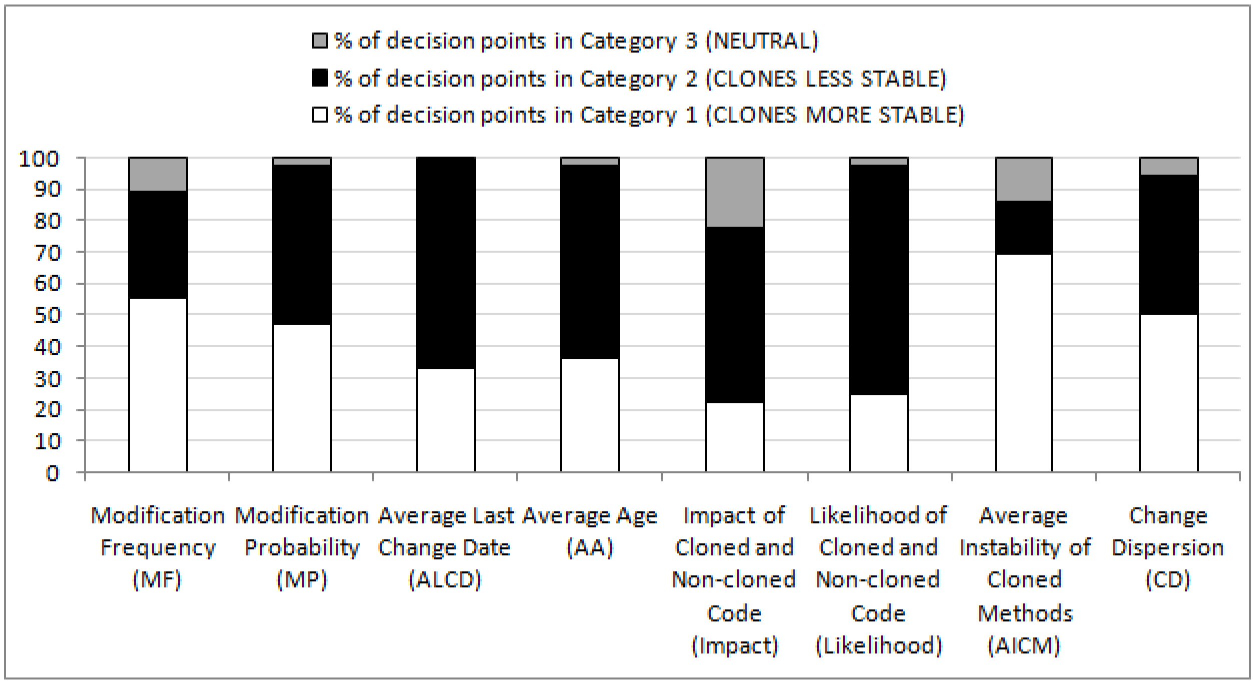


Figure 4.28: Proportions of decision points (Category 1 and Category 2) for 8 metrics considering the individual type results

Analysis on individual type results:

For each of the tables (corresponding to a particular metric) containing individual type results we determined the proportions of the decision points belonging to three categories: Category 1 (CLONES MORE STABLE), Category 2 (CLONES LESS STABLE), and Category 3 (NEUTRAL). Table 4.28 shows these proportions.

We observe that according to the individual type results obtained using NiCad, five metrics: Modification Probability, Average Last Change Date, Average Age, Impact, and Likelihood agree with higher instability of cloned code. The remaining three metrics: Modification Frequency, Average Instability per Cloned Method, and Dispersion of Changes suggest the opposite. So, in this case we again see that *the majority of the metrics agree with higher instability of cloned code.*

Analysis considering both individual type and combined type results:

All the 16 tables containing the individual type and combined type results of eight metrics have 480 decision points in total. Among these, 196 points belong to Category 1 (in favor of clones) and 238 points belong to Category 2 (against clones). The remaining 46 points are insignificant and belong to Category 3. So, *considering all the decision points we see that higher number of decision points agree with higher instability of cloned code.*

As we are observing the experimental results from different perspectives, the subsets of metrics agreeing and disagreeing with higher instability of cloned or non-cloned code are slightly different. For example, for the

Table 4.20: Comparison Between Original Findings and Our Findings Regarding the Candidate Metrics

Investigated Metric	**Original Findings (Combined Type Results)**	**Our Findings (Combined Type Results)**	**Our Findings (Combined Type Results with Tool Agreement)**	**Our Findings (Individual Type Results)**
Modification Frequency (MF) (Hotta et al. [47])	⊕	⊕	⊕	⊕
Modification Probability (MP) (Göde and Harder [40])	⊕	⊖	⊖	⊖
Average Last Change Date (ALCD) (Krinke [65])	⊕	○	○	⊖
Impact (Lozano and Wermelinger [75])	⊖	⊖	⊖	⊖
Likelihood (Lozano and Wermelinger [75])	⊖	⊖	⊖	⊖
Average Instability of Cloned Methods ((Lozano and Wermelinger [74])	⊖	⊕	⊕	⊕
Average Age (AA) (Our Proposed)		⊕	○	⊖
Change Dispersion (CD) (Our Proposed)		⊖	⊖	⊕

⊕ = *Instability of cloned code is less that the instability of non-cloned code*

⊖ = *Instability of cloned code is higher that the instability of non-cloned code*

○ = *Instabilities of cloned and non-cloned code are the same*

* *The original studies were performed considering the combined type clone results only.*

first two cases, we got the same set of metrics: Modification Probability, Impact, Likelihood, and Dispersion of Changes agreeing with higher instability of cloned code. But, for the third case this set consists of five metrics with three metrics: Modification Probability, Impact, and Likelihood common in two different sets. However, for each of the first three cases, *higher number of metrics agree with higher instability of cloned code.*

From a more critical observation we see that the metrics in favor of clones are not contradictory by nature to the metrics against clones. All of the metrics are independently calculated and evaluated on the same experimental setup. The studied metrics suggest that *clones generally exhibit more instability than non-cloned code in the maintenance phase.*

The Table 4.20 shows the findings of the original studies and our studies at a glance. We can see the eight candidate metrics in this table. The first six metrics were proposed and calculated by five existing studies.

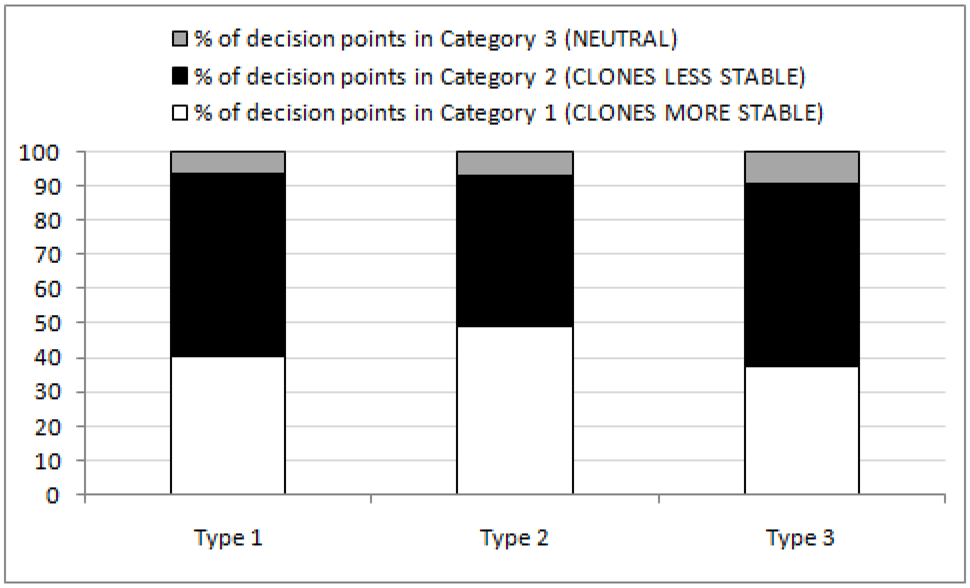


Figure 4.29: Proportions of decision points for each clone type considering 8 metrics

We also calculated these metric values using our implementation. The last two metrics are our proposed ones. While the original studies were performed on the combined type clone results only, we conducted our studies considering individual type results too. We see that most of the findings agree with higher instability of cloned code. We also observe that for three metrics, our experimental results are contradictory with the original findings. These metrics are: *modification probability*, *average last change date*, and *average instability per cloned method*. We consider *modification probability* at first. The original study performed by Göde and Harder [40] was conducted on two Java systems only. Also, the clone detection tool which was used in this study is different from our tools. These might be the possible reasons behind this contradiction. Secondly, *average last change date* was originally calculated by Krinke [65]. In this study Krinke used Simian [108] clone detector which can detect Type 1 clones only. Also, Krinke's findings were mainly based on the file level metrics (Section 4.3.3). However, in our experiment we considered three types of clones (Type 1, Type 2, Type 3) and emphasized on the system level metrics only (Section 4.3.3), because system level metrics provide us more appropriate results compared to file level metrics (elaborated in Section 4.3.3). Thus, our findings can be different from original findings. Thirdly, for calculating *average instability per cloned method*, Lozano and Wermelinger [74] used only CCFinderX for detecting clones and the experiment was performed on five Java systems only. We performed our experiment using both NiCad and CCFinderX on a different set of subject systems covering three programming languages. Possibly because of these differences between the original study [74] and our study, our findings are different from the original findings.

4.7.2 Type centric analysis

Our type centric analysis consists of the tables containing the individual type results. The graph in Fig. 4.29 shows the comparative instability of the three clone-types considering all three programming languages and

8 metrics. In a particular decision table (containing individual type results) corresponding to a particular metric a particular clone type contributes 12 decision points. So, $12 \times 8 = 96$ decision points are contributed by each clone type in aggregate. The proportions of the decision points belonging to Category 1, Category 2, and Category 3 for each type is shown in this graph.

According to this graph, *both Type 1 and Type 3 clones appear to be more unstable than Type 2 clones during software maintenance.* Explanation of getting such a scenario is elaborated below.

In the real sense of terms, cloning should not be harmful if it is done with proper consciousness of impacts. But Type 1 clones, according to the definition, might not ensure full awareness because these are the results of exact copy paste activities. If a code fragment contains some bugs and this code fragment is copied and pasted at several other places without any change, the bugs must also propagate to all other places. After the discovery of these bugs in any of these fragments, the fixation should take place to all other places. Thus Type 1 clones have high probability of getting more changes as well as of being more unstable and vulnerable in the maintenance phase. But, this is not the case for Type 2 clones because intentional Type 2 clones requires programmerś attention to be created. We know that if a code fragment is copied from one place and is pasted to another place with renaming variables and preserving the syntactic similarity, the pasted code fragment becomes Type 2 clone of the code fragment from which it is copied. As the intentional Type 2 clones ensure programmer consciousness, there is a probability that some existing bug in the previous code fragment will be removed from the newly created (pasted) fragment. The case of Type 3 clones is totally different from the two previous ones. Type 3 clones are created because of independent evolutions of Type 1 and Type 2 clones. If some buggy Type 1 clones receive independent evolutions without the awareness of existing bugs, fixation of these bugs becomes more difficult as compared with the bug fixation difficulty of the core Type 1 clones. Also, Type 3 clones have more tendency of being inconsistently changed.

According to the discussion, it is clear that both Type 1 and Type 3 clones have higher probabilities of being harmful for the maintenance phase. Our individual as well as cumulative statistics agree with this. So, we strongly suggest that regardless of programming languages programmers should refrain from creating exact clones and Type 3 clones. Exact clones can be easily avoided by refactoring which involves two activities: (i) creation of a method containing the Type 1 clone fragment and (ii) removal of all existences of Type 1 clone fragments with proper method calls. But, such straight forward refactoring is not possible for Type 3 clones because such clones contain non-cloned fragments within clone fragments. If we can avoid Type 1 clones from the very beginning of software development by the refactoring activities mentioned above, we can surely reduce the number of Type 3 clones to a great extent. Our observation also suggests that extensive refactoring of Type 2 clones is not required because Type 2 clones are not that much harmful.

4.7.3 Type centric analysis for each programming language

The graph in Fig. 4.30 shows the proportions of the decision points for each clone type and each programming language considering eight metrics. When we consider eight metrics, each clone type contributes 32 decision

Table 4.21: Fisher's Exact Tests by clone types for programming languages

	Java						C						C#					
	T 1	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**	**T 1**	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**	**T 1**	**T 2**	**T 1**	**T 3**	**T 2**	**T 3**
C 1	8	16	8	11	16	11	11	7	11	5	7	5	20	24	20	20	24	20
C 2	23	13	23	17	13	17	17	22	17	25	22	25	11	7	11	9	7	9
P	0.0342		0.4031		0.2924		0.263		0.0786		0.5321		0.2813		0.7881		0.5634	

C 1 = Category 1 C 2 = Category 2 T 1 = Type 1 T 2 = Type 2 T 3 = Type 3
P = P-value (or Probability value)

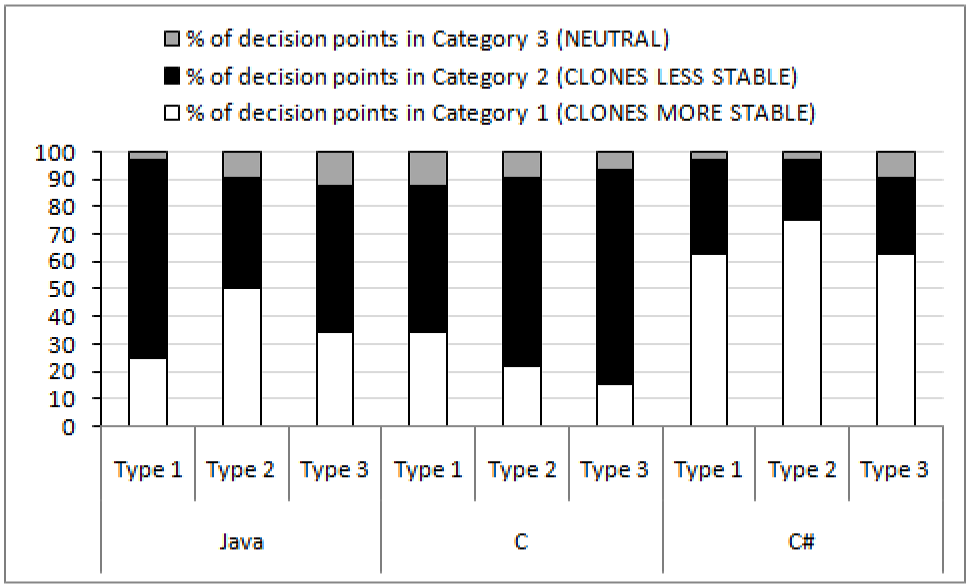


Figure 4.30: Proportions of decision points for three clone types of each language considering 8 metrics

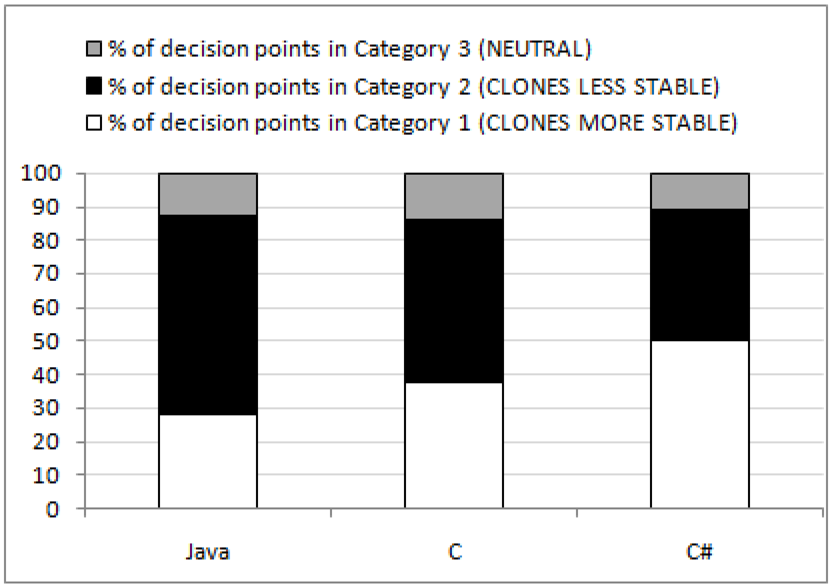


Figure 4.31: Proportions of decision points for each programming language considering combined type results of 8 metrics

points (four for each combination of clone-type and programming language) for a particular language. The graph (Fig.4.30) shows the proportions of the decision points (belonging to Category 1, Category 2, and Category 3) of these 32 points for each clone type of a language.

This graph conveys more specific information about the instabilities of three clone types for different languages. According to this graph , *each of the three clone types for C exhibits higher instability compared to non-cloned code in the maintenance phase. In case of Java, Type 1 and Type 3 clones show higher instability. However, each of the three clone-types of C# appears to be more stable than non-cloned code according to our studies.*

Fisher's Exact Test: To find the validity of the first null hypothesis (regarding RQ 9), we performed Fisher's exact tests for each pair of three clone types for each programming language. The test details for Java, C and C# are shown in Table 4.21. For the tests, we used the exact counts of the decision points belonging to Category 1 and Category 2 corresponding to each language and clone type.

According to the test results, *there is a statistically significant difference between Type 1 and Type 2 clones of Java (p-value = 0.0342 <0.05).* But, for other two languages, none of the three clone type pairs shows a significant difference.

4.7.4 Language centric analysis

We perform language centric analysis considering both combined type results and individual type results. The analysis procedure and observations are described below.

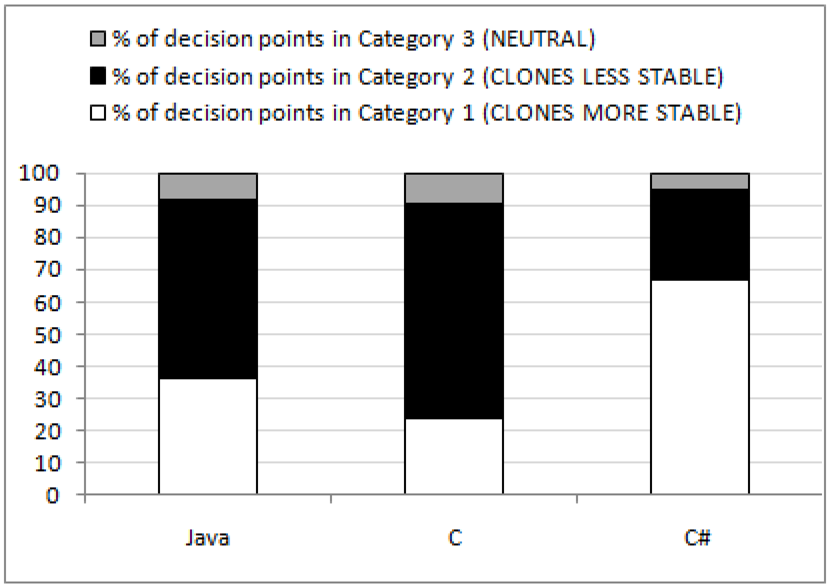


Figure 4.32: Proportions of decision points for each programming language considering individual type results of 8 metrics

Considering combined type results

The graph in Fig. 4.31 shows the proportions of the decision points of three categories (Category 1, Category 2, Category 3) for each programming language considering eight metrics. We see that each programming language contributes 8 decision points in each table containing the combined type results. So, if we consider all the eight tables (containing combined type results), a particular language contributes $8 \times 8 = 64$ decision points. This graph shows the proportions of the decision points for each language considering these 64 points.

