Management Aspects of Software Clone Detection and Analysis

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

By
Minhaz Fahim Zibran

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Copying a code fragment and reusing it by pasting with or without minor modifications is a common practice in software development for improved productivity. As a result, software systems often have similar segments of code, called software clones or code clones. Due to many reasons, unintentional clones may also appear in the source code without awareness of the developer. Studies report that significant fractions (5% to 50%) of the code in typical software systems are cloned. Although code cloning may increase initial productivity, it may cause fault propagation, inflate the code base and increase maintenance overhead. Thus, it is believed that code clones should be identified and carefully managed. This Ph.D. thesis contributes in clone management with techniques realized into tools and large-scale in-depth analyses of clones to inform clone management in devising effective techniques and strategies.

To support proactive clone management, we have developed a clone detector as a plug-in to the Eclipse IDE. For clone detection, we used a hybrid approach that combines the strength of both parser-based and text-based techniques. To capture clones that are similar but not exact duplicates, we adopted a novel approach that applies a suffix-tree-based k-difference hybrid algorithm, borrowed from the area of computational biology. Instead of targeting all clones from the entire code base, our tool aids clone-aware development by allowing focused search for clones of any code fragment of the developer's interest.

A good understanding on the code cloning phenomenon is a prerequisite to devise efficient clone management strategies. The second phase of the thesis includes large-scale empirical studies on the characteristics (e.g., proportion, types of similarity, change patterns) of code clones in evolving software systems. Applying statistical techniques, we also made fairly accurate forecast on the proportion of code clones in the future versions of software projects. The outcome of these studies expose useful insights into the characteristics of evolving clones and their management implications.

Upon identification of the code clones, their management often necessitates careful refactoring, which is dealt with at the third phase of the thesis. Given a large number of clones, it is difficult to optimally decide what to refactor and what not, especially when there are dependencies among clones and the objective remains the minimization of refactoring efforts and risks while maximizing benefits. In this regard, we developed a novel clone refactoring scheduler that applies a constraint programming approach. We also introduced a novel effort model for the estimation of efforts needed to refactor clones in source code.

We evaluated our clone detector, scheduler and effort model through comparative empirical studies and user studies. Finally, based on our experience and in-depth analysis of the present state of the art, we expose avenues for further research and development towards a versatile clone management system that we envision.

Acknowledgements

"Great things are not done by impulse, but by series of small things brought together..."
– Vincent van Gogh

I take much pleasure to express my profound gratitude to my Ph.D. thesis advisor Dr. Chanchal K. Roy for his persistent and inspiring supervision as well as financial support.

I also thank all the members of my thesis advisory committee, Dr. Kevin A. Schneider, Dr. Nathaniel Osgood, Dr. Nadeem Jamali, and Dr. Khan A. Wahid for their valuable advices and guidance in shaping my research towards a successful Ph.D. thesis. Special thanks to the external examiner of this thesis, Dr. Giuliano Antoniol. Thanks to the anonymous reviewers for their valuable comments and suggestions in improving the work and publications produced from this thesis. I would also like to thank Dr. James R. Cordy and Dr. Rainer Koschke who have also been my sources of valuable advices and inspiration.

Special thanks to NSERC (Natural Science and Engineering Research Council of Canada), the University of Saskatchewan, and the Walter C. Memorial Foundation for financial assistance during the progress of this thesis. I am also thankful to the ACM SIGAPP and the College of Graduate Studies and Research (CGSR) at the University of Saskatchewan for financially supporting my travels to national and international venues for presenting parts of this thesis work to the research community.

I must also thank the Schloss Dagstuhl (Leibniz Center for Informatics) for inviting me at the "Dagstuhl Seminar 12071: Software Clone Management Towards Industrial Application", and covering my conveyance and lodging during the week-long seminar in Germany. All the sessions and interactive brainstorming in the seminar were really useful in shaping this thesis, and I thank all the participants and organizers.

I am thankful to my fellow researchers, faculty members, and staff of the Department of Computer Science for their spontaneous cooperation and encouragement. I especially thank Ripon K. Saha, Muhammad Asaduzzaman, Tariq Muhammad, and Yusuke Yamamoto with whom I jointly worked on a couple of research projects. I also acknowledge the contributions of Sharif Uddin, Manishankar Mondal, Saidur Rahman, Mohammad A. Khan, Ripon K. Saha, Muhammad Asaduzzaman, and Khalid Billah who voluntarily participated in the user-centric studies of this thesis.

I am grateful to the members of my family, relatives and friends, especially to my wife Farjana Z. Eishita, my mother Pixy Naznin, and my father Mohammad Y. Ali, who did not get the share of my time and attention that they deserved. For those that I have not explicitly mentioned here, thank you for being a part of this thesis and helping me grow as a person and a researcher.

Above all, my sincere gratitude to the Almighty, who creates and makes things happen.

"Before I can have, I must do. Before I can do, I must be..."
– Jim Clemmer

I dedicate this thesis to my beloved mother, Pixy Naznin, whose selfless support has always been in each and every step of my life, and to my darling wife, Farjana Eishita, whose constant support, companionship and sacrifices made it possible to complete this work successfully.

Contents

List of Tables

List of Figures

List of Abbreviations

AG	Alive Genealogy
AI	Artificial Intelligence
ASP	Active Server Pages
CCG	Consistently Changed Genealogy
CG	Clone Genealogy
CP	Constraint Programming
CSOP	Constraint Satisfaction Optimization Problem
CSS	Cascaded Style Sheet
DG	Dead Genealogy
DP	Dynamic Programming
EM	Extract Method
ES	Extract Superclass
EU	Extract Utility-class
GA	Genetic Algorithm
IDE	Integrated Development Environment
IRS	Incident Reporting System
JSP	Java Server Pages
LIME	Language Independent Matching Engine
LCA	Lowest/Least Common Ancestor
LCE	Longest Common Extension
LOC	Source Line of Code
LP	Linear Programming
Mb	Megabit
MB	Megabyte
MILP	Mixed Integer Linear Programming
MVC	Model-View-Controller
OmeGA	Ordering Messy Genetic Algorithm
OO	Object-Oriented
OPL	Optimization Programming Language
PHP	Hypertext Preprocessor or Personal Home Page
PM	Pull-up Method
QMOOD	Quality Model for Object-Oriented Design
SD	Standard Deviation
SSG	Syntactically Similar Genealogy
TRS	Training Registration System
UPIT	Unique Percentage of Items Threshold

Chapter 1

Introduction and Motivation

"Do not go where the path may lead,
go instead where there is no path and leave a trail"
– Ralph Waldo Emerson

Copying existing code and pasting it in somewhere else followed by major or minor edits is a common practice that developers adopt to increase productivity. Such a reuse mechanism typically results in duplicate or very similar code fragments residing in the code base. Those duplicate or near-duplicate code segments are commonly known as code clones. There are many reasons why the developers intentionally perform such code cloning. Obvious reasons include reuse of existing implementation without "re-inventing the wheel". More comprehensive discussions on the reasons for code cloning can be found elsewhere [138, 233, 236]. Code clones may also appear in the code base without the awareness of the developers. Such unintentional/accidental clones may be introduced, for example, due to the use of certain design patterns, use of certain API's to accomplish similar programming tasks, or coding conventions imposed by the organization.

The reuse mechanism by code cloning offers some benefits [141, 142, 152]. For instance, cloning of existing code that is already known to be flawless, might save the developers from probable mistakes they might have made if they had to implement the same from scratch. It also saves time and effort in devising the logic and typing the corresponding textual code. Code cloning may also help in decoupling classes or components and facilitate independent evolution of similar feature implementations.

On the other end of the spectrum, code clones may also be detrimental in many cases [18, 85, 133, 141, 142, 187]. Obviously, redundant code may inflate the code base, and may increase resource requirements. This may be crucial for embedded systems and systems such as hand held devices, telecommunication switches, and small sensor systems. Moreover, cloning a code snippet that contains any unknown fault may result in propagation of that fault in all copies of the faulty fragment [133]. From the maintenance perspective, a change in one code segment may necessitate consistent changes in all clones of that fragment. Any inconsistency may introduce bugs or vulnerabilities in the system. Fowler et al. [81] recognize code clones as a serious kind of code smell.

However, during the software development process, duplication cannot be avoided at times. For example, duplication may be enforced by the limitation of the programming language's necessary mechanism to

implement an efficient generic solution of a problem at hand [141, 142]. Code generators may also generate duplicated code that the developers do not want to modify.

Previous research reports empirical evidences that a significant portion (generally 9%-17% [313]) of a typical software system consists of cloned code, and the proportion of code clones in the code base may be as low as 5% [236] and as high as even 50% [234]. Indeed, due to the negative impact of code clones in the maintenance effort, one might want to remove code clones by active refactoring, wherever feasible. However, in reality, aggressive refactoring of code clones appears not to be a very good idea [50], and not all clones are really removable through refactoring [44, 248]. Due to the dual role of code clones in the development and maintenance of software systems, as well as the pragmatic difficulty in avoiding or removing those, researchers and practitioners have agreed that code clones should be detected and managed efficiently [215, 310].

Indeed, clone management itself is a wide area of research and development. This thesis makes contribution in clone management with a new clone search technique integrated with IDE (Integrated Development Environment), in-depth analyses of code clones and their evolution to derive clone management implications, and a novel scheduler for scheduling of code clone refactorings based on cost-benefit analysis.

The remaining of this chapter is organized as follows. Section 1.1, describes the research problem addressed in this thesis. The research approach and contributions of this thesis are described in Section 1.2. Finally, in Section 1.3, presents the outline this thesis.

1.1 Problem Definition and Scope

Since the emergence of software clones as a research area in early 1990s, significant contributions over years made the field grow and become quite a mature field of research. However, the majority of the earlier work in the area were on the detection of a certain categories of code clones and analysis around only those categories of clones mainly to understand different cloning characteristics [311]. Relatively a fewer of the earlier research focused on software clones from the clone management perspective, and clone management has recently staged in the interest of the community. Although clones have negative impacts on the development and maintenance of source code, it is not always possible to avoid cloning or remove all clones from a system in practical settings due to a number of reasons [141, 142].

- The speed-up in the development process through reusing source code by cloning is an immediate and significant effect specially in tight development schedule.

- Sometimes the programming language, API libraries, and frameworks dictate the way the source code needs to be written, and thus the developers are forced to write source code having clones despite their fullest intention to avoid code clones.

- Programmers often deliberately create code clones to decouple program elements and facilitate their independent evolution. In such situations clones are rather desirable.

- Independent developers, specially in a distributed team environment, may produce similar source code while dealing with very similar problems. In such situations the developers even do not know about those *accidental* clones, which can later be revealed eventually by visual inspection or by a clone detection tool.

1.1.1 The Problem

Given that clones will exist in the source code, programmers will continue reusing source code by cloning, and that code clones have both positive and negative impacts on software development and maintenance, how can we efficiently manage clones to minimize their negative impacts while obtaining the best out of code cloning? Indeed, the problem of clone management encompasses a wide range of all process activities for the detection, analysis, and removal of code clones [86].

The difficulties in dealing with clones largely depend on their types. Exactly duplicate (except for their layouts/formatting) code fragments are *Type-1* clones. Code fragments that are very similar in their syntactic structure but varies only in the names of corresponding variables and identifiers are known as *Type-2* clones. Code fragments that exhibit similarity as of *Type-2* clones and also have other differences such as added, deleted or modified statements are called *Type-3* clones. The *Type-2* and *Type-3* clones are often referred to as *near-miss* clones [236, 242]. From an extensive literature survey [311], we found that although there have been some work towards clone management, there are still three major open sub-problems pertaining to dealing with different types of clones. This thesis addresses these sub-problems as described below:

Sub-problem 1: ***Integration with development process and dealing with Type-3 clones.***

Accurate detection of code clones is the first and fundamental step towards clone management. Till date, more than 40 clone detection tools and prototypes have been produced realizing a wide variety of techniques [236, 243]. However, from the perspective of clone management there still remains three issues. First, most of the clone detectors are developed as tool separate from IDEs. While such clone detectors have become useful in the analysis of clones, they lack the ability to provide IDE-integrated clone detection support for clone management during the software development phase. For proactive clone management, clone detection must be integrated with the development process so that the developers can deal with clone during their active interaction with the source code. Second, such tools adopt a post-mortem approach that aims to detect and report all clones from the entire source code. Typically a large number of clones are reported and it becomes difficult for a developer to focus on a certain set of clones of interest, which can be relevant to the developer's current work at hand. Third, most of the clone detectors, especially those few tools that are integrated with IDEs, exhibit limitations in detecting *Type-3* clones [236, 243].

Sub-problem 2: ***Understanding clones and their evolution.***

Integrated clone detection constitutes the merely the first step towards clone management. To devise effective techniques, tools, and strategies for clone management, we must understand the existence and evolution of clones. The significance of the clone management problem must be justified by knowing what proportion of source code are typically cloned. A deeper understanding on how the proportion of clones changes during the evolution of the software system is necessary to suggest the stages of the software lifecycle when it is the most appropriate to deal with clones. Moreover, investigation of the types of changes the individual clone fragments experience during their evolution is a must to determine what kinds of modification and refactoring support are required for effective clone management. However, a little is known about these phenomena, especially for the *near-miss* clones including *Type-3*.

Sub-problem 3: ***Ranking of clones for refactoring.***

Once the clones are detected, they can be better managed when the proportion of cloned code in the source code is kept to the minimum. The number of clones can be minimized by refactoring source code for extracting a generalized code fragment from multiple duplicate or near-duplicate code fragments. Among many general refactoring patterns [80, 81] the *extract method* refactoring is probably the most promising and popular technique for clone refactoring [90]. However, refactoring in general is often error-prone and not an easy task. Refactoring needs effort and care. Clone refactoring must deal with additional complexities due to the dispersion of the clones in different parts of the source code with respect to the file system hierarchy as well as inheritance hierarchy, and that there remains dependencies and conflicts among refactoring of the individual clone fragments, which must be resolved in the first place.

Typically, a large number of clones are found in the source code. Despite the developers' adoption of proactive clone management, a significant number of clones are likely to exist due to that lack of clone management ability of the individual programmers, coordination gap in team environment, and the like. Moreover, integrated clone management cannot be applicable for many legacy systems. The number of clones often proportionally increases with the increase in the size of the code base [313]. For clone refactoring, the developer must examine those reported clones to short-list a subset of feasible candidates for refactoring. In practice, all the reported clones do not appear to be feasible for removal by refactoring and there remain some deliberate clones that must be kept intact due to design choices. The cost-benefit rationale also plays a vital role in deciding which clones to refactor especially when the project runs in a limited budget. Therefore, prior to performing clone refactoring, one must take into account all the conflicts, dependencies, and priorities to produce a schedule of candidate clones for refactoring such that the cost of refactoring is minimized while the expected benefits are maximized.

In this context, there remain two difficulties. First, such a refactoring scheduling problem is known to be *NP-hard* [37, 176, 183], and thus, manual effort cannot be expected to perform well especially for medium to large software systems. Therefore, an automated (or semi-automated) clone refactoring scheduler that

takes into account all the conflict, dependencies, and criteria is necessary, which is missing in the literature. Second, there exists no established procedure for upfront estimation and quantification of the costs and expected benefits involved in code clone refactoring.

1.1.2 Scope

Software clones can be detrimental if not properly managed, and *the objective of this dissertation research is to help clone management by devising techniques realized into prototype tools and by revealing clone management implications from in-depth empirical studies performed in large scale.*

This thesis contributes to clone management research for effective and informed management of *Type-1*, *Type-2*, and *Type-3* code clones at the granularities of *functions* and *blocks*. Code clones at these granularities can be more meaningful with respect to reengineering and refactoring opportunities as further discussed in Chapter 7 (Section 7.2.1). The work described in this thesis deals with source code written in imperative programming languages such as Java, C, C++, and C#, as well as scripting code written in PHP. These imperative programming languages are widely used in general purpose software development, and PHP is a commonly used in the development of dynamic web applications. Although developed around the aforementioned programming languages, the techniques and findings from this dissertation research can also be made applicable for other languages and paradigms with minimal efforts.

Dealing with *Type-4* (semantically similar) code clones and clones in software artifacts other than the source code remains beyond the scope of the work presented in this thesis.

1.2 Research and Contributions

This thesis contributes towards clone management, in particular, we address the aforementioned three sub-problems through in-depth analysis and development.

Addressing Sub-problem 1: Integrated Detection of Exact and Near-miss Clones

To resolve the sub-problem 1, we develop a clone search tool that enables on-demand search for exact and *near-miss* clones including *Type-3*. The tool is developed as a plug-in to the popular Eclipse IDE (Integrated Development Environment), and thus clone detection is integrated with the development process. Instead of detecting and reporting all clones from the source code, the tool enables the developer to find clones of a particular code fragment. Thus, the number of reported clones is significantly reduced, which allows the developer to focus on a particular set of clones he or she is interested during the software development process. Further details about our algorithm for clone detection and approach for evaluation of the tool are described in Chapter 3.

Addressing Sub-problem 2: Clone Analysis

To address the sub-problem 2, we carry out a number of exploratory empirical studies. First, we conduct a large scale empirical study on a wide range of open-source application software systems written in different programming languages. We analyze the overall proportion of exact (*Type-1*) and *near-miss (Type-2 and Type-3)* clones in the series of releases of those software systems. In particular, we examine whether there is any impact of programming language or paradigm, program sizes on the proportion and evolution of code clones in software systems. We also investigate for any patterns in the proportion of code clones over the releases of the software system. The motivation and methodology of the study is further elaborated in Chapter 4. One of the findings from the study suggests significant impact of programming languages and paradigms on the existence of code clones in software systems. Beside studying clones in open-source systems, we also study the patterns of cloning in two industrial web applications.

From the first study, we also find some clone-groups that remain intact and propagate over a long series of releases of the evolving software systems. This finding inspired us to carry out a *third* empirical study (Section 4.3), where we include a deeper investigation on the evolution of individual clones. As we examine the evolution of individual clones, we again find long-lived clones that propagate across many releases of the system while many of the clones disappear in a few early releases.

Then, it becomes necessary to study the evolution of individual clone fragments and clone-groups to understand what type of changes those clones experience as they evolve across software releases, and what type of changes they undergo before their removal. Thus, we carry out a *fourth* empirical study to investigate the type of changes the individual clones experience as they evolve. The practical findings of the study suggest what type of clones can initially be considered to be attractive for refactoring and what type of tool support can be helpful for managing code clones. Further details of this empirical study is described in Chapter 5.

Addressing Sub-problem 3: Clone Refactoring

While a deeper understanding on the characteristics of clones and their changes can help point to interesting clones from management perspective, the problem of scheduling clone refactoring (sub-problem 3) still plays its role in practical settings where a large number of clones to need to be dealt with in limited budget (i.e., time, effort). Addressing sub-problem 3, we propose the first *effort model* for the estimation and quantification of the effort required to refactor code clones in object-oriented source code. For the efficient scheduling of code clone refactorings, we also develop a novel scheduler that takes into account the refactoring effort, effects, all possible conflicts and dependencies among the candidate clones. More details about the effort model, the clone refactoring scheduler, and their evaluation is discussed in Chapter 6.

1.3 Outline of the Thesis

The research presented in this thesis is composed of three major parts. In the first part, we develop an IDE-based clone search tool as a plug-in to the Eclipse IDE to support integration of clone detection with the actual development process. The second part of the research includes in-depth empirical studies on the characteristics of the existence and evolution of code clones in software systems. The empirical studies derive insights into the code clones and their maintenance implications to inform clone management. Driven by these implications, in the third phase of the dissertation research, we develop a parameterized effort model for the estimation of code clone refactoring effort, and then introduce a novel scheduler for conflict aware optimal scheduling of prioritized code clone refactorings.

In this chapter (Chapter 1), we introduced the research problem along with our motivation, research methodology, and expected contributions from this thesis. The remaining of this thesis is organized as follows.

- *Chapter 2* presents the terminology and concepts that constitute the foundation necessary to follow the remaining of the thesis.
- In *Chapter 3*, we present our IDE-integrated clone search tool and its evaluation.
- *Chapter 4* includes empirical studies on the existence and evolution of code clones in diverse software systems.
- An in-depth study on the evolution individual clones and their change patterns is presented in *Chapter 5*.
- In *Chapter 6*, we present a novel effort model for the estimation of clone refactoring efforts. We also present an automated approach for efficient scheduling of clone refactoring on the basis of cost-benefit analysis.
- Based on an in-depth survey [311] and our experience in the area, in *Chapter 7*, we present our vision for a versatile clone management system and expose avenues for further research and contributions.
- Finally, *Chapter 8* concludes this thesis.

Major parts of this thesis are already published in peer reviewed journals and international conferences. A list of publications that are outcomes of this dissertation research are presented in Appendix A. A User's Manual for the IDE-integrated clone search tool (as described in Chapter 3) is included in Appendix B. The OPL model of the clone refactoring scheduler (as described in Chapter 6) is presented in Appendix C.

Chapter 2

Background

"There is something fascinating about science.
One gets such wholesale returns of conjecture
out of such a trifling investment of fact"
– Mark Twain

In this chapter, we introduce the terminology and concepts necessary to follow the remaining of this thesis. We begin with the definition of code clones in Section 2.1. Section 2.2 and Section 2.3 respectively describes the patterns of cloning and negative impacts of clones. In Section 2.4, we clarify what is meant by clone management. Section 2.5 briefly describes different clone detection techniques along with their strengths and weaknesses. In Section 2.6, we introduce the clone genealogy model, which is used in some recent studies (including ours) on code clone evolution. Finally, Section 2.7 summarizes the chapter.

2.1 Definition of Code Clone

Duplicate or similar code fragments are roughly known to be code clones. Over more than a decade of research on code clones, the community [156, 162, 233, 236, 240, 249, 279, 313, 310] has accepted the following categorizing definitions of code clones.

Type-1 Clone: Identical code fragments except for variations in white-spaces and comments are *Type-1* clones. For example, the code fragments in Listing 2.1 and Listing 2.2 are *Type-1* clones despite the differences in their layouts and comments.

Type-2 Clone: Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments are called *Type-2* clones. For example, the code fragment in Listing 2.3 is a *Type-2* clone of the code fragments in Listing 2.1 and Listing 2.2.

Type-3 Clone: Code fragments that exhibit similarity as of *Type-2* clones and also allow further differences such as additions, deletions or modifications statements are known as *Type-3* clones. For example, the code fragment in Listing 2.4 is a *Type-3* clone of the code fragments in Listing 2.1, Listing 2.2, and Listing 2.3.

Type-4 Clone: Code fragments that exhibit identical functional behaviour but implemented through very different syntactic structure are known as *Type-4* clones. For example, the Java code fragment in Listing 2.5 implements the Bubble Sort algorithm using recursion, while the code fragment in Listing 2.6 implements the same algorithm using loops (iterations) only. These two code fragments are semantic clones since they are functionally identical (i.e., implements the Bubble Sort algorithm) but the structures of the program source code are very different.

Listing 2.1: Code fragment-1

```
final private boolean jj_3R_164()
{ // brace in separate line
   if (jj_3R_166())
      return true;
   Token xsp;
   while (true) {
      xsp = jj_scanpos;
      if (jj_3R_171()) {
          jj_scanpos = xsp;
          break;
      }
   }
   return false;
}
```

Listing 2.2: Code fragment-2

```
final private boolean jj_3R_164() {
   if (jj_3R_166())
      return true;
   Token xsp;
   while (true)
   { // brace in separate line
      xsp = jj_scanpos;
      if (jj_3R_171()) {
          jj_scanpos = xsp;
          break;
      }
   }
   return false;
}
```

Listing 2.3: Code fragment-3

```
final private boolean jj_3R_168() {
   if (jj_3R_170())
      return true;
   Token xsp;
   while (true) {
      xsp = jj_scanpos;
      if (jj_3R_182())
      { // brace in separate line
          jj_scanpos = xsp;
          break;
      }
   }
   return false;
}
```

Listing 2.4: Code fragment-4

```
final private boolean jj_3R_164() {
   if (jj_3R_166())
      return true;
   Token xsp;
   while (true) {
      xsp = jj_scanpos;
      if (jj_3R_171()) {
         jj_scanpos = xsp;
         break;
      }
      doProcess(); // added line
   }
   return false;
}
```

Listing 2.5: Recursive Bubble-sort

```
1  public int[] recursiveBubble(int[] args,
2              int startIndex, int endIndex) {
3      if(startIndex > endIndex){
4          System.out.println(Arrays.toString(args));
5          return args;
6      }
7      if (startIndex == endIndex - 1) {
8          recursiveBubble(args, 0, endIndex - 1);
9      } else if (args[startIndex] >
10                         args[startIndex+1]) {
11         int currentNumber = args[startIndex];
12         args[startIndex] = args[startIndex + 1];
13         args[startIndex + 1] = currentNumber;
14         recursiveBubble(args,
15                     startIndex + 1, endIndex);
16     } else {
17         recursiveBubble(args,
18                     startIndex + 1, endIndex);
19     }
20     return args;
21 }
```

Listing 2.6: Bubble-sort using loop

```
1  public int[] bubblesort(int[] args) {
2      System.out.println("Normal BubbleSort:");
3      for (int j = 0; j < args.length; j++) {
4          for (int i = 0; i < args.length; i++) {
5              int currentNumber = args[i];
6              if (i + 1 < args.length) {
7                  if (currentNumber > args[i + 1]) {
8                      args[i] = args[i + 1];
9                      args[i + 1] = currentNumber;
10                 }
11             }
12         }
13     }
14     System.out.println(Arrays.toString(args));
15     return args;
16 }
```

As described, the first three types of clones are defined based on the similarity in the program text, while *Type-4* clone is defined based on semantic similarity. Thus, *Type-4* clones are also called *semantic clones*. One the other hand, *Type-1* clones are also known as *exact clones*, *Type-2* clones are also sometimes called *renamed clones*, whereas, the *Type-2* and *Type-3* clones jointly are called *near-miss* clones [236]. The terms *parameterized clone* or *p-match clone* are often used in the community to refer to a subset of *Type-2* clones, where there must be a bijective mapping between the identifiers of the two *Type-2* clones [162]. Thus, renaming (arbitrary or systematic) of identifiers are allowed in *Type-2* clones, whereas for parameterized clones, those renaming must be consistent/systemic [236] conforming the required bijective mapping.

For *Type-3* clones, the deletion of statement from one code segment can be considered as an addition of the statement in the other. The difference in the additional or changed statements in the *Type-3* clones are often called the *gaps* [282], and thus *Type-3* clones are sometimes also referred to as *gapped clones* [162, 236, 282].

2.1.1 Clone Relationship

A clone relationship exists between two code segments that are clones to each other according to specification of similarity as prescribed by the definitions stated above. Such a clone relationship is *reflexive* (i.e., if code

segment A is a clone of B, then B is also a clone of A). Moreover, for *Type-1* and *Type-2* clones, the *transitive* relationship also exists (i.e., if code segment A is a clone of B, and B is a clone of C, then A is also a clone of C). However, such a transitive property may not hold for *Type-3* clones [34]. The aforementioned definitions also imply that a subset relationship exists among the *Type-1*, *Type-2*, and *Type-3* clones. Mathematically, Type-$i \subseteq$ Type-j, for $i \in \{1, 2\}$ and $j = i + 1$ [156].

Two code segments that are clones to each other (i.e., have clone relationship between them) are called a *clone pair*. A *clone-class* or *clone-group* is a set of code segments such that any two of them are clone pairs. The code fragments in Listing 2.1, Listing 2.2, Listing 2.3, and Listing 2.4 form a clone-group of four members[1].

2.1.2 Clone Granularity

The definitions of all four types of clones are based on the notion of code segment, and contiguous portion of code at different levels of granularity have been used in the literature. As concerned with source code, the most commonly used granularities are at the level of the entire source file, class definition, method body, code block, and statements, which yield the the notion of code clones of the following five types:

File clone: When two files are found to have contained similar enough source code, they are called file clones.

Class clone: Two classes of object-oriented source code can be considered as class clones if they have identical or near-identical code.

Function clone: Two functions are considered as clones when the bodies of the functions consist of code that are similar enough.

Block clone: When two blocks of code (marked with opening and closing braces or indentation, or the like) are similar enough, they are called block clones. To consider clones at the granularity of blocks, one must decide how to deal with nested blocks (i.e., blocks inside another block).

Arbitrary statements clone: When two groups of statements at arbitrary regions of the source file are found to be similar enough, they are also regarded as clones (`CCFinder` detects such clones).

2.1.3 Intentional and Accidental Clones

Code clones in a software system can appear in two ways. First, the programmer copies existing code, pastes it in another place, and thus reuses the implementation with or without further modifications. The resulting code may still remain similar to the original and form a clone-pair. Such clones created by the programmer's deliberate copy-paste-modification activities are known as *intentional clones* or *copy-pasted* clones [86, 117].

[1]The code fragments in Listing 2.1, Listing 2.2, Listing 2.3, and Listing 2.4 are extracted from the source code of an open-source project `JEdit` version 4.3-pre6.

There are a number of reasons why programmers create such intentional clones [162, 236]. An obvious reason is to reuse existing implementation without "re-inventing the wheel".

Indeed, similar code segments may also appear in the system without the programmer's intention, and often the developer can even be unaware of the creation of such clones [141, 142, 304]. For example, due to the developer's mental model, frequently used idioms are reproduced from memory instead of deliberate copy-paste [233]. Even different developers may also produce very similar code while solving a similar problem, or due to the dictation of certain APIs they use, or using the same design pattern [308]. Such similar code snippets that are not produced by deliberate copy-paste operations are known as *unintentional clones* or *accidental clones*.

In reality, there are many reasons behind the creation of intentional and accidental code clones in a software system. Further details on the root causes of cloning can be found elsewhere [141, 142, 162, 236, 304].

2.2 Patterns of Cloning

The *patterns of cloning* characterize the creation and distribution clones in the source code. *Forking* and *Templating* are two prominent patterns of cloning as described by Kapser and Godfrey [142]. In the process of *Forking*, a piece of existing code is cloned to a different location, which is expected to *evolve independently* from its original source. In *Templating*, similar behaviour is implemented by cloning existing code, which are then expected to *evolve together* with any future changes.

2.3 Negative Impacts of Code Clones

As mentioned before, the practice of code cloning offers reuse of source code and thus speed-up development process. Despite these immediate benefits, code clones may also have negative impacts in the long run on the maintenance of the software systems. If code clones are not carefully managed, they can introduce bugs in the system and can also cause propagation of bugs across different portions of the source code. Consider the example in Figure 2.1, where the developer produces the lines 10 through 19 by copying the preceding block (i.e., lines 1 through 9) and making necessary changes. However, the developer forgets to replace the variable `toExecMetricPrepSt` by `toExecRegMetricPrepSt` at line 17. Consequently, the statement at line 27 will never execute. Thus cloning may introduce new bugs in the system, when the developer fails to carefully manage code clones.

Cloning an existing piece of code that already had an unknown bug can cause the bug propagate to all copies of the buggy code. Consider the example in Figure 2.2, where the code fragment(1) performs file I/O (i.e., write) operations, but the file writer was never closed upon completion of the operations. A `writer.close()` statement can be expected at line 7, which is missing. When this code segment is copied to other places, the vulnerability existing in the original code is also replicated to all the copies, i.e., fragment(2), fragment(3), and fragment(4) in the figure.

```
1  if (tableName.equalsIgnoreCase(MetricTable.name())){
2      metricPrepSt.setInt(1, record.getId());
3      metricPrepSt.setFloat(2, record.getFreq());
4      metricPrepSt.setBoolean(3, record.getStatus());
5      metricPrepSt.setString(4, record.getDescription());
6      metricPrepSt.setDate(5, record.getDate());
7      metricPrepSt.setTimestamp(6, record.getTimestamp());
8      toExecMetricPrepSt = true;
9  }
10 else if (tableName.equalsIgnoreCase(RegMetricTable.name())){
11     regMetricPrepSt.setInt(1, record.getId());
12     regMetricPrepSt.setFloat(2, record.getFreq());
13     regMetricPrepSt.setBoolean(3, record.getStatus());
14     regMetricPrepSt.setString(4, record.getDescription());
15     regMetricPrepSt.setDate(5, record.getDate());
16     regMetricPrepSt.setTimestamp(6, record.getTimestamp());
17     toExecMetricPrepSt = true;
18     // should have been: toExecRegMetricPrepSt = true;
19 }
20 else {
21     logger.info("Invalid target table");
22 }
23 if(toExecMetricPrepSt){
24     metricPrepSt.executeBatch();
25 }
26 if(toExecRegMetricPrepSt){
27     regMetricPrepSt.executeBatch();
28 }
```

Figure 2.1: Bug creation through code cloning

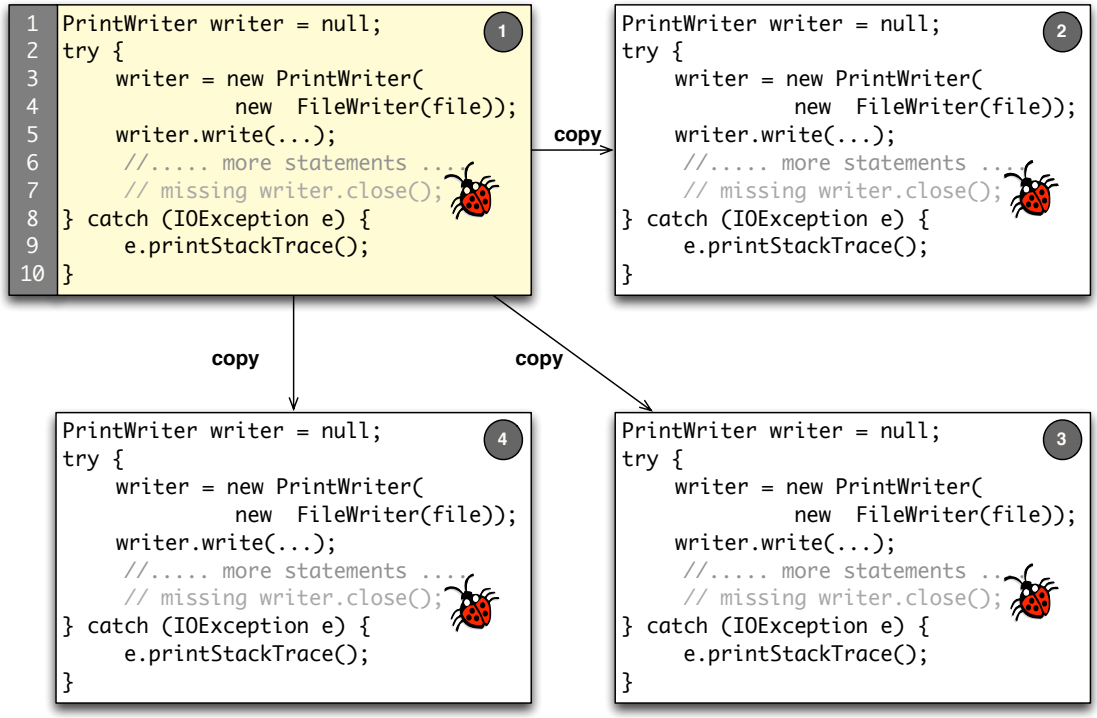


Figure 2.2: Bug propagation through code cloning

Such defects introduced in the source code by code cloning can be avoided or captured and repaired by applying clone management techniques. For example, when a bug is found in a certain piece of code, clone detection technique can be used to identify similar code fragments for detecting possible similar bugs hiding at different location of the source code.

2.3.1 A Real World Incident

In the early morning of October 12th, 2007, a catastrophic incident occurred in 662 railway stations in Tokyo, Japan [299]. As reported, more than 4,000 automatic gate machines at the railway stations failed to start-up correctly due to a defect in the source code of the software that controlled the boot-up process of the gate machines. The program vendor took more than half a day to locate and fix the single line defect in the source code. The failure affected around 2.6 million passengers.

Six days later, a similar start-up failure occurred in more than 100 fair adjustment machines in Tokyo's 65 railway stations, and caused inconveniences to about 400 passengers. The software controllers for the gate machines and fair adjusted machines were developed by the same vendor, and shared a similar source code to handle the similar boot-up processes. Immediately after fixing the software defect to recover from the first incident, the vendor also examined their other programs pertinent to automatic gate machines in search for any possible similar defects. Nevertheless, the second incident took place.

It was discovered that the software implementations for the similar boot-up processes had similar source code but used a different data format, which caused the failures occur in different days [299]. It is likely that the source code for one controller was developed by 'cloning' from the other followed by necessary modifications including those to accommodate the differences in the data format, and certain required changes somehow escaped leaving defects in the programs. Had the vendor detected the similar defects in the similar source code, at least the second incident could have been avoided.

2.4 Clone Management

"Clone management summarizes all process activities which are targeted at detecting, avoiding or removing clones" [86]. Thus, clone management encompasses a broad categories of activities having clone detection as the first step. Another major activity for clone management is the refactoring of code clones that aims to minimize the number of clones in the source code, thus keeping the negative impacts (e.g., code inflation) of clones in control. A deeper understanding the characteristics of clones and their implications can help identifying clones of interest and techniques to deal with them. Thus, clone analysis from the maintenance perspective also plays a vital role in clone management.

The primary objective for clone management includes minimization of the negative effects of clones while maximizing the benefits out of code cloning. Aside from the main purpose, there are some additional business values of clone management. The techniques, tools and information developed for clone management can be

applied for origin analysis (code provenance) [96], version merging [96], bug localization [117, 123, 181, 182], plagiarism (code copyright infringement) detection [210, 225], malware detection [42, 287], aspect mining [40, 41], library candidate extraction [108], cross-version analysis and version/branch merging in product line engineering [96].

2.5 Clone Detection

Clone detection is the most important and integral part of clone management. Clones from the source code must be identified first before they can be dealt with. Over more than a decade of code clone research a number of techniques have been devised for the detection of code clones and many clone detection tools have been developed. Prominent techniques can be categorized as shown in Table 2.1. The categorization is based on the type of information used in the analysis/comparison and the type of approach used for the analysis. The first technique (Tracking Clipboard Operations) in the table is an action-trace based technique and the rest are code similarity based techniques. In this section, we provide a brief summary of different clone detection techniques, and point out the strengths and weakness of those techniques in general. More detailed descriptions of those techniques can be found in the corresponding papers and elsewhere [236].

Tracking Clipboard Operations: This technique of clone detection is based on the assumption that programmers' copy-paste activities are the primary reason for the creation of code clones. So, the technique simply tracks clipboard activities in the editor (inside IDEs such as Eclipse) when a programmer copies a code segment and reuses by pasting it. The copied and the pasted code segments are recorded as clone-pairs.

Metrics Comparison: Metrics based techniques are usually used to detect function clones. The techniques are based on the assumption that similar code fragments should yield very similar values for different software metrics (e.g., cyclomatic complexity, fan-in, fan-out). Typically, for the code segments a set of metrics are gathered into vectors. The differences in the vectors are calculated, where close vectors (e.g., measured by Euclidean distance) indicate that their corresponding code fragments are clones.

Texual Comparison: Text based techniques compare program text, typically line by line, with or without normalizing the text by renaming the identifiers, filtering out the comments and differences in the layout.

Token Based Comparison: The entire program is transformed into a stream of tokens (i.e., individual units/words of meaning) through lexical analysis. Then the token stream is scanned to find similar token subsequences, and the original code portions corresponding to those subsequences are reported as clones.

Syntax Comparison: Syntax comparison based techniques are developed on the fact that similar code segments should also have similar syntactic structure. Thus, the program is parsed to produce syntax tree, where similar subtrees indicate that their corresponding code segments are clones.

PDG Based Comparison: For a given program, a set of PDGs (Program Dependency Graphs) [77] are produced based on the data and control dependencies among the statements of the program. The code segments corresponding to the isomorphic subgraphs are identified and reported as clones.

Table 2.1: Code Clone Detection Techniques (extended from [163])

Tracking Clipboard Operations	
copy-pasted code segments are recorded as clones	[48, 62, 114, 292]
Metrics Comparison	
comparing metrics for functions	[1, 160, 161, 172, 197]
comparing metrics for web sites	[66, 174]
Textual Comparison	
hashing of strings per line, then textual comparison	[126, 127]
hashing of strings per line, then visual comparison using dotplots	[70]
latent semantic indexing for identifiers and comments	[195]
code normalization, then textual comparison between lines	[238]
code normalization and comparison of fingerprinted lines using suffix tree	[310]
code normalization and comparison of fingerprinted statements	[212]
code normalization and hash based comparison of functions/blocks	[279]
Token-based (Lexical) Comparison	
suffix trees for tokens per line	[14, 15, 16]
token normalizations, then suffix tree/array based search	[27, 137, 145]
data mining for frequent token sequences	[182]
code block comparison based on ordered terms using MapReduce algorithm	[253, 254]
hash-based filtering followed by suffix-tree-based comparison	[164, 165]
token count based vector comparison using cosine similarity	[301, 302]
hashing of token sequences	[276]
Syntax Comparison	
hashing of syntax trees and tree comparison	[30, 298, 12]
coalescing all roots of isomorphic subtrees of abstract syntax tree	[273]
data mining for frequent syntax subtrees	[285]
serialization of syntax trees and suffix-tree detection	[74, 166, 179]
derivation of syntax patterns and pattern matching	[73]
metrics for syntax trees and metric vector comparison	[122, 215]
PDG Based Comparison	
approximate search for similar subgraphs in PDGs	[84, 107, 111, 157, 167]
Comparison of Low Level Form of Source Code	
comparing Java bytecode	[17, 56, 136, 146, 147, 255]
comparison of compiled (assembler) code	[60, 61, 76, 246]
comparing .NET CIL (Common Intermediate Language)	[5]

Comparison of Low Level Form of Code: Instead of analyzing and comparing textual source code, the techniques analyze the lower level code (e.g., assembly code, Java bytecode) as obtained from the transformation by the compiler.

Other Techniques: Beside the aforementioned prominent techniques for clone detection, other techniques, such as concolic analysis [171], anti-unification [43, 179], formal methods [255], and combination of distinct techniques [83, 155, 178] in hierarchical manner were also approached. Tracing of abstract memory states during the execution of the program was also attempted to detect semantic clones [151]. More details about software clone detection techniques and their comparison can be found in a recent survey of Rattan et al. [232] and elsewhere [34, 35, 236, 239, 243]

2.5.1 Strengths and Weaknesses of Clone Detection Techniques

Due to the orthogonal nature of the different approaches for clone detection, it is inappropriate rate one technique over another. Each of the techniques have their strengths and weaknesses. In this section, we briefly discuss the strengths and weaknesses of different clone detection techniques. More comprehensive qualitative evaluation of the existing clone detection techniques and tools can be found elsewhere [239, 243].

Clone detection using *clipboard operations (copy-paste)* has the benefit that clones are captured at the time of their creation. However, the technique poses a number of questions and limitations towards pragmatic design decisions to be made for realizing the technique in a tool:

- How much (granularity) of copy-pasted code should be recorded as clones?
- Should the copy-pasted code that spans beyond a syntactic block be considered as clone?
- Copy-pasted code may later be modified and become very different from original. Should the very dissimilar code segments still be treated as clones (as done in `CloneScape` [48]), or some similarity based decisions have to be made?
- Since the tracking of copy-paste activity is coupled with a certain editor, the technique is vulnerable to changes in the source code outside the editor.
- The technique cannot deal with unintentional/accidental clones, or clones in legacy systems.

The effectiveness of *metrics based* techniques depends on the selection of a broad set of orthogonal significant metrics. Typically, metrics based techniques may be computationally expensive for the overhead of calculating the chosen metrics. Moreover, due to the fact that different code segments may, at times, have the same value for certain metrics, the metric based techniques may report many false positives.

The advantage of simple *textual comparison* is that the technique, in general, is independent of programming languages and the program does not need to be complete or syntactically correct. However, the technique (without advanced normalization and filtering of differences in layouts) may susceptible to even minor changes in the source code [163]. Due to the use of lexical analysis only, the *token based* techniques can

operate by simple textual string comparisons and may detect code clones that span beyond the boundaries of syntactic blocks unless such clones are filtered out using an additional mechanism.

The techniques based on *syntax comparison* suffer from the overhead of invoking a parser to generate parse tree. Due to the use of a parser, the technique is coupled with programming language syntax. The parser based syntax comparison techniques are very accurate in detecting clones that have similar syntactic structure. Still, such techniques are vulnerable to even subtle differences in the nesting of code, and thus may fail to detect many potential clones. Earlier empirical evaluations [13, 34, 35] also suggest low recall of syntax comparison based clone detection techniques.

PDG based techniques analyze the source code at a level higher than that of syntax comparison based techniques, and thus can capture the program semantics to some extent. PDG based techniques can detect clones consisting of non-contiguous code. However, producing PDGs is computationally expensive. In addition, finding isomorphic subgraphs from PDGs is an NP-hard problem [163]. Therefore, approximation algorithms are used, which do not aim for the optimum. The PDG based approach is tolerant to reordering of certain program elements, which is both a strength and weakness of the technique. The ordering of statements specify the execution path of a program written in traditional imperative language, but the PDG based techniques often disregard such order and reports clones composed of *non-contiguous* code segments, and thus there remains quite good chance for false positives in detecting clones of practical significance.

Clone detection in compiled code (e.g., assembly code, binary executable, Java bytecode) requires the source code to be complete and syntactically correct. Moreover, compilation generates the lower level code that often normalizes syntactic variants of source code into a compact canonical representation free from comments and differences in the layouts as in the source code. This, in principle, should make it easier to capture semantically equivalent constructs that might "look different" in the source code written in a high level language [60]. On the contrary, small difference in the source code may produce very different sequences of bytes in the compiled code, and thus finding code clones based on byte code similarities may be even more difficult than finding clones in source code [17]. Moreover, clone detection based on the analysis of compiled code is typically performed at class level, because distinguishing the portion corresponding functions, blocks or other level of granularity and mapping those back to locations in the higher level source code can be very challenging.

The purpose of clone detection techniques is not limited to only clone management for improved software maintenance. The variable strengths of different clone detection techniques lead to their applications in solving diverse research problems such as plagiarism detection [39, 103], program fault localization [33, 46, 120, 123, 134], malware detection [42, 287], finding crosscutting concern code [41], aspect mining [148, 40], quality assessment of requirements specification [131], and more. Indeed, the choice of an appropriate clone detection technique largely depends on the context and the purpose [163].

2.6 Clone Genealogy

For in-depth investigation of the evolution of individual clone fragments, Kim et al. [153, 154] first coined the term *"clone genealogy"*, which refers to a set of one or more lineage(s) originating from the same clone-group. A *clone lineage* is a sequence of clone-groups evolving over a series of versions of the software system. The versions may be official releases of the software system, or revisions based on individual commit transactions or periodic (e.g., weekly, monthly) snapshots of the underlying code base. Thus, a clone genealogy (Figure 2.3) describes the evolution history of a particular clone-group over subsequent versions of the system. The extraction and study of clone genealogies from a series of versions of a program has been identified as an important approach to investigate the evolution of individual clone-groups.

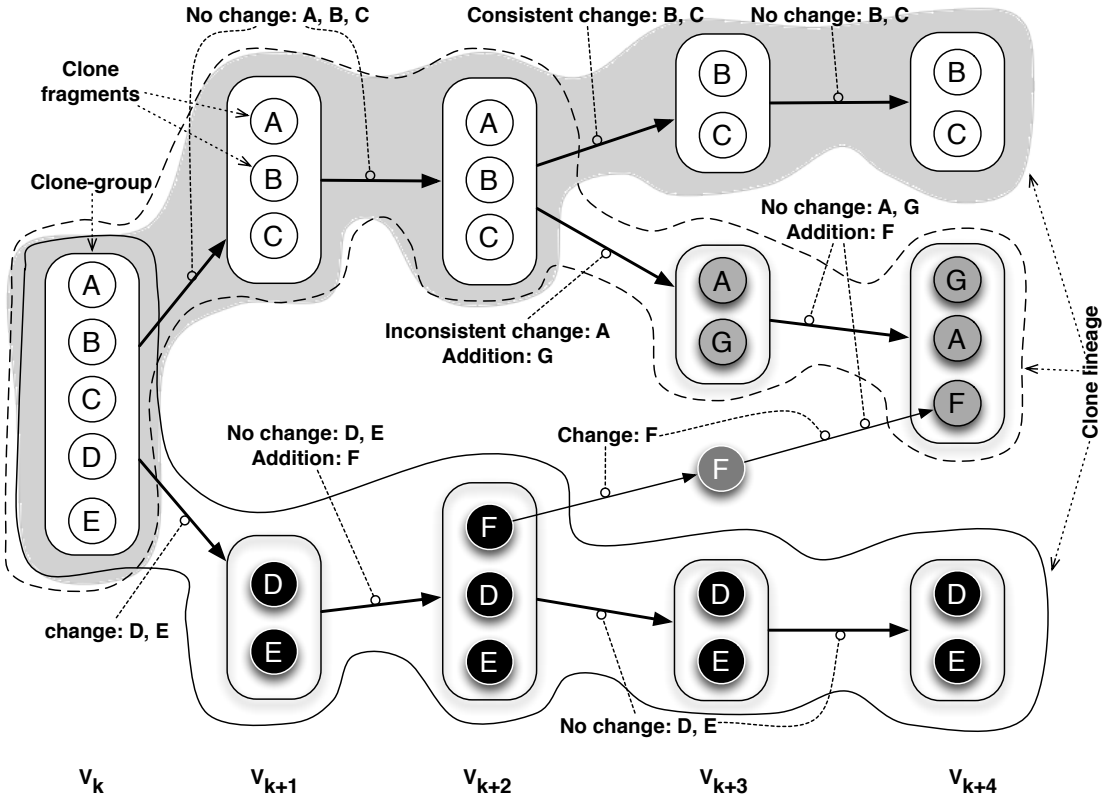


Figure 2.3: A clone genealogy with three lineages over versions V_k through V_{k+4}

2.6.1 Clone Genealogy Extraction

To map clones across subsequent versions of a program (i.e. extraction of clone genealogies) mainly four different approaches have been found in the literature, as summarized in Table 2.2. While most of these approaches [11, 18, 23, 154, 248] focused on genealogies of *Type-1* and *Type-2* clones, `gCad` [247, 249] is the only *Type-3* clone genealogy extractor to date released as a separate tool.

Table 2.2: Summary of techniques for clone genealogy extraction

Approach	Strength	Weakness	Citation
Separate clone detection from each version, and then similarity based heuristic mapping of clones in pairs of subsequent versions	flexible	quadratic runtime [98], susceptible to large change in clone	[18, 154, 248]
Clones detected from the first version are mapped to consecutive versions based on change logs obtained from source code repositories	faster than the above technique	can miss the clones introduced after the first version [247]	[11, 23, 168, 274]
Clones are mapped during the incremental clone detection that used source code changes between revisions	faster than the above two techniques	cannot operate on the clone detection results obtained from traditional non-incremental tools [247]	[93, 216]
Separate clone detection from each version, functions are mapped across subsequent versions, then clones are mapped based on the mapped functions	improved runtime	susceptible to similar overloaded/overriden functions	[188, 247, 249]

*A combination of the first and second approaches was also used in some studies [36]

2.6.2 Genealogy-based Clone Change Patterns

Recent genealogy based studies [154, 248, 249] on clone change patterns characterized the evolution of a particular clone-group based on the following six categories of transitions between subsequent versions of the software system.

Addition/Grow: One or more clone fragments is added to the clone-group in the next version.

Deletion/Shrink: One or more clones disappeared from the clone-group in the next version.

Consistent Change: All clone in the clone-group experience the same set of changes during transition to the next version.

Inconsistent Change: During transition to the next version, at least one clone in the clone-group experiences changes that are inconsistent with changes in other member(s) of the clone-group.

Same/No Change: The clone-group moves to the next version without experiencing any change in the members.

Syntactically Similar Change: The clone-groups propagate through subsequent releases either without any changes, or with changes only in formatting and identifiers (e.g., renaming of identifiers) in their code snippets.

On the basis of the aforementioned categorization, clone genealogies are classified as follows. Figure 2.4 shows those different types of genealogies.

Static Genealogy: A genealogy where the clones in the clone-group do not experience any change over the series of subsequent version.

Inconsistently Changed Genealogies: A genealogy where one or more clones in the clone-group experience inconsistent changes during transition between any two subsequent versions.

Consistently Changed Genealogies: A non-static genealogy where none of the clones in the clone-group experience inconsistent changes during transition to subsequent versions.

Dead Genealogy: A genealogy that does not survive till the last version (i.e. before the last version the clone-group disappears).

Alive Genealogy: A genealogy that survives at the last version of the software system.

Syntactically Similar Genealogy: A genealogy where all the clone fragments experience syntactically similar changes during their evolution.

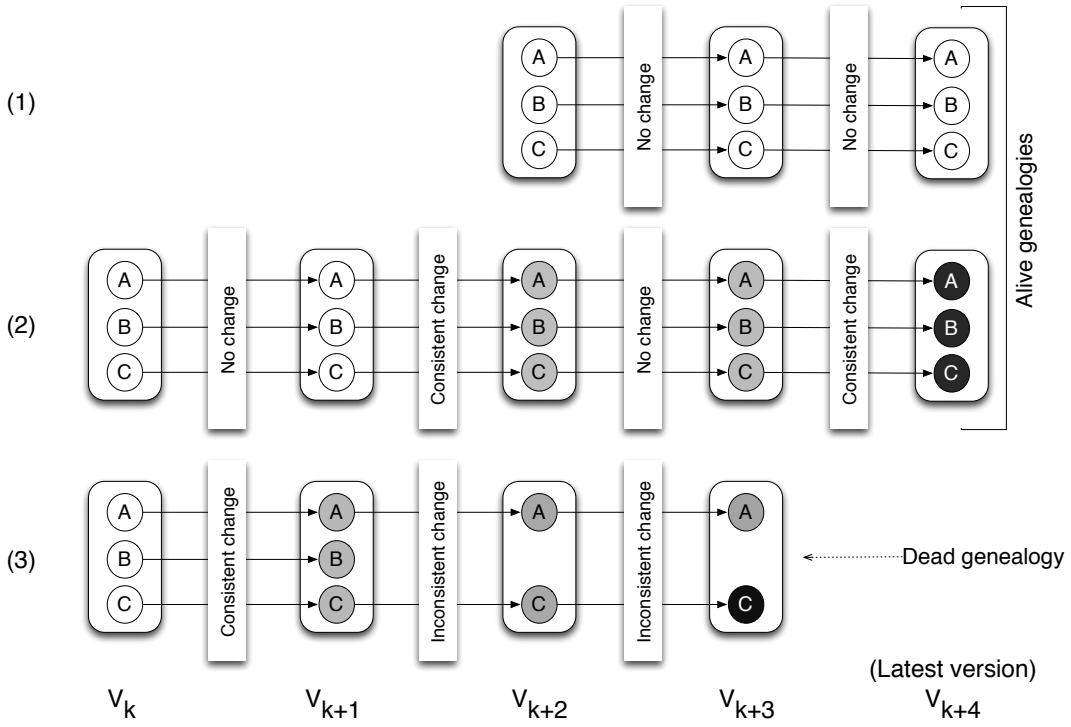


Figure 2.4: Clone change patterns and types of genealogies over versions V_k through V_{k+4}: (1) static genealogy, (2) consistently changed genealogy, (3) inconsistently changed genealogy

2.7 Summary

In this chapter, we have introduced the terminologies and preliminary concepts that form the foundation of this thesis. At the beginning of the chapter, we presented the definition of code clones and described different types of code clones, clone relationship, and granularity. We also described how clones can appear in the source code due to the developers' intention or even without their awareness. Next, we briefly illustrated the patterns of cloning followed by a discussion on the impacts of clones with a real-world example. We also clarified what is meant by "clone management", and we discussed the pros and cons of different clone detection techniques. Finally, we introduced the clone genealogy model that we use for in-depth study of code clone evolution.

Chapter 3

Integrated Clone Search

"The problem is never how to get new, innovative thoughts into your mind,
but how to get old ones out. Every mind is a building filled with archaic furniture.
Clean out a corner of your mind, creativity will instantly fill it"
– Dee Hock

The detection of code clones is a fundamental part of clone management, in particular, clone detection support within the IDE (Integrated Development Environment). Clones must be identified first to deal with them. In this chapter, we introduce a new IDE-integrated clone detection tool. We describe the theory and approach for clone detection adopted in our tool. Early evaluation of our prototype tool indicate that the approach is effective in detecting exact and near-miss clones with high accuracy.