We see that in case of Java, while only 28.1% (18 decision points) of the decision points belong to Category 1 (CLONES MORE STABLE), 59.4% (38 points) fall in Category 2 (CLONES LESS STABLE). These proportions for C are 37.5% (24 points) and 48.4% (31 points) respectively. However, an opposite scenario is exhibited by C#. While 50% (32 points) of the decision points of this language belong to Category 1, only 39% (25 points) points belong to Category 2. According to this graph, *clones in Java and C are possibly more unstable than the clones in C#.*

Considering individual type results

We can see that in each of the tables (corresponding to each of the eight metrics) containing individual type results a particular programming language contributes 12 decision points. So, if we consider all of the eight metrics, the number of decision points contributed by a particular programming language is 96. We observe that in case of Java, while only 35 decision points belong to Category 1 (in favor of clones) 53 decision points belong to Category 2 (against clones). The remaining 8 decision points are insignificant. For C, the counts

Table 4.22: Fisher's Exact Tests for prog. languages (Combined type case)

	Java	**C**		**Java**	**C#**		**C**	**C#**
Category 1	18	24		18	32		24	32
Category 2	38	31		38	25		31	25
	P = 0.2436			P = 0.0138			P = 0.2567	

Table 4.23: Fisher's Exact Tests for prog. languages (Individual type case)

	Java	**C**		**Java**	**C#**		**C**	**C#**
Category 1	35	23		35	64		23	64
Category 2	53	64		53	27		64	27
	P = 0.0773			P < 0.0001			P <0.0001	

of points belonging to Category 1 and Category 2 are 23 and 64 respectively. The corresponding counts for C# are 64 and 27. We determined the proportions of the decision points belonging to Category 1, 2, and 3 for each language and plotted these proportions in the graph of Fig. 4.32.

From the graph (Fig. 4.32) we again see that *clones in both Java and C languages exhibit higher instability compared to the instability exhibited by the clones of C#.*

Fisher's Exact Test: We performed Fisher's exact tests for both of the combined type case and individual type case to validate the null hypothesis 2 regarding the tenth research question (RQ 10). Table 4.22 contains the test details for combined type case and Table 4.23 contains the test details for individual type case. Each of these tables contains the details of the three tests corresponding to three language pairs. Each test was conducted on the exact counts of the decision points belonging to *Category 1* and *Category 2*. The p-values of the corresponding tests are shown along the last rows of the tables. If the p-value of particular test is less than 0.05, the difference between the observed data for that particular test is significant.

We see that for the combined type case, the test result corresponding to the language pair: Java and C# is significant. The p-value for this pair = 0.0138 (less than 0.05). For the individual type case, the test results for two language pairs: (1) Java and C# and, (2) C and C# are statistically significant because the p-values for these two pairs are less than 0.05.

Thus, we can say that *the instability exhibited by the clones in C# is significantly lower than the instability of clones in both Java and C.*

4.7.5 Clone detection tool centric analysis

This analysis is based on the tables containing the combined type results. There are eight tables containing the combined type results of eight metrics. In each of these tables, a particular clone detection tool contributes 12 decision points corresponding to 12 subject systems. So, the total number of decision points contributed by a particular tool in eight tables is 96 (= 12 x 8). Our tool centric analysis is based on 96 decision points contributed by each tool.

According to the significant decision points belonging to NiCad, 54.8% points (46 points) exhibit higher instability of cloned code than non-cloned code. The remaining 45.2% (38 points) of the significant points contributed by this tool suggest the opposite. These two percentages for CCFinderX are 57.1% (48 points) and 42.9% (36 points) respectively. Thus, *each clone detection tool individually suggests the higher instability of cloned code compared to non-cloned code.*

We also found the statistics contributed by the *consisstent decision points*. If for a particular subject system both of the tools have the same decision, then we call the corresponding two decision points *consistent decision points*. We found the number of cases for which both of the clone detection tools have the same decision. We considered the agreements regarding the significant decision points only. We observed that there are 59 cases where both of the clone detection tools take the same decision. Among these 59 cases, 57.6% (34 cases) suggests that cloned code is more changeable than non-cloned code in the maintenance phase. The remaining 42.4% (25 cases) suggest the opposite. Thus, *considering the agreement of two clone detection tools we also observe that cloned code is more unstable in the maintenance phase than non-cloned code.*

4.7.6 System centric analysis

In our system centric analysis we determined the agreement and disagreement of the eight metrics obtained for a particular system and a particular clone case (there are five clone cases in total as indicated in the Fig. 1). The agreement-disagreement scenario has been presented in Table 4.24. The construction of the table is explained below.

For a particular subject system and a particular clone case

- if majority of the metrics agree with higher instability of cloned code, the corresponding cell in the table is marked with '$\ominus$'.
- if majority of the metrics agree with higher instability of non-cloned code (lower instability of cloned code), the corresponding cell in the table is marked with '$\oplus$'.
- if the number of metrics agreeing with higher instability of cloned code is equal to the number of metrics with lower instability of cloned code, we marked the corresponding cell with '$\bigcirc$'.

The Table 4.24 contains 60 cells where each cell corresponds to a particular subject system and a particular clone case. We have the following observations from this table.

- While 31 cells (51.7%) are marked with '$\ominus$', 22 (36.7%) cells contain '$\oplus$', and the remaining 7 (11.7%) cells contain '$\bigcirc$'. Thus, *most of the cells indicate higher instability of cloned code.*
- We calculated the percentages of the cells containing the symbols $\oplus$ (CLONES MORE STABLE), $\ominus$ (CLONES LESS STABLE), and $\bigcirc$ (NEUTRAL) for each of the five clone cases and plotted these

Table 4.24: System centric analysis for five clone cases

Subject systems	T 1	T 2	T 3	NiCad (C)	CCFinder (C)
DNSJava	⊖	⊖	⊖	⊖	⊖
Ant-Contrib	⊖	⊕	⊖	⊖	○
Carol	○	⊖	⊖	⊖	⊖
Jabref	○	⊕	⊕	⊕	⊖
Ctags	⊖	⊖	⊖	⊕	○
Camellia	⊕	⊖	⊖	⊖	⊖
QMailAdmin	⊖	⊖	⊖	⊖	⊖
GNUMakeUniproc	⊕	⊖	⊖	⊕	⊕
GreenShot	○	⊕	⊖	⊖	⊕
ImgSeqScan	⊕	⊕	⊕	⊕	○
CapitalResource	⊕	⊕	⊕	⊕	⊖
MonoOSC	○	⊕	⊕	⊕	⊖

T1 = Type 1 case of NiCad   T2 = Type 2 case of NiCad

T3 = Type 3 case of NiCad   NiCad(C) = Combined case of NiCad

CCFinder(C) = Combined case of CCFinder

⊕ = For a particular subject system and clone case, majority of metrics agree with higher stability of cloned code

⊖ = For a particular subject system and clone case, majority of metrics agree with higher instability of cloned code

○ = For a particular subject system and clone case,the same number of metrics agree with both higher and lower stability of cloned code

Count of (⊕) = 22   Count of (⊖) = 31   Count of (○) = 7

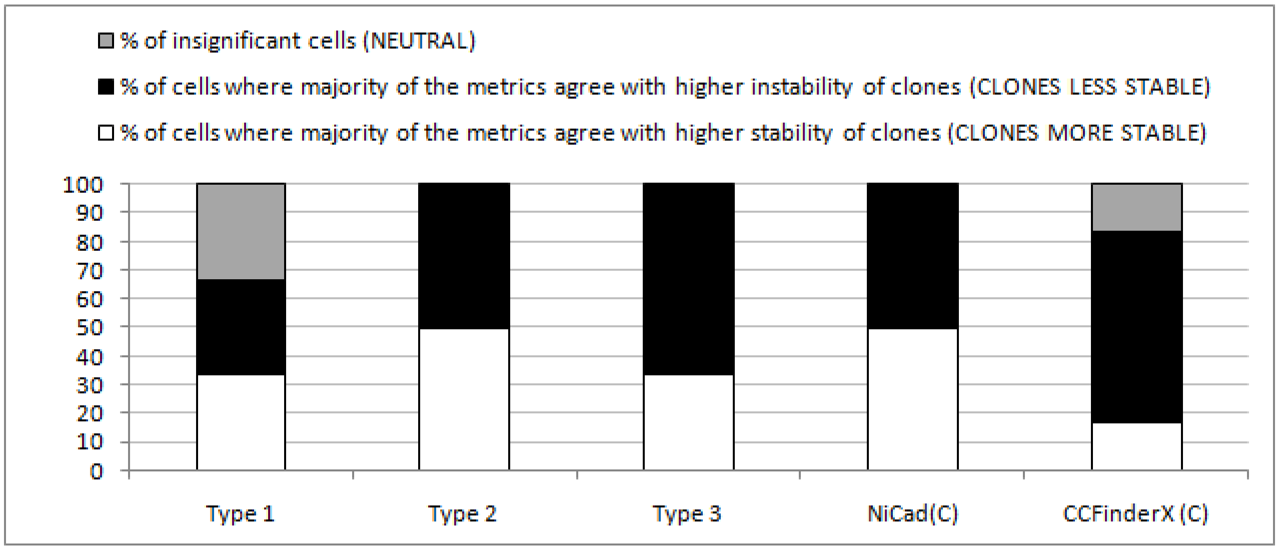


Figure 4.33: System centric statistics regarding five clone cases

percentages in the graph of Fig. 4.33. We see that *for most of the cells belonging to each of the two clone cases: Type 3 (NiCad), and Combined (CCFinderX), majority of the metrics agree with higher instability of cloned code.*

- We also calculated the percentages of the cells containing $\oplus$, $\ominus$, and $\bigcirc$ belonging to each of the three programming languages and plotted these percentages in the graph of Fig. 4.34. We see that *for most of the cells belonging to each of the two programming languages: Java, and C, majority of the metrics agree with higher instability of cloned code.* Such a scenario is also indicated by both Fig. 4.31 and Fig. 4.32. However, in Fig. 4.34, most of the cells belonging to C# agree with higher stability of cloned code.

4.7.7 Analysis related to the programming language paradigm

From Table 4.16 we calculated the average EPCMs (average extension of cloning per method) for procedural programming language (C) and object oriented programming languages (Java and C#) considering both of the two clone detection tools. The average EPCMs are 0.2898, 0.4112 and 0.5239 for C, Java and C# respectively. We observe that *average extension of cloning in C is significantly lower compared to the other two programming languages (Java and C#).*

We also calculated the averages of CD_cs (Dispersion of Changes in Cloned Code) for the three programming languages from Table 4.18. These averages are 0.2541, 0.1652 and 0.1640 for C, Java and C# respectively. In this case we see that the changes in the cloned code of C programming language are more dispersed than the changes in the cloned code of the other two programming languages.

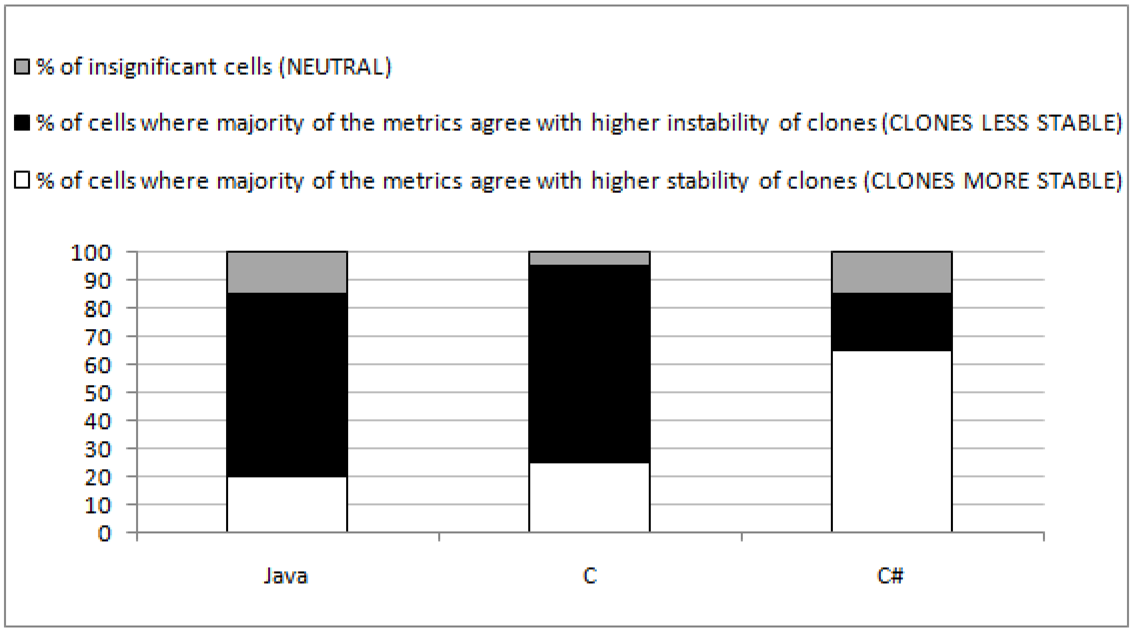


Figure 4.34: System centric statistics regarding three programming languages

From this we infer that *the extension (or proportion) of cloning in procedural programming language (c) is significantly lower than that of object oriented programming language (Java and C#) however, the changes to the clones in procedural language are more scattered compared to the changes to the clones in object-oriented languages.*

4.8 Threats to Validity

The number of subject systems investigated in our study is not sufficient for taking strong decision about the comparative impacts of cloned and non-cloned code. Also, some important factors such as type of developed software, expertise of the responsible programmers, allocated development time etc might have significant effects on cloning and stability of cloned and non-cloned code. But, we did not consider these factors in our study. However, our selection of subject systems emphasizing on the diversity of application domains, system sizes, implementation languages and the large number of revisions have considerably minimized these drawbacks.

Our observation regarding the programming language paradigm is based only on three programming languages. The observation could be more precise with some other programming languages from both procedural and object oriented paradigms.

All of the metrics investigated in our study are only related to stability (of cloned or non-cloned code). For determining the exact impact of cloned code on maintenance we should also investigate the relation of clones with bugs, faults and inconsistencies. However, according to our detailed explanation in the introduction, by comparing the instability of cloned code with that of non-cloned code we can determine the comparative harmfulness of these two code regions (cloned and non-cloned) on maintenance.

4.9 Related Work

We discussed the existing and our proposed stability measurement methodologies and metrics. This section briefly describes the outcomes of the existing studies with some other papers related to clone impact.

Hotta et al. [47] studied the impact of clones in software maintenance activities by determining the modification frequencies of the duplicated and non-duplicated code segments. Their implemented system works on different revisions of a subject system by automatically extracting the modified files across consecutive revisions. They conducted a fairly large study using different tools and subject systems which suggests that the presence of clones does not introduce extra difficulties to the maintenance phase.

Krinke [63] measured how consistently the code clones are changed during maintenance using *Simian* [108] (clone detector) and *diff* (file difference identifier) on Java, C and C++ code bases considering Type-I clones only. He found that clone groups changed consistently through half of their lifetime. In another experiment he showed that cloned code is more stable than non-cloned code [64].

In his most recent investigation [65] centred on calculating the average ages of the cloned and non-cloned code, he has shown cloned code to be more stable than non-cloned code by exploiting the capabilities of version controlling system.

In a recent study [40] Göde et al. replicated and extended Krinke's study [64] using an incremental clone detection technique to validate the outcome of Krinke's study. He supported Krinke by assessing cloned code to be more stable than non-cloned code in general.

Lozano and Wermelinger [75] experimented to assess the effects of clones on the changeability of software using CCFinder [54] as the clone detector. They calculated three stability measures—(i) likelihood and (ii) impact of a method change and (iii) work required for maintaining a method. According to their study, at least 50% of the cases clones did not increase the instability but sometimes instability seemed to increase for the part of the systems related to the cloned methods. In another experiment [73], they experienced that cloned code leads to more changes. In their most recent experiment [74] aiming to analyze the imprints of clones over time, they calculated the extension of cloning, and measured the persistence and stability of cloned methods by improving their previous studies. Their study suggests that cloned methods remain cloned most of their lifetime and that cloning introduces higher density of modifications during maintenance.

Kim et al. [59] proposed a model of clone genealogy to study clone evolution. Their study with the revisions of two medium sized Java systems showed that refactoring of clones may not always improve software quality. They also argued that aggressive and immediate refactoring of short-lived clones is not required and that such clones might not be harmful. Saha et al. [104] extended their work by extracting and evaluating code clone genealogies at the release level using 17 open source systems of four different languages. Their study reports similar findings as of Kim et al. and concludes that most of the clones do not require any refactoring efforts in the maintenance phase. On the other hand, Juergens et al.'s [52] study with large scale commercial systems suggests that inconsistent changes are very frequent to the cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault.

Kapser and Godfrey [55] identified different patterns of cloning and experienced that around 71% of the clones could be considered to have a positive impact on the maintainability of the software system.

Aversano et al. [5] combined clone detection and modification transactions on open source software repositories to investigate how clones are maintained during the evolution and bug fixing. Their study reports that most of the cloned code is consistently maintained. In another similar but extended study, Thummalapenta et al. [114] indicated that most of the cases clones are changed consistently and for the remaining inconsistently changed cases clones mainly undergo independent evolution.

We see that while the objective is the same—*determining the impacts of clones on software maintenance*, the researchers considered different approaches with different clone detection tools and subject systems, and finally reported contradictory findings. Our empirical study described in this chapter is an attempt to resolve the contradiction using a uniform framework.

4.10 Conclusion

In this empirical study, we implemented seven methodologies and calculated eight impact related metrics using a common framework. We implemented each of these methodologies using two clone detection tools: NiCad and CCFinderX and applied on each of the twelve subject systems written in three programming languages. We investigated total 480 decision points regarding sixteen tables of eight metrics. 434 points were significant decision points among which 238 points (54.84%) suggest cloned code to be more unstable than non-cloned code and the remaining 196 points (45.16%) suggest the opposite. Also, according to the cumulative statistics majority of the metrics suggest cloned code to be more unstable than non-cloned code. Each of the clone detection tools individually suggests cloned code to be more unstable. Moreover, our system centric analysis suggests higher instability of cloned code. This scenario disagrees with the already established bias [47, 65] and indicates that clones are not necessarily stable and most of the time more unstable than non-cloned code in the maintenance phase. So, clones should be managed with proper tool support.

According to our type-centric analysis Type 1 and Type 3 clones are more unstable compared to the Type 2 clones. Our Fisher's exact test results suggest that Type 1 clones of Java are significantly more unstable than Type 2 clones of this language. So, we should give more emphasis on managing Type 1 and Type 3 clones. As Type 2 clones appear to be less unstable, it should not be our primary target.

Our language centric analysis suggests that clones in Java and C programming languages are more unstable than the clones in C#. Our Fisher's exact test results regarding programming languages show that there are significant differences among the stabilities of the clones of different programming languages.

Thus, our type-centric and language-centric analyses together suggest that programmers working on Java and C languages should be more careful about Type 1 and Type 3 clones. Moreover, while deciding about clone management we should emphasize on the Type 1 and Type 3 clones of the systems developed using Java or C.

Finally, according to our result it seems that object oriented programming languages promote more cloning than procedural programming languages. However, changes to the clones in procedural programming language appear to be more scattered compared to the changes to the clones in object-oriented languages.