This chapter is organized as follows. First, we provide the motivation behind this particular work in Section 3.1. In Section 3.2, we present our approach for clone search and in Section 3.3, we describe the *k-difference hybrid dynamic programming algorithm* used in the detection of *Type-3* clones. Section 3.4 illustrates how we evaluate our clone search tool. In Section 3.4, we discuss the work relevant to ours, and finally Section 3.6 concludes this chapter.

3.1 Introduction

Over the past decade several techniques and tools for detecting code clones have been proposed, which we have described in Chapter 2. While most of them are capable of detecting *Type-1* (exactly similar code fragments except for white-spaces and formatting) and *Type-2* (syntactically similar code snippets, where identifiers/variables can be renamed) clones, only a few of them are reported to detect *Type-3* (where one or more lines of code can be added/modified/removed) clones. Indeed, due to the additional differences to be dealt with, the detection of *Type-3* clones is relatively more challenging than the detection of *Type-1* and *Type-2* clones.

Moreover, most clone detectors are developed as separate tools that facilitate 'postmortem' approach of clone detection after code development is complete [175]. While such tools are beneficial in the analysis and investigation of code clones and their evolution, they fail to provide necessary clone management support [309] for clone-aware development process, as they are not IDE-based. Those few tools that are integrated with

IDEs (Integrated Development Environments) are mostly focused on detecting *Type-1* and *Type-2* clones, and typically report all the clones in the entire code base. Such flooding of information may overwhelm the developer, who in practice, is likely to be interested in only the clones of a certain portion of code she deals with at a time.

To address these issues, we have developed an IDE-based clone search engine (as a plugin to the `Eclipse` IDE) to facilitate *focused* search of both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones during the *real development time*. By *"focused clone search"*, we mean searching and reporting the clones of a selected code segment only, something similar to that shown in Figure 3.1. This selected code segment is called the *seed fragment* for the search, and it encompasses one or more consecutive lines of code ignoring the comments. The *search space* may be the entire code base, or a portion of it (e.g., set of directories/packages, projects, files) according to the user's choice. Thus, such focused clone search avoids the unnecessary computation overhead that the traditional clone detectors would perform in finding all the clones from the entire code base.

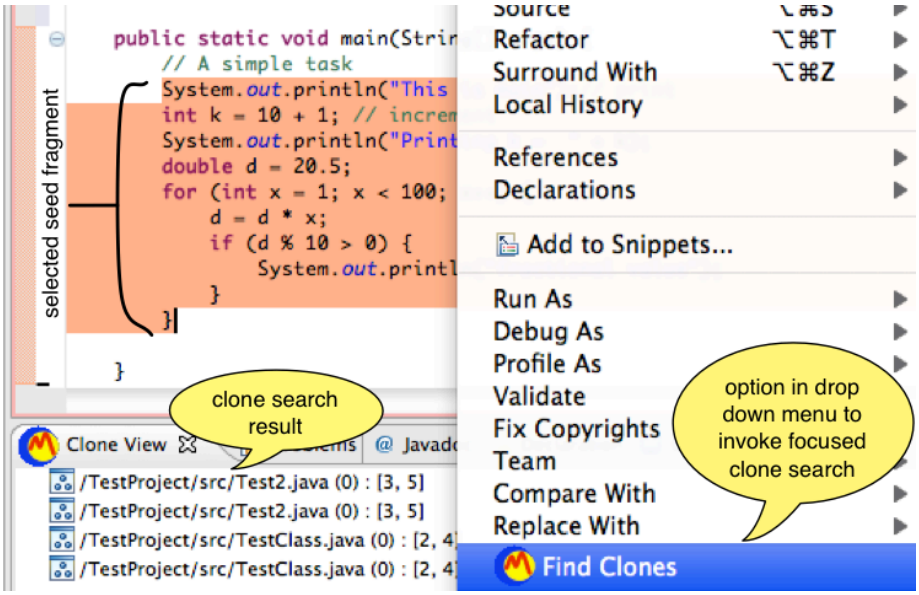


Figure 3.1: Focused clone search facility

The tool support for focused search for clones of a particular code fragment has a number of use cases pertaining to clone management, some of which are listed below.

- Upon identifying the desired clones of a chosen code fragment of interest, the developer may decide to perform proactive clone refactoring, or make reference to existing code instead of introducing a new clone fragment. Thus unnecessary creation of new clones can be avoided.
- While fixing a bug in a certain code fragment, the developer may search for its clones to examine the presence of similar bugs elsewhere in the source code.

- During the development process, when the developer encounters a code fragment that he or she struggles to understand, the developer can find similar implementations for investigation, and when necessary can consult the authors of those clone fragments for discussion or help.
- To explore variant usage of a particular API, the developer can first find a code fragment where the API is used, and then can initiate focused search for clones of the code fragment to find more example usages of the concerned API from the same project or even from different projects.
- The focused search for clones can also be used for verifying code plagiarism and code provenance.

3.2 Clone Detection

Clone detection techniques can broadly be categorized as token-based, text-based, tree-based, graph-based, and metric-based [243], which have their advantages and weaknesses, as discussed in Chapter 2 (Section 2.5). For clone detection, we adopt a hybrid approach combining strengths of multiple techniques. Figure 3.2 presents a schematic diagram of the major algorithmic modules and the process of clone detection used in our approach.

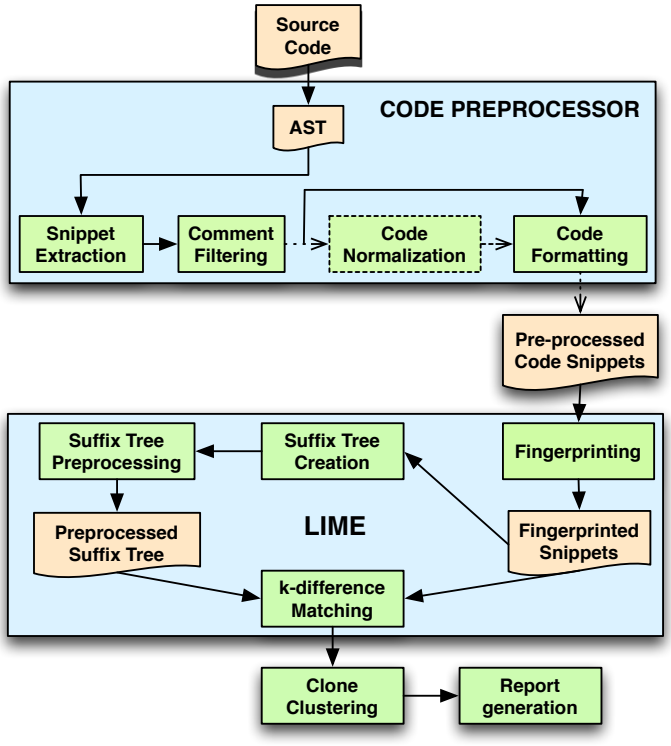


Figure 3.2: Schematic diagram of our approach for near-miss code clone detection

Given a *seed fragment* our technique finds all of its near-miss clones residing in the bodies of the functions within the user-defined search space. For clone identification, each function within the search space is

examined with respect to the *seed fragment*. In the following subsections we describe our clone detection procedure with an illustrative example. First, consider the top two code fragments in the Figure 3.3.

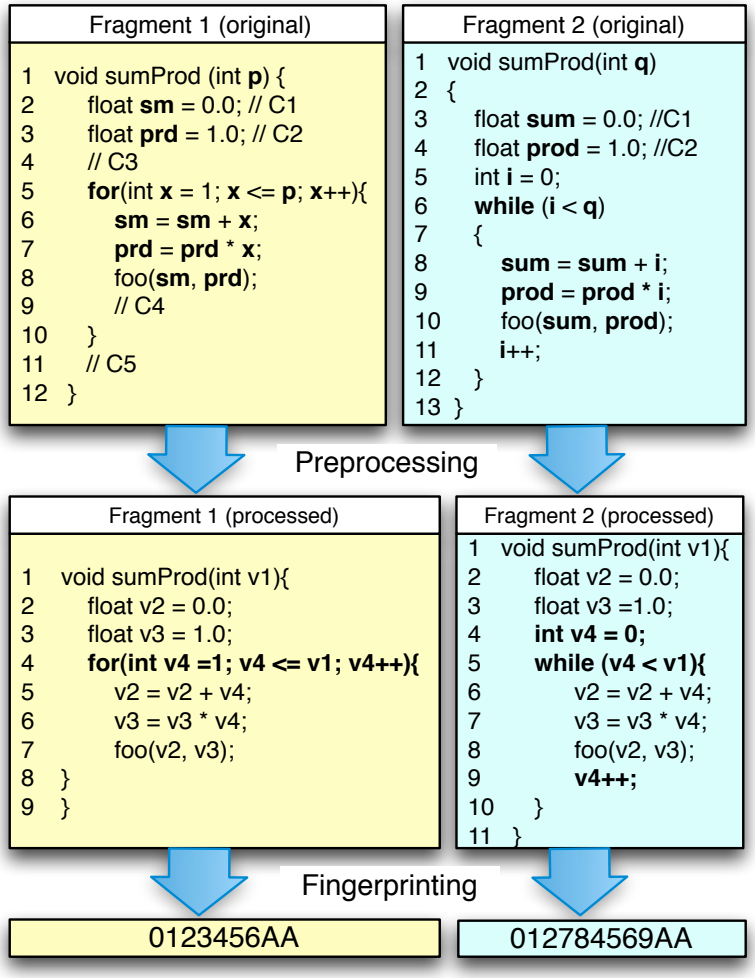


Figure 3.3: Code preprocessing and fingerprinting

These two fragments represent the editing scenario $S4(d)$ of the clone detection technique evaluation framework proposed by Roy and Cordy [243]. They showed that most of the existing clone detection tools did not perform well to detect clones in this scenario. However, we will now show how our multi-phase technique can efficiently detect them as clones.

3.2.1 Code Preprocessing

At the very first phase, using Eclipse's JDT API's, we generate ASTs (Abstract Syntax Trees) for the source code within the user-defined search space. At this point, we filter out the comments and blank lines, and extract all the functions/methods. Then using the Eclipse's refactoring API, we further *normalize* the code

by consistently renaming identifiers and variable names. Such normalization is applied to the *seed fragment* as well. The normalized code fragments are then uniformly formatted with the help of Eclipse's code formatter API. Thus, upon completion of the preprocessing, the original code segments are transformed to fragments free from variations in variable names, comments and layout, as shown in the Figure 3.3. For our current example, the original code snippets (top two in the Figure 3.3) are preprocessed to the transformed fragments (the two fragments in the middle of the Figure 3.3). The preprocessed code segments are then fed to the next module, `LIME` (Language Independent Matching Engine) [309], which performs further computation and comparison.

3.2.2 Source Code Fingerprinting

Using Rabin's fingerprinting algorithm [226], `LIME` computes fingerprints for each line of all the preprocessed code fragments, and thus generates unique integer value for every distinct line of code. By using fingerprints instead of original lines of source code, we avoid the overhead of comparatively expensive pairwise string comparisons. Suppose, for the preprocessed code fragments shown in Figure 3.3, the computed fingerprints for all distinct lines of both the fragments are as presented in Table 3.1. The actual fingerprints are very different from what we show in the table. But for the clarity of description, here we show fingerprints having single digit hexadecimal integer values.

Table 3.1: Hypothetical fingerprints for distinct lines of code

Line of Code	**Fingerprint**
void sumProd(int v1){	0
float v2 = 0.0;	1
float v3 = 1.0;	2
for (int v4 = 1; v4 <= v1; v4++){	3
v2 = v2 + v4;	4
v3 = v3 * v4;	5
foo(v2, v3);	6
int v4 = 0;	7
while (v4 < v1){	8
v4++;	9
}	A

Having the lines source codes 'fingerprinted', for each code fragment, we get a sequence of fingerprints. For the preprocessed fragments of Figure 3.3 the sequence of fingerprints we get are "0123456AA" for the fragment 1, and "012784569AA" for the fragment 2 in accordance with the demonstrations in Table 3.2 and Table 3.3..

Table 3.2: Hypothetical fingerprint for each LOC of fragment 1

Source lines of fragment 1	**Fingerprint**
void sumProd(int v1){	0
float v2 = 0.0;	1
float v3 = 1.0;	2
for (int v4 = 1; v4 <= v1; v4++){	3
v2 = v2 + v4;	4
v3 = v3 * v4;	5
foo(v2, v3);	6
}	A
}	A
Fingerprint sequence: 0123456AA	

Table 3.3: Hypothetical fingerprint for each LOC of fragment 2

Source lines of fragment 2	**Fingerprint**
void sumProd(int v1){	0
float v2 = 0.0;	1
float v3 = 1.0;	2
int v4 = 0;	7
while (v4 < v1){	8
v2 = v2 + v4;	4
v3 = v3 * v4;	5
foo(v2, v3);	6
v4++;	9
}	A
}	A
Fingerprint sequence: 012784569AA	

3.2.3 Creation of Suffix Tree

Upon fingerprinting the preprocessed code fragments, we prepare a generalized sequence of fingerprints by concatenating fingerprint-sequences from all the fragments. To separate fingerprint-sequences from subsequent fragments, we use a distinct terminator (for the current example, say the '$' and '#' symbol) after each of the fingerprint-sequences[1]. So, for the preprocessed fragments of Figure 3.3, the generalized fingerprint-sequence we get is "0123456AA$012784569AA#".

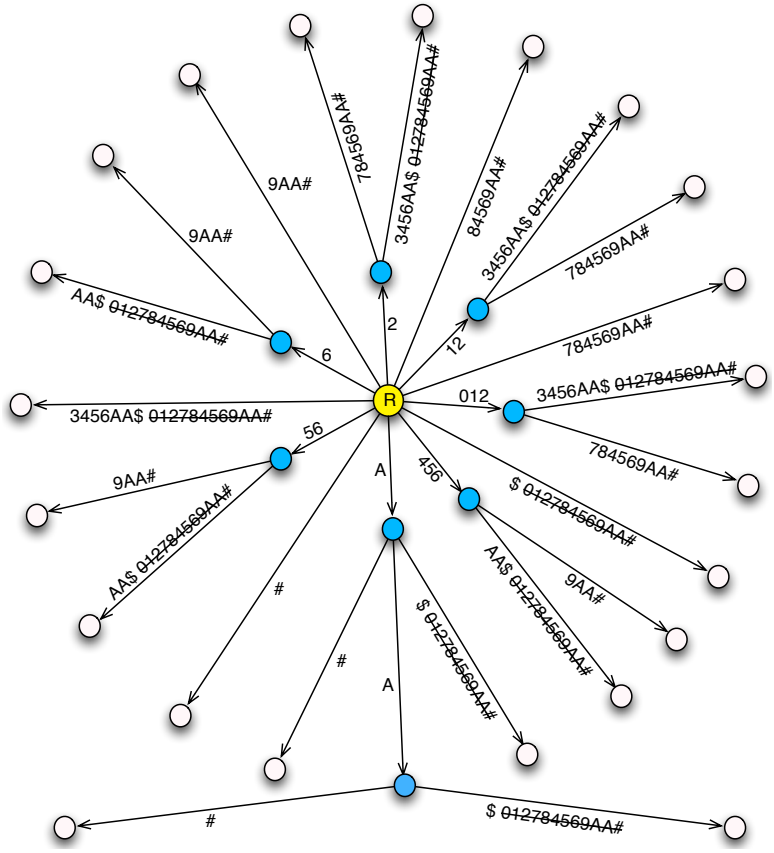


Figure 3.4: The suffix tree for the generalized sequence "0123456AA$012784569AA#"

Next, for the generalized fingerprint-sequence, we construct a generalized *suffix tree* using Ukkonen's online algorithm [283], which runs in linear time. A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves. Each internal node of the suffix tree, other than the root, has at least two children, and each edge is labeled with a nonempty substring of S. Moreover, no two edges out of a node have edge-labels beginning with the same character. Each leaf corresponds to a suffix of S that can be obtained by concatenation of the edge-labels on a path from the root to the leaf. A direct edge from a non-leaf node to a leaf node is called a *leaf edge*.

[1]The algorithm for creating a suffix-tree requires that the last symbol of the sequence must be unique.

In the Figure 3.4, we present the suffix tree for the sequence "0123456AA$012784569AA#" having the root at the center marked with 'R'. Upon creation of the suffix tree, we discard from each of the edge-labels the portion following the first (left-most) terminator symbol. Those discarded portions are shown in the figure with strike-through text. The label of each leaf edge thus ends with a terminator symbol.

3.2.4 Finding Largest Common Subsequences

The concatenation of edge-labels on paths from the root to the non-leaf internal nodes (filled circles in the figure) of the suffix tree gives a set of all sub-sequences common in the fingerprint-sequences. Which fragments participate in a certain common sub-sequence can be determined by looking at the terminator symbols on the labels of the leaf edges on the path from corresponding internal node. For our current example, {2, 12, 012, 6, 56, 456, A, AA} is the set of common sub-sequences we find. We further filter out this set by removing all those sequences that are subsumed by any other sequences and those which are smaller than a user-defined minimum size ω. For our example, the resulting set of the largest common sub-sequences becomes {012, 456} for $\omega \geq 3$ LOC (Source Lines of Code).

From this set of fingerprint-sequences we trace back the actual lines of the original code fragments, and thus we find that the first three lines of fragment 1 is a *Type-2* clone of the first four lines of the fragment 2. Similarly, lines six through eight of the fragment 1 and the lines eight through ten are also *Type-2* clones of each other. To the best of our knowledge, `CCFinder`, `CloneDigger`, `Dup`, `RTF` and the rest of the suffix-tree-based clone detectors including that of Tairas and Gray [267] exploit suffix trees up to this level with or without fingerprinting the source code [243]. But, the usage of suffix trees up to this point can facilitate the detection of *Type-1* and *Type-2* clones only. The detection of *Type-3* clones requires further computations.

3.2.5 Approximate Matching

For detecting *Type-3* clones, we invoke a *k-difference hybrid algorithm* (described later in Section 3.3) on pairs of fingerprinted code fragments. The approximate matching algorithm finds all the occurrences of one sequence inside another allowing at most k differences. Based on a user-defined *dissimilarity threshold*, we compute k for each pair of code fragment as, $\lceil k = \frac{l \times threshold}{100} \rceil$, where, l is the number of lines in the smaller fragment. The threshold signifies what percentage of different lines of code be allowed in the approximate matching. For example, consider two fragments of source code having 100 and 120 lines respectively, and the user's given threshold is 10%. Then the value of $k = \frac{100 \times 10}{100} = 10$, which means in the approximate matching, difference of at most 10 lines will be tolerated. Thus our computation of k is sensitive to the size of the code fragments.

For our current example, the corresponding fingerprint-sequences "0123456AA" and "012784569AA" approximately match (with $k \geq 3$) having three differences, yielding a global alignment, for instance, as shown in Table 3.4.

Table 3.4: Global alignment of *0123456AA* and *012784569AA*

0	1	2	**7**	**8**	4	5	6	**9**	A	A
0	1	2	**3**	_	4	5	6	_	A	A

Here, '_' denotes a space

Thus, given either of the original fragments of Figure 3.3 as seed, we accurately detect the other as its *Type-3* clone. All the detected clones are then sorted according to their similarity with the given seed fragment, and the result is then displayed in the Eclipse's interactive Tree View. Choosing a clone from the Tree View highlights the corresponding code fragment in the editor.

3.3 K-difference Hybrid Algorithm

We adapted the k-difference hybrid algorithm from the work of Landau and Vishkin [173]. The algorithm combines the advantage of further preprocessed suffix tree with dynamic programming (DP). Readers interested in the details of the algorithm are referred to elsewhere [97, 173]. In this section, we provide a brief description of how we adapted the algorithm for near-miss clone detection.

3.3.1 Preprocessing of Suffix Tree

We enumerate all the r nodes of the suffix tree $\mathcal{T}$ according to the standard *depth-first (preorder)* numbering as shown in Figure 3.5. Thus each node v of $\mathcal{T}$ is numbered with a $\lceil log_2 r \rceil$ bit integer.

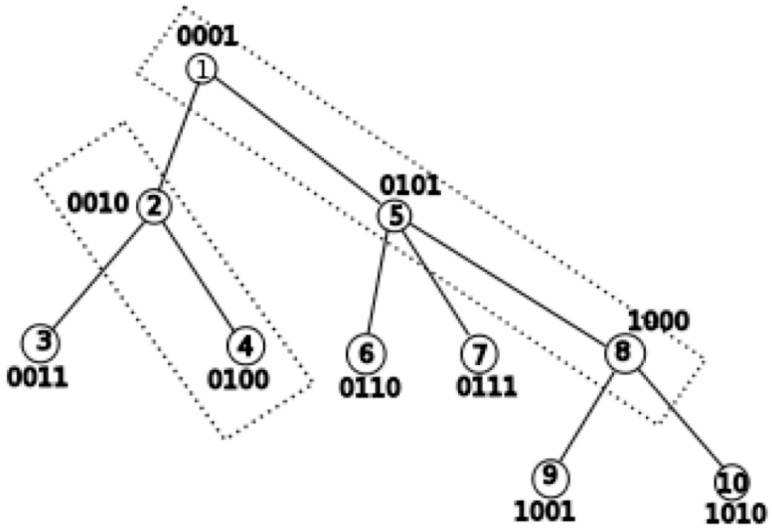


Figure 3.5: Enumerated suffix tree having nodes partitioned into Runs marked with dotted rectangles [97]

Then, we map the enumerated suffix tree $\mathcal{T}$ to a hypothetical rooted complete binary tree $\mathcal{B}$, whose vertices are enumerated according to *inorder* numbering, as shown in Figure 3.6. Throughout the remainder of the chapter, we will simply use "node v", to refer to the node in the enumerated rooted tree having node number v.

Definition 1 (lowest common ancestor) *The lowest or least common ancestor (LCA) of any two nodes u and v in a rooted tree Υ, denoted as $LCA_{\Upsilon}(u, v)$, is the deepest node x in the tree, which is an ancestor of both u and v [97].*

Definition 2 *For any node k, $h(k)$ is the position (from right) of the least significant 1-bit in the binary representation of k. For a node k in $\mathcal{B}$, $h(k)$ equals the height of the node in $\mathcal{B}$ [97].*

Definition 3 *For a node v of $\mathcal{T}$, $I(v)$ is the node w in $\mathcal{T}$ such that $h(w)$ is the maximum over all nodes in the subtree of v including v itself [97].*

Definition 4 *A run in $\mathcal{T}$ is a maximal subset of nodes of $\mathcal{T}$, denoted as T_r, such that, $\forall \langle u, v \rangle \in T_r, I(u) = I(v)$. The* head *of a run is the node closest to the root [97].*

We map each node v in $\mathcal{T}$ to a node $I(v)$ in $\mathcal{B}$. Figure 3.5 presents an example of partitioning the nodes of suffix tree $\mathcal{T}$ into seven different runs. Figure 3.6 shows a complete binary tree $\mathcal{B}$, which the nodes of the suffix tree $\mathcal{T}$ of Figure 3.5 are mapped to. In the Figure 3.6, the binary representation of the number a node v in $\mathcal{B}$ is shown if there is a node in $\mathcal{T}$ that maps to v.

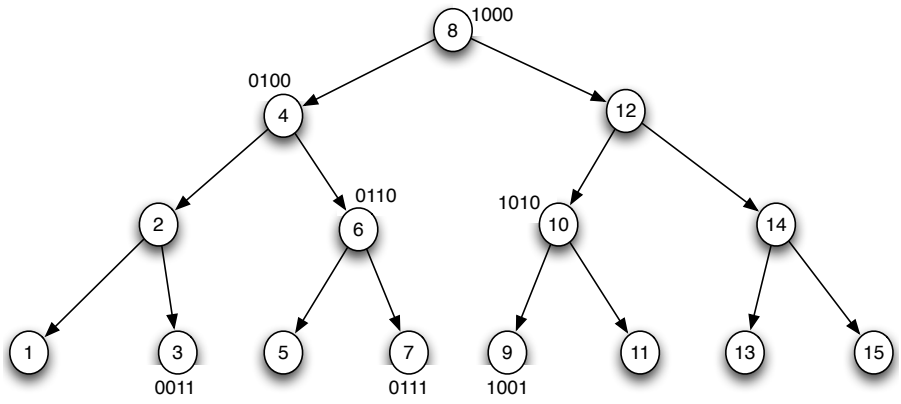


Figure 3.6: Binary tree with nodes enumerated according to inorder numbering [97]

Moreover, for each node v in $\mathcal{T}$, we create an $O(log_2 r)$ bit number A_v. Bit $A_v(i)$ is set to 1 if and only if node v has some ancestor in $\mathcal{T}$ that maps to height i in $\mathcal{B}$, i.e., if and only if v has an ancestor u such that $h(I(u)) = i$.

Computing LCA in Binary Tree

Having the inorder numbering of the nodes of binary tree $\mathcal{B}$, we compute $LCA_{\mathcal{B}}(u, v)$ in constant time using shift and XOR operations. We first determine if u is an ancestor of v, or vice versa. Using $h(u)$ and $h(v)$ we determine which of the nodes u and v is higher (closer to the root). If u is the higher node, we find the number of edges on the path from the root to node u. Then we take the XOR of u and v, and find the position k of the left-most 1-bit (counting from the left). Node u is an ancestor of node v if and only if k is larger than the number of edges on the path from the root to node u. In this case node u is the LCA of nodes u and v.

In the cases, where none of u and v is an ancestor of the other, we XOR u and v, find the left-most 1 bit in the k^{th} position in the result. Then we right shift u by $d - k$ places (where $d = \lceil log_2 r \rceil$ is the number of bits used in numbering the nodes of $\mathcal{B}$), set the right-most bit to 1, and left shift it back by $d - k$ places.

Computing LCA in Suffix Tree

Using the following procedure [97], we find z in constant time, where $z = LCA_{\mathcal{T}}(x, y)$ for the suffix tree $\mathcal{T}$ preprocessed as described before.

1. Find $b = LCA_{\mathcal{B}}(I(x), I(y))$.
2. Find the smallest position $\rho \geq h(b)$ such that both numbers A_x and A_y have 1-bits in position ρ.
3. Find x', the node closest to x on the same run as z (unknown yet) as follows.
   (a) Find location ρ_r of the right most 1-bit in A_x.
   (b) If $\rho_r = \rho$, then set $x' = x$, and go to step 4.
   Otherwise continue to next steps.
   (c) Find the position ρ_l of the left-most 1-bit in A_x, which is to the right of position ρ. Form the number i_w consisting of the bits of $I(x)$ to the left position of ρ_l, followed by a 1-bit in position ρ_l, followed by all zeros. Then find the head w of the run containing node i_w. Set node x' to be the parent of node w in $\mathcal{T}$.
4. Find y', the node closest to y on the same run as z, using the same approach as in step 3.
5. If $x' < y'$ then $z = LCA_{\mathcal{T}}(x, y) = x'$, otherwise, $z = LCA_{\mathcal{T}}(x, y) = y'$.

Finding Longest Common Extension

Suppose, $S = [s_1\ s_2\ s_3 \ldots s_m]$ is a sequence of length m, and $[s_i\ s_{i+1}\ s_{i+2} \ldots s_j]$ (where, $1 \leq i \leq j$ and $1 \leq j \leq m$) is a subsequence of S, which we denote as $S[i \ldots j]$. Thus, a suffix $[s_i\ s_{i+1}\ s_{i+2} \ldots s_m]$ starting from element s_i is denoted by $S[i \ldots m]$, and $S(i)$ refers to the i^{th} element in sequence S. Given a pair of

sequences S_1 (of length m) and S_2 (of length n), and an index pair $\langle i, j \rangle$ where i and j refer to element-positions in S_1 and S_2 respectively, the *longest common extension (LCE)* between the sequences (with respect to the given index pair) is the longest subsequence of S_1 starting at position i that matches a subsequence of S_2 starting at the j^{th} position; in other words, the longest prefix of $S_1[i \dots m]$ that matches a prefix of $S_2[j \dots n]$.

Using a preprocessed generalized suffix tree $\mathcal{T}$ for the sequences, given an index pair $\langle i, j \rangle$ corresponding to sequences S_1 and S_2, the LCE can be computed in constant time. We first find the LCA, z of the leaves of $\mathcal{T}$ that correspond to $S_1[i \dots m]$ and $S_2[j \dots n]$. The concatenation of the edge-labels on the path from the root to z yields the LCE we want.