We see that although clones are more unstable compared to non-cloned code in general, many decision points in this experiment suggest higher stability of clones. Thus, our outcomes from this empirical study do not necessarily solve the controversy regarding clone stability. To get a deep insight in this matter we performed two more empirical studies that are elaborated in the next two chapters.

Chapter 5
Dispersion of Changes in Cloned and Non-cloned Code

5.1 Motivation

From our investigation presented in the previous chapter (Chapter 4) we see that although clones are more unstable than non-cloned code in general, clones sometimes appear to exhibit higher stability during maintenance. Our previous findings do not necessarily solve the controversy regarding clone stability. To get a deeper insight into the changes occurring to the clones we performed another in-depth investigation on one of our proposed metrics, change dispersion [87]. We described change dispersion in detail in Section 4.3.7 of the previous chapter (Chapter 4). In this study based on change dispersion we extensively analyzed whether the presence of clones in methods increase method instability. Our study involves the manual investigation of all the changes occurred to the clones of one of our subject systems Ctags [33].

Our experimental results on sixteen subject systems covering four different programming languages (Java, C, C#, and Python) considering all three major types (Type 1, Type 2, Type 3) of clones involving two clone detection tools (CCFinderX [19] and NiCad [96]) indicate that:

- Higher dispersion of changes is a possible indicator of higher source code instability.
- Dispersion of changes in the cloned code is sometimes higher than the dispersion of changes in the non-cloned code. In other words, the percentage of methods affected by changes in cloned code is sometimes higher than the percentage of methods affected by the changes in non-cloned code. Thus, cloned code is possibly more harmful than non-cloned code during the maintenance phase.
- Clones in the subject systems written in Java and C have a higher probability of having more dispersed changes compared to the clones in C# and Python systems. As higher change dispersion indicates higher instability, the clones in Java and C systems are more unstable compared to non-cloned code and possibly require more maintenance effort than non-cloned code.
- Type 3 clones exhibit higher change dispersion compared to Type 1 and Type 2 clones.

According to our investigation on method instability based on change dispersion, *both fully cloned and partially cloned methods of Java and C systems exhibit higher instability compared to the fully non-cloned methods.*

Table 5.1: Subject Systems

	Systems	Domains	LOC	Revisions
Python	Noora	Development Tool	14,862	140
	Marimorepy	Collection of Libraries	13,802	49
	Pyevolve	Artificial Intelligence	8.809	200
	Ocemp	Game	57,098	438

According to our investigation on all the changes occurring to the clones of CTAGS [33] during its evolution

- A significant portion (48.57%) of changes in cloned code were made so that the clone fragments in a particular clone class remain consistent.
- Consistency ensuring changes mainly took place to the fully cloned methods.
- A significant proportion (63.15%) of the consistency ensuring changes happened to the Type 3 clone fragments.

5.2 Study Setup

Our setups for clone detection tools have been described in Section 4.4.4. We investigate sixteen subject systems in this study. Twelve subject systems written in Java, C, and C# have been listed in Table 4.3. The remaining four systems written Python are shown in Table 5.1.

5.3 Experimental Results and Discussion

We detect clones from the subject systems using our clone detection tools: NiCad and CCFinderX and then, we determine change dispersions of cloned and non-cloned code.

Change: A change is considered as was defined by Hotta et al. [47]. According to their definition a change can affect multiple consecutive lines. Suppose, n lines of a method (or any other program entity) were changed through additions, deletions or modifications. If these n lines are consecutive then, the count of change is one. If these n lines are not consecutive then, the count of changes equals to the number of unchanged portions within these n lines plus one.

Change dispersions (normalized within zero and one) of the cloned and non-cloned code in the Python systems are shown in Table 5.2. This table shows change dispersions considering the clone detection results of NiCad only. Change dispersions of the remaining twelve systems (corresponding to NiCad results) have been shown in Table 4.19 in Chapter 4. CCFinderX cannot detect clones from Python systems. Change

Table 5.2: Dispersion OF Changes USING NiCad Results

	Systems	Type 1 (NiCad)			Type 2 (NiCad)			Type 3 (NiCad)		
		CD_c	CD_n	Rem	CD_c	CD_n	Rem	CD_c	CD_n	Rem
Python	Noora	0.2012	0.1788	⊖	0.1452	0.2181	⊕	0.2372	0.1744	⊖
	Marimorepy	0.0232	0.1583	⊕	0	0.1666	⊕	0.375	0.1512	⊖
	Pyevolve	0.2549	0.2511	○	0.250	0.3012	⊕	0.2298	0.2739	⊕
	Ocemp	0.2294	0.6659	⊕	0.4228	0.6554	⊕	0.3867	0.6245	⊕

CD_c= Change Dispersion in Cloned Code.

CD_n= Change Dispersion in Non-cloned Code. *Rem = Remark*

⊕= CD_c <CD_n (Category 1, CLONES MORE STABLE)

⊖= CD_c >CD_n (Category 2, CLONES LESS STABLE)

○= The decision point falls in Category 3

Count of (⊕) = 8 *Count of (⊖) = 4* *Count of (○) = 0*

dispersions using CCFinderX results of the other twelve systems written in Java, C and C# are shown in Table 4.18 in Chapter 4. The decision points and their categories mentioned in Table 5.2 have been described in Section 4.6.

5.3.1 Overall analysis

The Tables 5.2 (12 decision points), 4.19 (36 decision points), and the CCFinderX results in Table 4.18 (12 decision points) contain 60 decision points in total. Among these points, 57 points are significant and fall in Category 1 (CLONES MORE STABLE) or Category 2 (CLONES LESS STABLE). We ignored the remaining three insignificant decision points belonging to Category 3 (NEUTRAL), because for each of these decision points, the difference between the change dispersions of cloned (CD_c) and non-cloned (CD_n) code is not significant according to the equation Eq. 4.22.

According to 49.1% (28 points) of the significant points (57 points), dispersion of changes in cloned code is less than the dispersion of changes in non-cloned code (Category 1, CLONES MORE STABLE). The opposite is true for the remaining 50.88% points (29 points). Though the difference between the percentages is very small, it indicates that *the changes in the cloned portions of our investigated subject systems are sometimes more scattered than the changes in the non-cloned portions.* In other words, *the proportion of methods affected by the changes in cloned code is sometimes greater than the proportion of methods affected by the changes in the non-cloned code.*

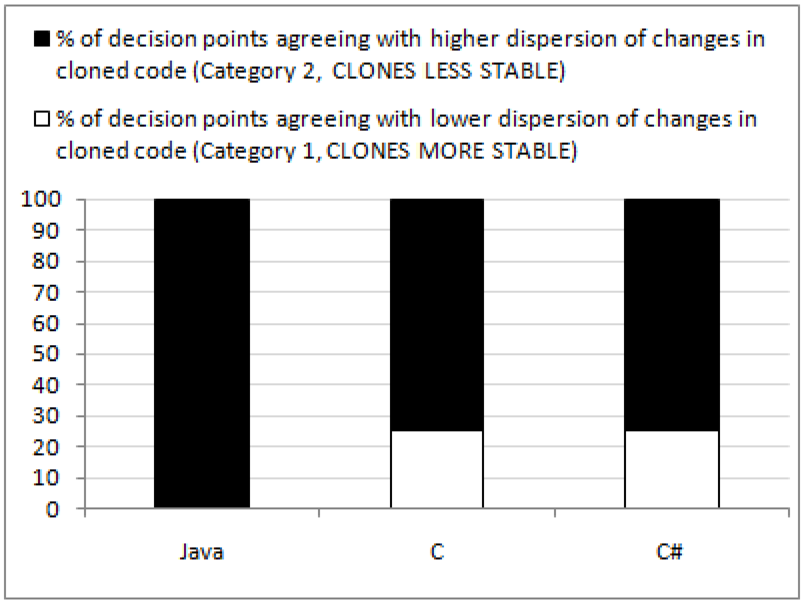


Figure 5.1: Programming language centric statistics on CCFinder results

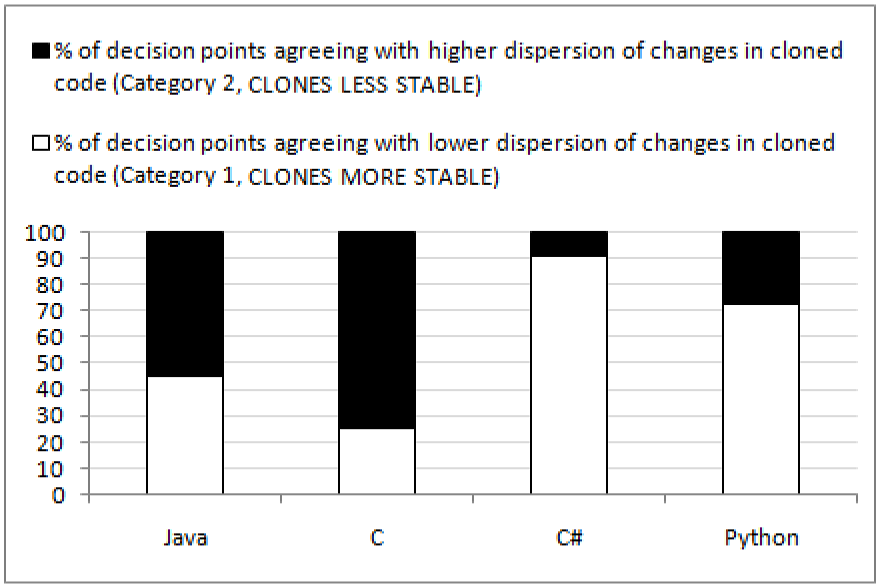


Figure 5.2: Programming language centric statistics on NiCad results

5.3.2 Language Centric Analysis

From the Tables 5.2, 4.19, and the CCFinderX results in Table 4.18 we obtained language centric statistics of the dispersion of changes happened in the cloned and non-cloned code.

Considering the significant decision points belonging to a particular programming language in a particular table we measured two proportions: **(1)** proportion of the significant decision points agreeing with higher dispersion of changes in cloned code (Category 2, CLONES LESS STABLE), and **(2)** proportion of the significant decision points agreeing with higher dispersion of changes in non-cloned code (Category 1, CLONES MORE STABLE).

According to the language centric statistics graph of Fig. 5.1 obtained from CCFinderX results (Table 4.18), in case of each of the three programming languages: Java, C, and C#, most of the decision points agree with higher dispersion of changes in cloned code (Category 2, CLONES LESS STABLE).

The statistics that we obtain from NiCad results are shown in the graph of Fig. 5.2. According to this graph higher proportion of decision points belonging to both Java and C agree with higher dispersion of changes in cloned code (Category 2, CLONES LESS STABLE). However, an opposite scenario can be observed in case of each of the other two programming languages: C# and Python. For each of these two languages, higher proportion of the significant decision points agree with lower dispersion of changes in cloned code (Category 1, CLONES MORE STABLE).

Thus, considering the graphs in Fig. 5.1 and Fig. 5.2 we can come to the conclusion that:

- *cloned code of the subject systems written in Java and C languages has higher probability of getting more dispersed changes compared to the cloned code of the other two languages (C# and Python). So, cloned code in Java and C systems is more likely to require higher effort than non-cloned code during maintenance phase.*

- However, *considering the statistics of the two graphs, clones of C# and Python are not likely to exhibit higher probability of getting more dispersed changes compared to the cloned code of Java and C*. So, *clones in the subject system written in these two programming languages (C# and Python) should not necessarily be considered harmful for maintenance according to our change dispersion analysis.*

5.3.3 Type centric analysis

Our type centric analysis depends only on NiCad results. Considering all the significant decision points belonging to a particular clone type (Type 1, Type 2, Type 3) in tables: Table 5.2 and Table 4.19 we calculate two measures: **(1)** the proportion of decision points agreeing with higher dispersion of changes in cloned code, and **(2)** the proportion of decision points agreeing with lower dispersion of changed in cloned code. We plot these two measures for each clone type in Fig. 5.3. According to this graph, for most of the subject systems the dispersion of changes in Type 3 clones is higher than the dispersion of changes in the corresponding non-cloned code. However, an opposite scenario can be observed for each of the other

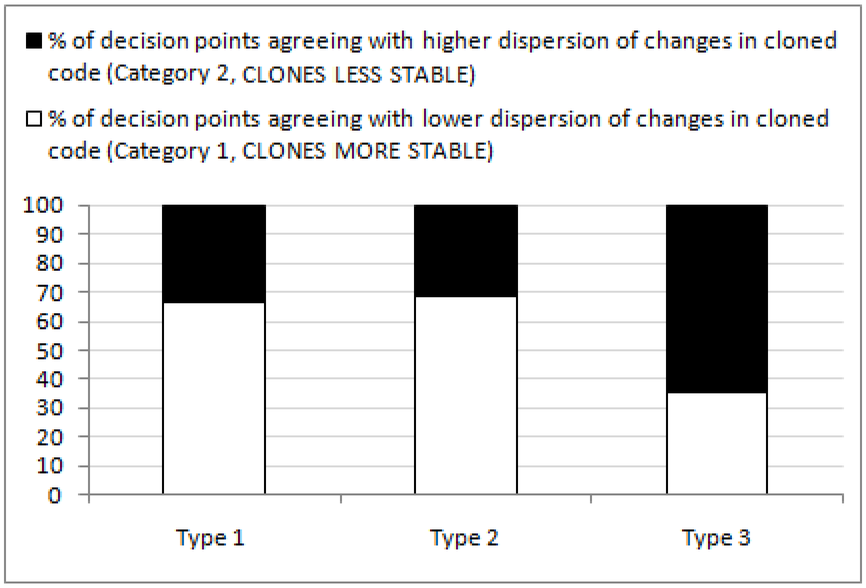


Figure 5.3: Overall type centric analysis on NiCad results

two clone types. From this scenario we decide that Type 3 clones possibly require more maintenance effort compared to the Type 1 and Type 2 clones.

We draw another graph (Fig. 5.4) from tables: Table 5.2, and Table 4.19 showing the type-wise change dispersion for each programming language. Considering the significant decision points belonging to a particular clone type and a particular language in these tables we measure: (1) the percentage of decision points deciding higher dispersion of changes in cloned code, and (2) the percentage of decision points deciding higher dispersion of changes in non-cloned code. According to this graph, in case of both Java and C systems, each type of clone has the probability of getting more dispersed changes compared to the corresponding non-cloned code. We also see that Type 3 clones have the highest possibility of getting more dispersed changes compared to the other two clone types for these two languages (Java and C). However, for the other two programming languages (C# and Python) cloned code has lower probability of getting more dispersed changes in general except for Type 3 clones of Python.

5.3.4 Findings

According to the above type and language centric analyses we come to the following decisions.

- Clones sometimes receive more dispersed changes compared to the change dispersion in non-cloned code. Thus, clones are likely to require higher maintenance effort compared to the non-cloned code.

- Type 3 clones have higher probability of getting more dispersed changes compared to the other two clone types: Type 1 and Type 2. Thus, Type 3 clones possibly require more maintenance effort compared to

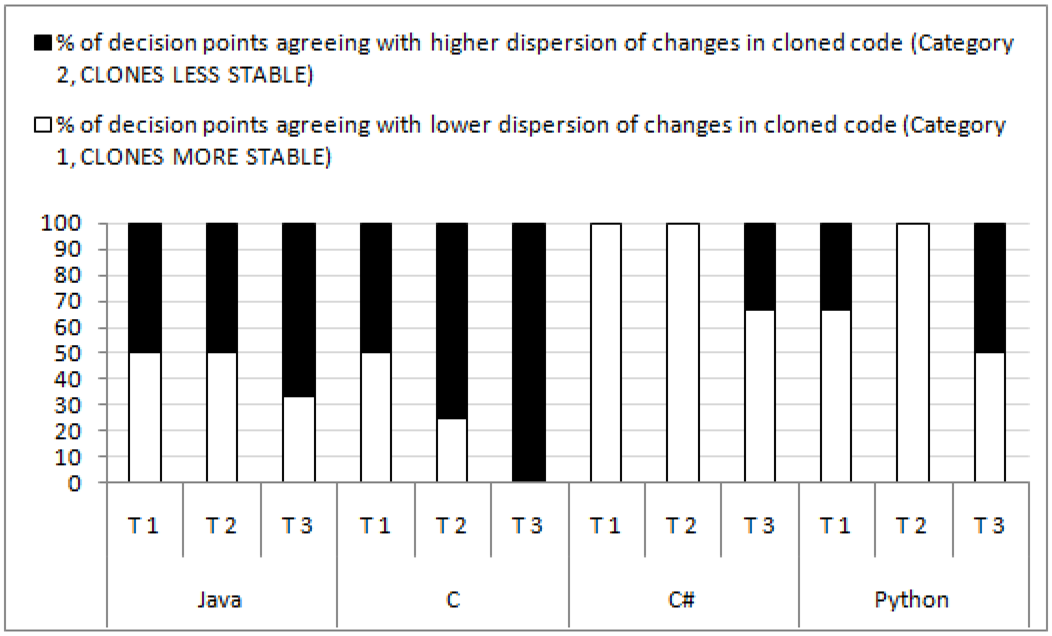


Figure 5.4: Type centric analysis on NiCad results

the other two clone types. In other words, Type 3 clones are more harmful than the other two types of clones according to our analysis.

- Each clone type of the subject systems written in both Java and C has higher probability of getting more dispersed changes compared to the clone types in other two languages: C# and Python. Thus, according to our analysis clones in the subject systems written in Java and C are more harmful (require more maintenance effort) compared to the clones in other two languages.

We suggest that programmers should be more careful while creating Type 3 clones so that: **(1)** the clone fragments do not have bugs, and **(2)** a particular change to a Type 3 clone fragment is properly propagated (if necessary) to other clone fragments in the same clone class. In the following section we present an in-depth analysis on method instability on the basis of change dispersion.

5.4 Investigation on method instability based on change dispersion

According to Eq. 4.6 and Eq. 4.7, higher dispersion is an implication of higher proportion of changed methods during evolution. We have already mentioned higher change dispersion as a possible indicator of increased effort for understanding and analyzing the consequences (or impacts) of changes. In the following subsection we investigate whether higher dispersion indicates higher instability of source code because, if higher change dispersion indicates higher code instability then, higher dispersion should also be an indicator of higher maintenance effort.

Table 5.3: Corelation Between Dispersion and Code Instability

	Systems	*Dispersion*	*Instability*
Java	DNSjava	0.3316	6.88
	Ant-Contrib	0.029	2.41
	Carol	0.187	10.07
	JabRef	0.3597	12.11
C	Ctags	0.4185	4.95
	Camellia	0.671	25
	QMailAdmin	0.4916	20.67
	Gnumakeuniproc	0.0279	5.1
C#	GreenShot	0.2852	4.39
	ImgSeqScan	0.037	5.27
	CapitalResource	0.0472	4.92
	MonoOSC	0.2715	4.77
Python	Noora	0.2345	11.89
	Marimorepy	0.1652	3.81
	Pyevolve	0.3238	4.36
	Ocemp	0.6762	20.64
Correlation coefficient between dispersion and instability = 0.8000			

5.4.1 Correlation of Dispersion with Code Instability

To determine how change dispersion is correlated with instability of source code we calculated the following two measures.

- **The dispersion of changes to the methods (DCM) considering the whole code base:** In the previous tables regarding change dispersion, we showed the dispersion of changes to the cloned and non-cloned methods separately. But, for investigating the correlation between change dispersion and code instability we calculate change dispersion considering all methods of a subject system.

  If a subject system has G method genealogies (cloned or non-cloned) in total and G_C of these genealogies received some changes during the evolution, the dispersion of changes to the methods (DCM) of this subject system can be calculated by the following equation.

$$DCM = \frac{G_C \times 100}{G} \tag{5.1}$$

- **Average Number of Changes per Commit operation (ANCC):** For each of the subject systems, we determined the average number of changes happened per commit operation considering only those commits where there were some changes to the source code.

Here, $ANCC$ is a source code instability metric. After calculating the $ANCC$ and DCM for each of the candidate systems we determined the Pearson correlation between $ANCC$s and the corresponding dispersion values (DCMs). The correlation is shown in Table 5.3. We used the change dispersion values by normalizing them within zero to one. The correlation co-efficient is positive (coefficient = 0.800) and it indicates a good correlation between DCM and $ANCC$. So, we see that higher change dispersion is an indicator of higher instability (or change-proneness) of source code.

From the above correlation scenario we come to the following decisions.

- As higher dispersion of changes indicates higher source code instability, higher dispersion is also a possible indicator of higher maintenance effort and costs.

- We have already observed that dispersion of changes in cloned methods is sometimes higher than the dispersion of changes in non-cloned methods. More specifically, cloned (fully or partially) methods in Java and C systems have a higher likelihood of getting more dispersed changes compared to the cloned methods in C# and Python systems. From this we suspect that clones in Java and C systems are possibly more unstable and require higher maintenance effort and costs compared to the non-cloned code.

We further investigate on the instability of cloned and non-cloned methods of Java and C systems because, we wanted to be confident about whether the presence of clones in methods in these systems increases method instability. We limited our investigation to those cloned or non-cloned methods to which the changes were dispersed disregarding the methods that did not get any changes during the evolution.

5.4.2 Instability of Cloned and non-cloned methods

So far we have empirically shown that (1) changes in the cloned methods in Java and C systems are generally more dispersed compared to the changes in non-cloned methods, (2) changes in the cloned methods in C# and Python systems are generally less dispersed compared to the changes in non-cloned methods, and (3) higher change dispersion is an indicator of higher source code instability. So, cloned methods in the subject systems written in Java and C might have higher instability compared to the non-cloned methods. To get a deeper insight into this matter we performed the following investigations on the instability of cloned and non-cloned methods considering the Java and C systems only. We excluded Python and C# systems from the following investigations, because cloned methods in these systems appear to get less dispersed changes compared to non-cloned code. Thus, the cloned methods in these systems are expected to be more stable compared to the non-cloned methods. In the following investigations we used the combined type clone results of NiCad.

5.4.3 Investigation on the instability of cloned and non-cloned methods

We at first separated the method genealogies detected in a candidate Java or C system into two disjoint sets: (1) cloned (fully or partially) method genealogies (CMG), and (2) fully non-cloned method genealogies (NMG). If any method instance in a particular method genealogy contains a clone, that genealogy is considered as a CMG. On the other hand, if no method instance in a particular genealogy contains a clone, that genealogy is considered an NMG. Then, for each of these sets (CMG and NMG) we calculated an instability metric considering method level granularity. The metric was calculated considering two things: (1) method longevity, and (2) method size. A method with higher longevity has the probability of getting more changes than a method with comparatively lower longevity. Also, method size might have an effect on method instability. We normalized the effects of method size and longevity in the instability metric in the following way.

Method Instability considering Longevity and Size (MILS): For a particular set of method genealogies, we calculate the average number of changes received per 100 LOC of a method instance per 100 commit operations. Here, we note that each particular method genealogy consists of several method instances. For the two sets of genealogies (CMG and NMG) we calculate $MILS_{CMG}$ and $MILS_{NMG}$ in the following way.

$$MILS_{CMG} = \frac{NOC_{CMG} \times 10000}{|CMG| \times ALS_{CMG} \times AS_{CMG}} \tag{5.2}$$

Here, NOC_{CMG} denotes the total number of changes happened to all method instances of the method genealogies in the set CMG, ALS_{CMG} denotes the average life span per genealogy in CMG, and AS_{CMG} is the average size of a method instance in CMG. A particular method genealogy in CMG can have several method instances. While determining AS_{CMG} we at first find the summation of the sizes of all method instances of all genealogies. Then, we divide this sum by the total count of all method instances of all genealogies in CMG to get AS_{CMG}. We calculate ALS_{CMG} according to the following equation.

$$ALS_{CMG} = \frac{\sum_{g \epsilon CMG} LS(g)}{|CMG|} \tag{5.3}$$

Here, g is a particular method genealogy in the set CMG. $LS(g)$ is the life span of g. $LS(g)$ is the count of commit operations for which the genealogy g remained alive during evolution.

Justification of $MILS$ metric: Now, we have a close look at Eq. 5.2. The term $NOC_{CMG}/(|CMG| \times ALS_{CMG})$ gives us the count of changes received by a method instance (of CMG) per commit operation. AS_{CMG} is the average size of a method instance. So, we divide the term $NOC_{CMG}/|CMG| \times ALS_{CMG}$ by AS_{CMG}, we will get the count of changes happened per LOC of a method instance (of CMG) per revision. At last, by multiplying 10000 with the result we get the count of changes happened per hundred LOC of a method instance per hundred commit operations. Thus, Eq. 5.2 reasonably calculates $MILS_{CMG}$.

Similarly, we calculate $MILS_{NMG}$ according to the following equation.

$$MILS_{NMG} = \frac{NOC_{NMG} \times 10000}{|NMG| \times ALS_{NMG} \times AS_{NMG}} \tag{5.4}$$

Here we should mention that a particular genealogy might not live for 100 commits and also might not consist of 100 LOC. However, multiplication of 10000 in the above equations gives us more understandable values.

We calculated the values of $MILS_{CMG}$ and $MILS_{NMG}$ using two clone detection tools for each of the Java and C subject systems and plotted these values in the graphs of Fig. 5.5 (for CCFinder) and Fig. 5.6 (for NiCad). We see that for most of the systems, $MILS_{CMG} > MILS_{NMG}$ in these two graphs.

Findings: The graphs imply that cloned methods in Java and C systems receive higher amount of changes compared to the non-cloned methods. In other words, *the instability of cloned methods of the subject systems written in Java or C is most of the time higher (except Camellia in NiCad result in Fig. 5.6) than the instability of non-cloned methods in these systems.*

5.4.4 Investigation on the instability of fully cloned and partially cloned methods

In the previous subsection we observed that cloned methods (in other words, cloned method genealogies (CMG)) exhibit higher instability compared to the non-cloned methods (or non-cloned method genealogies (NMG)). We suspected clones to be a possible cause of this higher instability of cloned (fully or partially) methods. For justifying whether clones are really responsible for higher instability of cloned methods, we determined the instability of fully cloned and partially cloned methods separately. Then, we made the following two comparisons.

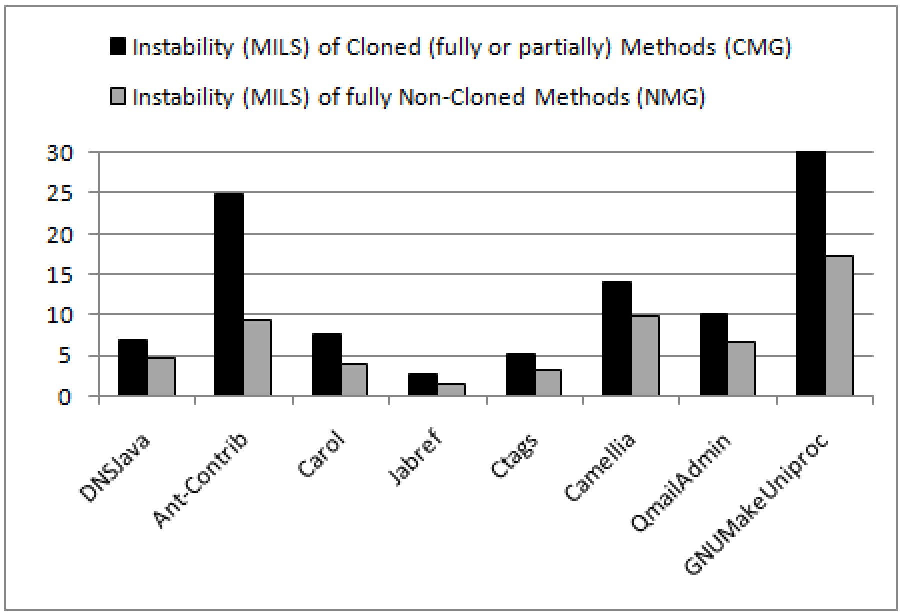


Figure 5.5: Instability (MILS) of cloned (fully or partially) and non-cloned method genealogies (for CCFinder Results)

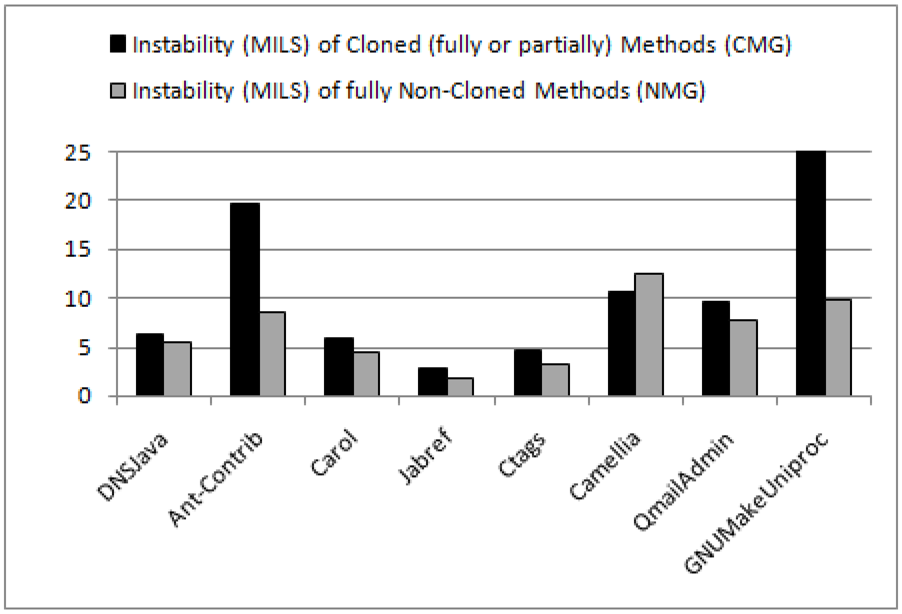


Figure 5.6: Instability (MILS) of cloned (fully or partially) and non-cloned method genealogies (for NiCad Results)

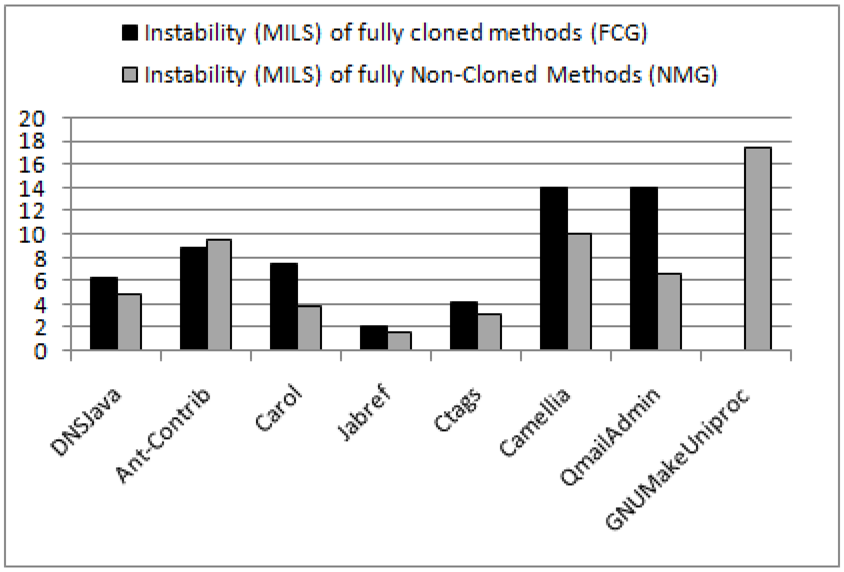


Figure 5.7: Instability (MILS) of fully cloned and fully non-cloned method genealogies (for CCFinderX Results)

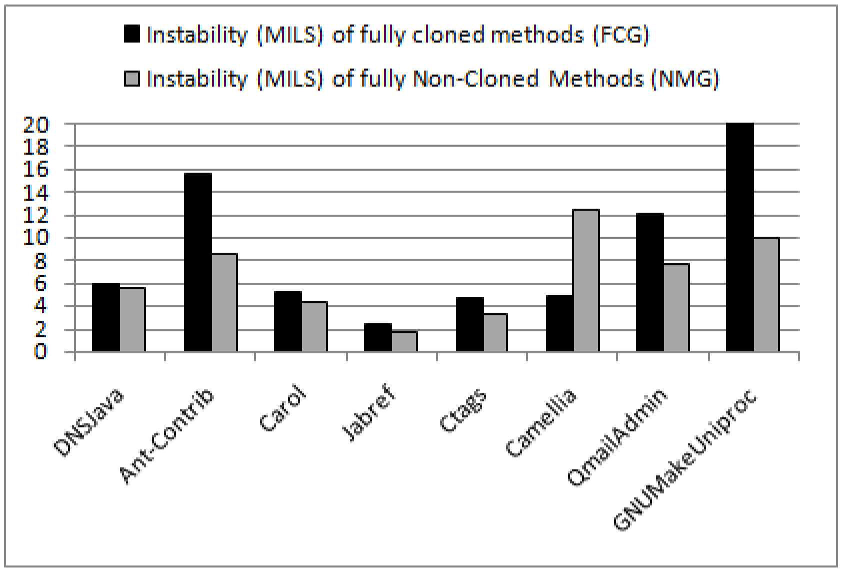


Figure 5.8: Instability (MILS) of fully cloned and fully non-cloned method genealogies (for NiCad Results)

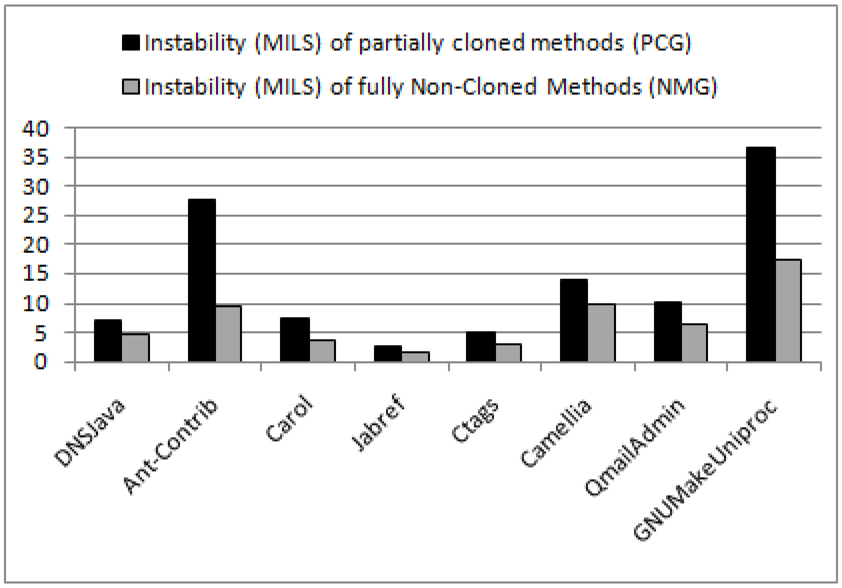


Figure 5.9: Instability (MILS) of partially cloned and fully non-cloned method genealogies (for CCFinderX Results)

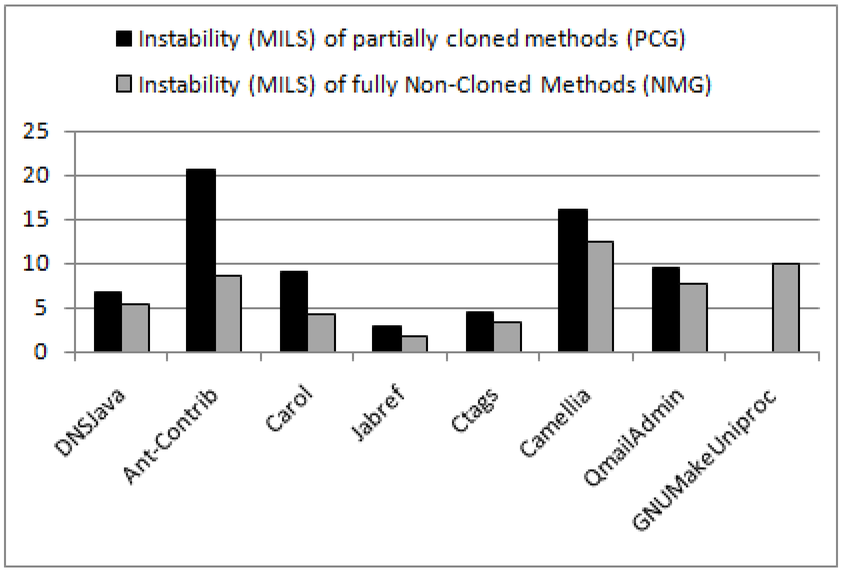


Figure 5.10: Instability (MILS) of partially cloned and fully non-cloned method genealogies (for NiCad Results)

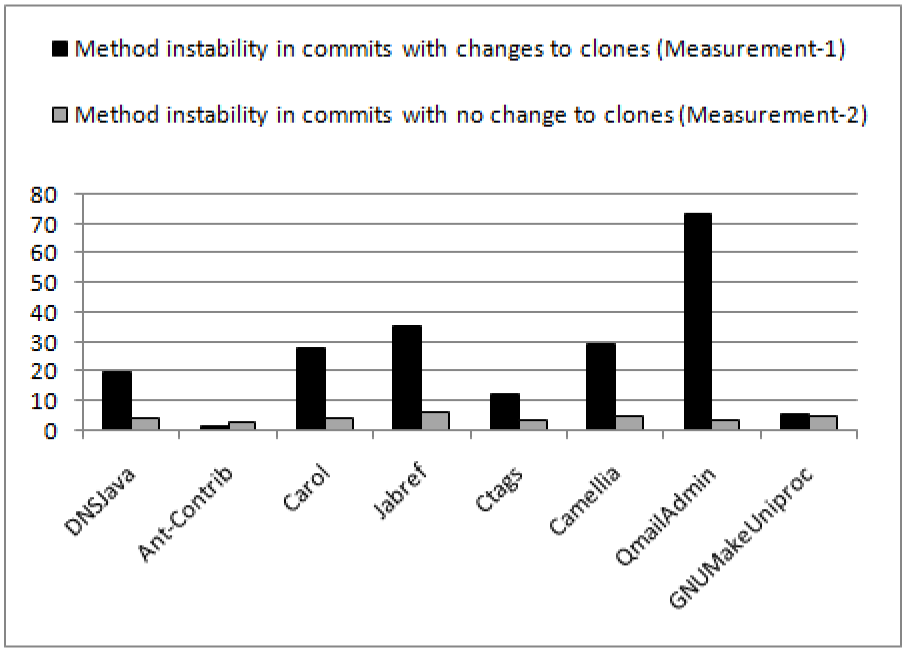


Figure 5.11: Average number of changes per commit operation (for CCFinderX Results)

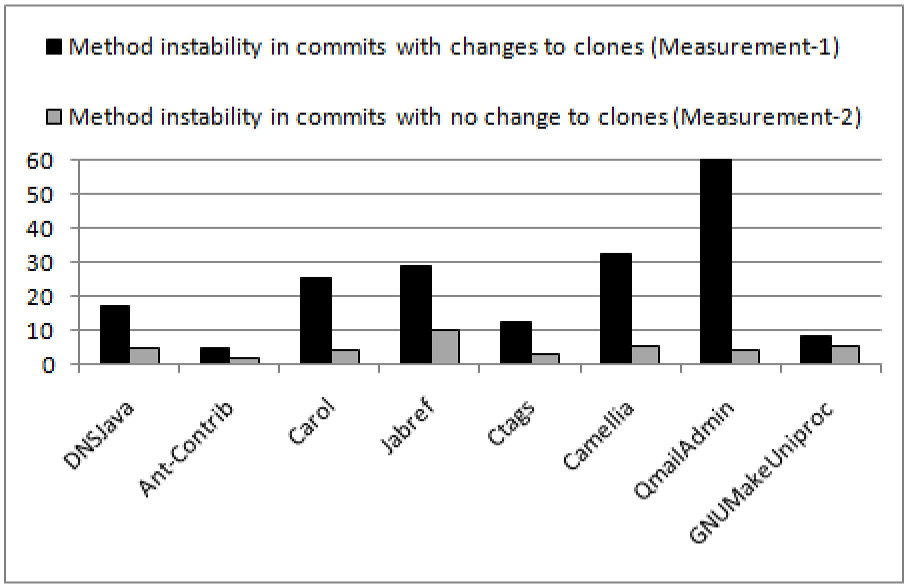


Figure 5.12: Average number of changes per commit operation (for NiCad Results)

- comparison of the instabilities of fully cloned and fully non-cloned method genealogies
- comparison of the instabilities of partially cloned and fully non-cloned method genealogies

The intuition behind these comparisons was that if both fully cloned and partially cloned methods exhibit higher instability compared to the fully non-cloned methods, clones can be a possible cause of the higher instability of cloned (fully or partially) methods.