3.3.2 Hybrid Dynamic Programming

A classical way to represent a sequence matching problem is to express it in terms of *global alignment*. Given a couple of sequences S_1 and S_2, their global alignment is obtained by first inserting spaces in chosen places (between elements, at the end or beginning of S_1 and S_2) to make the resulting sequences S_1' and S_2' have equal length l, and then superimposing one above the other so that every element or space in either sequence is opposite a unique sequence or a unique space in the other sequence [97]. Table 3.4 presents an example of global alignment. Typically, a score $g(x, y)$ is associated with the alignment of each pair $\langle x, y \rangle$ of elements. The score of an alignment of S_1 and S_2 is computed as $\sum_{i=1}^{l} g(S_1'(i), S_2'(i))$, and the alignment with the optimal score yields the optimal global alignment.

Let $V(i, j)$ denotes the score of the optimal global alignment of $S_1[1 \dots i]$ and $S_2[1 \dots j]$. The optimal global alignment of the sequences having score $V(m, n)$ can be computed in $O(mn)$ time by traditional dynamic programming (DP) using the following recurrence.

$$V(i,j) = min \begin{cases} V(i-1, j-1) + g(S_1(i), S_2(j)), \\ V(i-1, j) + g(S_1(i), \_), \\ V(i, j-1) + g(\_, S_2(j)) \end{cases}$$

where the base conditions are,

$$V(0,j) = \sum_{1 \le p \le j} g(\_, S_2(p)) \text{ and } V(i,0) = \sum_{1 \le 1p \le i} g(S_1(p), \_).$$

We formulate the focused clone searching problem as a variation of the optimal global alignment problem. Let S_2 and S_1 be the fingerprint-sequences of respectively the seed and another fragment in the search space. We define a scoring scheme as follows:

$$g(S_1(i), S_2(j)) = \begin{cases} 0, & \text{if } S_1(i) = S_2(j) \\ 1, & \text{if } S_1(i) \neq S_2(j) \end{cases}$$

$$g(S_2(i), \_) = 1$$

$$g(\_, S_1(j)) = \begin{cases} 0, & \text{if } \forall j' < j, S_1(j') \text{ aligned to '\_'} \\ & \quad \text{or, } j > m \\ 1, & \textit{otherwise.} \end{cases}$$

Instead of finding their optimal global alignment, we need to find all the global alignments with scores no more than the k. The k-difference hybrid approach makes use of suffix trees to solve subproblem of computing the LCE queries within the framework of DP. Consider an $m \times n$ traditional DP table for S_1 and S_2. The solution to the optimal global alignment problem computed using traditional DP approach yields a path on the DP table specifying the computed optimal alignment as well as giving the number of pairwise character matches and differences. The same concept of path is used in the hybrid algorithm.

For the sake of the hybrid algorithm, we enumerate the diagonals of the DP table as follows. The main diagonal of the table consisting of cells $\langle i, i \rangle$, for $0 \leq i \leq n \leq m$, is numbered diagonal 0. The diagonals above the main diagonal are numbered 1 through m; the diagonal starting at cell $\langle 0, i \rangle$ is diagonal i. The diagonals below the main diagonal are enumerated -1 through $-n$; the diagonal starting at cell $\langle i, 0 \rangle$ is diagonal $-i$.

Definition 5 (d-path) *A d-path in the DP table is a path that starts in the row zero and specifies a total of exactly d mismatches and spaces [97].*

Definition 6 (farthest reaching d-path) *A d-path is farthest reaching on diagonal i, if it is a d-path ending on diagonal i, and the index of its ending column c along diagonal i is the maximum among all the d-paths ending on diagonal i [97].*

For $d > 0$, three distinct d-paths on diagonal i can be computed from $(d-1)$-paths on diagonals $i-1, i$, and $i+1$ [97]:

R_v-path is composed of the farthest-reaching $(d-1)$-path on diagonal $i+1$, trailed by a vertical edge (corresponding to a space in S_1) to diagonal i, and a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

R_h-path consists of the farthest-reaching $(d-1)$-path on diagonal $i-1$, trailed by a horizontal edge (corresponding to a space in S_2) to diagonal i, and a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

R_d-path is made up of the farthest-reaching $(d-1)$-path on diagonal i, trailed by a diagonal edge (corresponding to a mismatch between an element in S_2 and an element in S_1) along diagonal i, followed by a maximal extension along diagonal i that corresponds to identical subsequences in S_2 and S_1.

With these specifications, in Algorithm 1, we describe the k-difference hybrid algorithm.

Algorithm 1 : k-Difference Hybrid Algorithm [97]

for $i = 0$ *to* m **do**
    find the *LCE between* $S_1[i \ldots m]$ *and* $S_2[1 \ldots n]$ *by* LCE *query to suffix tree. This specifies the end column of the farthest reaching 0-path on diagonal* i.
end for
Any path reaching row n *in column* c*, defines an exact match of* S_2 *in* S_1 *ending at* $S_1(c)$.
for $d = 1$ *to* k ***do***
    for $i = -n$ *to* m ***do***
        Find the end on diagonal i *of paths* R_v, R_h*, and* R_d*. The farthest-reaching of these three paths is the farthest-reaching* d*-path on diagonal* i.
    end for
    Any path reaching row n *in column* c*, defines an approximate match of* S_2 *in* S_1 *ending at* $S_1(c)$ *with at most* k *differences.*
end for

3.4 Evaluation

In this section, we present both theoretical and empirical evaluation of our approach in terms of performance, accuracy, and usability.

3.4.1 Algorithmic Complexity

Suppose, we have $\mathcal{F}$ code fragments in the search space including the seed fragment, and fragment f has l_f lines of code. The total number of lines over all code fragments is $l_{\mathcal{F}} = \sum_{f=1}^{\mathcal{F}} l_f$. Without loosing generality, we can assume that each line of code has c characters.

Code preprocessing is done in linear time. Using Rabin's fingerprinting algorithm (which runs in linear time [226]), we fingerprint a line of code in $O(c)$ time, and thus for computing fingerprints of all $l_{\mathcal{F}}$ lines it takes $O(c \times l_{\mathcal{F}})$ time. The number of characters in a nonempty line of source code typically vary between 1 and 20. So, we may consider c to be invariant, and thus the running time of Rabin's fingerprinting algorithm over all lines of source code becomes $O(l_{\mathcal{F}})$.

To construct the generalized suffix tree we use Ukkonen's online algorithm [283], which also runs in linear time. Hence the construction of the generalized suffix tree also takes $O(l_{\mathcal{F}})$ time. Exact matches are readily detected as we construct the generalized suffix tree. Since, the fragments are preprocessed to discard variations in variable names and formatting, all *Type-1* and *Type-2* clones across all code the fragments are also detected in $O(l_{\mathcal{F}})$ time.

Preprocessing of the generalized suffix tree also takes $O(l_{\mathcal{F}})$ time. Given a text S_1 of length m and a pattern S_2 of length n, in $O(km)$ time and $O(m + n)$ space, the k-difference hybrid algorithm can find all end locations in S_1 where S_2 matches with at most k differences [97, 173]. Based on the algorithm, our

implementation takes $O(kl_{\mathcal{F}})$ time and $O(l_{\mathcal{F}})$ space to find all occurrence of k-difference near-miss clones of a chosen seed fragment in all other fragments in the search space.

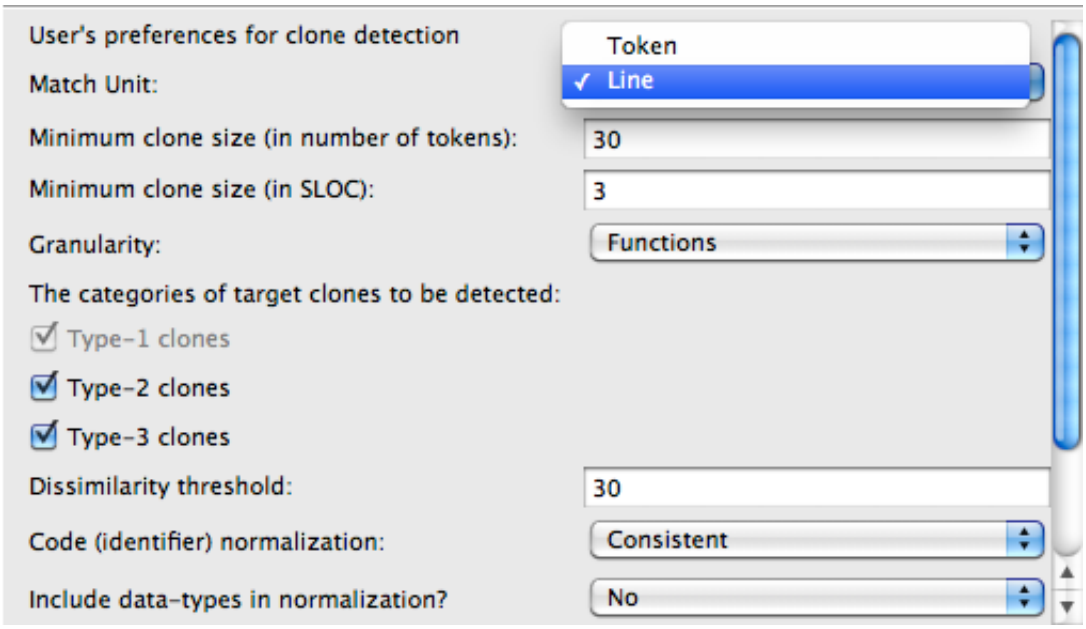


Figure 3.7: Customization options for clone detection

3.4.2 Usability and Customization

One of the primary objectives of our work is to support clone-aware programming during the real development activities. To make the *focused clone search* facility flexibly usable, our tool enables the user to simply select a code fragment and invoke *focused search* for it's near-miss clones by choosing the appropriate option from a drop-down menu (Figure 3.1). The user can also define the search space (i.e., projects, directories, or files). Moreover, the tool offers a variety of customization options through an Eclipse-preferences user-interface (UI) as shown in the Figure 3.7.

These customizations define a set of search options such as the type of the target clones (i.e., *Type-1*, *Type-2*, and/or *Type-3*), dissimilarity threshold, code normalization options, minimum clone size, and match-unit (i.e., lines or tokens). Alhough, in Section 3.2, we described our clone detection approach to have each line of code as a match-unit, the technique can also use each token/word as a match-unit, and thus can offer more rigorous search if needed. For the token-based operation, the code formatting phase can be skipped, and the rest of the phases remain as they are. The code normalization phase is skipped if only the *Type-1* clones need to be identified, and the k-difference hybrid algorithm is invoked only when the *Type-3* clones have to be detected.

Table 3.5: Subject systems for the study on comparison with `NiCad`

Subject Systems	Total LOC	# of Functions	# of Clone	
			Groups	Fragments
weltab	9936	123	20	68
PostGreSQL	154843	4689	203	519

3.4.3 Empirical Evaluation

Since our tool is developed as a plugin to Eclipse, it is tedious and time consuming to use the Eclipse's editor UI to manually select code fragments, find their clones and then verify accuracy. Therefore, we created a variant of our tool by replacing the Eclipse-coupled "code preprocessor" module by a separate *TXL*[2] implementation. Such an implementation can handle any programming language, as long as the appropriate grammar and the transformation rules are defined. Ours can handle C, C#, and Java. Also note that, except the "code preprocessor" module, the rest of our tool is language independent.

The stand-alone variant of our tool can operate outside IDE, and we used this to empirically evaluate the accuracy of clone detection (Section 3.4.3 and Section 3.4.3). We also carry out a user-study (Section 3.4.3) to evaluate the usability of our tool and its usefulness in context.

Comparison with NiCad

Traditional clone detectors are not directly comparable to our tool, as the objective of our tool is to enable focused clone search instead of the detection of all clones from the entire code base. The work of Lee at al. [175] could have been comparable, but their implementation was not available in public. So, we devised a technique for comparison with `NiCad` [238], a state-of-the-art tool for detecting exact (*Type-1*) and near-miss (*Type-2*, and *Type-3*) clones. We chose `NiCad` for two main reasons. First, the threshold schemes of our tool and `NiCad` are similar and comparable. Second, as a clone detector, `NiCad` was reported to have high precision [238, 242] and recall [240].

For both `NiCad` and our tool, we set the minimum clone size to three LOC in the pretty-printed format [238] of the source code. Moreover, the UPIT (Unique Percentage of Items Threshold) for `NiCad` and the dissimilarity threshold for our tool, both were set to 30%. Since `NiCad` performs line-based comparison, the match-unit for our tool was also set to 'line'. Then we instructed `NiCad` to detect all near-miss *function* clones from the two subject systems, `weltab` and `PostGreSQL` (Table 3.5). These systems obtained from www.bauhaus-stuttgart.de/clones/, are two of those used in the clone detectors comparison framework of Bellon et al. [35]. `NiCad` reports the clone detection result having the clones clustered into clone-groups.

[2]http://www.txl.ca/

Table 3.6: Accuracy of clone detection based on mutation-based evaluation

Measurement	*Type-1*	*Type-2*	*Type-3*	Overall
precision	1.0	1.0	1.0	1.0
recall	1.0	0.96	0.90	0.94
f-score	1.0	0.97	0.94	0.96

The number of clone-groups and the total number of individual clone fragments as identified by `NiCad`, are presented in the right-most two columns of the Table 3.5.

From each of the clone-groups, we randomly picked a fragment, passed it to our tool as the *seed fragment*, and invoked the tool to perform a *focused search* for all the clones of the seed. Then we verified the search result to determine whether our tool detected all other members of the clone-group (as of the seed), and whether any false positives were reported. For each seed fragment obtained from every clone-group over both the subject systems, our tool did find all the remaining members of the corresponding group, and did not report any false positives.

Mutation-based Evaluation

The aforementioned comparative study evaluated our tool's accuracy in clone detection with respect to that of `NiCad`. Using a variant of the mutation framework proposed by Roy and Cordy [240], we further evaluated the precision and recall of our tool. As the code base subject to this study, we chose `JHotDraw-5.4b1`, which had 20,613 LOC Java code.

We selected, as *seed fragments*, 10 arbitrary functions ($f_1 \dots f_{10}$) of different sizes from different locations of the code base. Then we made 15 copies ($f_i^1 \dots f_i^{15}$) of each of those 10 fragments, and modified them following a subset of mutation operators [240]. Thus, we produced five *Type-1* clones, five *Type-2* clones, and five *Type-3* clones of each *seed fragment* f_i, resulting a total of 150 synthetic function clones. Then we injected these clones into different locations of the original code base, resulting a *mutated* code base.

We configured our tool according to the specifications shown in the Figure 3.7. Then we invoked focused clone search on the mutated code base, providing each of the 10 seed fragments, one at a time. For each of the seed fragments, we investigated whether all of its synthetic clones (*Type-1*, *Type-2*, and *Type-3*) were reported in the search result. Some additional fragments were also reported as clones, as they existed in the code base but we did not know about them in advance. We manually verified those additional clones for any possible false positives.

With respect to the injected synthetic clones, we compute the precision, recall (i.e., sensitivity), and f-score of our tool using the following equations:

$$p = \frac{|c_s \cap c_d|}{|c_d|}, \quad r = \frac{|c_s \cap c_d|}{|c_s|}, \quad \exists = \frac{2 \times p \times r}{p + r}$$

where p, r, and $\exists$ denotes precision, recall, and f-score respectively while c_s and c_d are respectively the sets of injected synthetic clones, and those that our tool detects. In the Table 3.6 we present the accuracy of our tool in detecting those synthetic clones.

Our tool missed two of the *Type-2* and five of the *Type-3* synthetic clones. We manually investigated those clones, and found that while creating the *Type-2* clones, we altered the order of declaration of variables. The normalization based on "consistent renaming" of identifiers is sensitive to such orders, and this was set in the configuration. This is why our tool could not detect those as clones, but it was able to capture them when later we applied normalization using the "blind renaming" operation. The five *Type-3* clones escaped due to the fact that upon modification, those fragments simply went beyond the 30% dissimilarity threshold we used.

Table 3.7: Demographic information about the participants of the user-study

User	Age (in years)	Software development experience		Known Programming Languages
		in industry (months)	in academia (months)	
1	27	0	96	C, C++, Java, Visual C++, JavaScript
2	30	3	72	C, C++, Java, C#, TXL
3	33	96	24	C, C++, Java, C#, Python
4	35	6	14	C, Java, PHP, SQL, Matlab
5	31	57	96	C, C++, Java, C#, Perl, Python, TXL, VB
6	34	30	156	C, C++, Java, Python, SQL
7	35	0	144	C, C++, Java, VB, Scheme, Assembly, PHP, JavaScript
8	31	18	84	C, C++, Java, C#, JavaScript, PHP, HTML, ColdFusion

User Study

We carried out a user study involving eight programmers who were graduate students having years of experience in programming under different IDEs including Eclipse. All the participants are familiar with code clones and their impacts on software development and maintenance. Demographic information about the participant of the user-study is presented in Table 3.7.

We provided our Eclipse plugin to them, and they used it for an extended period of time while doing some programming tasks. Then we interviewed them with a questionnaire comprising both open and closed questions. The closed questions included a Likart-scale query saying, "*How will you rate this tool's usability*

and usefulness?". The participants had to choose one of the answers: very poor, poor, moderate, good, very good, or excellent. In response, five of the participants chose 'good', two chose 'moderate', and the other participant chose 'very good'. A noteworthy comment from one of the participants was, "*...this [tool] is intuitive!*".

3.5 Related Work

There are many clone detection tools out there and a comprehensive list can be found elsewhere [243, 311]. In Chapter 2 (Section 2.5), we discussed different clone detection techniques and pointed to their strengths and limitations. Among all those clone detectors, those which are IDE-based are the most relevant to our work on integrated clone search as presented in this chapter.

`CloneBoard` [62], `CPC` [292], and `CloneScape` [48] are Eclipse plugins that can detect and track clones based on clip-board (copy-paste) activities of programmers. Based on programmer' copy-paste activities, `CReN` [117] (another plugin to Eclipse), offers some sort of clone refactoring support by consistent renaming of identifiers. While such an approach based on programmers' copy-paste activities may be able to handle *intentional clones*, they cannot deal with *unintentional* clones. Moreover, such tools may not be suitable for distributed development, as they may fail to combine information about clones separately created by distinguished developers working in a distributed environment.

`CPD`[3] is a part of Java source code analyzer, `PMD`. `SDD` [177] and `Simian`[4] are plugins to Eclipse. Tairas and Gray [267] also developed a suffix-tree based clone detector as a plugin for the Microsoft Phoenix framework. Bahtiyar developed `JClone` [12] as a plugin to the Eclipse IDE for detecting code clones from Java projects. All of these clone detectors can detect *Type-1* and *Type-2* clones only, but not *Type-3* [12, 36]. Another Eclipse plugin, `CloneDR`[5], is an AST-based clone detector that can detect *Type-1* and *Type-2* clones, but it also fails to detect Type-3 clones in many scenarios [243]. `CloneDetective` [132] uses a suffix-tree-based algorithm to detect *Type-1* and *Type-2* clones but "probably not" *Type-3* [243]. However, `SimScan`[6], which is a parser-based tool available as plugin to Eclipse, IDEA, or JBuilder, can detect *Type-1*, *Type-2*, and possibly a subset of *Type-3* clones.

`SHINOBI` [145] is a plugin to Microsoft Visual Studio. It internally uses `CCFinderX`'s preprocessor, and thus it can detect *Type-1* and *Type-2* clones only, but not *Type-3*. It was developed as a client(IDE)-server(CVS) application to relocate the the clone detection overhead from the client to a central server. A similar clone detection tool is `JSync` [215] that is available as an add-on to Subclipse/SVN, an Eclipse plugin software configuration management tool. However, such client-server configuration may not be suitable for those individual practitioners who work on their separate stand-alone machines.

[3] http://pmd.sourceforge.net/cpd.html
[4] http://www.harukizaemon.com/simian
[5] http://www.semdesigns.com/Products/Clone/
[6] http://blue-edge.bg/download.html

Table 3.8: Summary of clone detection support from IDE-integrated clone detection tools

Tool	Integration with	Clone detection approach	Supported clone types			
			Type-1	Type-2	Type-3	Type-4
`CloneDetective` [132]	ConQAT	suffix tree based	●	●	●	○
`SimScan`	Eclipse, JBuilder, intelliJ IDEA	AST based	●	●	●	○
`Simian`	Eclipse	unknown	●	●	○	○
`DupMan` [87]	Eclipse	internally uses `SimScan`	●	●	●	○
`CloneScape` [48]	Eclipse	copy-paste	◐	◐	◐	○
`CloneBoard` [62]	Eclipse	copy-paste	◐	◐	◐	○
`CPC` [292]	Eclipse	copy-paste	◐	◐	◐	○
`CnP` [114]	Eclipse	copy-paste	◐	◐	◐	○
`SHINOBI` [145]	Microsoft Visual Studio	suffix array based	●	●	○	○
`Wrangler` [179]	Eclipse, Emacs	AST, anti-unification	●	●	○	○
`JClone` [12]	Eclipse	AST based	●	●	○	○
`JSync` [215]	SVN	AST, LSH, feature vector	●	●	◐	○
`CPD`	PMD	fingerprinting	●	●	○	○
`SDD` [177]	Eclipse	indexing, n-neighbour	●	●	○	○
`CloneDR`	Eclipse	AST based	●	●	◐	○
`CloneDigger`	Eclipse	AST, suffix tree, anti-unification	●	●	○	○
`CodeRush`	MS Visual Studio	unknown	●	●	◐	○
AgentRalph	ReSharper	internally uses `Simian`	●	●	○	○
Tool of Tairas and Gray [267]	MS Phoenix framework	suffix tree based	●	●	○	○
Tool of Zibran and Roy [310]	Eclipse	fingerprinting AST, suffix tree	●	●	●	○

Legends: ●= supported, ◐= partially supported for a limited subset, ○= not supported

Earlier prototypes of JSync [215] appeared as Clever [218] and Cleman [217]. JSync detects clones based on similarities among the feature vectors computed over AST representation of the code fragments. To attain computational efficiency, JSync applies $N(N = 16)$ locality sensitive hashing (LSH) functions to first cluster the code fragments into N buckets. Indeed, the effectiveness of such a clustering depends on the definition of those N individual hash functions. But specifications of those hash functions are not reported, and so no further comments can be made on their effectiveness. Upon having the fragments clustered into N buckets, the fragments in each bucket are pairwise compared to determine if they are clones. Finally, the clone pairs are merged to form clone-groups based on the assumed transitive property among the clones. However, such transitive property may not hold for *Type-3* clones, while it may hold for *Type-1* and *Type-2* clones only. This implies that JSync's approach for clone detection may perform well enough for *Type-1* and *Type-2* clones, but not for *Type-3*. The authors of JSync also noted, "For *Type-3* clones, in JSync, the changes are limited to the minor ones with less than a small pre-defined threshold of added/removed program elements" [215]. Moreover, the accuracy of clone detection is proportional to the value of N, and the increase of N also increases computational overhead.

CloneTracker [68] uses SimScan as the underlying clone detector. CeDAR [268] can incorporate the results from different clone detection tools (e.g., CCFinder, CloneDR, DECKARD, Simian, or SimScan) and can display properties of the clones in an IDE. DupMan[7] is an Eclipse duplication management framework for clone detection and removal. It also works on top of other clone detectors (e.g., SimScan, CCFinder, CloneDR, Dup, Duploc, Duplix). Doppel-Code [79] is another Eclipse plugin developed mainly for presenting clone detection result obtained from a separate clone detector Simian. While all these four tools are developed as plugins to Eclipse, they suffer from the limitations of the underlying clone detectors they use internally. AgentRalph[8] is a clone detector developed as a plug-in to the ReSharper add-on of Microsoft Visual Studio. AgentRalph can detect *Type-1* and *Type-2* function clones only from source code written in C#.

Li and Thompson developed Wrangler [179], a tool for supporting refactoring of functional programs written in Erlang. Wrangler can be integrated with Emacs or Eclipse IDE through the ErlIDE plugin. Wrangler can detect clones from Erlang programs in several phases. First, the program is parsed to generate AST. Then the AST is normalized (generalized) by replacing certain expression by placeholders. The source code corresponding to the generalized AST is then pretty-printed and serialized into a single sequence of delimited expressions. Each expression statement is then hashed to generate a sequence of hash values, which are then used to build a suffix-tree. The clones detected from the suffix-tree are then examined through anti-unification for filtering out the false positives. Thus Wrangler can detect *Type-1* and *Type-2* clones only, but probably not *Type3*. In addition, the clone detection approach of Wrangler is computationally expensive. The anti-unification technique for the examination of a single clone-group having n members itself has the $O(n^2)$ worst case runtime complexity [179].

[7] http://sourceforge.net/projects/dupman/
[8] https://code.google.com/p/agentralphplugin/

All of the aforementioned tools adopt the traditional 'postmortem' approach to detect clones from the entire code base, which is overkill for a *focused search* of only the clones of an individual code fragment [175]. Most of those tools have limitations in detecting *Type-3* clones, as can be seen more clearly in Table 3.8. Ours is the first that enables efficient *focused search* not only for *Type-1* and *Type-2* clones, but also for *Type-3* [310]. Moreover, our tool, unlike `SHINOBI` and `Clever`, can be conveniently adopted in both distributed and centralized development environment. Lee et al. [175] used an algorithm based on feature-vector computation over AST and finds the k most similar clones of a given code segment. But their tool was not reported to be integrated with IDE. On the contrary, integration with IDE was a main objective of our work. Moreover, using a suffix-tree-based hybrid algorithm, our tool finds *all* the clones of a chosen code fragment for a given similarity threshold. A similar clone detection technique as of ours was applied by Bazrafshan et al. [32] for approximate code search in program histories.

3.6 Summary

To facilitate proactive clone management, clone detection must be integrated with the regular software development process. Since a developer typically works inside IDE, clone detection facility must be integrated with the IDE. Instead of overwhelming the developers by reporting all clones from the entire code base, the integrated clone detector must allow the *focused* search of clones of a particular code fragment that the developer is interested at some point.

In this chapter, we have presented an IDE-integrated *focused clone search* tool implemented based on a suffix-tree-based k-difference hybrid algorithm. Our novel approach for clone detection exploits the benefits of both AST-based techniques as well as text-comparison-based techniques, and is capable of detecting exact (i.e., *Type-1*) and near-miss (i.e., *Type-2* and *Type-3*) clones of a particular code fragment of the developer's choice made by selection in the editor. The detected clones are reported in the descending order of their similarity with the initially selected *seed* code fragment of the developer's choice.

Using an asymptotic complexity analysis, we have shown that our approach is efficient in terms of both time and memory. Performing a mutation-based evaluation, we have also demonstrated that our tool is highly accurate in terms of precision and recall. This was further confirmed by a case study comparing our tool with `NiCad` [238], a state-of-the-art near-miss clone detector. Moreover, from a user-centric pilot study, we found that our tool is flexibly usable for clone-aware software development and thus clone management. A user manual including instructions for the installation and usage of our clone search tool is available in Appendix B.

Chapter 4

Analysis of Existence and Evolution of Clones

"Details matter, its worth waiting to get it right"
– Steve Jobs

In the last chapter (Chapter 3), we presented our IDE-integrated tool for the detection of both exact and near-miss code clones. While clone detection is a significant phase in the early stage of clone management, we must understand and determine how we should deal with the detected clones. Therefore, we need to study the characteristics of clones and their evolution in software systems to explore patterns and trends that can suggest potential strategies and techniques for clone management. In this chapter, we present empirical studies on the existence and evolution of code clones in open-source and industrial software systems from diverse application domains and written in different programming languages.

The rest of the chapter is organized as follows. In Section 4.1, we introduce our motivation and context. In Section 4.2, we provide the details of the experimental setup and then Section 4.2.5 presents the findings of our study. Section 4.5 shows the threats to the validity of our results. In Section 4.6, we discuss the related studies on clone evolution and attempt to place our work in that context by providing comparative discussion. Finally, Section 6.10 concludes the chapter.

4.1 Introduction

Until recently, many studies on code clones reported observations on the effect of program size and programming language/paradigm on code cloning. However, to develop a more confident understanding on such phenomena, a larger scale structured study with statistical analysis on many *releases* of diverse systems is still required. Most of the other earlier studies [89, 153, 154, 169, 168, 188, 201] on clone evolution investigated how individual cloned fragments evolve across subsequent CVS commit transactions or CVS snapshots over weekly intervals or so.