The set CMG contains both fully cloned and partially cloned method genealogies. A method genealogy is termed as fully cloned genealogy if each method instance of this genealogy is fully cloned. On the other hand, in a partially cloned method genealogy there is at least one method instance that is not fully cloned. The sets of fully cloned and partially cloned genealogies are termed as FCG and PCG respectively. For each of these sets we computed the value of the already defined instability metric $MILS$ according to the following equations.

$$MILS_{FCG} = \frac{NOC_{FCG} \times 10000}{|FCG| \times ALS_{FCG} \times AS_{FCG}} \tag{5.5}$$

$$MILS_{PCG} = \frac{NOC_{PCG} \times 10000}{|PCG| \times ALS_{PCG} \times AS_{PCG}} \tag{5.6}$$

We calculated $MILS_{FCG}$ and $MILS_{PCG}$ for each of the candidate systems using two clone detection tools. The comparison between the instabilities of fully cloned and fully non-cloned method genealogies has been shown in the graphs of Fig. 5.7 (CCFinderX) and Fig. 5.8 (NiCad). According to each of these graphs, *fully cloned methods exhibit higher instability compared to the fully non-cloned methods for most of the subject systems.*

We compare the instabilities of fully non-cloned and partially cloned method genealogies in the graphs of Fig. 5.9 (CCFinderX) and Fig. 5.10 (NiCad). According to these two graphs, *partially cloned methods appear to exhibit higher probability of being more unstable than the fully non-cloned methods.*

From the above scenario we infer that clones can be a possible cause of making the cloned (fully or partially) methods more unstable.

5.4.5 Investigation on the commits having changes to the cloned portions of the cloned methods

We also investigated the commit operations of the Java and C subject systems to determine whether clones are responsible for higher instability of cloned methods. For the purpose of this investigation we separated the commit operations happened to a subject system into the following two disjoint sets.

Set 1: The commits with some changes to the cloned portions of the cloned methods are contained in this set.

Set 2: This set consists of the commits with no changes to the cloned portions of the cloned methods

Then, we calculated the following two measures for these two sets.

Measurement-1: Average number of changes to methods (cloned or non-cloned) per commit operation of Set 1.

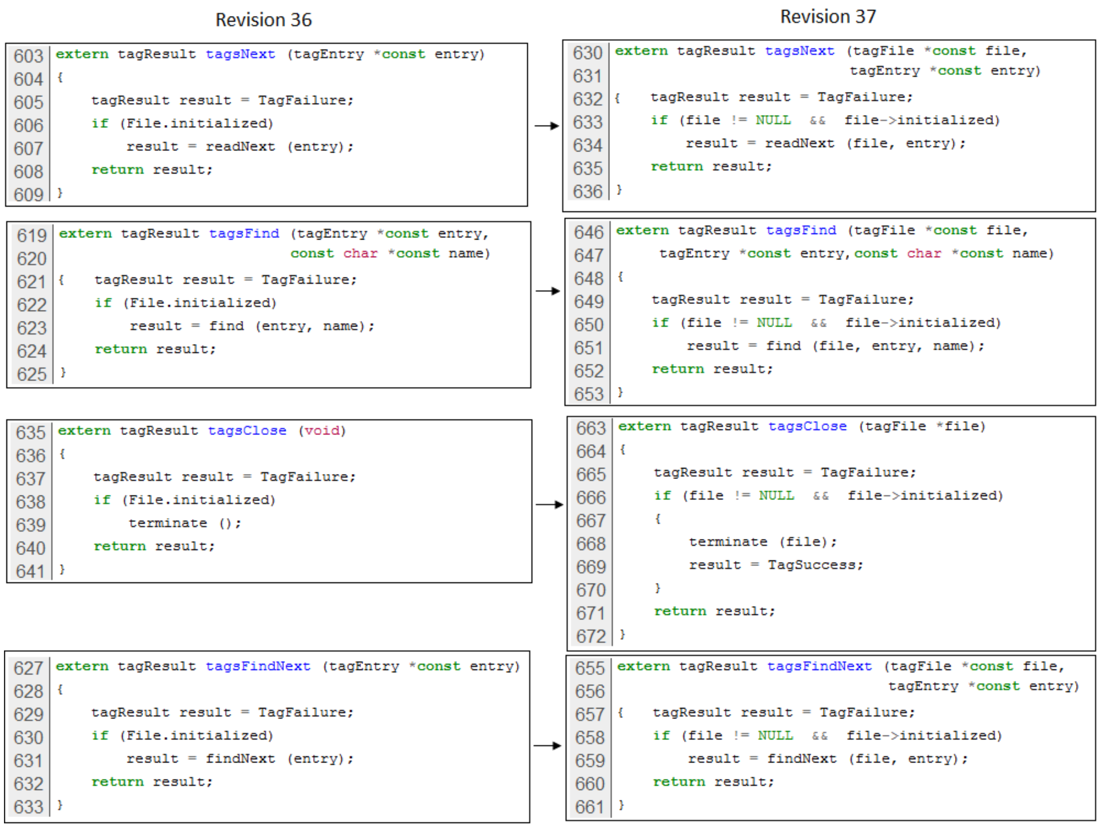
Measurement-2: Average number of changes to methods (cloned or non-cloned) per commit operation of Set 2.

We calculated the values of these two measurements using two clone detection tools for each of the subject systems and plotted these values in two graphs (Fig. 5.11 and Fig. 5.12). Each of these two graphs demonstrates that Measurement-1 is much higher than Measurement-2. We performed MWW (Mann Whitney Wilcoxon) tests [78, 79] on the observed values of these two measures to see whether Measurement-1 is significantly higher than Measurement-2. The p-values (probability values) of the tests corresponding to the results obtained for CCFinderX and NiCad are 0.0147 and 0.0018 respectively. Both of these values are less than 0.05 and it indicates that Measurement-1 is significantly higher than Measurement-2. In other words average number of changes to the commits where there are some changes to the clones is significantly higher than the average number of changes to the commits with no changes to the clones. While calculating Measurement-1 we are considering the commits with changes to clones. To calculate Measurement-2 we are considering those commits where there are no changes to the clones. Thus, it might seem that Measurement-1 will generally be greater than Measurement-2. However, the percentage of cloned code in the subject systems is significantly lower. According to some studies [6, 10], only 7% to 23% of a codebase can be cloned code. Thus, our calculation process of Measurement-1 does not necessarily bias it to be greater than Measurement-2.

From our observation on Measurement-1 and Measurement-2 we conclude that changes to the cloned portions of methods are always associated with higher amount of changes. Such a finding is consistent with that of Lozano and Wermelinger [74]. To find the reason why the commits with changes to the clones are always associated with higher amount of changes we performed the following manual investigation on the commits happened to Ctags [33].

5.4.6 Manual Investigation on the commits having changes to the cloned portions of cloned methods

We manually investigated all of the changes happened to the clones of the subject system Ctags [33]. We choose this subject system because it is relatively small in size and has a reasonably long revision history. Our investigation was based on the combined type clone results of NiCad. As indicated in Table 5.1, we analyze 774 revisions as well as commits of Ctags [33]. We found 61 commit operations where there were some changes to the cloned portions of cloned methods. We separated these commits into two sets. One set consists of those commits each of which had changes to a single clone fragment only. We term this set as *Set-1*. The other set contains those commits each of which had changes to more than one clone fragments. We term this set as *Set-2*. We found respectively 33 and 28 commits in Set-1 and Set-2. Then we analyzed



```
extern tagResult tagsNext (tagEntry *const entry)
{
    tagResult result = TagFailure;
    if (File.initialized)
        result = readNext (entry);
    return result;
}
```

```
extern tagResult tagsNext (tagFile *const file,
                           tagEntry *const entry)
{   tagResult result = TagFailure;
    if (file != NULL  &&  file->initialized)
        result = readNext (file, entry);
    return result;
}
```

```
extern tagResult tagsFind (tagEntry *const entry,
                           const char *const name)
{   tagResult result = TagFailure;
    if (File.initialized)
        result = find (entry, name);
    return result;
}
```

```
extern tagResult tagsFind (tagFile *const file,
     tagEntry *const entry, const char *const name)
{
    tagResult result = TagFailure;
    if (file != NULL  &&  file->initialized)
        result = find (file, entry, name);
    return result;
}
```

```
extern tagResult tagsClose (void)
{
    tagResult result = TagFailure;
    if (File.initialized)
        terminate ();
    return result;
}
```

```
extern tagResult tagsClose (tagFile *file)
{
    tagResult result = TagFailure;
    if (file != NULL  &&  file->initialized)
    {
        terminate (file);
        result = TagSuccess;
    }
    return result;
}
```

```
extern tagResult tagsFindNext (tagEntry *const entry)
{
    tagResult result = TagFailure;
    if (File.initialized)
        result = findNext (entry);
    return result;
}
```

```
extern tagResult tagsFindNext (tagFile *const file,
                               tagEntry *const entry)
{   tagResult result = TagFailure;
    if (file != NULL  &&  file->initialized)
        result = findNext (file, entry);
    return result;
}
```

Figure 5.13: The changes in CTAGS happened to the commit operation applied on revision 36

the commits in Set-2 to determine whether the clone fragments changed in a particular commit belong to the same clone class. Our intention was to find the reason why changes are happening to multiple clone fragments at a time and whether there is any relationship among these changes.

We identified 19 commit operations in Set-2 where more than one clone fragments belonging to the same clone class were changed in the same way. The pattern of changes indicate that they were made because of maintaining consistency in the clone fragments. The changes happened in the commit operation applied on revision-36 have been shown in Fig. 5.13. The figure shows four methods in revision 36 and their corresponding snap-shots in revision 37. According to NiCad result these four methods are four clone fragments (Type 3 clones) that belong to the same clone class in revision 36. The commit operation on revision 36 changed each of these methods by adding an extra parameter *file*. We can easily understand the changes happened to the methods in revision 36 by comparing the corresponding snap-shots in revision 37. The changes imply that they were made to all the clone fragments for ensuring consistency. We consider these 19 commit operations in a separate set and term this set as CEC (The set of Consistency Ensuring Commits).

However, for each of the remaining 9 commit operations in Set-2, the clone fragments that received some changes belonged to different clone classes. In other words, no two clone fragments in such a commit belonged to the same clone class.

We observe that in all 61 commits (the commits with some changes to the cloned portions of the cloned methods) the cloned portions of the cloned methods received 175 changes in total. Among these changes, 85 (48.57%) changes happened during the 19 commit operations (the CEC set) for ensuring the consistency among the cloned fragments. We thus see that 31.14% of the 61 commits are for ensuring consistency of the changes among the cloned fragments. From this scenario we come to the conclusion that a major portion of the changes happening to the cloned code are made for ensuring the consistency of the clone fragments. These consistency ensuring changes possibly require additional effort during maintenance.

We investigated the clone fragments appeared in the set CEC to see whether we can categorize the clone fragments that require consistency ensuring changes. We found that for 12 commits (63.15%) in CEC the clone fragments are full methods (not just a portion of method). For the remaining commit operations, the clone fragments were either condition (if / else) blocks or case statements. From this we conclude that consistency ensuring changes mainly happen to the fully cloned methods.

We further analyzed the clone fragments changed in commits of the set CEC to determine which type(s) of clones mainly require the consistency ensuring changes. According to our observation,

- 12 commits (63.15%) in CEC contained changes to the Type 3 clone fragments.
- 8 commits (42.1%) in CEC contained changes to the Type 2 clone fragments.
- 5 commits (26.31%) in CEC contained changes to the Type 1 clone fragments.

From this scenario we see that highest proportion of the consistency ensuring commits involve changes to the Type 3 clone fragments. However, this percentage for the Type 2 clones should also be taken into account. The lowest proportion of commits (in *CEC*) contained changes to the Type 1 clones. Possibly because of such a scenario in other systems, Type 3 clones appear to exhibit the highest probability of getting more dispersed changes compared to the other two types of clones (Fig. 5.3).

5.5 Threats to Validity

We calculated and analyzed the dispersions of changes of cloned and non-cloned code for only 16 subject systems which are not sufficient enough for taking any general decision about different types of clones and programming languages. Also, some other important factors such as programmer expertise, application domain, programmer‘s knowledge about application domain were not considered in our experiment. But, our selection of subject systems considering thirteen application domains, four programming languages, diversified sizes and revisions have considerably minimized these drawbacks, and thus we believe that our findings are significant.

5.6 Related Work

Over the last several years, the impact of clones has been an area of focus for software engineering research resulting in a significant number of studies and empirical evidence. Kim et al. [59] proposed a model of clone genealogy. Their study with the revisions of two medium sized Java systems showed that refactoring clones may not always improve software quality. They also argued that aggressive and immediate refactoring of short-lived clones is not required and that such clones might not be harmful. Saha et al. [104] extended their work by extracting and evaluating code clone genealogies at the release level of 17 open source systems involving four different languages. Their study reports similar findings to Kim et al. and concludes that most of the clones do not require any refactoring effort.

Kapser and Godfrey [55] strongly argued against the conventional belief of harmfulness of clones. In their study they identified different patterns of cloning and showed that about 71% of the cloned code has a kind of positive impact in software maintenance. They concluded that cloning can be an effective way of reusing stable and mature features.

Lozano and Wermelinger [75] developed a prototype tool to track the frequency of changes of cloned and non-cloned code with method level granularity. On the basis of their study on four open source systems they concluded that the existence of cloned code within a method significantly increases the required effort to change the method. In a recent study [74] they further analyzed clone imprints over time and observed that cloned methods remain cloned most of their life time and cloning introduces a higher density of modifications in the maintenance phase.

Juergens et al. [52] studied the impact of clones on large scale commercial systems and suggested that inconsistent changes occurs frequently with cloned code and nearly every second unintentional inconsistent change to a clone leads to a fault. Aversano et al. [5] on the other hand, carried out an empirical study that combines the clone detection and co-change analysis to investigate how clones are maintained during evolution or bug fixing. Their case study on two subject systems confirmed that most of the clones are consistently maintained. Thummalapenta et al. [114] in another empirical study on four subject systems concluded that most of the clones are changed consistently and other inconsistently changed fragments evolve independently.

In a recent study [40] Göde and Harder replicated and extended Krinke's study [64] using an incremental clone detection technique to validate the outcome of Krinke's study. They supported Krinke by assessing cloned code to be more stable than non-cloned code in general while this scenario reverses with respect to deletions.

Hotta et al. [47] studied the impact of clones by measuring the modification frequencies of cloned and non-cloned code of several subject systems. Their study using different clone detection tools suggests that the presence of clones does not introduce extra difficulties to the maintenance phase.

Krinke [63] measured how consistently the code clones are changed during maintenance using *Simian* [108] and *diff* on Java, C and C++ code bases considering Type-I clones only. He found that clone groups changed consistently through half of their lifetime. In another experiment he showed that cloned code is more stable than non-cloned code [64]. In his most recent study [65] he calculated the average last change dates of the cloned and non-cloned code and observed that cloned code is more stable than non-cloned code.

None of the existing studies measured the dispersion of changes. But, without measuring dispersion we cannot accurately measure the impact of a particular region (cloned or non-cloned). We have introduced and measured dispersion. Our experimental results suggest that the changes in the cloned regions are sometimes more dispersed than the changes in the non-cloned regions of a subject system.

5.7 Conclusion

In this chapter we performed a more in-depth investigation on one of our proposed metrics: *change dispersion.* Intuitively, higher dispersion of changes indicates higher maintenance effort and costs. According to our empirical study and analysis on 16 subject systems written in four different programming languages (Java, C, C# and Python) involving two clone detection tools (CCFinderX and NiCad):

- *higher dispersion of changes is a strong indicator of higher instability in source code (the correlation coefficient between change dispersion and code instability = 0.800012).*

- dispersion of changes in cloned code is sometimes higher than the dispersion in non-cloned code. More specifically, the clones in Java and C systems exhibit a higher likelihood of having more dispersed changes compared to the clones in C# and Python systems. From this we suspect that *clones in the subject systems written in Java and C possibly require higher maintenance effort and cost compared to the non-cloned code.*

- Type 3 clones exhibit more dispersed changes compared to Type 1 and Type 2 clones.

We further investigate the instability of those cloned and non-cloned methods to which the changes were dispersed during evolution disregarding those methods (in these regions) that did not change. According to our observation, both fully cloned and partially cloned methods have a higher likelihood of being more unstable compared to the fully non-cloned methods. From this we conclude that *clones are a possible cause of instability in fully cloned and partially cloned methods.*

We also observed that the commit operations with some changes to the cloned portions of the cloned methods always contain significantly higher amount of changes (shown with statistical significance test) compared to the commit operations with no-changes to the cloned portions. From this we conclude that *changing a clone results in a higher amount of change.*

According to our manual investigation on the commit operations of Ctags [33],

- A significant portion (48.57% according to our observation on Ctags) of the changes occurring to clones is made to ensure that clone fragments belonging to the same clone class remain consistent.

- Consistency ensuring changes mainly take place in fully cloned methods.

- A significant proportion (63.15%) of the consistency ensuring changes happen to the Type 3 clones.

Finally we conclude that clones are sometimes more unstable as well as harmful for the maintenance phase compared to non-cloned code. Our introduced metric *change dispersion* also reflects this. Dispersion has the potential to assist in fine grained calculation of the impacts of cloned and non-cloned code in the maintenance phase.