Although such fine-grained studies offer important insights into the maintenance implications of code clones, a broader picture through analysis of clone densities at *release level* is also necessary to capture more stable characteristics of clone evolution over a longer period. Such an study can provide an overall but quick understanding of clones, which might help in estimating the maintenance cost of the software. Since the

number of cloned code is believed to have significant effect on software development and maintenance effort, project planning and management activities need deeper understanding on how many clones there may be in an evolving software. The ability to predict the amount of clones in the future releases would be useful in taking decisions on software cost estimation and project planning, specially for large and complex systems.

Most of the previous work studied the evolution of *Type-1* (identical code fragments except for variations in whitespace and comments) and/or *Type-2* (where syntactically similar fragments are also considered clones) clones. The studies of Antoniol et al. [7, 8] might have included *Type-3* (statements added/deleted/modified in copied fragments) clones, but their metric based approach for clone detection must have included many false positives.

In this chapter, we present an empirical study on the existence and evolution of exact and near-miss clones over 1,636 *releases* (and pre-releases) of 18 large open source software systems of diverse categories written in three different languages namely Java, C#, and C. To the best of our knowledge, this is the largest study so far on code clone evolution. Moreover, this study includes not only *Type-1* and *Type-2* clones, but also *Type-3*. Our study has two goals: first, using regression analysis we aim to compute one step ahead forecast on clone-density in subsequent releases starting from a very early release of the system. Second, applying statistical methods we investigate cloning property for understanding to what extent previous observations hold over a large number of releases of a wide variety of systems written in diverse programming languages. In particular, we focus on the following research questions:

(a) Using a simple statistical model how accurately can we predict the amount of code clones in future releases?

(b) Is there any common pattern in the evolution of clone-density over releases of evolving software systems?

(c) Is there any significant difference between the existence and evolution of exact clones and near-miss clones?

(d) Do programming languages/paradigms have any effect on the amount and evolution of code clones in the evolving systems? and

(e) How do the sizes of systems and functions affect density of exact and near-miss clones?

A good understanding on these facts should be helpful for project managers in project planning, cost estimation, and planning for clone management strategies. It is believed that the effort required to evolve and maintain a software project is affected by the proportion of code clones present in the current code base and the proportion of clones results in the next version. Hence, an estimation of the probable amount of clones in the next version might be useful for estimating required effort (e.g., cost, risk, time, number of developers) for future development and maintenance activities [7].

Through extensive quantitative analysis and manual investigation over 1,636 releases of the 18 software systems, we draw the following conclusions.

- Using simple regression analysis technique it is possible to make fairly accurate one step ahead forecast of clone-density in future versions of software systems. The forecast errors can indicate the magnitude of irregularities in the creation of clones in the evolving software system.

- Programming language/paradigm is found to have significant effect on code cloning. Java systems are found to have the highest proportion of function clones, C systems have the lowest clone-density, and the C# systems fit in the middle. Systems developed using object-oriented (OO) language/paradigm (Java and C#) have higher proportion of exact clones than near-miss clones, whereas the opposite holds for systems written using procedural C language. During evolution, Java and C# systems exhibit higher variation of clone densities than C systems. Moreover, there exists little or no effect of system's size on the regularity in the evolution of clone-density.

- With the growth of software systems, as the number of functions increases, the number of both exact and near-miss cloned fragments also increases, indicating a very strong positive correlation between them. On the other hand, the correlation between clone-density and number of functions is positive, but fairly weak, and larger systems tend to exhibit less clone-density. As the similarity threshold decreases from near-miss clones to exact clones, the correlation between clone-density and number of functions gradually gets weaker.

- The rate of change in clone-density of near-miss clones is relatively irregular than that of exact clones in all the subject systems regardless of languages/paradigms and types of systems.

- There are some common patterns in the evolution of clone-density across subsequent versions. For instance, relatively higher rate of changes in clone densities is found over early versions of software evolution. In the later releases, there exists long sequences of versions having relatively much less variations in clone densities.

The findings from the study drives us to carry out follow up investigations, one with two industrial web applications and the other based on clone genealogies in open-source software systems.

4.2 Analyzing and Forecasting Near-miss Clones

In this study, we applied the `NiCad` clone detection tool [238] to find *function* level clones in release versions of a number of open source systems. Then we used a clone-density metric, Pearson's correlation coefficient, and regression analysis technique to examine the results. This section introduces the studied systems and describes the methodology and metrics used, including a brief overview of our approach for manual verification of the detected clones.

4.2.1 Clone-density and its Correlation Coefficient

We conducted our analysis using the notion of clone-density, which is the percentage of cloned functions over all functions in the system. Mathematically,

$$clone\ density = \frac{f_c \times 100}{f_c + f_{nc}} \tag{4.1}$$

where, f_c denotes the number of cloned functions, and f_{nc} refers to the number of non-cloned functions.

According to the above definition, having the number of cloned functions unchanged, if the number of total functions increases, clone-density is expected to decrease in the subsequent releases. Intuitively, an increase in the total number of functions also increases the chance of more clones. Hence, the investigation of changes in clone-density necessitates understanding the change relationship among total number of functions, number of cloned functions, and clone-density. To examine this, we used *Pearson product-moment correlation coefficient* [209], which is a well-established statistical measurement to examine linear relationship between variables. We chose to use Pearson coefficient because it is suitable for interval data. The *Pearson product-moment correlation coefficient* r_{xy} between variables x and y is calculated by [6]:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2 \sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{4.2}$$

where x_i and y_i are values of the variables x and y, n is the number of samples (values) available for those variables, $\bar{x}$ and $\bar{y}$ are respectively the mean of all n values of x and y.

The value of r_{xy} ranges between $+1.0$ and -1.0 and indicates to what extent the variables are positively or negatively correlated. Two variables are positively correlated if one increases, then the other also increases. Negative correlation between variables implies if one gets larger, then the other gets smaller. Positive value of r_{xy} implies positive correlation and negative value implies negative correlation. The closer r_{xy} to ± 1.0 the stronger the correlation relationship is. A value of r_{xy} close to zero indicates very weak or no correlation between the variables.

For exact and near-miss clones in the series of releases of each system, we computed *Pearson coefficient* r_{fc} between the number of functions and the number of cloned functions. Similarly, we computed *Pearson coefficient* r_{fd} between the number of functions and clone-density. These two measurements help understand the change of clone-density. Positive correlation between the number of functions and the number of cloned functions indicates that as new functions are created, many of them are possibly created by copy-paste operations, or those functions are accidental clones due to incorporation of similar product feature, or tackling similar problems. Positive correlation between the number of functions and the density of clones implies the number of new clones is higher than the amount of newly created functions.

4.2.2 Forecasting Clone-density

We applied regression analysis [209] for forecasting clone-density in the subsequent versions of the each software system. Using regression analysis we estimated one step ahead clone-density in the subsequent versions starting from the third version of the underlying software system. The model gradually adapts the trend as more data becomes available, and thus error of estimation is expected to be reduced in the forecasts for the subsequent versions.

For each software system in our study, we plotted the clone densities against subsequent versions both for determining the suitable regression function and for identifying the components of the time series. As expected, we did not find any seasonality or cyclic component, but we found randomness and linear trend. Hence, we decided on linear regression model with single predictor variable, written as [208],

$$\hat{y_i} = \hat{\alpha_i} + \hat{\beta_i} x_i$$

where $\hat{y_i}$ is the response (estimation), x_i is the predictor variable (sequence number of a version), $\hat{\alpha_i}$ and $\hat{\beta_i}$ are regression coefficients at the i^{th} estimation step. The values of α_i and β_i (at the i^{th} step) are estimated using the following equations [208]:

$$\begin{aligned}
\hat{\alpha}_{i+1} &= \bar{y_i} - \hat{\beta}_{i+1}\bar{x_i} \\
\hat{\beta}_{i+1} &= \frac{\sum_{k=1}^{i}(x_k - \bar{x_k})(y_k - \bar{y_k})}{\sum_{k=1}^{i}(x_k - \bar{x_k})^2} \\
\bar{x_k} &= \frac{\sum_{j=1}^{k} x_j}{k} \\
\bar{y_k} &= \frac{\sum_{j=1}^{k} y_j}{k}
\end{aligned}$$

where y_i is the actual value (clone-density) observed at step i. The hat ($\hat{.}$) symbol over a variable indicates estimated value. At the i^{th} step we compute the standard error of estimation σ_i using the following formula [6]:

$$\sigma_i = \sqrt{\frac{\sum_{k=1}^{i}(y_k - \hat{y_k})^2}{i}} \tag{4.3}$$

And for a given system s, the average standard error of estimate ξ_s is given by:

$$\xi_s = \frac{\sum_{i \in V_s} \sigma_i}{|V_s|} \tag{4.4}$$

where V_s is the set of all versions of the system s, for which forecasts were made. The regression model tries to find the best fit of the clone-density evolution patterns in the software systems. Hence, forecast errors are expected to be relatively low for those systems having very regular clone evolution patterns. Thus the magnitudes of the forecast errors over subsequent versions indicate how irregular the underlying evolution pattern is.

Table 4.1: Subject systems for the study based on clone-density

Lang.	Systems	Types/domains of systems	Releases		Avg. LOCs per release	Functions (Avg.)	
			Ranges	Total		Total	Sizes
Java	Apache-Ant	Build tool	1.1 to 1.8.0.1	20	80544	7246	11
	ArgoUML	Modeling tool	0.8.1 to 0.30.1-beta	145	136737	10048	14
	Commander4j	ERP system	2.2 to 2.97	37	45164	2373	19
	DavMail	Email client	1.4.0 to 3.6.5-1000	20	8324	469	18
	JasperReports	Reporting tool	0.x.x to 3.7.2	70	81853	5848	14
	JEdit	Editor	30-pre-4 to 43-pre-18	66	74261	4131	18
	Over all six Java systems			358	71147	5019	16
C	Conky	System monitor for X based on torsmo	1.1 to 1.8.0	70	19987	485	41
	GCC	Compiler	1.21 to 4.5.0	79	915438	16661	55
	GIT	Version control system	0.01 to 1.7.0.5	154	89896	2121	42
	Linux kernel	Operating System	0.01 to 2.6.0-test11	459	823021	19359	43
	PostGreSQL	Database	1.08 to 9.0.6	157	391628	9070	43
	Samba	File and print server	1.6.07 to 3.5.2	180	294227	9016	33
	Over all six C systems			1099	422366	9452	43
C#	NANT	Build tool	0.1.3 to 0.90.1	22	27850	1026	27
	CruiseControl	Continuous integration server	0.7 to 1.5.6804.1	24	59905	3545	17
	iTextSharp	Editor	0.01 to 5.0.1.1	39	131657	5691	23
	ProcessHacker	Process viewer and memory editor	1.0.0.0 to 1.3.9.0	35	63385	473	134
	WixEdit	Editor	0.1.1 to 0.7.3	32	10999	556	20
	ZedGraph	Drawing library	1.1 to 5.0.4	27	37702	505	75
	Over all six C# systems			179	55250	1966	49
Total number of release and pre-release versions across all systems is 1,636							

4.2.3 Subject Systems

Table 6.4 describes the 18 software systems, their average sizes, and number of releases we used as subjects of our study. The number of lines presented in the Table 6.4 excludes blank lines and commented lines. Furthermore, we consider .c, .java and .cs files only. Our choice of subject systems was based on three criteria: (a) availability of many release versions over long life span of the systems, (b) size of the systems to be sufficiently large for clone analysis, and (c) diversity in types (i.e., OS, editor, database, etc.) of systems to minimize domain effect on the results.

4.2.4 Clone detection

We used the `NiCad` clone detector [238] for detecting near-miss clones. For consistency with our earlier studies [237, 242], we considered all non-empty functions of at least 3 lines in pretty-printed format. We then use size-sensitive UPI (Unique Percentage of Items) thresholds [238] to find exact and near-miss function clones. For example, if the UPI threshold (UPIT) is 0%, we detect only exact clones; if the UPIT is 10%, we detect two functions as clones if at least 90% of the pretty-printed text lines are the same (i.e., if they are at most 10% different). In this study, we used the representative set of UPITs 0%, 10%, 20% and 30%, corresponding to editing changes of from 0% to 30%, or 0 to 3 lines in every ten. It was not possible to manually validate all the detected clones in all releases of all the systems. We validated the clones of three releases (the first and the last releases and another randomly selected release) for each system and experienced almost no false

positives. We used `NiCad`'s interactive HTML output to obtain an overall view of the original source of the clone classes. Then, we carried out pairwise comparisons on the original source code of the functions in each clone class using Linux *diff*, followed by manual examination with a greater difference limit than the UPIT. Manual validation of precision using this method is both easy and efficient [242], but measuring recall over all the functions was not possible due to the difficulty level of the approach. However, `NiCad`, as a clone detector, was reported to have high recall [240] and precision [238, 242].

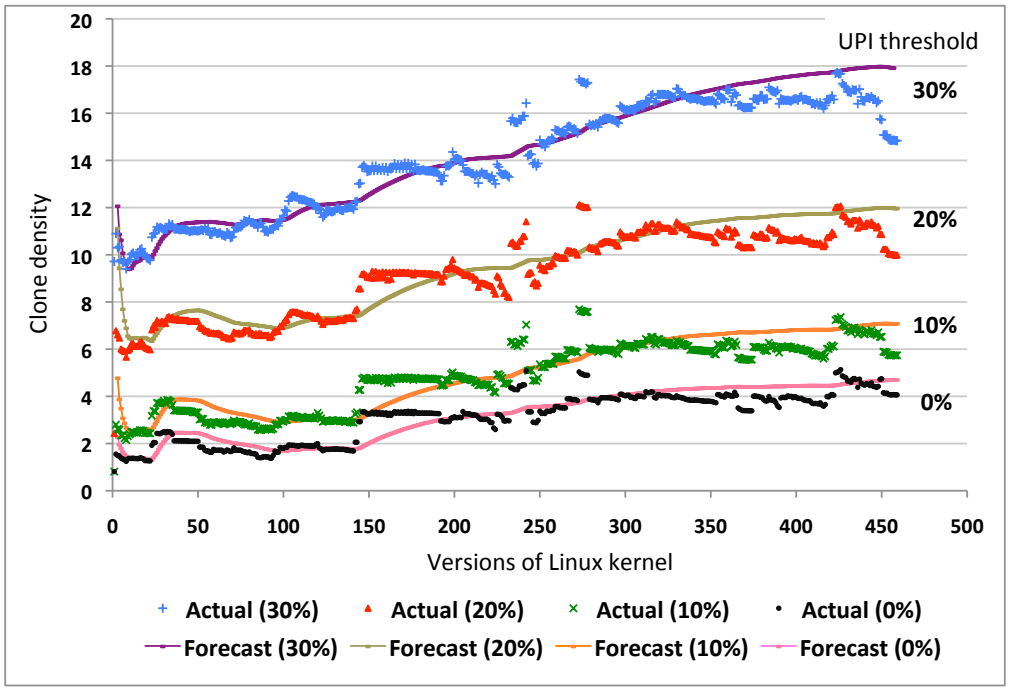


Figure 4.1: Actual and forecasted density of exact and near-miss clones in Linux kernel

4.2.5 Findings

For every release of each of the 18 systems, we computed density of clones (using equation 4.1) at different UPITs. Then for each system, at each different UPIT, we computed the average density of clones over all releases, as presented in Table 4.2. Over all releases of each system, we computed the *Pearson correlation coefficient* between number of functions and clones (r_{fc}), as well as between number of functions and clone-density (r_{fd}), as shown in Table 4.4. Applying the regression analysis model, we made one step ahead forecast on clone-density in subsequent releases of the systems. For each release of a system we calculated the total number of functions, the number of cloned functions, the actual and forecasted clone densities.

We compared each forecasted clone-density with the actual density and computed the standard error of estimate using Equation 4.3. For each of four different UPITs, using Equation 4.4, we calculated the average standard error of estimate over all releases of a system. The regression analysis model enabled us to obtain one step ahead forecast on clone densities with reasonable level of accuracy. The standard error of estimation

Table 4.2: Average clone density in each system

Prog. Lang.	Subject Systems	Densities for UPI threshold			
		30%	**20%**	**10%**	**0%**
Java	Apache-Ant	20.43	17.49	15.99	15.73
	ArgoUML	26.79	21.59	16.85	15.83
	Commander4j	43.76	38.33	33.98	32.63
	DavMail	13.74	6.93	4.42	3.64
	JasperReports	40.39	38.18	35.32	33.72
	JEdit	13.57	9.20	5.98	5.69
	Average	26.45	21.95	18.76	17.87
C	Conky	9.04	5.96	2.63	1.98
	GCC	18.61	13.93	9.31	6.77
	GIT	2.92	1.74	1.03	0.84
	Linux Kernel	14.15	9.19	4.85	3.14
	PostGreSQL	13.01	7.27	3.03	1.93
	Samba	15.39	8.84	3.28	1.78
	Average	12.19	7.82	4.02	2.74
C#	NANT	16.73	12.63	8.69	8.42
	CruiseControl	10.87	6.80	4.46	4.38
	iTextSharp	18.43	15.05	12.59	11.76
	ProcessHacker	7.32	3.20	2.17	2.17
	WixEdit	15.44	12.11	7.65	6.93
	ZedGraph	17.19	13.89	9.67	9.33
	Average	14.33	10.61	7.54	7.17
Average		17.65	13.46	10.11	9.26

Table 4.3: Clone density categorized by system-size

Sizes (LOC/release)	Densities for UPI threshold			
	30%	**20%**	**10%**	**0%**
Below 50K	19.32	14.97	11.17	10.49
Between 50K and 100K	15.92	12.77	10.83	10.42
Above 100K	17.73	12.64	8.32	6.87

Table 4.4: Pearson coefficients r_{fc} and r_{fd}

Prog. Lang.	**Subject Systems**	r_{fc}		r_{fd}	
		30%	**0%**	**30%**	**0%**
Java	Apache-Ant	0.999	0.997	0.75	0.53
	ArgoUML	0.81	0.63	-0.22	-0.15
	Commander4j	0.998	0.99	0.67	0.50
	DavMail	0.98	0.97	0.88	0.92
	JasperReports	0.999	0.998	0.87	0.87
	JEdit	0.95	0.89	-0.55	0.42
C	Conky	0.89	0.12	0.47	-0.24
	GCC	0.99	0.99	0.94	0.91
	GIT	0.80	0.48	-0.76	-0.59
	Linux Kernel	0.995	0.99	0.69	0.68
	PostGreSQL	0.98	0.83	0.78	0.03
	Samba	0.97	0.91	0.89	0.16
C#	NANT	0.996	0.99	0.81	0.62
	CruiseControl.NET	0.84	0.23	0.02	-0.39
	iTextSharp	0.997	0.99	0.83	0.81
	ProcessHacker	0.997	0.99	-0.45	0.84
	WixEdit	0.94	0.88	0.38	0.39
	ZedGraph	0.92	0.84	0.92	0.82

averaged over all four levels of UPITs and all the systems is 2.35. Table 4.5 presents the average forecast errors at different UPITs for all the 18 systems. Figure 4.1 plots the actual and forecasted density of clones in all 459 releases of Linux kernel (the system in our study having the highest number of releases).

Table 4.5: Average forecast errors (ξ_s) for each subject system

Prog. Lang.	**Subject Systems**	**ξ_s for UPI threshold**				**Avg.**
		30%	**20%**	**10%**	**0%**	ξ
Java	Apache-Ant	2.61	2.27	2.20	2.10	2.29
	ArgoUML	5.01	5.30	5.38	5.24	5.23
	Commander4j	0.44	0.46	0.60	0.56	0.52
	DavMail	2.23	2.00	1.25	1.20	1.67
	JasperReports	6.41	6.48	6.53	6.49	6.48
	JEdit	0.43	0.44	0.38	0.37	0.41
	Average	2.86	2.82	2.72	2.66	2.77
C	Conky	2.18	1.83	1.27	1.20	1.62
	GCC	1.93	1.64	1.43	1.09	1.52
	GIT	1.44	1.31	1.05	1.07	1.22
	Linux Kernel	0.55	0.85	0.62	0.46	0.62
	PostGreSQL	0.92	0.87	0.78	0.69	0.82
	Samba	2.89	2.72	2.32	2.23	2.54
	Average	1.65	1.54	1.25	1.12	1.39
C#	NANT	4.05	4.07	3.56	3.60	3.82
	CruiseControl	2.29	2.10	1.77	1.76	1.98
	iTextSharp	4.55	4.62	4.01	3.74	4.23
	ProcessHacker	0.09	0.15	0.15	0.15	0.13
	WixEdit	4.48	4.76	4.36	3.73	4.34
	ZedGraph	3.32	3.13	2.68	2.61	2.93
	Average	3.13	3.14	2.76	2.60	2.91
Overall		2.55	2.50	2.24	2.13	2.35

Growth of Clones

For all the systems, with the growth of the source code, the number of clones also tends to increase. As can be observed in Table 4.4, for all the subject systems, the Pearson correlation coefficients (r_{fc}) between the number of functions and the number of functions clones are quite high, and for clones detected at UPIT 30% (and for some systems, clones at UPIT 0%), the correlation coefficients are found to be above 0.9

and close to 1.0 for some systems. Such high Pearson correlation coefficients suggest that at almost all the transitions between subsequent releases, whenever the number of functions increases or decreases in the source code, the number of function clones also increases or decreases in the same direction. The plot in Figure 4.2 demonstrates the phenomena with respect to Apache-ANT for which both the Pearson correlation coefficients (i.e., r_{fc} and r_{fd}) are above 0.99.

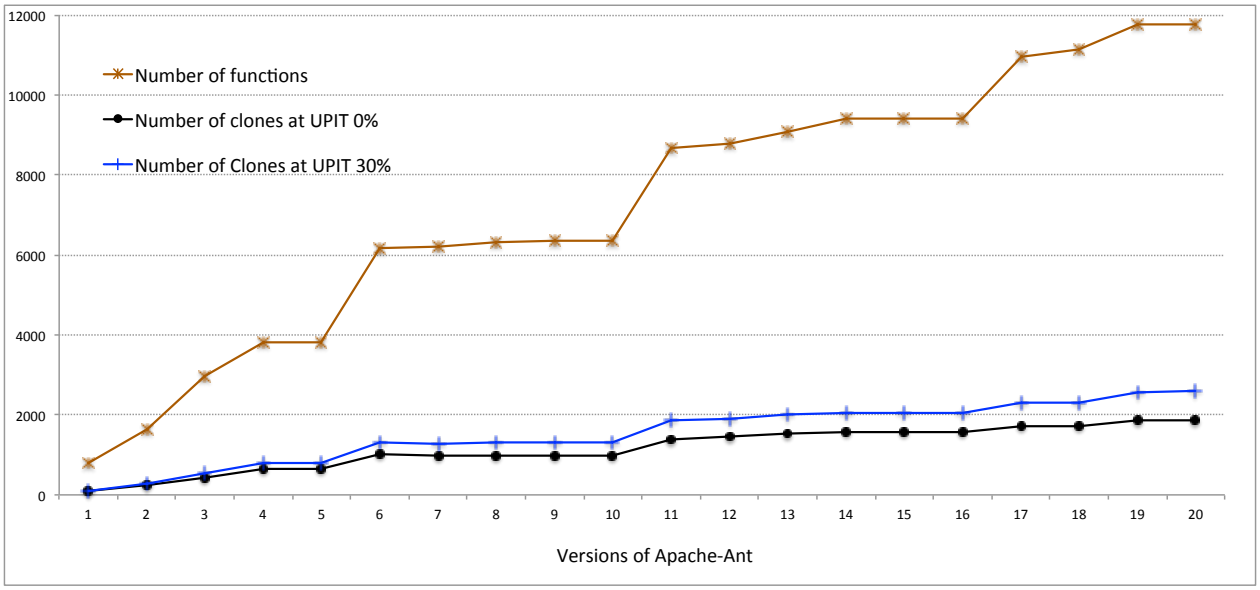


Figure 4.2: Number of functions and clones in subsequent releases of Apache-ANT

As we consider the Pearson correlation coefficient (r_{fd}) between the number of functions and clone-density (right-most two columns in Table 4.4), we find relatively lower coefficient values, still most positive values with a few negative values for some of the systems (e.g., ArgoUML, Jedit, Conky, GIT, CruiseControl.NET and ProcessHacker). Thus, we can interpret that with some exceptions, in most of the cases, the proportion of clones (i.e. clone-density) also increases (or decreases) with the increase (or decrease) in the number of functions in the source code during transitions between subsequent releases.

Effect of Programming Language/Paradigm

In Figure 4.3, we present the density of exact and near-miss clones averaged over all systems categorized by programming languages. The standard deviations of the distributions are also marked by the respective horizontal bars. We found that for all of the four UPITs, Java systems have the highest clone-density, C systems have the lowest clone-density, and C# systems fall in between. This finding is consistent with that reported by Roy and Cordy [242], where they mentioned that the existence of accessor methods/functions in Java and C# systems might be a possible reason for object-oriented systems to have higher number of clones. Morever, due to the presence of encapsulation, object-oriented source code may have similar functions defined in different classes, if appropriate abstraction mechanism was ignored or not possible.

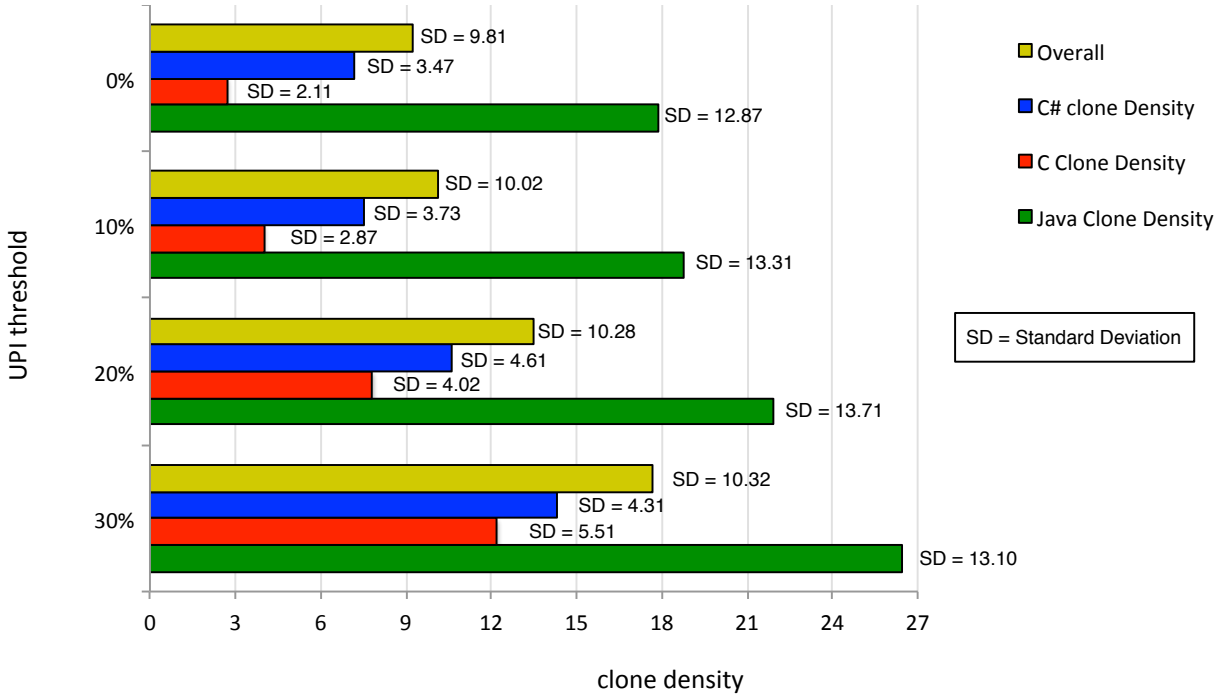


Figure 4.3: Average density of exact and near-miss clones at different UPITs (dissimilarity levels)

The average function size in the Java systems in our study is much lower than those in the systems written in C and C# (Table 6.4). The effect of programming paradigm (OO versus procedural) or practice might have caused such differences. The existence of many getter and setter methods in the OO systems might also have played a role. Intuitively, the smaller the sizes of the functions, the higher the the statistical probability of their being clones. This may be another reason for comparatively higher clone-density in Java systems in our study. As the UPIT increases from 0% to 10%, 20%, and 30%, the density of clones largely increases in C systems. On the contrary, exact clones dominate in Java and C# systems, as with the increase of UPIT clone-density does not increase much for these systems.