As clones appear to be potentially harmful for the maintenance phase, we also investigate whether clones have any effect on the co-changeability of program artifacts. Co-changeability of program entities is another aspect of stability. In the next chapter we define co-changeability and describe our investigation regarding the effects of clones on co-changeability.

Chapter 6

Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems

6.1 Motivation

From our previous investigations presented in Chapters 4 and 5 we found that clones are generally more unstable compared to the non-cloned code and also, most of the changes occurring to the clones are made so that the clone fragments belonging to a particular clone class remain consistent. These consistency ensuring changes possibly take part in increasing the instability of software systems.

However, co-changeability (the likelihood of changing together) of program artifacts (such as: files, methods, classes) is another aspect of stability. Intuitively, higher co-changeability of program artifacts during commit operations is an indication of higher instability of source code. As clones appear to increase system instability considering the previous stability aspects (discussed in Chapters 4 and 5), we also investigated whether clones promote co-changeability of program artifacts. We present our investigation regarding this issue in this chapter.

There are many empirical studies regarding the co-changeability of program artifacts. The existing studies have focused on programmer awareness of dependencies among program artifacts. There are numerous studies on: (i) the detection of class, method or file level interdependencies and co-change patterns [14, 16, 17, 39, 117, 122], (ii) visualization of these dependencies and patterns [12, 13, 23], (iii) impacts of changing program components [17, 18], and (iv) propagation of changes [42] based on the software evolution history. However, the existing studies fail to focus on the following important issues.

(1) Investigation on the effect of clones on co-changeability: There is no study investigating whether clone has any effect on co-changeability of program artifacts. From our previous investigations we found that presence of clones is sometimes a potential threat to the stability of software systems. Thus, this is also important to investigate whether clones promote co-changeability of program artifacts. Intuitively, higher co-changeability of program artifacts is an indication of higher instability of software systems.

(2) Minimization of changes: Each of these studies support programmer awareness about which other entities might need to be modified while modifying a particular entity. But, none of these studies investigated the reasons why some software systems exhibit higher changeability than others. None of these studies provide a way to reduce the number of modifications.

(3) Minimization of module dependencies: None of these studies provide a way to minimize dependencies (couplings) among program modules. Intuitively, higher dependency among program modules makes the modifications to the modules more difficult. There are many refactoring mechanisms but these cannot remove complex dependencies in many situations. Suppose a particular user-defined method is being used by two different functionalities (or tasks). While making changes to this method we should be concerned about all other methods implementing these two functionalities. In a real scenario, a particular method might be used by many functionalities or tasks (defined in Section 6.2) and in that case, modifications to the shared method will become more difficult. If we cannot eliminate such complex dependencies, the software may become increasingly complex over time, and may go beyond the point of maintainability. None of the existing studies show possible ways of minimizing module dependencies.

Focusing on the above issues we performed an in-depth empirical study with the following contributions.

(1) We investigate the effect of clones on the co-changeability of program artifacts.

(2) Our study discovers a potential cause of increased modifications to program artifacts.

(3) We propose a possible way to minimize module dependencies (or couplings) as well as source code modifications.

We performed our investigation considering method level granularity. We introduce two metrics: **(1)** *COMS* (Co-changeability of Methods), and **(2)** *CCMS* (Connectivity of Co-changed Method Groups). ***COMS*** quantifies the extent to which a particular method co-changes with other methods. ***CCMS*** measures the sharing of methods among different functionalities or tasks. In other words, CCMS is a measure of dependencies (or couplings) among methods. In this research work, we extensively investigated the influence of **CCMS** on both **COMS** and source code modifications.

We performed a case study with hundreds of revisions of six open source software systems written in three different languages and evaluated the introduced metrics in three ways: **(i)** for the whole software system, **(ii)** for the cloned regions of the system, and **(iii)** for the non-cloned regions of the system. We also measured a change related metric ***CMP*** (Code Modification Probability) and observed whether higher connectivity among co-changed method groups causes increased source code modifications.

According to our experimental results, **(i)** *higher **CCMS** causes higher **COMS***, **(ii)** *higher **CCMS** is also a possible cause of increased source code modifications as well as efforts in the maintenance phase (empirically evaluated with statistical support)*, **(iii)** *the **COMS** in the cloned regions of a software system is negligible compared to the **COMS** in the non-cloned regions* and **(iv)** *cloning can be a possible way of minimizing **CCMS** for those situations where functionalities or tasks are connected but are likely to evolve independently.*

6.2 Terminology

6.2.1 Task

Before the commencement of a particular project it is generally decomposed into multiple smaller tasks which are assigned to the programmer responsible for the task. If we consider an example project 'Library Management System', an example task could be 'User Login' or 'Issuing a Book' etc. While accomplishing a particular task a developer can further decompose it into multiple methods or classes or other language specific structures which together perform the particular task. In this research work we focus on the methods that belong to a particular task.

6.2.2 Co-changed Method Group

If a particular project is developed under version controlling system such as SVN, the responsible programmer commits the relevant source code files (possibly with other non-source-code files if necessary) to SVN after partial or full implementation of each task. So it is very likely that the methods which are added or modified in a particular commit operation belong to a particular task. According to our definition these methods which are added or modified in a particular commit operation form a co-changed method group.

During the evolution of a software system multiple revisions of it are created because of multiple commit operations. We denote the revisions by $revision(i)$ where $1 \leq i \leq n$. Here n is the total number of revisions of the software system created so far. A commit operation $commit(i)$ on $revision(i)$ causes the next revision $revision(i{+}1)$ to be created. For most of the cases a particular commit operation consists of several changes to the source code. The methods that are created or that receive some changes in a particular commit operation generally belong to a particular task. But, a single commit operation might also affect multiple tasks. Such commits are termed 'atypical commits' [75]. For excluding atypical commits Lozano and Wermelinger [75] discarded 2.5% of the largest commit operations while analyzing the revisions of a subject system.

Detection of Co-changed Method Groups: To determine the co-changed method group for a particular commit operation $commit(i)$ we need to accomplish several tasks: (i) detection of all methods in $revision(i)$ with corresponding beginning and ending line numbers, (ii) determination of changes in $revision(i)$ with corresponding line numbers, (iii) mapping these changes to the detected methods of $revision(i)$ and at last (iv) retrieval of the changed methods. For a sequence of n revisions of a subject system we will get a sequence of $n - 1$ commit operations. After discarding the commit operations with atypical changes we will obtain our target commit operations. If the count of these target commit operations is t, we will have t co-changed method groups. But, it is very likely that a particular co-changed method group will appear multiple times in this sequence of t groups because, changes in multiple commit operations might be centered around the same or similar tasks. So, we need to determine the unique co-changed method groups for a sequence of commit operations. Unique co-changed group identification process is elaborated in *Algorithm* 1 (Section 6.4).

Difference Between Our Proposed Methodology and Existing Methodologies: Our process of detecting co-changed method groups is a variant of the association rules introduced by Zimmerman et al. [122]. The original association rule considers the co-changes of all types of entities including packages, methods, classes etc. In our implementation we have considered the co-changes among methods only. The methodology proposed by Zimmerman et al. [122] is not suitable for the investigation regarding method sharing. Because, it does not show how methods are grouped for accomplishing different functionalities. Also, no other existing methods tried to extract possible groups of co-changing entities. Our proposed algorithm (Algorithm 1) detects possible groups of co-changing methods. From these groups we can easily identify shared methods (the methods that are shared by more than one group) and can calculate **CCMS** and **COMS**.

Minimization of the probability of false associations: Generally while developing or making changes to a particular functionality a programmer writes code, checks whether the code is working properly and continues this process until they achieve the required goal. During this process of writing and checking the programmer might erroneously change program artifacts that are not related to the functionality. These errors can be fixed through a continuous checking and writing process. When the programmer commits the changes to SVN, they try to be sure that the changes they make do not contain bugs because the committed code is likely used by other programmers. So, we see that before committing the programmer remains more concerned about the accuracy of the code. So, if we associate the entities changed in a commit operation, it is likely that we will avoid irrelevant or false associations. Also, discarding atypical commits increases the probability of avoiding false associations.

6.2.3 Co-changeability of Methods (COMS)

Consider that we detected m unique co-changed method groups. Each of these groups is denoted by $g(i)$ where $1 \leq i \leq m$. The count of elements in a particular group $g(i)$ is denoted by $|g(i)|$. If we want to change a particular method in a group $g(i)$, we also need to be concerned about the other $|g(i)| - 1$ members belonging to this group. So, we calculate the COMS for group $g(i)$ according to Eq. 6.1.

$$COMS(g(i)) = |g(i)| \times (|g(i)| - 1) \tag{6.1}$$

Here, $COMS(g(i))$ is the COMS for group $g(i)$. The total COMS of all groups in a software system can be calculated using the following equation.

$$Total\ COMS = \sum_{i=1}^{m} COMS(g(i)) \tag{6.2}$$

Here m is the total number of unique co-changed method groups in the software system. Co-changeability is influenced by group connectivity and this influence is not trivial. In the following subsections we at first define CCMS and then mathematically show the influence of CCMS on COMS.

6.2.4 Connectivity of Co-changed Method Groups (CCMS)

If two co-changed method groups share a common subset of methods, we say that these two groups are connected. We define the connectivity of this connection constructed by these two co-changed method groups by the count of methods shared between these two groups. A particular co-changed method group can have multiple connections with multiple other groups.

Suppose a group $g(i)$ has n connections. The subsets of methods corresponding to a connection is denoted by $s(j)$ where $1 \leq j \leq n$. The connectivity of this group $g(i)$ can be calculated by the summation of the connectivities of these connections in the following way.

$$CCMS(g(i)) = \sum_{j=1}^{n} |s(j)| \tag{6.3}$$

Here, $CCMS(g(i))$ is the connectivity of the co-changed method group g(i).

We should note that this equation (Eq. 6.3) gives more emphasis on those methods that are included in higher number of connections. Suppose, a particular method group has n connections with n other groups. If a particular method in this group remains included in m of these connections where $m \leq n$, this method will be considered m times by this equation.

6.2.5 Influence of CCMS on COMS

$CCMS$ can sometimes influence $COMS$. According to the equation Eq. 6.1, $COMS$ of a particular method group can be increased by increasing the size of the method group. In the following three examples we will see how higher connectivity among method groups (higher $CCMS$) increase $COMS$. In each of these examples we will see the grouping of ten methods in two groups.

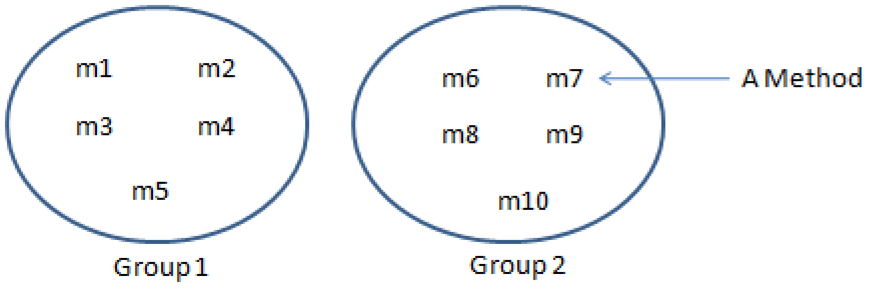


Figure 6.1: Two groups with no common methods.

In the first example (Fig. 6.1) we see that each of the two groups contains five methods and the groups are not connected. Total $COMS$ of these two groups $= 5 \times 4 + 5 \times 4 = 40$.

In the second example (Fig. 6.2) we see that each of the two groups contains six methods and the groups have two common methods. Total $COMS$ of these two groups $= 6 \times 5 + 6 \times 5 = 60$.

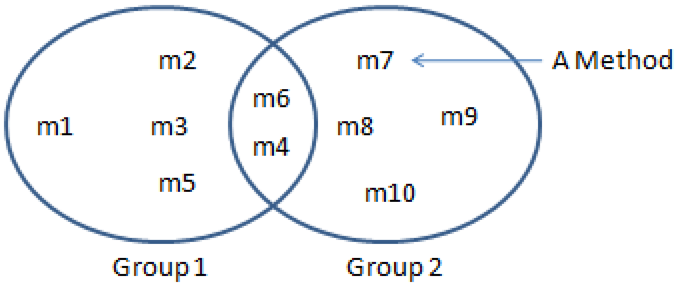


Figure 6.2: Two groups with two common methods.

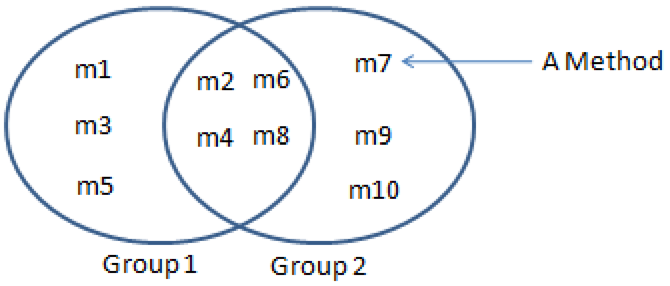


Figure 6.3: Two groups with four common methods.

In the third example (Fig. 6.3), each of the two groups contains seven methods and four methods are common. Total $COMS$ of these groups = $7 \times 6 + 7 \times 6 = 84$.

Thus we can see that higher connectivity in method groups, in other words higher $CCMS$, can sometimes increase $COMS$ (co-changeability of methods).

However, in real scenario, a single group might have several connections with several other groups which will increase the probability of method co-changes to a great extent. So, *by reducing CCMS we can reduce COMS.*

6.3 COMS in Cloned and Non-cloned Code

We derived the equation for calculating COMS for the whole software system from its observed unique co-changed method groups. We can also calculate COMS separately for cloned and non-cloned regions of a software system. We have done this because there are many empirical studies [40,47,52,55,65,74,75,83,86,87] with controversial outcomes about the impacts of clones in the maintenance phase. Some studies [52, 74, 75, 83,87] imply that that cloned code is more harmful than non-cloned code while others [40,47,55,65] report the opposite. We wanted to find whether cloned code exhibits more co-changeability of methods than non-cloned code or not.

We have already discussed the procedure for calculating the co-changed method group for a particular commit operation *commit*(i) applied on a revision *revision*(i). If we apply a clone detection tool on *revision*(i), we can separate the cloned and non-cloned blocks belonging to *revision*(i). By mapping these blocks to the methods of this revision we can determine which methods are cloned and which methods are not. We also need to know which lines of a particular method are cloned. According to our consideration, a cloned method might be fully or partially cloned. A partially cloned method has some non-cloned blocks in it. As we know the cloned lines of each method, we can map the changes belonging to commit operation *commit*(i) to the cloned and non-cloned portions of the methods. Thus, we can determine the group of methods which have changes in their cloned portions and also the group of methods which have changes in their non-cloned portions. We call these groups co-changed method groups of cloned and non-cloned code respectively. A partially cloned method which has some changes in its non-cloned portions but not in its cloned portions will belong to the co-changed method group of non-cloned code not to the group of cloned code. By determining the unique co-changed method groups in cloned and non-cloned regions we can determine the COMS for cloned and non-cloned regions.

6.4 Experimental Steps

For determining the unique co-changed method groups, their connectivities and co-changeabilities for a particular software system we performed several sequential steps including: (1) download preprocessing of subject systems, (2) method detection and extraction, (3) clone detection, (4) detection and reflection of changes, (5) storage of methods, (6) method genealogy detection, (7) determination of unique co-changed method groups, and (8) calculation of metrics. However, we have described the first six steps in Section 4.4 of Chapter 4. For this reason we skipped these steps here.

6.4.1 Determination of unique co-changed method groups

As we inspect the methods affected by the commit operations sequentially, we are store and update the co-changed method groups according to the algorithm *Algorithm* 1. For eliminating the effects of atypical changes [75] that affect multiple functionalities (or goals) at a single commit operation we discarded 2.5% of the largest commit operations for each of the subject systems from our consideration as was done by Lozano and Wermelinger [75]. From each of the commit operations of our target set obtained by eliminating the commit operations with atypical changes, we extracted the co-changed method groups sequentially. Method genealogy extraction was necessary to determine whether a currently detected group has already appeared previously.

Suppose, we have already detected some unique co-changed method groups by examining some commit operations. We call this list of existing groups *existing list.* After getting a new group from the next commit operation, we at first check whether this group is a proper subset of any group in the *existing list.* If this is true, we ignore this new group, otherwise we check the *existing list* to find any group which is a proper subset of this new group. We discard these groups from the *existing list* and add the new group to it. Then, we proceed with the next commit operation. However, at the very beginning of this process (while examining the first commit operation), the *existing list* remains empty. These sequential steps are elaborated in the *Algorithm* 1.

Though we have discarded 2.5% of the largest commit operations (atypical commits), there is little probability that a co-changing method group will contain unrelated methods. However, the algorithm ensures the detection of all possible groups.

Algorithm 1 Determine unique co-changed method groups

```
Require: The sequence of commit operations, ExistingList of groups (initially empty)
Ensure: Unique co-changed method groups.
  for each commit operation $commit(i)$ do
    $NewGroup \leftarrow$ the list of changed methods
    for each group $g(j)$ in ExistingList do
      if $NewGroup \subset g(j)$ then
        Ignore NewGroup
        exit the loop.
      else
        if $g(j) \subset NewGroup$ then
          Delete $g(j)$ from ExistingList
        end if
      end if
    end for
    if NewGroup is not ignored then
      Add NewGroup to the ExistingList.
    end if
  end for
```

6.4.2 Metric Calculations

Calculation of CCMS and COMS

After determining the unique co-changed method groups we calculated the values of our proposed metrics, CCMS and COMS, for all method groups in total. However, while finding correlations, we calculated the average values per co-change method group for these metrics.

Calculation of CMP (code modification probability)

For finding correlations of code modifications with connectivity, we calculated the code modification probabilities (CMP) for each of the candidate software systems using the following equation.

$$CMP = \frac{\sum_{c \epsilon CMS} CML(c)}{|CMS| \times \sum_{c \epsilon CMS} LOC(c)} \tag{6.4}$$

In the above equation (Eq. 6.6), CMP is the code modification probability. $CML(c)$ denotes the count of modified (added, deleted or changed) lines of a software system during the commit operation c. CMS is the set of all commits where there were some modifications to the source code. $LOC(c)$ is the count of total lines of code of a candidate system during commit operation c.

We see that Eq. 6.6 calculates the source code modification probability by considering only those commit operations where there were some modifications to the source code. For each of the commit operations with some modifications to the source code we calculated the following two measurements.

(1) Total number of lines in that revision on which the commit was applied ($LOC(c)$).

(2) Total number of lines modified because of the commit operation ($CML(c)$).

We know that each commit operation creates a new revision. For calculating the number of lines modified in a particular commit operations we at first identify two revisions: (1) the revision on which the commit operation was applied and (2) the revision that was created just after applying the commit operation. Then, we use UNIX *diff* to identify the lines that were modified in the older revision to create the newer one.

6.5 Experimental Setup

We have already described our implementation framework and NiCad setup in Chapter 4. In this experiment we investigate six subject systems written in three different programming languages. The subject systems are: (1) Carol, (2) DNSJava, (3) Ctags, (4) Camellia, (5) GreenShot, and (6) MonoOSC. The descriptions of these systems have already been given in Table 4.3.

6.6 Experimental Results

We applied our methodology to six open source software systems and calculated the following for each of them.

(i) The count of unique co-changed method groups

(ii) Total COMS for all co-changed method groups

(iii) Total CCMS of all co-changed method groups

(iv) Code modification probability (CMP).

We provide these values in Tables 6.1 and 6.2.

Table 6.1: Total CCMS and Total COMS for different subject systems

Language	Subject Systems	UGC	COMS	CCMS
Java	Carol	157	1578	118
	Dnsjava	329	4010	956
C	Ctags	142	1532	920
	Camellia	67	938	526
C#	GreenShot	241	3262	1741
	MonoOSC	105	2246	1785

UGC = Unique co-changed Group Count

Table 6.2: Source code modification probabilities for different subject systems

Language	Subject Systems	NCMS	CMP
Java	Carol	383	3.23E-06
	Dnsjava	1254	1.75E-06
C	Ctags	447	3.63E-06
	Camellia	147	5.24E-06
C#	GreenShot	586	2.99E-06
	MonoOSC	236	2.35E-05

CMP = Code modification probability

NCMS = Number of commits with modifications in the source code

We determined the Pearson correlations between CCMS and the other 2 measures in the above list (excluding the count of unique co-changed method groups) and recorded the results in Table 6.3. For calculating the correlations, we calculated the average values of CCMS and COMS per method group for each subject system. We also calculated the COMS and CCMS separately for cloned and non-cloned code for each of the subject systems.

6.6.1 Analysis of Experimental Results

We determined the strengths of correlations between CCMS and two other measures: COMS and CMP. The correlations recorded in Table 6.3 are explained below.

Table 6.3: Correlation of CCMS with COMS and CMP

Lang.	Systems	CCMS	COMS	CCMS	CMP
Java	Carol	0.7515	10.0509	0.7515	3.23E-06
	Dnsjava	2.9057	12.1884	2.9057	1.75E-06
C	Ctags	6.4788	10.7887	6.4788	3.63E-06
	Camellia	7.8507	14	7.8507	5.24E-06
C#	GreenShot	7.22	13.5352	7.22	2.99E-06
	MonoOSC	17	21.3904	17	2.35E-05
Correlation Co-efficient		**0.941557804**		**0.901590668**	

CCMS = Connectivity of Co-changing method groups

COMS = Co-changeability of methods

CMP = Code modification probability

Correlation between CCMS and COMS

From Table 6.3 we see that there is a strong correlation between connectivity and co-changeability. The Pearson correlation co-efficient between these two is 0.9415. Such a strong correlation is expected. Also, we have mathematically shown that connectivity has direct influence on the co-changeability. Thus we can say that *co-changeability can be minimized by minimizing connectivity.*

Correlation of CCMS with CMP

To determine whether higher connectivity is an indicator of higher modification of source code, we calculated the correlation between *CMP* (code modification probability) and *CCMS*. We observed that *CMP* is strongly correlated with connectivity with a correlation co-efficient of 0.9015. So, higher changeability (modification probability) in source code is an indicator of higher connectivity among co-changed method groups. As *CCMS* measures the intensity of method sharing among co-changing method groups, we can say that *higher method sharing might be a possible cause to higher instability in source code.*

6.6.2 Clone related analysis

We calculated the COMS for non-cloned code and three types (Type 1, Type 2 and Type 3) of cloned code of a subject system separately to compare the COMS in cloned and non-cloned code. The average COMS of different types of cloned code and corresponding non-cloned code have been plotted in the graph in Fig. 6.4. In general we see that the COMS of each type of cloned code (except Type 1 clones of Dnsjava) is much smaller than non-cloned code. Also, we have observed that the numbers of co-changed method groups that we found for different types of cloned code are negligible compared to the group counts of non-cloned code.

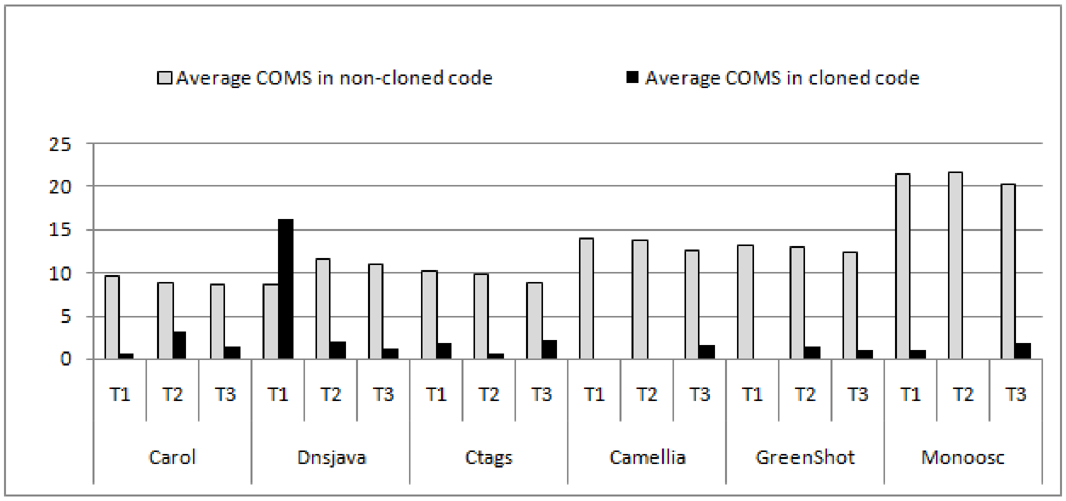


Figure 6.4: Comparison of COMS in non-cloned code and three types of cloned code.

From this we decide that the COMS in cloned code is much lower than the COMS in non-cloned code. Clones are generally created for serving different tasks or functionalities independently, which indicates that cloned methods should have no coupling. This is obviously a good characteristic of clones, which can be carefully used to minimize the COMS in non-cloned code.

6.7 Minimization of COMS

From our mathematical derivation in Section 6.2.5 we see that a higher CCMS (Connectivity of Co-changed Method Groups) causes a higher COMS (co-changeability of methods). Intuitively, higher co-changeability of methods should cause increased modifications to the source code. Also, in the analysis part we have seen that higher instability in source code is an indicator of higher CCMS. COMS can be minimized by minimizing CCMS. There are two ways of minimizing CCMS. These are explained below.

6.7.1 Cloning

Minimization of CCMS is tricky. Cloning is a possible way of minimizing CCMS. If a particular method takes part in implementing several tasks, we can make a separate copy of this method for each of these tasks or functionalities. This is a way of minimizing CCMS without increasing group size. In this way, separate copies of this method will evolve independently with the evolutions of separate tasks. But for this to be a fruitful approach we need to be sure of the following.

(1) The method that is being replicated does not contain a bug. Identification of a bug in any of these copies will require the propagation of bug correction activities to all copies, which will likely increase change efforts.

Table 6.4: Example of connected functionality

Revision 140. Goal: Addition of new administrator	
File path	**Method signature**
qmailadmin/auth.c	set_admin_type()
qmailadmin/autorespond.c	int show_autorespond_line (char*,char*,time_t,char*)
qmailadmin/user.c	int addusernow()
Revision 144. Goal: Mail forwarding	
File path	**Method signature**
qmailadmin/alias.c	show_dotqmail_lines (char*,char*,time_t,char*)
qmailadmin/autorespond.c	int show_autorespond_line (char*,char*,time_t,char*)
qmailadmin/command.c	process_commands()
qmailadmin/forward.c	int show_forwards (char*,char*,time_t,char*)
qmailadmin/qmailadmin.c	main(argc,argv)
qmailadmin/util.c	int count_stuff(void)

(2) Cloning can be applied to minimize CCMS only for those situations where the tasks or functionalities that share the common method (or common set of methods) are likely to evolve independently. Otherwise, the synchronization of modifications among the cloned methods will increase the instability of the source code as well as the maintenance effort.

In Table 6.4 we provide a simple example of two connected functionalities in QmailAdmin written in C. Each goal consists of methods from multiple files. The method 'int show_autorespond_line(char*, char*, time_t, char*)' of file 'autorespond.c' connects these two functionalities. This is a very simple connection, but when tasks (or functionalities) become in this way connected we can eliminate connectivity by cloning (assuming the shared method needs to evolve independently for each goal). Just for explanation we can say that a separate copy of this method can be created so that two copies can serve two purposes (which might not be necessary for this case because of the simple connection between the two functionalities).

6.7.2 Method breakdown

If a single method contains multiple sections for multiple functionalities (which is a common case for the C# systems according to our observation), we can split this method into several relevant methods containing the sections corresponding to those functionalities. In this way, we minimize method couplings as well as ensure securities for different functionalities such that while changing the portion of a particular functionality other portions of the other functionalities will not be affected mistakenly.

6.8 Threats to Validity

The sample size of our study is not sufficient enough to draw a general conclusion. Furthermore, the level of expertise of the involved programmers and the nature of applications might also have some effects on the experimental results. However, our selection of six different subject systems of three different programming languages considering the diverse variety of sizes, application domains and hundreds of revisions should minimize these drawbacks considerably.

According to our definition and calculation procedure of co-changed method groups there is very little probability that a co-changed method group will contain unrelated methods and that co-changeability will be over-estimated. Since we are identifying and updating the co-changed method groups from the very beginning of the development phase, we can retrieve all the existing connectivities as well as method sharing among different functionalities. Thus, we believe that we could determine the actual effect of connectivities on co-changeabilities.

6.9 Related Work

Studying the impacts of co-changes is not a new topic. Jafar et al. [48] performed a comprehensive study on macro co-changes considering file level granularity. They introduced two metrics *MCC* (macro co-changes) and *DMCC* (diphase macro co-changes) and using their proposed approach *Macocha* they detected how many files exhibit *MCC* and *DMCC*. Their introduced metrics can assist in mainly two ways: (i) by managing development teams, and (ii) by managing bugs and change propagations.

Zimmermann et al. [122] implemented a tool called *ROSE* and integrated it with *ECLIPSE* as a plugin to achieve three aims: (i) prediction of future changes, (ii) determination of component (file, method, variable etc) couplings that are difficult to detect by program analysis, and (iii) prevention of errors because of incomplete changes. Their *ROSE* prototype could predict which files need to be modified for a particular change request for 26% of cases.

Beyer [12] implemented and described a co-change visualization tool *CCVISU* that can extract the underlying clustering of artifacts in a software system by analyzing CVS log files. *CCVISU* can help us in two ways: (i) understanding the relationships among different software artifacts such as files, classes, methods and packages which is useful for reverse engineering, and (ii) helpful guidance of changes happening in the maintenance phase.

Canfora and Cerulo [17] proposed an impact analysis approach that retrieves the set of affected source files from a change request by mining the information stored in the bug tracking and versioning systems.

Ying et al. [117] proposed and developed a methodology which was capable of recommending relevant source files for a particular modification task by querying previously stored change patterns. The change patterns were extracted by mining data from a software configuration management (SCM) system by applying an association rule mining algorithm. Their recommendation system is intended to reveal valuable dependencies among files.

Gall et al. [38] introduced an approach of discovering logical dependencies and change patterns among different program modules by using the information in the release history of a system. Their logical coupling identification approach is intended to be used to restructure systems to minimize structural problems.

D'Ambros et al. [23] implemented *evolution radar* to visualize module level and file level logical couplings. They argued that their tool integrated with a development environment can support restructuring, re-documentation and change impact estimation.

Hassan and Holt [42] conducted an empirical study on change propagations in the software systems. They have shown that historical co-change information can be used to help developers during the change propagation process.

Zhou et al. [118] presented a Bayesian network based approach for predicting change coupling behaviour between source code artifacts. On the basis of a set of extracted features such as: static source code dependency, frequency of previous co-changes, change significance level, age of change and co-change entities their approach models the uncertainty of change coupling process.

We can see that none of the existing studies focused on how the connectivities among different functionalities affect co-changeability of methods and source code modifications. Also, there is no study on the effects of clones on method co-changeability. Our study investigates these issues.

6.10 Conclusion

In this research work, we described an in-depth investigation on how the sharing of methods among different functionalities affects method co-changeability and modification of source code. For this purpose we proposed and empirically evaluated two metrics: (i) COMS (Co-changeability of Methods) and (ii) CCMS (Connectivity of Co-changed Method groups). The first metric measures the extent to which a particular method co-changes with other methods while the other one quantifies the sharing of methods among functionalities. We analyzed the influence of CCMS on both COMS and modifications of source code. According to our analysis, CCMS causes higher COMS as well as increased modifications in the source code.

We observed that COMS in cloned code is negligible compared to that of non-cloned code and also, cloning can be a solution to minimizing method group connectivity as well as couplings (or dependencies) among methods. Although our study is not exhaustive it contributes in two ways: **(1)** it identifies a possible cause of higher source code modifications and **(2)** it suggests a possible way of minimizing method couplings and code modifications. Our suggested solution has the potential to increase the stability of software systems in the maintenance phase. As future work we are planning to integrate our proposed methodology with the Eclipse IDE as a plugin so that we will be able to visualize the functionalities and their connectivities to find the target functionalities to minimize their connectivities.

Chapter 7
Conclusion

7.1 Concluding Remarks

Software maintenance is one of the most important phases of the software development life cycle. Changes are inevitable during this phase. However, changes in this phase without proper awareness of consequences are sometimes risky. Frequent modifications to a program entity that is logically coupled with several other entities might leave the related entities in an inconsistent state. According to a number of empirical studies [7, 8, 46, 52, 70, 73–75, 87], code clones are responsible for introducing additional modification challenges.

Cloning has been investigated in different studies [5, 7, 8, 11, 40, 46, 47, 49, 52, 55, 59, 63–65, 70, 73–75, 83, 87, 94, 104, 106, 114] with contradictory outcomes regarding its impact on software maintenance and evolution. While there is empirical evidence [7, 8, 46, 70, 73, 75, 87] of several harmful impacts of clones on maintenance, a number of studies [5, 11, 47, 49, 59, 63, 94, 104] have also argued that clones are not harmful and even beneficial for the maintenance phase from different perspectives. Recently, software researchers are measuring the stability of cloned and non-cloned code separately and comparing them with the underlying idea that if cloned code exhibits higher instability in the maintenance phase compared to the non-cloned code, clones should be considered as harmful, because in that case clones require higher effort and cost to be maintained than non-cloned code. However, the stability related studies [40, 47, 65, 74, 75] could not also come to a consensus and there was no concrete answer to the long lived research question "Is cloned or non-cloned code more stable during software maintenance?".

We identified some possible reasons for the contradictory outcomes and also some drawbacks of the existing stability related studies. Focusing on these we performed a series of empirical studies. In our first study we investigated eight stability measurement metrics by implementing seven methodologies that calculate these metrics on the same experimental setup. Six metrics are pre-existing and the remaining two are our proposed new ones. We evaluated these metrics using two clone detection tools (CCFinder [19], and NiCad [96]) and investigated the instability of three major types of clones (Type 1, Type 2, Type 3). We have also investigated the instability scenarios of different clone types in different programming languages to identify which clone types show higher instability and in which programming languages. Through our investigations on the hundreds of revisions of sixteen subject systems written in four different programming languages we answered ten research questions. The research questions and answers are shown in Table 7.1.

Table 7.1: Research Questions and Answers

	Research Question	Answers
RQ1	Which code changes more frequently, cloned or non-cloned?	Modification frequency of cloned code is generally smaller than that of non-cloned code (CLONE MORE STABLE).
RQ2	Which code exhibits higher modification probability, cloned or non-cloned?	Modification probability of cloned code is generally higher than that of non-cloned code (CLONE LESS STABLE).
RQ3	Which code changed more recently, cloned or non-cloned?	Cloned code is generally changed more lately compared to non-cloned code (CLONE LESS STABLE).
RQ4	Which code remains unchanged for greater lengths of time, cloned or non-cloned?	Cloned code generally lives longer than non-cloned code (CLONE MORE STABLE).
RQ5	Which method exhibits higher impact of changes in it, cloned or non-cloned?	Impact of cloned methods is generally higher than that of non-cloned methods (CLONE LESS STABLE).
RQ6	Which method is more likely to change, cloned or non-cloned?	Cloned methods are more likely to change compared to non-cloned methods (CLONE LESS STABLE).
RQ7	Which code in partially cloned methods exhibits higher average instability, cloned or non-cloned?	Instability of cloned methods due to their cloned portions is generally smaller than the instability of cloned methods due to their non-cloned portions (CLONE MORE STABLE).
RQ8	Which code gets more scattered changes, cloned or non-cloned?	Cloned code generally gets more scattered changes compared to non-cloned code (CLONE LESS STABLE).
RQ9	Do different types of clones exhibit different stability?	Type 1 and Type 3 clones exhibit higher instability compared to the instability of Type 2 clones.
RQ10	Do clones of different programming languages show different stability?	Clones in both Java and C systems exhibit significantly higher instability compared to the instability of the clones in C# systems.

From our findings in Table 7.1 we can say that clones are often more unstable than non-cloned code in the maintenance phase. We suggest the followings according to our findings.

(1) Programmers should be more careful while creating Type 1 and Type 3 clones so that the code fragments that are being copied do not contain any bug. Also, as many of the Type 3 clones are created from Type 1 clones, if we can eliminate (or avoid) Type 1 clones, a considerable amount of Type 3 clones will not be created and software stability risks will be reduced as well.

(2) The clones in Java and C programming languages are more unstable compared to the clones in C#. Thus, managers can impose system-wide rules before the commencement of Java or C projects indicating that the programmers should avoid clones or should remain more conscious when creating clones.

(3) Type 1 and Type 3 clones should be our primary refactoring candidates. Sometimes Type 3 clones become more difficult to be re-factored because these are created due to the independent evolutions of Type 1 and Type 2 clones. So, if we can limit Type 1 clones by applying refactoring we can avoid a significant amount of Type 3 clones.

We also found that object oriented programming languages promote more cloning compared to procedural programming languages.

As clones are generally less stable than non-cloned code in the maintenance phase, clones should be properly managed with necessary tool support. However, although clones are generally more unstable they sometimes appear to exhibit higher stability compared to non-cloned code during maintenance. To further investigate this matter we performed an in-depth study on one of our proposed metrics, change dispersion, involving a manual investigation of all the changes occurred to the clones in our candidate system Ctags [33] (developed in C) during its evolution.

According to this investigation, the presence of clones in the methods of the subject systems written in Java and C increases method instability. According to our manual analysis on the changes to the clones, about 50% of changes to the clones were made so that the clone fragments in a particular clone class remain consistent. Most of the consistency ensuring changes occurred to the Type 3 clones.

As we have found that clones are sometimes responsible for increasing system instability we further investigated the impact of clones on the co-changeability of program artifacts (such as: files, methods, classes). Co-changeability is another aspect of stability. Intuitively, higher co-changeability of program entities is an indication of higher instability of source code. We investigated the effect of clones on method co-changeability. According to our investigation, clones do not have any negative effect on method co-changeability. In other words, clones do not increase method co-changeability. Moreover, method cloning can be a possible way of minimizing method co-changeability when the cloned methods are likely to evolve independently.

Finally we see that clones have both positive and negative effects on software stability. Our empirical studies clearly demonstrate this. The findings of our studies show how we can use the positive sides of clones (e.g. minimization of method co-changeability using method clones) by minimizing their negative effects (e.g. minimization of Type 1 and Type 3 clones because they are highly unstable) for better software maintenance.

7.2 Future Work

We plan to investigate the following regarding clones.