To examine the statistical significance of the impact of programming languages over clone densities, we conducted *Kruskal-Wallis* [6] tests with significance level α set to 0.05. We separately applied the statistical test over the densities of clones categorized by languages at each of the four UPITs (i.e., 0% to 10%, 20%, and 30%). The P-values obtained from the tests are presented in Table 4.6.

Table 4.6: P-values from *Kruskal-Wallis* tests over the densities of clones categorized by languages

UPIT	**0%**	**10%**	**20%**	**30%**
P-value	0.01	0.034	0.071	0.097

As we interpret the results reported in Table 4.6, the differences in the average densities of clones in the systems categorized by the underlying programming languages are found to be statistically significant for

exact (*Type-1*) clones and for near-miss clones at UPIT 10% ($p < \alpha$). However, for the clones with higher dissimilarities (i.e., at UPITs 20% and 30%), the differences do not appear to be statistically significant ($p > \alpha$).

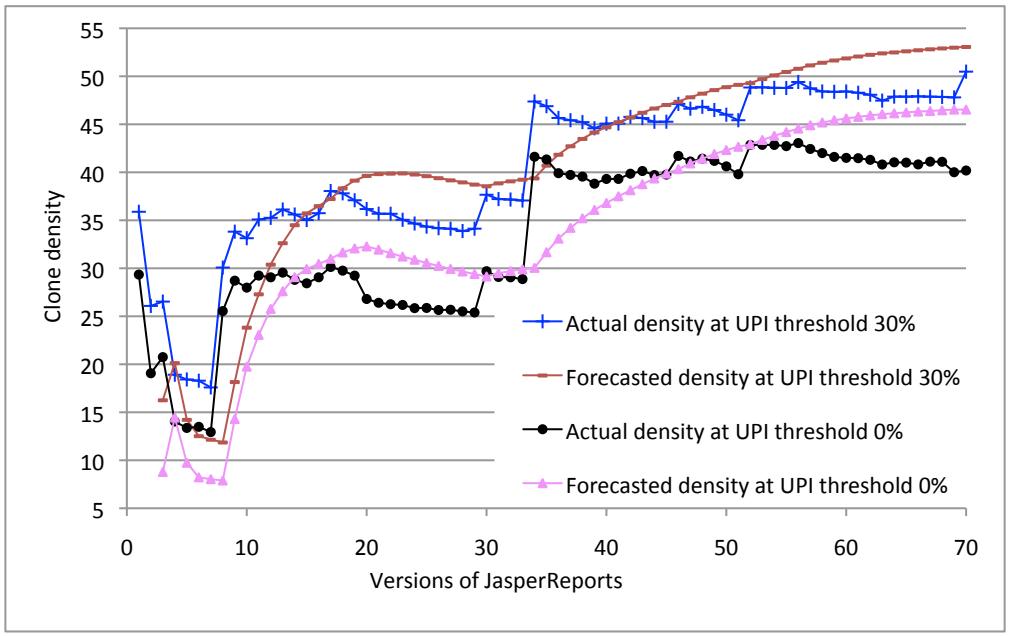


Figure 4.4: Actual and forecasted density of exact and near-miss clones in JasperReports

Regularity in Evolution Patterns

The Java system JasperReports contributed the most in the standard error of estimate mentioned above. For this system, the average standard error of estimate is the highest (6.48), which also indicates high irregularities in its clone evolution, as shown in Figure 4.4. To keep the figure legible, we plotted the actual and forecasted density of clones only for exact clones and near-miss clones with threshold 30%. The *Pearson coefficients* r_{fc} and r_{fd} (Table 4.4) indicates that over subsequent releases of JasperReports as the number of functions increased or decreased, the number of clones as well as clone-density also changed in the same direction. This suggests that the fluctuations in JasperReports' clone-density is due to major changes in the number of functions in the subsequent releases. This is further validated by the data for individual releases. For example, we found that between JasperReports 0.2.5 and JasperReports 0.3.1 the number of functions increased from 796 to 1,147 and the number of exact cloned functions increased from 103 to 293 resulting increase in clone-density from 12.9% to 25.5%. Looking at the change log for JasperReports 0.3.1, we found that besides bug fixes and architectural improvements from earlier release, support for XML output was added in release 0.3.1, which may be a reason for such large increase in number of functions and clones.

Effect of Programming Languages on Irregularity of Clone Evolution: As we see in Table 4.5, the forecast errors averaged over C systems are much lower (1.39) than that of Java and C# systems. For

Java and C# systems the average forecast errors (2.77 and 2.91 respectively), are more than twice as for C systems. This implies that the clone evolution patterns in Java and C# systems are more irregular than in C systems. However, among Java systems JasperReports and ArgoUML produced the highest average forecast errors, 6.48 and 5.23 respectively. On the other hand, WixEdit and NANT among all C# systems produced the most forecast errors, 4.34 and 3.82 respectively.

Table 4.7: Avg. forecast error (ξ) categorized by system-size

Sizes	**Avg. ξ for UPI threshold**			
(LOC/release)	**30%**	**20%**	**10%**	**0%**
Below 50K	2.78	2.71	2.29	2.15
Between 50K and 100K	2.29	2.21	2.05	2.01
Above 100K	2.64	2.67	2.43	2.24

Effect of Program Size on Irregularity of Clone Evolution: Table 4.7 presents the forecast errors averaged over systems categorized by their sizes. We see that for all four UPITs systems of sizes between 50 KLOC/release and 100 KLOC/release exhibit the least average forecast errors. For UPIT 0% and 10% the systems of sizes above 100 KLOC/release yield more forecast errors than systems having less than 50 KLOC/release. On the contrary, for UPITs 20% and 30% we found the opposite. Hence, it is difficult to say anything about the effect of system size on the regularity of clone evolution, and further investigation may be required in this regard.

As the UPIT increases from 0% to 30%, for most of the systems across all three languages the average forecast errors (Table 4.5) also increase indicating that the evolution of near-miss clones is more irregular than that of exact clones. The reason may be the fact that any two fragments have higher chance of being similar than being exactly same, and so, creation or deletion of a functions may have higher effect on the amount of near-miss clones than exact clones. Possibly, due to the same reason, for almost all systems, as UPIT increases from 0% to 30%, the *Pearson coefficient* r_{fc} (between number of functions and number of cloned functions) gets closer to +1.0 (Table 4.4). We see that for all systems, r_{fc} is positive for both exact and near-miss clones. For all systems r_{fc} is +0.80 or more for near-miss clones (UPIT 30%), which suggests a very strong positive correlation between the number of functions and the number of near-miss cloned functions. In case of exact clones, r_{fc} is +0.5 or higher for 15 out of 18 systems, which also suggests fairly strong positive correlation.

On the other hand, the value of r_{fd} (correlation coefficient between the number of functions and clone-density) is +0.5 or higher for 11 out of 18 systems at UPIT 30%, and for four systems r_{fd} is negative. This may be interpreted with the fact that for the near-miss clones at UPIT 30%, there exists weak positive correlation between the number of functions and clone-density. For exact clones half of the systems exhibit r_{fd} values less than +0.5, and four of the systems has negative r_{fd}. This shows relatively weaker positive

correlation between number of functions and density of exact clones. We also see that with the decrease in UPIT from 30% to 0%, positive correlation between number of functions and clone-density gets weaker. However, looking at average clone densities categorized by systems' size (Table 4.3) we see that for UPIT 0%, 10%, and 20% average clone-density decreases as the systems' size increases from below 50 KLOC to over 100 KLOC. This further weakens the positive correlation between number of functions and clone-density. A possible explanation may be that large systems in our study tend to have functions of larger sizes, and larger functions have statistically less probability of being clones than smaller functions. In our study the four largest systems are C systems namely GCC, Linux kernel, PostGreSQL, and Samba having average function size of 43 LOC, much higher than average Java function size (16 LOC). For instance, the "void ffebld_constantarray_prepare (...)" function in "gcc/f/bld.c" source file of GCC-3.0.1 has 361 pretty-printed LOC consisting of long sequences of switch statements.

Clone Evolution Patterns

Looking at the clone densities over all the systems across subsequent releases, we found interesting patterns in the evolution of clone-density. For each system, clone densities for the four UPITs exhibit the same evolution pattern varying only in their magnitudes. For example, Figure 4.1 plots clone densities at UPITs 0%, 10%, 20%, and 30% with roughly parallel lines. We also observed that for most of the systems, the density of exact clones does not differ much from the density of clones at UPIT 10%. In Apache-Ant, NANT, JEdit, ArgoUML, GIT, GCC, ProcessHacker, WixEdit, CruiseControl.NET, and ZedGraph clone densities at UPIT 10% is found to be almost equal to the densities of exact clones. For other systems, we found differences, which is much lower than the differences of clone densities between UPITs 10% and 20%, or between threshold 20% and 30%.

Having some short term fluctuations of increase and decrease, over long term clone-density tend to increase linearly in Linux kernel (Figure 4.1). Previous studies on Linux kernel reported roughly linear (super linear) growth of Linux kernel source code in terms of LOC [95]. The linear growth of both line of source code as well as density of code clones indicate that cloning activities continued in a similar pace as the development and maintenance activities took place in Linux kernel code base. Similar geometric evolution of clone-density is found in JEdit. For this system, at all four UPITs clone-density gradually increased over the early 38 releases between JEdit-3.0-pre4 and JEdit-4.2-pre2. Then over the later 28 releases between JEdit-4.2-pre3 and JEdit-4.3-pre18 clone-density decreased. However, we found roughly linear growth in the number of functions over all releases of JEdit. This suggests possibly more copy-paste-modification activities took place during the early development of JEdit due to active development of similar features implementation, and as the project became stable, later development activities were possibly dominated by maintenance, and performance enhancement rather than new feature incorporation.

For all the other 16 systems (except Linux kernel and JEdit), we observed some distinguishing patterns in the evolution of clone densities. In all these 16 systems, we noticed major change in clone densities over early

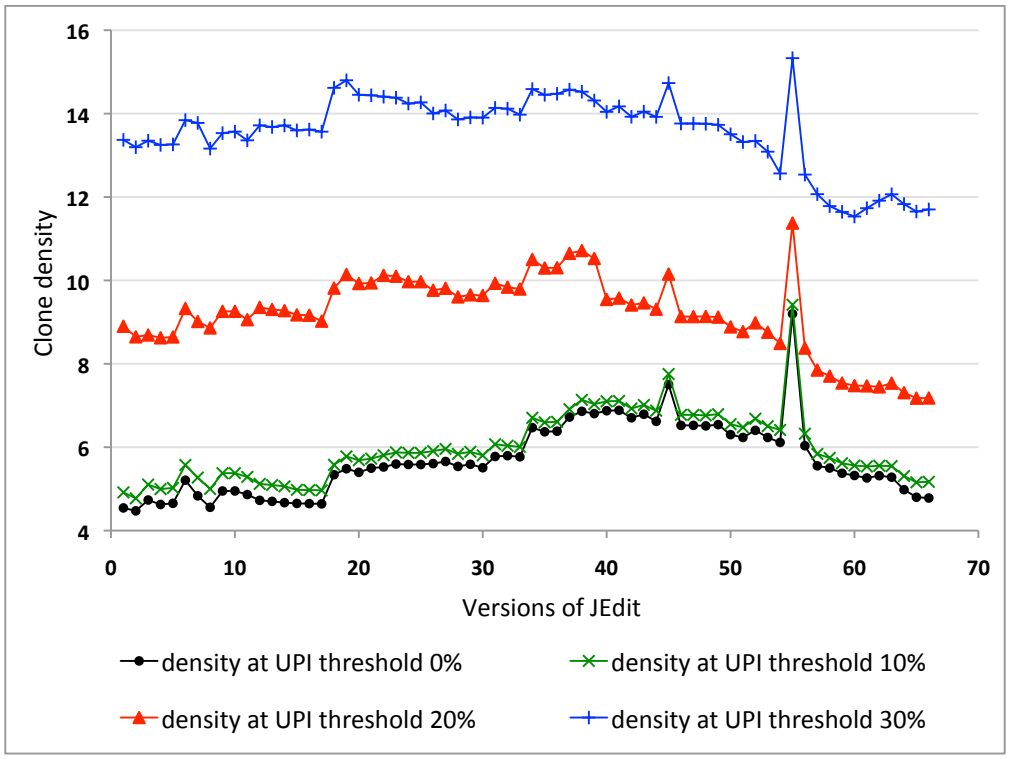


Figure 4.5: Exact and near-miss clone-density in JEdit

releases, and relatively smaller variations over sequence of many later releases. A reason to this fact may be during the early releases active development and feature implementations dominated in software growth, whereas maintenance activities dominated in the later releases. As shown in Figure 4.6, clone-density strictly increased in Apache-Ant from release 1.1 (10.6% exact clones) to release 1.4 (16.8% exact clones) and did not vary much (amount of exact clones remained between 15.38% 16.8%) over the later 16 releases. Such pattern was also found in Commander4j. Similar pattern of increasing clone-density was also found in DavMail, Conky, NANT, and WixEdit. However, in these systems, the early change in not strictly increasing, rather overall increase with short term fluctuations is found. For example, clone-density in NANT (Figure 4.7) increased from release 0.1.3 (5.1% near-miss clones at UPIT 30%) to 0.8.3 (20.9% near-miss clones at UPIT 30%), and did not exhibit much variations over the later releases (density of near-miss clones at UPIT 30% remained between 20.9% and 23.4%). However, between NANT-0.1.3 and NANT-0.8.3, there exists some fluctuations of increase and decrease in clone-density. From NANT-0.1.4 clone-density increased in NANT-0.1.5, again decreased through release 0.5 and 0.6, which followed an increased density in NANT-0.7.7. This is possibly due to active development and refactoring activities during those early releases of NANT.

A *second pattern* of clone evolution during early developments was found in CruiseControl.NET, ArgoUML, GIT, JasperReports, PostGreSQL, and ZedGraph. For all these systems, during the early releases, clone densities tend to decrease along with some short term fluctuations. Figure 4.8 shows this pattern plotting the evolution of clone-density for CruiseControl.NET. For this system, as we see, clone-density largely

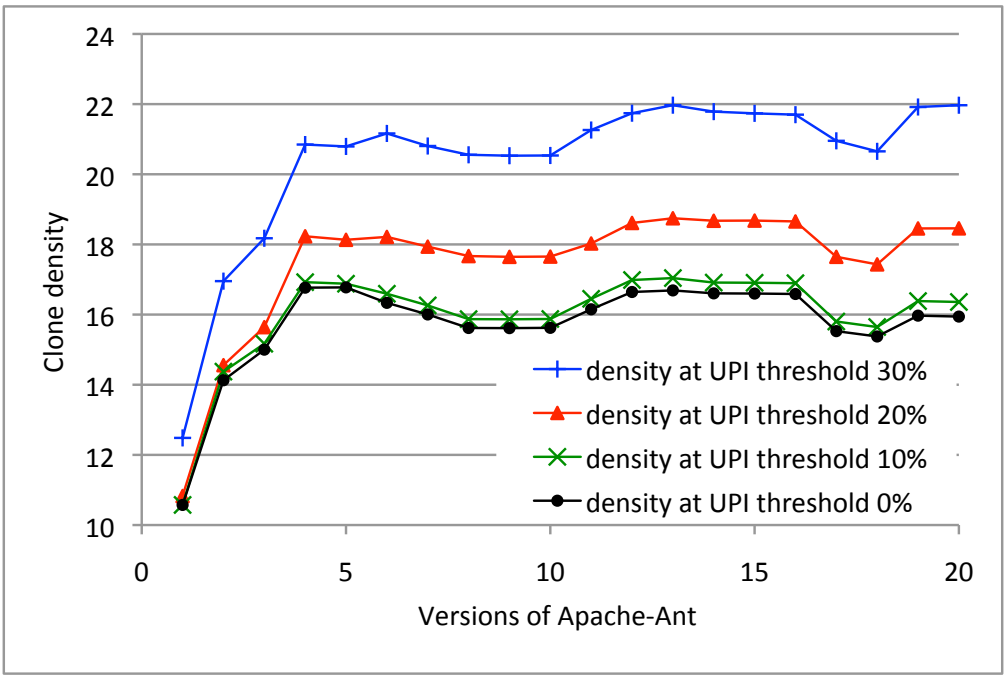


Figure 4.6: Exact and near-miss clone-density in Apache-Ant

decreased from release 0.7 (9.6% exact clones) through release 1.0 (2.8% exact clones), and over later 16 releases (release 1.0.1 through release 1.5.0) density of exact clones remained between 2.3% and 3.6%. However, we found that over those releases of CruiseControl.NET, the number of functions increased from 2,473 to 2,859, which indicates the possibility of not much copy-paste-modification activities took place during the early developments of CruiseControl.NET.

After the early phase of software evolution, over long term relatively less variation in clone-density was found in general. However, two interesting patterns are found. The first pattern exhibits roughly steady clone-density over long sequence of releases with small irregular variations. Such pattern is found in WixEdit, iTextSharp, DavMail, Commander4j, Ant (Figure 4.6), NANT (Figure 4.7), CruiseControl.NET (Figure 4.8), and JasperReports (Figure 4.4). In the second pattern, clone-density is found not to vary that much over long sequence of releases, and relatively large changes found between such release sequences. Similar pattern is found in GCC, ArgoUML, GIT, JasperReports, PostGreSQL, Samba, and ProcessHacker. Scatter plot of such pattern yields a stair-like shape, as shown in Figure 4.9, which shows exact and near-miss clone densities in 79 releases of GCC.

There are two possibilities to cause such steady clone-density: when neither the number of functions nor the number of cloned fragments changes, or when both changes in a ratio such that clone-density remains steady. Interestingly, both of these facts are found in systems exhibiting such patterns. For example, density of exact clones remained between 3.78% and 3.84% over seven releases of GCC, GCC-2.8.0 through GCC-2.95.3. Among these releases, between GCC-2.8.1 and GCC-2.95 the number of functions increased from 5,893 to 11,268, and the number of cloned functions also increased from 223 to 433 causing no significant

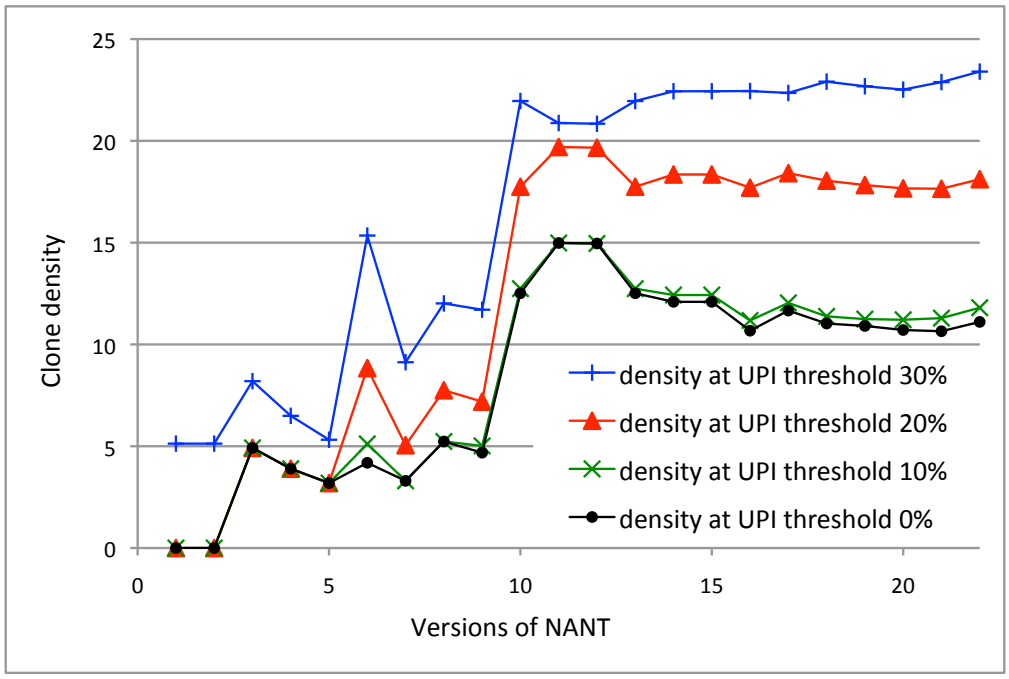


Figure 4.7: Exact and near-miss clone-density in NANT

change in clone-density. Again, from GCC-2.95.3 to GCC-3.0 the number of functions increased from 11,300 to 15,936, and number of cloned functions increased from 431 to 1161, causing the clone-density jump from 3.81% to 7.28%. Over the following four releases, GCC-3.0.1 through GCC-3.0.4 the total number of functions varied a bit between 16,016 and 16,134. The number of cloned functions remained unchanged at 1,170 and consequently the clone-density varied a little between 7.25% and 7.31%.

Besides the above mentioned patterns in the evolution of clone-density in the later releases, some systems exhibit sudden significant increase in clone densities between subsequent releases followed by large decrease in the next release. Such sudden variations are found in JEdit, ArgoUML, GIT, ZedGraph, and Samba. For instance, the large spike in Figure 4.5 corresponds to JEdit-4.3-pre6 between JEdit-4.3-pre5 and JEdit-4.3-pre7. In JEdit-4.3-pre6, the number of clones increased significantly compared to its previous and later releases.

Clone Propagation Across Releases

In an earlier work [153], Kim et al. tracked individual clone-groups (genealogies and lineages [153]) over releases and characterized them in terms of dead, alive, consistently changed, and syntactic similar genealogies. However, the length of the succession of individual clone-groups over releases is yet to be investigated to understand how persistent the clone-groups are across releases. A clone-group in a later release is said to be a successor of a clone-group of an earlier release, if the code snippets in the earlier release do not change more than a threshold (30% lines in our study) in the later release. The length of succession of a clone-group is the maximum number of subsequent releases in which a successor of that clone-group exists.

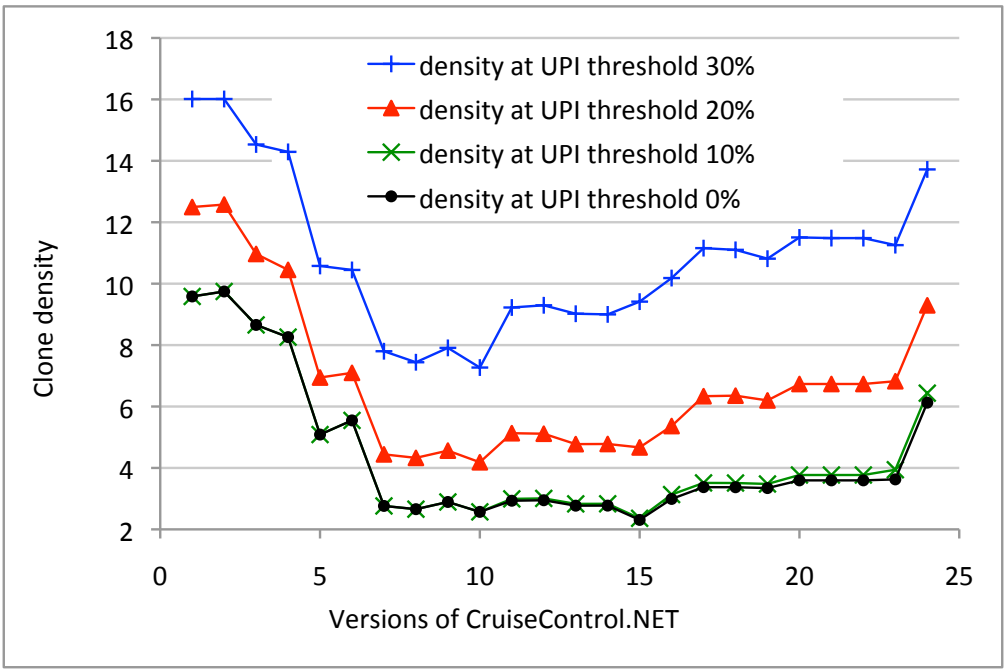


Figure 4.8: Exact and near-miss clone-density in CruiseControl.NET

For the two Java systems DavMail and Apache-Ant, we tracked individual clone-groups, and found that a significant amount of clone-groups have succession length over 60% of the total number of releases. In DavMail 38% of the clone-group successions have length 80% of the releases or more. In Figure 4.10, we present an example of such a succession of near-miss clones in releases of DavMail. The clone-group consisting of the two shaded functions in the figure have succession spanned over releases 2.0.0 through 2.1.1. The other two functions joined the clone-group in release 3.0.0, and the succession continued to release 3.6.5 resulting a total succession length of 18 releases out of 20.

4.3 Genealogy-based Investigation of Clone Propagation

The long succession of some clone-groups as observed from the study described above, inspired us to follow it up with a genealogy-based study[1] to obtain a deeper understanding over the longevity of evolving individual clone-groups. As subjects to our study, we used releases of 17 open-source software systems (Table 4.8) written in four different programming languages namely C, C++, Java, and C#.

[1]This genealogy-based study was performed jointly with Ripon K. Saha and Muhammad Asaduzzaman, who were two M.Sc. students at the Department of Computer Science, University of Saskatchewan, Canada. As this is a joint work, we kept its description concise. This particular work of ours appeared in a peer-reviewed conference [248], where further details of the study can be found.

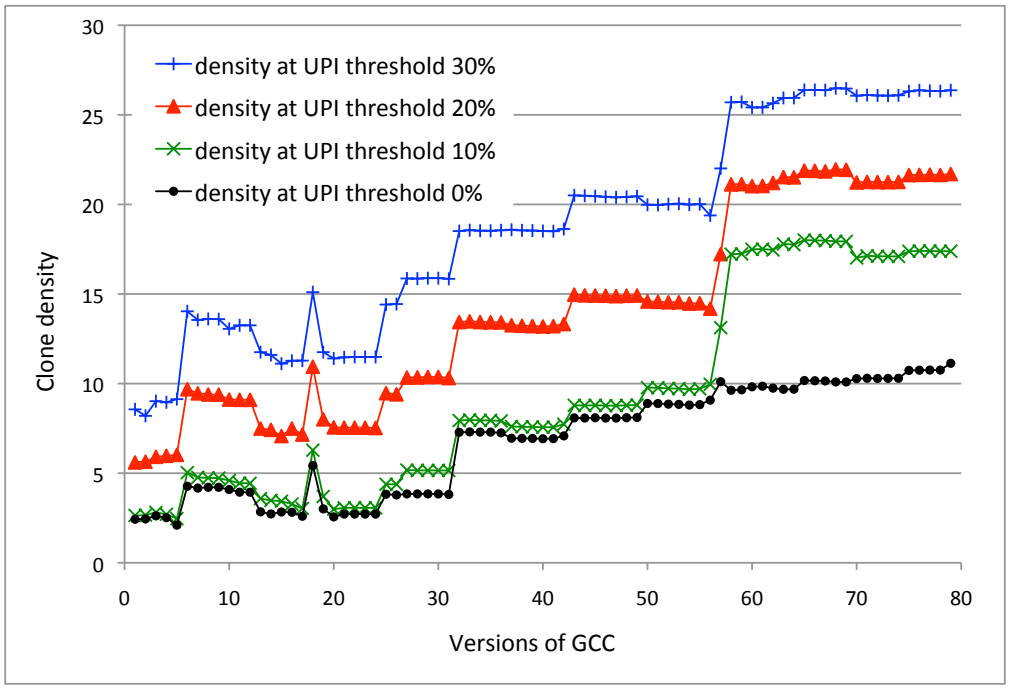


Figure 4.9: Exact and near-miss clone-density in GCC

We used `CCFinderX`[2] to detect code clones in each release of the software systems under study. Then for each of the subject systems, to construct the clone genealogies, we mapped the clones-groups detected from successive releases using based on textual similarity of the code snippets in addition to a snippet-matching technique that match code snippets by comparing the names of identifiers. Finally, we characterize the extracted clone genealogies according to the change patterns described in Section 2.6.2 of Chapter 2. The size number of total and different categories of clone genealogies found in each of the subject systems are presented in Table 4.8.

From our analysis of the characterized clone genealogies, we found many alive and long-lived genealogies while many of the genealogies disappeared within a few early releases. 47.57% of the alive genealogies spanned over at least half of the total number of releases of the respective software systems. In most of the clone genealogies extracted from the subject systems, clone-groups propagated across releases either without any change or with changes only in the names of identifiers. Hence, it is possible that these types of genealogies do not need any extra care during software maintenance. Also, they are less likely to be removed from the systems, and on average almost 69% of them reached to the final release. As many as 11% to 38% of all the extracted genealogies experienced consistent changes. Moreover, on average nearly 67% of total alive genealogies did not experience any addition or deletion of lines, or renaming of identifiers.