(1) Relationship of clone instability with unintentional inconsistent changes and bug propagation: We have already observed that clones exhibit higher instability compared to non-cloned code. We would like to investigate how clone instability is related with unintentional inconsistent changes and hidden bug propagation. We also plan to investigate which type(s) of clones is (are) mainly responsible for unintentional inconsistent changes and bug propagation.

(2) Detection of refactorable candidates: All the clone fragments detected by a clone detection tool are not potential candidates for refactoring. As we have observed, for some clone fragments refactoring is even impossible. So, identification of refactorable clones can be a promising research topic. We would like to explore this topic.

(3) Development of an effort calculation model: There are many existing effort calculation models but these can not be used to calculate the efforts required for cloned and non-cloned code separately. For this reason, we plan to develop an effort calculation model which will facilitate the calculation of efforts for cloned and non-cloned code separately. We have already started working on it and had a considerable amount of progress.

(4) Investigation on the co-changeability of program artifacts: We investigated the impact of clones on method co-changeability and found that clones can possibly decrease method co-changeability. We would like to further investigate the effects of clones on the co-changeability of other program artifacts such as classes, and packages.

Bibliography

[1] Actor Architecture platform. http://www.docstoc.com/docs/5693312/Actor-Architecture

[2] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson. Near-miss Model Clone Detection for Simulink Models. In *Proceeding of the 6th International Workshop on Software Clones (IWSC'12)*, pp. 78 – 79, 2012.

[3] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling. Detecting Clones across Microsoft .NET Programming Languages. In *Proceeding of the 19th Working Conference on Reverse Engineering (WCRE'12)*, pp. 405 – 414, 2012.

[4] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison Wesley Longman, Inc, 2000.

[5] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proceeding of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pp. 81–90, 2007.

[6] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceeding of the 2nd Working Conference on Reverse Engineering (WCRE'95)*, pp. 86–95, 1995.

[7] T. Bakota, R. Ferenc, and T. Gyimóthy. Clone Smells in Software Evolution. In *Proceeding of the 23rd IEEE International Conference on Software Maintenance (ICSM'07)*, pp.24–33, 2007.

[8] L. Barbour, F. Khomh, and Y. Zou. Late Propagation in Software Clones. In *Proceeding of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pp. 273–282, 2011.

[9] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond Templates: A Study of Clones in the STL and Some General Implications. In *Proceeding of the 27th ACM/IEEE International Conference on Software Engineering (ICSE'05)* , pp. 451 – 459, 2005. ACM.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceeding of the 14th IEEE International Conference on Software Maintenance (ICSM'98)*, pp. 368–377, 1998.

[11] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proceeding of the 16th Working Conference on Reverse Engineering (WCRE'09)*, pp. 85–94, 2009.

[12] D. Beyer. Co-change visualization, In *Proceeding of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, Industrial and Tool volume, pp. 89–92, 2005.

[13] D. Beyer, and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceeding of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pp. 199–210, 2006.

[14] S. Bouktif, Y. G. Guèhèneuc, and G. Antoniol. Extracting changepatterns from cvs repositories. In *Proceeding of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pp. 221–230, 2006.

[15] F. Calefato, F. Lanubile, and T. Mallardo. Function Clone Detection in Web Applications: A Semiautomated Approach. *Journal of Web Engineering*, 3(1), pp. 3–21, 2004.

[16] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta. Using multivariate time series and association rules to detect logical change coupling: An empirical study. In *Proceeding of the 26th IEEE International Conference on Software Maintenance (ICSM'10)*, pp. 1–10, 2010.

[17] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proceeding of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 29, 2005.

[18] M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. An eclectic approach for change impact analysis. In *Proceeding of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)* , pp. 163–166, 2010.

[19] CCFinderX. `http://www.ccfinder.net/ccfinderxos.html`

[20] M. Chilowicz, E. Duris, and G. Roussel. Syntax Tree Fingerprinting for Source Code Similarity Detection. In *Proceeding of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*, pp. 243 – 247, 2009.

[21] J. Cordy. The txl source transformation language. *Science of Computer Programming Journal*, 61(3), pp. 190–210, 2006.

[22] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceeding of the 34th annual ACM symposium on Theory of computing (STOC'02)*, pp. 380 – 388, 2002.

[23] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering (TSE)*, 35(5), pp. 720–735, 2009.

[24] N. Davey, P. Barson, S. Field, R. Frank, and S. Tansley. The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, 1:219–236, 1995.

[25] I. J. Davis, and M. W. Godfrey. Clone Detection by Exploiting Assembler. In *Proceeding of the 4th International Workshop on Software Clones (IWSC'10)*, pp. 77 – 78, 2010. ACM.

[26] I. J. Davis, and M. W. Godfrey. From Whence It Came: Detecting Source Code Clones by Analyzing Assembler. In *Proceeding of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pp. 242 – 246, 2010.

[27] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model Clone Detection in Practice. In *Proceeding of the 4th International Workshop on Software Clones (IWSC'10)*, pp. 57 – 64, 2010.

[28] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceeding of the 30th ACM/IEEE International Conference on Software Engineering (ICSE'08)*, pp. 603 – 612, 2008.

[29] C. Domann, E. Juergens, and J. Streit. The Curse of Copy&Paste Cloning in Requirements Specifications. In *Proceeding of the 3rd International Symposium on Empirical Software Engineering and Measurement, (ESEM'09)*, pp. 443 – 446, 2009.

[30] E. Duala-Ekoko, and M. P. Robillard. Clone Region Descriptors: Representing and Tracking Duplication in Source Code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1), pp. 1–31, 2010.

[31] E. Duala-Ekoko, and M. P. Robillard. Tracking Code Clones in Evolving Software. In *Proceeding of the 29th ACM/IEEE International Conference on Software Engineering (ICSE'07)*, pp.158 – 167, 2007.

[32] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceeding of the 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pp. 109 – 118, 1999.

[33] Exuberant Ctags: `http://ctags.sourceforge.net/`

[34] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3), pp. 319–349, 1987.

[35] Fisher Exact Test: http://in-silico.net/tools/statistics/fisher_exact_test

[36] R. Falke, P. Frenzel, and R. Koschke. Empirical Evaluation of Clone Detection using Syntax Suffix Trees. *Empirical Software Engineering*, 13(6), pp. 601 – 643, 2008.

[37] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida. Code Clone Graph Metrics for Detecting Diffused Code Clones. In *Proceeding of the 16th Asia-Pacific Software Engineering Conference (APSEC'09)*, pp. 373 – 380 , 2009.

[38] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceeding of the 14th IEEE International Conference on Software Maintenance (ICSM'98)*, pp. 190–199, 1998.

[39] D. M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3), pp. 369–393, 2006.

[40] N. Göde, and J. Harder. Clone Stability. In *Proceeding of the 15th European Conference on Software Maintenance and Re-engineering (CSMR'11)*, pp. 65–74, 2011.

[41] N. Göde, R. Koschke. Studying Clone Evolution Using Incremental Clone Detection. *Journal of Software Maintenance and Evolution: Research and Practice*, 28 pp., 2010, In Press, (DOI: 10.1002/smr.520).

[42] A. E. Hassan and R. C. Holt, Predicting change propagation in software systems. In *Proceeding of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 284–293, 2004.

[43] Y. Higo, and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proceeding of the 15th European Conference on Software Maintenance and Re-engineering (CSMR'11)*, pp. 75 – 84 , 2011.

[44] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring Support Based on Code Clone Analysis. *Lecture Notes in Computer Science*, 3009:220–233, 2004.

[45] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental Code Clone Detection: A PDG-based Approach. In *Proceeding of the 18th Working Conference on Reverse Engineering (WCRE'11)*, pp. 3 – 12 , 2011.

[46] W. Hordijk, M. Ponisio, and R. Wieringa. Harmfulness of Code Duplication - A Structured Review of the Evidence. In *Proceeding of the 13th International Conference on Evaluation and Assessment in Software Engineering (EASE'09)*, pp. 88–97, 2009.

[47] K. Hotta, Y. Sano, Y. Higo, S. Kusumoto. Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software. In *Proceeding of the Joint ERCIM Workshop on Software Evolution (EVOL '10) and International Workshop on Principles of Software Evolution (IWPSE '10)*, pp. 73–82, 2010.

[48] F. Jaafar, Y. Gueheneuc, S. Hamel, G. Antoniol. An Exploratory Study of Macro Co-changes. In *Proceeding of the 18th Working Conference on Reverse Engineering (WCRE'11)*, pp. 32–334, 2011.

[49] S. Jarzabek , and Y. Xu. Are clones harmful for maintenance? In *Proceeding of the 4th International Workshop on Software Clones (IWSC'10)*, pp. 73-74, 2010.

[50] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *Proceeding of the 29th ACM/IEEE International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[51] J. H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceeding of the 10th IEEE International Conference on Software Maintenance (ICSM'94)*, pp. 120–126, 1994.

[52] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proceeding of the 31st ACM/IEEE International Conference on Software Engineering (ICSE'09)*, pp. 485– 495, 2009.

[53] N. Juillerat, and B. Hirsbrunner. An Algorithm for Detecting and Removing Clones in Java Code. *International Workshop on Software Evolution through Transformations*, pp. 63–74, 2006.

[54] 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 (TSE)*, 28(7), pp. 654–670, 2002.

[55] C. Kapser, and M. W. Godfrey, "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6), pp. 645–692, 2008.

[56] C. J. Kapser, and M. W. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *Journal of Software Maintenance and Evolution*, 18:61 – 82, 2006.

[57] I. Keivanloo, C. K. Roy, and J. Rilling. SeByte: A Semantic Clone Detection Tool for Intermediate Languages. In *Proceeding of the 20th IEEE International Conference on Program Comprehension (ICPC'12)*, pp. 247 – 249, 2012.

[58] I. Keivanloo, C. K. Roy, and J. Rilling. Java Bytecode Clone Detection via Relaxation on Code Fingerprint and Semantic Web Reasoning. In *Proceeding of the 6th International Workshop on Software Clones (IWSC'12)*, pp. 36 – 42, 2012.

[59] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceeding of the10th European Software Engineering Conference (ESEC'05) held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'05)*, pp. 187–196, 2005.

[60] R. Koschke. Large-Scale Inter-System Clone Detection Using Suffix Trees. In *Proceeding of the 16th European Conference on Software Maintenance and Re-engineering (CSMR'12)*, pp. 309 – 318, 2012.

[61] R. Koschke. Survey of Research on Software Clones. *Dagstuhl Seminar: Duplication, Redundancy, and Similarity in Software*, 2006.

[62] R. Koschke, R. Falke, and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceeding of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pp. 253–262, 2006.

[63] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceeding of the 14th Working Conference on Reverse Engineering (WCRE'07)*, pp. 170–178, 2007.

[64] J. Krinke. Is cloned code more stable than non-cloned code? In *Proceeding of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pp. 57–66, 2008.

[65] J. Krinke. Is Cloned Code older than Non-Cloned Code? In *Proceeding of the 5th International Workshop on Software Clones (IWSC'11)*, pp. 28–33, 2011.

[66] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceeding of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pp. 301 – 309, 2001.

[67] B. Lagüe, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceeding of the 13th IEEE International Conference on Software Maintenance (ICSM'97)*, pp. 314 – 321, 1997.

[68] H. Lee, and K. Doh. Tree-Pattern-based Duplicate Code Detection. In *Proceeding of the ACM first International Workshop on Data-intensive Software Management and Mining (DSMM'09)*, pp. 7 – 12, 2009. ACM.

[69] S. Lee, and I. Jeong. SDD: High Performance Code Clone Detection System for Large Scale Source Code. In *Proceeding of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pp. 140–141, 2005.

[70] J. Li, and M. D. Ernst. CBCD: Cloned Buggy Code Detector. University of Washington Department of Computer Science and Engineering technical report UW-CSE-11-05-02, (Seattle, WA, USA), May 2, 2011. Revised October 2011.

[71] 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 *Proceeding of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, 6:20 – 20,2004.

[72] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting Duplications in Sequence Diagrams Based on Suffix Trees. In *Proceeding of the 13th Asia Pacific Software Engineering Conference (APSEC'06)*, pp. 269 – 276, 2006.

[73] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the Harmfulness of Cloning: A Change Based Experiment. In *Proceeding of the 4th International Workshop on Mining Software Repositories (MSR'07)*, pp. 18, 2007.

[74] A. Lozano and M. Wermelinger. Tracking clones imprint. In *Proceeding of the 4th International Workshop on Software Clones (IWSC'10)*, pp. 65–72, 2010.

[75] A. Lozano, and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceeding of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, pp. 227–236, 2008.

[76] G. A. D. Lucca, M. D. Penta, and A. R. Fasolino. An Approach to Identify Duplicated Web Pages. In *Proceeding of the 26th IEEE International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMOSAC'02)*, pp. 481–486, 2002.

[77] U. Manber. Finding similar files in a large file system. In *Proceeding of the USENIX Winter 1994 Technical Conference (WTEC'94)*, pp. 1–10, 1994.

[78] Mann-Whitney U-Test calculator: `http://elegans.som.vcu.edu/~leon/stats/utest.html`

[79] Mann-Whitney U-Test details: `http://www.experiment-resources.com/mann-whitney-u-test.html`

[80] A. Marcus, and J. I. Maletic. Identification of High-Level Concept Clones in Source Code. In *Proceeding of the 16th IEEE international conference on Automated software engineering (ASE'01)*, pp. 107–114, 2001.

[81] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceeding of the 12th IEEE International Conference on Software Maintenance (ICSM'96)*, pp. 244–253, 1996.

[82] R. C. Miller and B. A. Myers. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceeding of the General Track: 2002 USENIX Annual Technical Conference*, pp. 161 – 174, 2001.

[83] M. Mondal, C. K. Roy, S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceeding of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*, pp. 1227–1234, 2012.

[84] M. Mondal, C. K. Roy, and K. A. Schneider. An Empirical Study on Clone Stability. *ACM SIGAPP Applied Computing Review (ACR)*, 12(3), pp. 20-36, 2012.

[85] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of Co-changed Method Groups: A Case Study on Open Source Systems. In *Proceeding of the 22nd Annual International Conference of the Center for Advanced Studies on Collaborative Research, IBM Canada Software Laboratory (CASCON'12)*, pp. 205 – 219, 2012.

[86] 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 *Proceeding of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*, pp. 242 – 245, 2011.

[87] M. Mondal, C. K. Roy, and K. A. Schneider. Dispersion of Changes in Cloned and Non-cloned Code. In *Proceeding of the 6th International Workshop on Software Clones (IWSC'12)*, pp. 29 – 35, 2012.

[88] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Proceeding of the 12th International Conference on Fundamental Approaches to Software Engineering (FASE '09): Held as Part of the Joint European Conferences on Theory and Practice of Software, (ETAPS'09)* , pp. 440 – 455, 2009.

[89] A. J. Offutt, M. J. Harrold, and P. Kolte. A software metric system for module coupling. *Journal of Systems and Software*, pp. 295–308, 1993.

[90] M. Page-Jones. *The practical guide to structured systems design.* YOURDON Press, New York, NY, 1980.

[91] W. Pan. Quantifying the Stability of Software Systems via Simulation in Dependency Networks. *World Academy of Science, Engineering and Technology*, Issue 60, pp. 1513 – 1520, 2011.

[92] J. Patenaude, E. Merlo, M. Dagenais, and B. Laguë. Extending Software Quality Assessment Techniques to Java Systems. In *Proceeding of the 7th International Workshop on Program Comprehension (IWPC'99)* pp. 49 – 56, 1999.

[93] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceeding of the 31st ACM/IEEE International Conference on Software Engineering (ICSE'09)*, pp. 276 – 286, 2009.

[94] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that Smell. In *Proceeding of the 7th International Workshop on Mining Software Repositories (MSR'10)*, pp. 72 - 81, 2010.

[95] Recent statistics: http://stackoverflow.com/questions/3477706/development-cost-versus-maintenance-cost

[96] C.K. Roy and J.R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceeding of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*, pp. 172–181, 2008.

[97] C. K. Roy. Detection and Analysis of Near-Miss Software Clones. In *Proceeding of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, pp.447–450, 2009.

[98] C. K. Roy, and J. R. Cordy. A mutation / injection-based automatic framework for evaluating code clone detection tools. In *Proceeding of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'09)* , pp. 157–166, 2009.

[99] C.K. Roy, and J.R. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, School of Computing, Queen's University, September 2007, 115 pp.

[100] C. K. Roy, and J. R. Cordy. An Empirical Study of Function Clones in Open Source Software. In *Proceeding of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pp. 81–90, 2008.

[101] C. K. Roy, and J. R. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3), pp.165–189, 2010.

[102] 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*, 74:470-495, 2009.

[103] A. Santone. Clone detection through process algebras and Java bytecode. In *Proceeding of the 5th International Workshop on Software Clones (IWSC'11)*, pp. 73 – 74, 2011.

[104] 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 *Proceeding of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*, pp. 87–96, 2010.

[105] Scorpio: http://sdl.ist.osaka-u.ac.jp/ higo/cgi-bin/moin.cgi/scorpio-e/

[106] 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 *Proceeding of the 17th Working Conference on Reverse Engineering (WCRE'10)*, pp.13-21, 2010.

[107] D. Shawky, and A. Ali. An Approach for Assessing Similarity Metrics Used in Metric-based Clone Detection Techniques. In *Proceeding of the 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT'10)*, 1:580 – 584, 2010.

[108] Simian-similarity analyser. `http://www.redhillconsulting.com.au/products/simian/`

[109] SourceForge. `http://sourceforge.net/`

[110] H. Störrle. Towards clone detection in UML domain models. In *Proceeding of the 4th European Conference on Software Architecture (ECSA'10)*, pp. 285 – 293, 2010.

[111] Strike A Match algorithm: `http://www.catalysoft.com/articles/StrikeAMatch.html`.

[112] A. Saebjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting Code Clones in Binary Executables. In *Proceeding of the 18th ACM International Symposium on Software Testing and Analysis (ISSTA'09)*, pp. 117 – 128, 2009.

[113] R. Tairas, and J. Gray. Phoenix-Based Clone Detection Using Suffix Trees. In *Proceeding of the 44th Annual Southeast Regional Conference (ACM-SE'06)*, pp. 679 – 684, 2006.

[114] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1), pp. 1–34, 2009.

[115] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle. On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems. In *Proceeding of the 18th Working Conference on Reverse Engineering (WCRE'11)*, pp. 13 – 22, 2011.

[116] R. Wettel, and R. Marinescu. Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments. In *Proceeding of the 7th IEEE International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'05)*, pp. 63 – 70, 2005.

[117] A. T. T. Ying, G. C. Murphy, R. Ng, M. C. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9), pp. 574–586, 2004.

[118] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü. A bayesian network based approach for change coupling prediction. In *Proceeding of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pp. 27–36, 2008.

[119] M.F. Zibran, and C.K Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proceeding of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, pp. 105 – 114, 2011.

[120] M.F. Zibran, and C.K Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach. In *Proceeding of the 19th IEEE International Conference on Program Comprehension (ICPC'11)*, pp. 266-269, 2011.

[121] M.F. Zibran, and C.K. Roy. Towards Flexible Code Clone Detection, Management, and Refactoring in IDE. In *Proceeding of the 4th International Workshop on Software Clones (IWSC'11)*, pp. 75 – 76, 2011.

[122] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller. Mining version histories to guide software changes. In *Proceeding of the 26th ACM/IEEE International Conference on Software Engineering (ICSE'04)*, pp. 563–572, 2004.