[2] `CCFinderX` is an extended version of the clone detector `CCFinder` [137]

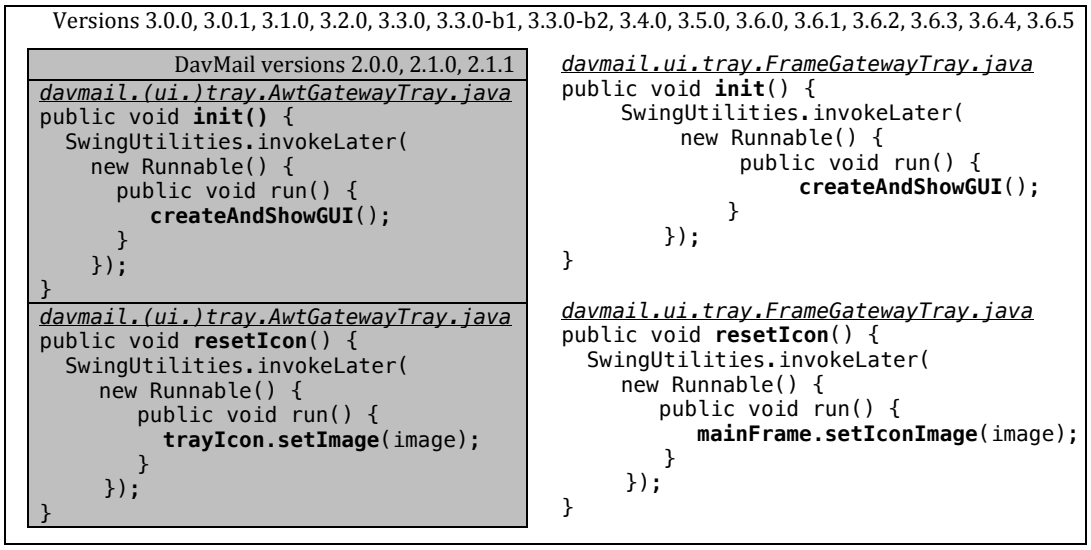


Figure 4.10: Group of near-miss function clones in releases of DavMail

Table 4.8: Genealogy-based study on clone evolution

Prog. Lang.	Subject System	Size (LOC)	Duration (yyyy-mm-dd) From	To	No. of rel.	Genealogy statistics Total	AG (%)	DG (%)	SSG (%)	CCG (%)
Java	JUnit	2,179–8,785	2003-05-12	2009-12-08	20	127	78.74	21.26	81.89	15.75
	CAROL	2,812–11,694	2002-11-12	2005-04-13	10	141	44.68	55.32	56.73	38.30
	dnsjava	11,025–23,334	2001-03-29	2009-11-21	22	417	82.97	17.03	85.37	12.23
	JabRef	11,352–74,104	2003-11-30	2010-04-14	33	1132	73.41	26.59	66.25	26.06
	iText	51,860–82,164	2002-03-07	2008-01-25	49	1568	68.75	31.25	74.62	20.22
	Average for systems written in Java				26.80	677	71.43	26.57	72.67	21.77
C++	KeePass	14,789–43,644	2003-11-17	2006-10-14	35	790	70.76	29.24	73.54	20.63
	Notepad++	26,937–81,980	2003-11-25	2007-02-04	30	977	81.99	18.01	73.69	19.86
	7-Zip	71,638–100,823	2003-12-11	2009-02-03	45	1427	65.38	34.62	64.62	24.46
	eMule	6,803–203,780	2002-07-07	2010-04-07	73	3547	66.08	33.92	59.57	29.86
	Average for systems written in C++				45.75	1685.25	68.79	31.21	64.32	26.18
C	Wget	14,209 - 40,021	1998-09-23	2009-09-22	17	206	57.77	42.23	60.68	28.64
	Conky	7,029–42,060	2005-07-20	2010-03-30	70	1328	53.69	46.31	82.45	11.97
	ZABBIX	12,468–70,890	2004-03-23	2010-01-27	28	1026	65.79	34.21	49.31	28.65
	Claws Mail	126,247–203,783	2005-03-19	2010-01-31	47	2363	85.57	14.43	63.26	27.08
	Average for systems written in C				40.50	1230.75	76.00	24.00	77.55	16.84
C#	NAnt	686–52,533	2001-07-19	2007-12-08	22	625	76.96	23.04	55.20	34.08
	iTextSharp	33,545–163,890	2003-02-04	2007-03-08	26	2666	77.16	22.84	86.38	10.43
	ProcessHacker	10,349–123,878	2008-10-17	2010-01-23	38	950	71.79	28.21	73.26	20.32
	ZedGraph	2,439–26,433	2004-08-02	2008-12-12	28	374	76.74	23.26	62.83	24.87
	Average for systems written in C#				28.50	1153.75	68.75	31.25	74.62	20.22
Avg. over all the Systems					34.88	1156.71	70.33	29.67	66.56	24.28
Total number of releases across all the subject systems is 593										

Here, rel. = releases, AG = Alive Genealogy, DG = Dead Genealogy, SSG = Syntactically Similar Genealogy, CCG = Consistently Changed Genealogy

4.4 Clones in Industrial Web Applications

Unlike the traditional software applications written in a particular language, web applications these days typically have multilingual implementations, where a single source file may contain tangled snippets of source code written in different programming languages such as HTML, PHP, and JavaScript. Most of the earlier studies (including ours as described in Section 4.2 and Section 4.3) on the analysis of code clones focused on open-source traditional applications written in a particular programming language such as Java, C, C++, and C#. While such studies provide useful insights into clones in those kinds of software systems, relatively fewer research aimed to investigate clones in web applications, even though web applications are becoming more ubiquitous these days than in the past.

Those few studies [194, 229, 231] on the analysis of code clones in web applications mostly analyzed only *Type-1* and *Type-2* clones or the set of clones based on the authors' distinct definitions of code similarity. Moreover, those studies were limited in the investigation of clones only in *static* HTML pages, or in the reference graph derived from the links among different pages and resources.

Thus, we became motivated to study clones in industrial web applications, in particular to understand if there is any significant difference between web applications and traditional software systems in terms of the existence of clones and patterns of cloning. In this study[3], investigate the patterns of both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) code clones in two industrial web applications: a `Training Registration System (TRS)`[4] and an `Incident Reporting System (IRS)`[5].

The systems are developed at the Information and Communications Technology division of the University of Saskatchewan (UofS), Canada. `TRS` was developed using the traditional page-based copy-paste style where HTML mark-up and PHP code were put together on dynamic web pages. `IRS` was developed following a more sophisticated approach using the MVC (Model-View-Controller) pattern that resulted in a relatively more modularized implementation.

Table 4.9: Block clones in industrial web applications

Subject Systems	Development Style Experienced	Number of LOC	.php files	PHP blocks	clones	cloned files	% of cloned files
`IRS`	Modular implementation	6,848	75	644	240	30	40%
`TRS`	Page-based copy-paste	5,295	27	868	212	19	70%

[3]This study on the clones in web applications is a joint work with Tariq Muhammad and Yosuke Yamamoto, two Ph.D. students at the Department of Computer Science, University of Saskatchewan, Canada. Since this is a joint work, we kept its description short. This particular study of ours appeared in a peer-reviewed conference [211], where further details can be found.

[4]http://www.usask.ca/dhse/trainingcourses

[5]http://lamp.usask.ca/dhse/ZF1.0.1

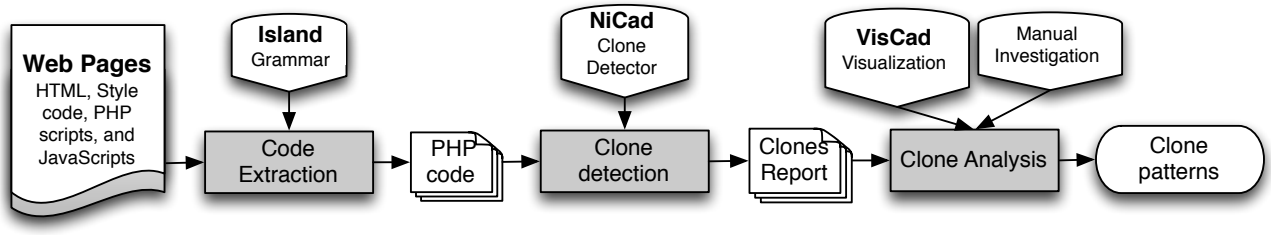


Figure 4.11: Procedure of the empirical study on code clones in web applications

Using island grammar [265], we extracted the PHP code from the tangled source code of the dynamic web pages. Then using the `NiCad` [238] clone detector, we detected clones in the PHP code at the granularity of syntactic blocks, and analyzed them with `VisCad` [10], a clone analysis and visualization tool. We investigated the possible instances of different patterns of cloning such as *Forking* and *Templating* [142] (see Section 2.2). In particular, we aimed to examine whether there is any significant difference in the existence and patterns of cloning in the two web applications that underwent technically two different development approaches. A schematic diagram portraying the procedural steps of our empirical study is presented in Figure 4.11.

The sizes of the web application, the number and proportion of clones found in each of the subject systems are presented in Table 4.9. From the study, we found that despite the architectural differences and development styles, both the web applications had significant number of code clones. However, the system developed using the traditional page-based approach had relatively higher number of the clones very scattered over different areas of the code base. On the contrary, the modular implementation following the MVC pattern resulted in relatively less scattered clones in the other system. The dispersion of clones in the later system was dictated by the underlying framework, and those clones appeared to be relatively easier to manage compared to those in the other system.

4.5 Threats to Validity

In our clone-density-based study (Section 4.2), we examined multiple releases of six systems for each of three programming languages (C, C#, and Java) which leads to a total of 18 subject systems. This number of systems for each language may not be enough to derive a decisive conclusion on the effect of programming languages/paradigms on code clone evolution. However, this clone-density-based empirical study was the largest study in clone evolution at the time this work was published [313]. The genealogy-based study was also a large-scale study with 593 releases of 17 software systems. Indeed, the findings from our study on clones in web applications (Section 4.4) are based on only two industrial systems, and thus can be argued against their *generalizability*.

There exists an open question about what should be the appropriate level of similarity to consider two code fragments as near-miss clone-pair. To address this concern, in our clone-density-based study (Section 4.2),

we examined the evolution of code clones at four different levels of similarities: exact clones, and near-miss clones with 70%, 80%, and 90% similarities on the pretty-printed source lines of code (see [242, 238] for details).

The types of subject systems might have domain influence on the results. To minimize such domain effects, in both the clone-density-based study (Section 4.2) and the genealogy-based study (Section 4.3), we included a large number of releases of software systems from diverse application domains. The clone-density-based study included a total of 1,636 releases over 18 subject systems and the genealogy-based study was on the basis of 593 releases of 17 software systems written in diverse programming languages.

Although we carried out some manual verifications, exhaustive manual validation over all the clones found in all the releases of all the systems was not feasible due to the huge amount of time and effort required for this. However, in our clone-density-based study (Section 4.2) as well as in the study of clones in web applications (Section 4.4), we used the `NiCad` clone detection tool, which was reported to have high precision [238, 242] and recall [240] in finding clones. Thus, we gain confidence about the validity of the results from these studies.

However, in our genealogy-based study (Section 4.3), we used `CCFinderX` for clone detection, which is recognized to have high recall in clone detection, although its precision can be lower that some other clone detectors [35]. As `CCFinderX` cannot detect *Type-3* clones, our genealogy-based study does not include genealogies of *Type-3* clones. For this particular study, we chose `CCFinderX` to be able to compare our results with the study of Kim et al. [154]. Maintaining consistency with the their study [154], we carefully set `CCFinderX`'s parameters to keep the probable false positives to the minimum.

4.6 Related Work

Over the last decade, many studies were conducted to reveal the patterns and characteristics of code clones in software systems. However, we confine our discussion on only those studies that reported the management implications of code clones and their evolution. To keep the discussion easy to follow, we organize our discussion in accordance with the analytical studies presented in this thesis.

4.6.1 Analysis of Clones and their Evolution

Over the past decade, many empirical studies have been conducted, majority of which were exploratory studies for understanding the existence and properties of clones in software systems. Relatively recent studies have aimed to investigate the evolution of clones and their implications on software maintenance. In this section, we discuss only those work that appear to be relevant to our study on clone evolution presented in Section 4.2.

Roy and Cordy [242] conducted an empirical study on 23 systems written in three different languages, and examined distribution of clones in the systems as well as the effect of programming language and system's size. Later they conducted another study [241] on eight systems written in Python, and found that

cloning properties of Python systems are not really different from previous observations for C, Java, and C#. However, both the studies examined code clones in single version of each system, without investigating the evolution clones across software releases.

Laguë et al. [172] studied the evolution of clones with six versions of a large telecommunication software system and concluded that although a significant number of clones were removed during the evolution, the overall clone-density increased over time. Antoniol et al. [8] and Li et al. [182] studied the evolution of the Linux kernel and observed that although clone coverage (i.e., the proportion of cloned code over the entire source code) increased early in the development, it stabilized over time. Our study differs from theirs because we not only study clone-densities but also forecast clone-densities for the future releases of the software and that we conduct the study with 18 diverse categories of systems of three different languages and we use a hybrid clone detection tool, `NiCad` [238], which has high precision [242] and recall [240].

Göde [89] examined the evolution of individual exact cloned fragments in several open-source systems written in C, C++, and Java using `iClones` [92]. The ratio of exact clones over system size decreased in majority of the systems he studied. Krinke [169] on the other hand, conducted an empirical study to investigate the type of changes taking place on cloned versus non-cloned code to determine whether cloned code is more stable than non-cloned code. For clone detection, he used Simian, a text-based clone detector capable of identifying almost identical clones. He observed that changes to the evolving software is dominated by massive deletion of cloned code, and if deletions are ignored, in terms of line addition and modification, cloned code is more stable than non-cloned code. Krinke [168] also analyzed many revisions of five open-source software systems and found that half of the changes to code clone-groups are inconsistent and that corrective changes following inconsistent changes are rare.

A number of tools and frameworks [67, 217] have been developed for aiding clone tracking, analysis, evolution and management. Kim et al. [153, 154] coined the terms "clone lineage" and "clone genealogy" to describe relationship between clone-groups in subsequent versions of evolving software. To investigate the clone evolution, they developed a clone genealogy extractor using `CCFinder` [137], a token based *Type-1* and *Type-2* clone detector. To evaluate their approach, they extracted clone genealogy from two open-source Java projects, `Carol` and `DnsJava`. They found that many genealogies disappear in a relatively short time after their birth.

Saha et al. [248] conducted a follow-up study at release level. Lozano and Wermelinger [188] analyzed commit-by-commit evolution of five open-source Java projects over at least thirty months. They also used `CCFinder` [137] for detecting clones in methods. They found that cloned methods change more than non-cloned method, which contradicts Krinke's findings [169]. They further reported that cloned methods tend to remain cloned most of their lifetime, which also contradicts the findings of Kim and Notkin [154].

Bakota et al. [18] proposed a machine learning approach for detecting inconsistent clone evolution situations and found different bad smells using twelve versions of Mozilla Firefox. Thummalapenta et al. [274] proposed an automatic approach for classifying the evolution of cloned fragments and reported an analysis

using four different Java and C software systems for investigating to what extent clones are consistently propagated or evolved independently. Bettenburg et al. [36] studied the inconsistent changes of clones at the release level. They noted that the number of defects due to inconsistent changes in clones is substantially lower at the release level than at the revision level.

While these studies provide important insights on the fine-grained evolution of clones and their maintenance implications, our study, as described in Section 4.2, significantly differs from them in several aspects. First, instead of providing in-depth analysis on the evolution of clones, we provide an overall analysis on the evolution of clone-densities and attempt to forecast such densities in the future releases for 18 diverse varieties of systems of three different programming languages. None of the studies above attempted to forecast clone-densities as ours.

There exists a common issue with most of these studies that they are based on CVS snapshots over certain intervals. The interval of commit-by-commit transactions or snapshots of CVS/SVN repository over certain periods may be too frequent for analyzing clone evolution [36]. Even a large number of such versions of the source code can span over only a few stable releases. In addition, the source code under study may not experience significant development/changes during some of those intervals. Moreover, commit transactions are sensitive to developer's commit style. Therefore, we conducted our study (Section 4.2) at the release level to capture stable changes in the clones over a longer period. Finally, most of the earlier studies focused on the evolution of *Type-1* and/or *Type-2* clones, whereas we studied both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones at different similarity levels using a hybrid clone detection tool.

Antoniol et al. [7] applied *ARIMA(p,d,q)* [38] time series analysis to model code clone evolution for predicting proportion of code clones in 27 subsequent versions of MiniSQL (written in C) using a metrics-based clone detection tool. Again, our study (Section 4.2) significantly differs from them in examining 18 systems of diverse varieties written in three different programming languages, and using a parser-based but text-line comparison clone detection tool with different similarity levels. Moreover, the *ARIMA* process is elegant but complicated, and applicable only when there is enough historical data available to generate a reliable model. The very first step in the *ARIMA* process is to identify a tentative model through the analysis of historical data, and it is recommended that 50, preferably more historical observations be considered at this step [209].

Shawky et. al. [260] modelled clone evolution in two software systems (i.e., 50 versions of FileZilla and 100 versions of VLC) using *chaos theory* for prediction in new versions. *Chaos theory* also requires historical data to build the initial model. This implies that neither *ARIMA* nor *chaos theory* is suitable for estimation during the early stage in the evolution, whereas the regression analysis technique is applicable from the very early releases of the software evolution.

4.6.2 Study of Clones in Web Applications

Very few studies are found in the literature that investigated code clones in web applications. In this section, we discuss the work relevant to our study on code clones in industrial web applications, as presented in Section 4.4.

One of the earliest works in this regard was performed by Lucca et al. [190]. Their work focused on the detection of similar static web pages as clones. Their approach was based on serialization of HTML tags in web pages and then pairwise computation of Levenstein distance between tag-sequences. Later, they conducted a follow up work [66], in which they attempted to detect clones in ASP pages. They viewed an ASP page as a sequence of references to different ASP objects. They hypothesized that ASP pages having same behaviour host the same sequence of such references. They constructed "ASP-strings" composed of the sequence of references extracted from ASP pages, and detected similar ASP pages applying a similar approach based on Levenstein distance between "ASP-strings". Our work significantly differs from those of Lucca et al. [190, 66]. While the objective of their work was to identify similarities in the entire static HTML pages and ASP pages, in our work, as described in Section 4.4 of Chapter 4, we investigate the patterns of cloning in the scripting code embedded in dynamic web applications.

Lucia et al. [191, 193] proposed a method to identify clones in the navigational patterns in web applications. They extracted a navigational graph from the references among the different web pages and resources. Then they applied a graph-based pattern matching algorithm to search for pairs of maximal clones in the navigational graph. Later, they extended their work to develop a tool [192] to facilitate the analysis of clones in the navigational patterns. However, our study (Section 4.4) is different from theirs in the fact that we investigate cloning patterns in the PHP scripts, whereas their work was towards the detection and analysis of clones in the navigational patterns.

Lanubile and Mallardo [174] applied a metric based approach (with the help of a tool called `eMetric`) to detect function level clones in the JavaScript and VBScript code in three web based systems. Rajapakse and Jarzabek [229] conducted a case study on 17 web applications. The objective of their work was mainly to understand to what extent there existed cloned code in web applications, and they found a cloning rate of up to 63%. In a later study [231], they explored the effectiveness of the PHP Server Pages technique in minimizing clones in web applications. They concluded that clone unification using Server Pages technique can reduce the number of clones, but such a reduction may also negatively affect the system qualities (e.g., runtime performance), which may not be acceptable in many practical situations [230, 231].

Our work (Section 4.4) significantly differs from those studies in a number of ways. The metric based clone detection technique used in the study of Lanubile and Mallardo [174] might have reported many false positives and missed many potential clones, as the metric based techniques are known to have low accuracy in clone detection [243, 311]. The studies of Rajapakse and Jarzabek [229, 231] were only on *Type-1* and *Type-2* clones, as the clone detector `CCFinder` used in their study can detect only *Type-1* and *Type-2* clones, but not *Type-3*.

However, in our study (Section 4.4), we use a hybrid clone detector, `NiCad` [238], which was reported to have high accuracy in detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones [238, 243]. Moreover, the objective of Lanubile and Mallardo [174] was the detection of clones in source code written in JavaScript and VBScript code, whereas Rajapakse and Jarzabek [229, 231] focused on the existence of clones and the effectiveness of Server Pages technique in clone minimization. On the contrary, our work (Section 4.4) is on the *analysis* of cloning patterns in PHP scripts in industrial web applications that underwent distinct development processes.

Similar to our study (Section 4.4), the work of Mao et al. [194] also used TXL [51] implementation of island grammar [265] to extract style information from HTML files. Then they identified the exact (*Type-1*) clones in the style code segments by pairwise comparison, and factored out the styles and adjusted the `div` tags that referred to the styles. The objective of their work was to convert the table-based layout websites to standards-compliant modern CSS stylesheet-based websites. Our work, as described in Section 4.4 of Chapter 4, is orthogonal to theirs. We use island grammar to extract scripting code from the dynamic web applications, we detect both the exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones using the `NiCad` [238] clone detector, and analyze those clones with the help of `VisCad` [9, 10].

4.6.3 Genealogy-based Study of Clones

The maintenance implications of clones are studied more in the recent years than in the past. Such studies typically investigate how individual clones experience changes during their evolution and the impacts of such changes on software maintenance. Our genealogy-based empirical study was conducted with the same objective. In this section, we discuss those work that we find relevant to our genealogy-based study as described in Section 4.3.

Kapser and Godfrey [141, 142] conducted large-scale empirical studies and concluded that clones are not necessarily harmful and found several patterns of clones that could be useful in many cases. Juergens et al. [133], on the other hand, argued that unintentionally created inconsistent clones always lead to faults, and concluded that clones could be harmful in software maintenance. While we also studied the maintenance implications of clones, our study (Section 4.3) significantly differs from theirs in the sense that they did not study the evolution of clones.

The study of Kim et al. [154] is the most closely related to our genealogy-based study presented in Section 4.3 of Chapter 4. However, they studied only two small Java systems and at the revision level. On the other hand, we conducted a larger study with 17 diverse open-source systems written in four different programming languages. Furthermore, instead of location mapping, we have used snippet matching together with text similarity for mapping the clone-groups from one version to the next. This allows us to map clone-groups even when lines are modified or reordered in the next version.

Aversano et al. [11] extended the clone evolution model of Kim et al. [154] by grouping inconsistent changes to independent and late evolution classes. Again, they studied only two open-source Java systems

namely `ArgoUML` and `dnsjava` and reported contradictory findings for the consistently changed clone-groups. Lozano et al. [187, 189] carried out several studies on the maintenance implications of clones. While they could not find any systematic relationships between cloning and maintenance efforts, they concluded that change efforts might increase for a function/method when it has clones. However, the work of Lozano et al. [187, 189] studied clones in revisions of software systems written in Java only, and their focus was on the changes in all functions/methods irrespective of whether they had clones or not. On the contrary, our work (Section 4.3) was particularly focused only on the cloned code in releases of software systems written in four different languages.

Göde [89] proposed an incremental approach that models *Type-1* clone evolution based on the changes made to the source code at consecutive revisions of several open-source systems. While he concluded that the ratio of clones decreased in the majority of the systems and cloned fragments survived more than a year on average, no general conclusion on the consistent or inconsistent changes to clone-groups was reported. Bazrafshan [31] studied the daily snapshots of the source code repositories of seven open-source software systems to investigate the differences in the evolution of exact and near-miss clones. Unlike the work of Göde [89] or Bazrafshan [31], our study (Section 4.3) took into account both *Type-1* and *Type-2* clones in stable releases of software systems. Krinke [168] analyzed many revisions of five open source software systems and found that half of the changes to code clone-groups are inconsistent and that corrective changes following inconsistent changes are rare. In another study [169], he found that cloned codes are more stable than non-cloned codes and thus require less maintenance effort compared to non-cloned code. Bakota et al. [18] proposed a machine learning approach for detecting instances of inconsistent clone evolution situations, and based on a study over twelve monthly revisions of Mozilla Firefox, reported different "bad smells". However, they studied the evolution patterns of cloned fragments whereas we studied clone-groups.

Note that, all the aforementioned studies on clone evolution were based on revisions of the software systems under study. Such revisions are composed of periodic snap-shots of the underlying code base or the developers' commit transactions. Many of such revisions can constitute only a few stable releases of the software system, and those revisions often contain temporary clones that the programmer create for experimentation only [36]. To avoid, such effects of short-lived temporary clone, in our studies, we investigated clones in the stable releases of the software systems.

Bettenburg et al. [36] studied the inconsistent changes of clones in releases of two open-source software systems. They noted that the number of defects through inconsistent changes is possibly substantially lower at the release level than at the revision level. However, the study of Bettenburg et al. [36] was based on releases of only two software systems and they examined the possible relationship between inconsistent changes in clones and software defects. We conducted a larger study (Section 4.3) on 17 software systems written in four different languages, and we particularly focused on evaluating clone genealogies.

For investigating to what extent clones are consistently propagated or independently evolved, Thummalapenta et al. [274] performed an empirical study on four open-source software systems written in C and

Java. While they focused on identifying evolution of cloned codes over time and relating the evolution patterns with other parameters (e.g., clone granularity, clone radius and cloned code fault-proneness), in our work (Section 4.3), we focused on examining clone genealogies with 17 open-source software systems covering four popular programming languages.

4.7 Summary

In this chapter, we presented empirical studies on the existence and evolution of code clones in open-source and industrial software systems. The first work is a clone-density-based empirical study on the evolution of exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) code clones in 18 large open-source software systems of diverse categories written in three different languages namely Java, C#, and C. For each software system our study included many releases (and pre-release) ranging between 20 and 459, summing up to 1,636 in total. For each system we investigated how the density of code clones changes between subsequent releases at four different similarity levels. Using the *Pearson product moment correlation coefficient* we examined the relationship between software growth (in terms of number of functions) and changes in the amount of clones.

We found that with the increase in the number of functions, the number of cloned fragments also increases in all systems. However, between clone-density and the number of functions, very weak positive correlation has been identified. Moreover, the average clone-density for larger (in terms of LOC) systems are found to be less than that for smaller systems. Using a simple regression, analysis we were able to make one step ahead forecast on the densities of both exact and near-miss clones in subsequent releases. The average standard error of estimate over all releases of all systems was 2.35. We tracked the forecast errors over subsequent releases of each system and examined the regularity of changes in clone-density from release to release.

Programming languages/paradigms are found to have significant impact of the existence and evolution of clones. The average density of exact clones over all releases of all systems is found to be 9.26%. The average density of near-miss clones was 10.11% at UPIT 10%, 13.46% at UPIT 20%, and 17.65 at UPIT 30%. For all the four UPITs Java systems are found to have the highest density, and C systems have the lowest density of clones. With the increase of UPIT the increase in clone-density is found to be much higher in C systems than Java and C# systems. The evolution of clone-density in Java and C# systems is found to be more irregular than C systems. Moreover, with the increase of UPIT, irregularity in the change in clone-density is found to have increased in all systems. These results suggest that one should expect higher number of code clones in software systems written in object-oriented languages such as Java and C#. Thus, such systems can be more vulnerable to the negative impacts of code clones and therefore active clone management is more important for those systems.

We also identified some interesting patterns in the evolution of clone-density over subsequent releases. For instance, we found major changes in clone-density over few early releases of software life time, and over the later releases there exists long sequences of releases among which clone-density does not vary that

much. We also found several instances of where a particular clone-group propagated over a long series of releases. Our genealogy-based follow-up study (Section 4.3) also revealed many long-lived clones. From the other follow-up study on clones in web applications (Section 4.4), we found even higher proportion of clones in the industrial web applications compared to the traditional open-source software systems. Due to the tangled multilingual nature of the source code for web applications, tracing and managing those clones in such applications demand specialized tool support that should be able to perform language specific separation of code from the multilingual program source, and handle them separately while still preserving the inter-language relationships of the code clones.

The presence of a significant number of long-lived clones and the irregularities in the evolution of clone densities found in the studies drives us to carry out a more in-depth investigation on the changes and removal of individual clones, as presented in the next chapter (Chapter 5).

Chapter 5

Investigation of Clone Change Patterns

"A fact is a simple statement that everyone believes. It is innocent, unless found guilty.
A hypothesis is a novel suggestion that no one wants to believe. It is guilty, until found effective"
– Edward Teller

In the last chapter (Chapter 4), we presented empirical studies on the existence and evolution code clones in terms of clone-density as well as clone genealogy. The studies revealed irregularities in the changes of clone densities across software releases and reported many clones that propagated over long sequences of software releases. Therefore, we became interested to carry out a deeper level study on the changes and removal of individual clone-groups.

In this chapter, we present a genealogy-based study on the evolution and mutation of clones across releases of open-source software systems written in different programming languages. Addressing a number of research questions, this study offers insights into the mutation and removal of code clones in practice and points to pragmatic possibilities for clone management. In Section 5.1, we introduce our motivation and the research questions. In Section 5.2, we present the terminology and metrics used in our study. In Section 5.3, we describe the setup and procedure of our empirical study. Section 5.4 presents the findings our study. In Section 5.5, we discuss the possible threats to the validity of our study, Section 5.6 accommodates related work, and Section 5.7 concludes the chapter with a summary.

5.1 Introduction

Despite ongoing research on the positive and negative effects of code clones [133, 142, 154, 201, 202, 222, 227, 274], researchers and practitioners have come to an accord for the need of active and informed clone management [309, 310] including documentation and removal of clones through refactoring. Out of a recent study, Fontana et al. [78] also reported that the removal of code clones improves code quality in *most* cases but not all. However, code clones can often be desirable, and aggressive removal of clones through refactoring may not be a good idea [154, 308, 307, 312], given the risks and efforts involved in such activities. In this regard, a number of classification schemes [20, 139, 158, 258], metric based selection approaches [19, 49, 104], and an effort model [308, 307, 312] have been proposed to identify potential clones for refactoring. Still, for many systems, clone management and removal is yet to be a part of the daily maintenance activities [90]. Despite

more than a decade of software clone research, clone management remains far from industrial adoption, and this area has gained more focus from the community in recent years [311].

A deep understanding of how individual clones change during their evolution, and which criteria cause their removal from the system, can help in devising effective strategies and tool support for clone management. A number of studies on near-miss clone evolution [203, 204, 205, 206, 207, 248, 251, 313, 314] are found in the literature, which attempt to inform clone management [284, 309, 311]. These studies on clone evolution and programmers' psychology lead to some common beliefs and at times even contradictions about the traits of clone evolution. For example, the study of Kim et al. [154] suggests that many clones are *volatile* (i.e., disappear shortly after they are created), while the study of Lozano and Wermelinger [188] suggests otherwise.

This chapter focuses on the patterns of changes and removal of code clones during the evolution of software systems. In particular, we formulate the following eight research questions to capture different characteristics of clone change and removal. Some of the research questions correspond to common beliefs (or, contradictions) in the community [45]; but we want to develop empirical evidence based on a systematic genealogy-based study on clone change and removal in evolving software systems.

RQ1: *Do the sizes of the groups of clones make any difference in clone removal in practice?* — Kim et al. [152] suspected that frequently copied code fragments (i.e., larger clone-groups) can be good candidates for clone refactoring.

RQ2: *Do the sizes of the individual clone fragments in terms of the number of lines impact clone removal in practice?* — Larger clone fragments can be attractive candidates for refactoring, as conjectured by Kim et al. [152].

RQ3: *For a group of clones, does the distribution of the clones in the file system hierarchy impact their removal in practice?* — Göde [90] reported that the developers were more interested in refactoring closely located clones.

RQ4: *Is there any relationship between any particular type of changes in the clones and their removal?* — This is still an open question, as far as we are concerned. If there exists any relationship between a particular type of changes and clone removal, the clone management tools can focus on supporting that category of changes.

RQ5: *How frequently do the clones experience changes before they are removed from the system?* — There is an ongoing debate on the stability of cloned code as compared to the non-cloned code [91, 169, 201, 202].

RQ6: *Does the granularity (entire function bodies or syntactic blocks) of clones make any difference in their removal in practice?* — A recent study of Göde [90] reported many instances of removal of block clones by *extract method* refactoring.

RQ7: *Does the textual similarity in the source code of the clones have any effect in the removal of clones in practice?* — Very similar (e.g., *Type-1*) clones can be expected to be easier to refactor than very dissimilar (e.g., *Type-3*) clones.

RQ8: *During the evolution of the software systems, when does clone removal take place?* — This question addresses the aforementioned contradiction about the volatility of clones.

To address the research questions, we carry out a systematic study based on code clone *genealogy* [154, 249], which maps the individual clone fragments across subsequent releases over their evolution. We investigate the changes and removal of individual clones in 329 releases of nine diverse open-source software systems written in Java, C, and C#. Then we analyze them against a wide range of metrics and characterization criteria. In the light of a combination of qualitative analysis, quantitative analysis and statistical tests of significance, we derive the answers to the research questions.

We believe, such an empirical study on the characteristics of changes and removal of individual near-miss clone fragments is timely and addresses a gap in the literature. Our study is based on genealogies of near-miss clones including not only *Type-1* and *Type-2* clones, but also *Type-3* clones.

Table 5.1: Software systems subject to our empirical study

Prog. Lang.	Subject System	No. of Releases	Releases		Dates (mm/dd/yy)		Duration (months)	Source Lines of Code (LOC ranges)		
			From	To	From	To				
Java	dnsjava	50	0.9.2	2.1.1	04/19/99	02/10/11	131	6,290	–	15,018
	JabRef	27	1.5	2.4.2	08/15/04	11/01/08	50	22,041	–	69,170
	ArgoUML	48	0.27.1	0.32.beta2	10/04/08	01/24/11	26	176,618	–	202,555
C	ZABBIX	31	1.0	1.8.4	03/23/04	06/01/11	86	9,252	–	62,845
	Conky	28	1.1	1.8.1	06/20/05	10/05/10	62	6,555	–	39,810
	Claws Mail	44	2.0.0	3.7.9	06/30/06	04/09/11	63	1,33,642	–	1,89,786
C#	CruiseControl	31	0.7.rc1	1.8.4	11/08/04	09/01/13	98	35,895	–	1,82,032
	iTextSharp	22	5.0.0	5.4.4	12/08/09	09/16/13	45	1,72,573	–	2,17,328
	ZedGraph	48	1.1	5.1.5	08/02/04	12/12/08	52	2,439	–	26,433

5.2 Terminology and Metrics

In this section, we introduce the terminology and metrics used in this work to characterize the changes and removal of code clones. Some of the metrics and and criteria are adopted from earlier studies found in the literature [44, 90, 94, 154]. For the purpose of our analysis, we categorize the evolving clone-groups in accordance with the change patterns in the corresponding genealogies, as described in Chapter 2.

Clone Genealogy: Recall from Chapter 2 that a set of clone fragments that are clones of each other form a *clone-group*. A *clone genealogy* refers to a set of one or more lineage(s) originating from the same clone-group, whereas, a *clone lineage* is a sequence of clone-groups evolving over a series of releases of the software system. Figure 2.4 shows several examples of clone genealogy.

Consistent and Inconsistent Change: Recall from Chapter 2 that if all clones in the clone-group experience the same set of changes during the transition between releases, then such changes are characterized as being a *consistent change*, otherwise the changes are regarded as being *inconsistent*.

Consistently Changed Clone-Group: If the genealogy of a clone-group has any consistent change pattern(s) but does not have any inconsistent change patterns during evolution, it is classified as a *consistently*

changed clone-group. The clone-group associated with the second genealogy in Figure 2.4 is an example of a consistently changed clone-group as there are consistent changes between versions (i.e., releases) V_{k+1} and V_{k+2} and between versions V_{k+2} and V_{k+3} as well as between versions V_{k+3} and V_{k+4} while there is no inconsistent change at all during the evolution of the clone-group.

Inconsistently Changed Clone-Group: If the genealogy of a clone-group has any inconsistent change pattern(s) throughout the entire evolution period, it is characterized as an *inconsistently changed* clone-group. The clone-group associated with the third genealogy in Figure 2.4 is an inconsistently changed clone-group as there are inconsistent changes between versions (i.e., releases) V_{k+1} and V_{k+2} as well as between versions V_{k+2} and V_{k+3}.

Static, Alive, Dead Clone-Group: Static clone-groups are those which propagate through subsequent releases having no textual change in the clones. A clone-group is called *dead* if it disappears before reaching the final release under consideration, otherwise the clone-group is considered *alive*. The clone-groups associated with the first, second and third genealogies in Figure 2.4 represents static, dead, and alive clone-groups respectively.

Textual Similarity: The textual similarity between two code snippets S_1 and S_2, denoted by $\int(S_1, S_2)$, is determined by calculating the identical lines with respect to their sizes, as defined by the following formula[1].

$$\int(S_1, S_2) = \frac{2 \times |\ell_1 \cap \ell_2|}{|\ell_1| + |\ell_2|} \tag{5.1}$$

where ℓ_1 and ℓ_2 are the ordered sets of pretty-printed (i.e., normalized) lines in S_1 and S_2 respectively. $|\ell_1 \cap \ell_2|$ is the number of common ordered lines between ℓ_1 and ℓ_2, calculated using the longest common subsequence (LCS) algorithm. The textual similarity of a clone-group G, denoted as $\int(G)$ is the average of the textual similarities between all clone pairs in that group. Mathematically,

$$\int(G) = \frac{\sum_{S_i, S_j \in G} \int(S_i, S_j)}{\binom{|G|}{2}} \tag{5.2}$$

Entropy of Dispersion: We used an entropy measure to characterize the file level physical distribution of the clones in a clone-group. Such an entropy measurement, sometimes referred to as *Shannon entropy*, is commonly used in the area of Information Theory. In this work, the entropy of dispersion of the clones in clone-group G is calculated using Equation 5.3 as follows:

$$entropy(G) = \sum_{i \in \mathcal{F}_G} -p_i \log(p_i) \tag{5.3}$$

where, $\mathcal{F}_G$ denotes the set of distinct files hosting the clones in clone-group G, and p_i denotes the probability of the clones being located in file i.

For example, if all the clone fragments reside in the same file, the dispersion entropy will be 0.0. If the entropy is low, clones are densely located in only a few files. If the entropy is high, the clones are scattered across different files.

[1] In the area of Information Retrieval, this similarity measurement is known as the *Dice Coefficient*.

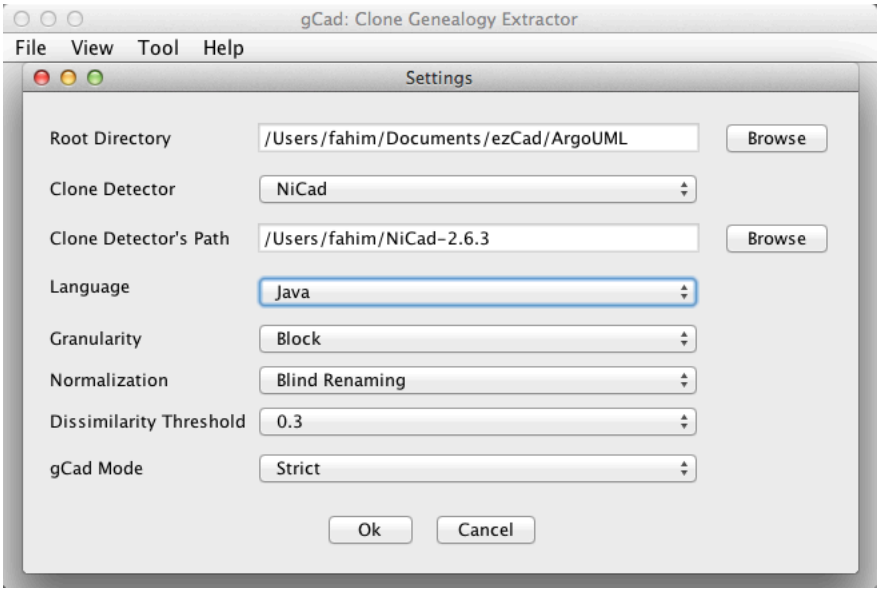


Figure 5.1: `gCad` settings for clone genealogy extraction

5.3 Study Setup

To investigate the research questions outlined in Section 5.1, we study the clone genealogies across releases of nine diverse open-source software systems (Table 5.1) written in Java, C, and C#.

In the selection of the subject systems, we followed a number of criteria. First, we tried to include software systems that had reasonably large sizes and large number of releases. In computation of a system's size, we took into account only the source code lines (LOC) written in the particular programming language that the software system is categorized in Table 5.1. We excluded comments, blank lines, and lines of code written in any other programming languages. Second, in our study, we tried to include subject systems from diverse application domains. Third, we preferred those open-source software systems, which were used in earlier studies [154, 248, 249, 313] reported in the literature.

5.3.1 Extraction of Genealogies

For the extraction of clone genealogies, we used an extended version of `gCad` [249] clone genealogy extractor that we developed. `gCad` can construct and classify genealogies of all three types (*Type-1, Type-2*, and *Type-3*) of clones that we are interested in. Details of how `gCad` operates and computes clone genealogies can be found elsewhere [249]. As per the need of this study, we significantly extended and customized the tool with a carefully designed graphical user interface (GUI), and a set of appropriate features to compute the necessary metrics. For the purpose of our study, we carefully chose a set of `gCad`'s configuration parameters as shown in Figure 5.1.

For the detection of code clones, we selected the `NiCad-2.6.3` [54, 55, 238] clone detector, which is a state-of-the-art clone detection tool reported to be effective in detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones with high precision and recall [240, 238]. `gCad` invokes `NiCad` to separately detect clones from every release of each of the subject systems. In invoking the clone detector, some of the `gCad` parameters are passed to `NiCad` to guide the process of detecting *Type-1, Type-2,* and *Type-3* clones at the chosen granularity (syntactic blocks for our study) and dissimilarity threshold (0.3 for our study).

The dissimilarity threshold (Figure 5.1) is a size-sensitive dissimilarity threshold that plays a vital role in guiding `NiCad` in the detection of *Type-3* clones. For our study, the dissimilarity threshold was set to 0.3, which signifies that `NiCad` detects two code fragments as clones if at least 70% of their pretty-printed text lines are the same (i.e., if at most 30% lines are different). The normalization option "blind-renaming" tells `NiCad` to ignore the differences in the names of identifiers/variables, and thus it is a significant parameter for the detection of *Type-2* clones.

From the clone detection results obtained from `NiCad`, for each of the subject systems, we separately constructed the clone genealogies using `gCad`. We operated `gCad` in 'strict' mode to construct and characterize clone genealogies. In 'strict' mode, `gCad` captures and takes into account all types of changes in the source code lines of clone-pairs, irrespective of whether those changes took place in their corresponding similar or dissimilar lines of code. Details of how `gCad` operates in different modes can be found elsewhere [249].

5.3.2 Investigation

We examined all the dead genealogies to see how the clones were removed. We also examined how the individual clone fragments changed during their evolution over a series of releases. Since, the inconsistent changes to clones are believed to be a common phenomena that produce vulnerabilities in a system [313, 308, 312], we characterized the clone changes as consistent versus inconsistent. In addition, we captured how frequently a clone-group changes during the evolution before its removal. For quantitative analysis, we computed the necessary metrics according to the categorization described in Section 5.2.

5.4 Findings

The findings of our study are derived from qualitative and quantitative analyses of the changes and removal of individual clone fragments. We also apply the statistical *Mann-Whitney-Wilcoxon (MWW)* test [6] with $\alpha = 0.05$, to determine the statistical significance of the findings. Using the *Shapiro-Wilk* test [6] and *Q-Q* plot [6], we examined the distribution of the data, and found that some of the observations exhibited normal distributions while some others did not. Therefore, we chose to use the *MWW* non-parametric test, which does not assume the normal distribution of the data, and thus, is appropriate for data that exhibit or do not exhibit normal distribution.

Table 5.2: Sizes of removed and alive clone-groups

Prog. Lang.	Subject System	Avg. Sizes of Clone-groups	
		Removed	**Alive**
Java	dnsjava	2.25	2.75
	JabRef	2.31	4.17
	ArgoUML	2.12	9.12
C	ZABBIX	2.31	4.53
	Conky	2.37	9.31
	Claws Mail	2.88	2.95
C#	CruiseControl	2.32	3.58
	iTextSharp	2.21	5.80
	ZedGraph	2.15	2.50

5.4.1 Size of the Clone-Groups

To capture the relationship between the number of fragments in a clone-group and clone removal, we computed the average number of fragments in the removed clone-groups and that of the alive clone-groups for each of the subject systems (Table 5.2). As seen in Table 5.2, for all the subject systems, the average size of the alive clone-groups is higher than those of the removed clone-groups. During our manual investigation, we found that the developers refactored clone-groups that had only two or three clone fragments. Similarly, we found that in `JabRef`, there were 74 clone-groups having more than three fragments, and only four of them were refactored. This gives the impression that developers are more inclined to remove smaller clone-groups. To statistically verify this, we address the second research question *RQ1*, and formulate our null hypothesis as follows.

H_0^1**:** *The size of a clone group does NOT make a difference in clone removal in practice.*

A *MWW* test ($P = 0.35$) *fails to reject* (as, $P > \alpha$) the null hypothesis, which implies that the difference is not statistically significant. Hence, we answer the research question *RQ1* as follows.

Ans. to RQ1: *The size of the clone-groups (in terms of the number of member clone fragments) does not make a statistically significant difference in clone removal in practice.*

Although, in our study, the sizes of the removed clone-groups (in terms of the number of clone fragments) appears to be consistently lower than the alive clone-groups, this might have happened simply by chance in the software systems in our study. A larger study with many software systems may be required to further investigate the possibility of statistical significance of the pattern we found between the sizes of the clone-groups and their removal.

Table 5.3: Average sizes (LOC) of clone fragments

Prog. Lang.	Subject System	Removed		Alive	
		Average	SD	Average	SD
Java	dnsjava	10.00	3.00	11.00	5.00
	JabRef	17.00	15.00	13.00	9.00
	ArgoUML	16.00	13.00	15.00	19.00
C	ZABBIX	26.00	25.00	21.00	21.00
	Conky	20.00	23.00	15.00	7.00
	ClawsMail	15.00	9.00	16.00	18.00
C#	CruiseControl	8.85	4.50	9.46	5.61
	iTextSharp	13.99	13.62	10.76	11.00
	ZedGraph	15.70	15.45	12.34	12.67

SD = Standard Deviation

5.4.2 Size of the Clone Fragments

The sizes of the clone fragments can be expected to have a relationship with the refactoring effort, especially when the candidate clone-group includes near-miss (*Type-2* and *Type-3*) clones beyond *Type-1*.

To examine the relationship between clone removal and the LOC per clone fragment in the clone-groups, we separately computed the average number of pretty-printed LOC per fragment for the removed clones as well as for the alive clones. We also calculated the standard deviations for each of the measurements to capture the degree of variations. The results are presented in Table 5.3.

As can be observed from Table 5.3, there are subtle differences in the sizes of the clone fragments of both the removed and alive clone-groups. For six of the nine subject systems (`ZabRef`, `ArgoUML`, `ZABBIX`, `Conky`, `iTextSharp`, and `ZedGraph`), the average sizes of clone fragments of removed clone-groups appear to be significantly higher than those of the alive clone-groups. Hence, the anticipation of Kim et al. [152] saying – developers are more interested in getting rid of larger clones – appears to be true.

Addressing the research question *RQ2*, we now formulate our null hypothesis as follows.

H_0^2**:** *The size of the individual clones in terms of number of lines does NOT impact clone removal in practice.*

A *MWW* test ($P = 0.001$) over the series of sizes for the removed and alive clones *rejects* (as, $P < \alpha$) the null hypothesis. From the analysis described above, we answer research question *RQ2* as follows.

Ans. to RQ2: *The size of the individual clones in terms of number of lines does have a statistically significant impact on clone removal in practice, and larger clone fragments appear to be attractive for removal in practice.*

5.4.3 Entropy of Dispersion

In Table 5.4, we present the entropy of dispersion of both the removed and alive clones for all the subject systems. From the developer's perspective, refactoring/removal of co-located clones may require less effort than that needed for refactoring clones scattered over the code base. This can be expected to hold true due to several reasons. In the refactoring of scattered clones the developers might need to spend much time and effort to navigate to, understand the contexts, and make careful modifications at different locations of the code base.

In Table 5.4, we see that for each of the subject systems, the average entropy of dispersion for the removed clones is much lower than that for the alive clones. This indicates those clone-groups whose member clone fragments are closely located in the code base are relatively more attractive for refactoring/removal. To determine whether the initial observation *significantly* supports the expectation, we again conducted a *MWW* test with the null hypothesis as follows.

H_0^3**:** *For a group of clones, the distribution of individual clones in the file system hierarchy does NOT impact their removal in practice.*

The hypothesis addresses the research question *RQ3*. A *MWW* test ($P = 0.233$) between the entropy values for both the removed and alive clones (over all the systems) *fails to reject* (as, $P > \alpha$) the null hypothesis. This implies that there exists no relationship between the entropy of dispersion and clone removal in practice. Therefore, we derive the answer to research question *RQ3* as follows.

Ans. to RQ3: *For a group of clones, the distribution of individual clones in the file system hierarchy does not have a statistically significant impact on their removal in practice.*

As we delved deeper through manual investigation, we found a strange phenomenon in the relationship between entropy and the number of clone fragments that were removed. Most of the removed clone-groups had two fragments, if their entropy was greater than zero, i.e., they were not really located in the same file. For example, in `JabRef` and `ZABBIX`, developers refactored 37 and 43 clone-groups respectively, all of which had entropy higher than zero. Among them only two clone-groups in JabRef and 10 clone-groups in `ZABBIX` had three clone fragments, while the rest had only two fragments.

5.4.4 Change Patterns

Despite the realized advantages of code cloning, it is also true that code clones may have a significant impact on software development and maintenance in several ways. First, the reuse by copy-pasting of any code segment that already contains unknown faults, results in propagation of those faults to all the copies. Second, when a change is made in a code fragment, consistent changes are often expected in all its clone fragments, while any inconsistencies may introduce new faults. Third, if a bug is found in a certain code fragment, there remains a possibility that similar bugs can be found in the clones of the fragment, and thus may necessitate consistent propagation of that bug-fix to all the clones.

Table 5.4: Comparison of entropy of dispersion

Prog. Lang.	Subject System	Clones	
		Removed	Alive
Java	dnsjava	0.71	0.90
	JabRef	0.53	0.98
	ArgoUML	0.82	1.30
C	ZABBIX	0.35	0.53
	Conky	0.18	0.24
	Claws Mail	0.30	0.70
C#	CruiseControl	0.18	0.23
	iTextSharp	0.22	0.38
	ZedGraph	0.17	0.10

Table 5.5: Removal of clone-groups classified by change patterns

Prog. Lang.	Subject System	Static Clone-Groups			Consistently Changed CG			Inconsistently Changed CG		
		Total	Removed	[%]	Total	Removed	[%]	Total	Removed	[%]
Java	dnsjava	60	27	45.00	8	3	37.50	49	27	55.10
	JabRef	217	52	23.96	53	3	5.66	132	15	11.36
	ArgoUML	1435	109	7.60	39	4	10.26	440	19	4.31
C	ZABBIX	166	88	53.01	61	18	29.51	109	35	32.11
	Conky	121	44	36.36	19	7	36.84	37	16	43.24
	ClawsMail	445	58	13.03	172	7	4.07	304	7	2.30
C#	CruiseControl	528	187	35.41	119	35	29.41	388	133	34.27
	iTextSharp	1259	99	7.86	103	8	7.76	325	66	20.31
	ZedGraph	225	161	74.22	33	12	36.36	79	45	56.96

Table 5.6: *MWW* tests over categories of changes

Change Types	No Change	Consistent Change	Inconsistent Change
No Change	-	$P = 0.003$	$P = 0.158$
Consistent Change	$P = 0.003$	-	$P = 0.042$
Inconsistent Change	$P = 0.158$	$P = 0.042$	-

Table 5.7: *MWW* tests over removal of clones

Clone Categories	Static	Consistently Changed	Inconsistently Changed
Static	-	$P = 0.31$	$P = 0.695$
Consistently Changed	$P = 0.31$	-	$P = 0.536$
Inconsistently Changed	$P = 0.695$	$P = 0.536$	-

Thus, whether the clones changed consistently, inconsistently, or remained static during the evolution of a software system, may have implications in clone management in future releases. Therefore, we categorized the clones based on whether they remained unchanged, or changed consistently or inconsistently, and what percentage of such clones were actually removed during the evolution of the system. For each of the systems, the total number of clones of each of these three categories and the percentage of them that were removed, are presented in Table 5.5.

As we can see in Table 5.5, for each of the subject system, the number of static clone-groups is the highest while the number of the consistently changed clone-groups is the lowest. To examine any trends in the existence of static, consistently changed, and inconsistently changed clone-groups in the systems, we again conducted *MWW* tests between each two of the three categories of changes (total number) occurred in the clone-groups over all the systems. The results of the *MWW* tests are presented in Table 5.6, which suggest significant difference in occurrence of the three categories of changes (as, $P < \alpha$), except that the difference in the number of inconsistent changes and no-changes are found to be statistically insignificant (as, $P > \alpha$).

With respect to clone removal, from Table 5.5, we see that for six of the nine systems (`JabRef, ZABBIX`, `ClawsMail`, `CruiseControl`, `iTextSharp`, and `ZedGraph`), the majority of the removed clones are static clone-groups. The removal of inconsistently changed clone-groups were found to occur most often in two of the systems (`dnsjava` and `Conky`), whereas, the removal of consistently changed clones dominated in `ArgoUML`.

A high-level perception from the results in Table 5.5 may indicate that the static clone-groups can be more susceptible to removal. To verify such an observation, we carried out *MWW* tests between each pair of the three categories of clone *removal* over all the systems. The results of the *MWW* tests, as presented in Table 5.7, also suggest that there is no significant difference in the removal of static, consistently changed and inconsistently changed clone-groups (as, $P > \alpha$ in all cases).

These observations lead to the answer to the research question *RQ4* as follows.

Ans. to RQ4: *The majority of the clones do not experience any changes during their evolution. Those clones that experience changes, majority of those clone-groups undergo inconsistent changes. However, there is no statistically significant relationship between any particular type (i.e., consistent or inconsistent) of changes in the clones, and their removal at a later release.*

5.4.5 Frequency of Changes

The frequency of changes to the clone-groups is an important criterion in clone management, since changing source code can be expensive, while making consistent changes to clones may involve significant effort and risks. Indeed, the modifications of a clone fragment needing effort, and the required effort can be multiplied by the size of the corresponding clone-group, to make consistent changes to all clone fragments in the clone-group. This is one of the areas where clone management tool support may contribute by facilitating clone merging, or consistent change propagation.

Thus, we examined how frequently the clone-groups underwent changes before their removal. In Table 5.8, we present the number of clone-groups that, before removal, underwent changes only once, twice, and more than twice. As seen in the table, most of the removed clones were changed only once. For the clone-groups that changed at least once, their average change frequency is less than two, over all the subject systems. From our manual verification, we found that very few clone-groups underwent changes more than twice before their removal. On the other hand, we also found many clone-groups remained alive although they experienced minor or significant changes. However, we confined our focus to the changes of the removed clone-groups to get a complete picture over the entire life-time of the clone-groups. Now, we derive the answer to the research question *RQ5* as follows.

Ans. to RQ5: *Most clones do not undergo frequent changes before their removal.*

5.4.6 Level of Granularity

The *extract method* refactoring pattern is perhaps the most highlighted technique for removing clones at the granularity of syntactic blocks. Thus, we may expect evidence of many instances of block clone removal.

Table 5.8: Frequency of changes before removal

Prog. Lang.	Subject System	Change Frequency			
		1	**2**	**>2**	**Average**
Java	dnsjava	16	9	5	1.80
	JabRef	11	4	3	1.72
	ArgoUML	17	4	2	1.48
C	ZABBIX	30	16	7	1.74
	Conky	10	8	5	1.95
	ClawsMail	9	2	5	1.57
C#	CruiseControl	103	45	20	0.75
	iTextSharp	62	11	1	0.50
	ZedGraph	40	12	5	0.39

Alternatively, functions typically contain a somewhat complete implementation of certain features or program logic and so it may be easier to remove/refactor clones at the granularity of entire function bodies, rather than at the granularity of smaller syntactic blocks.

To determine whether there exist any relationships between clone removal and clone granularity, we examined both levels of granularities – function/method and syntactic block. Note that the body of a function also constitutes a block. Therefore, we distinguish *true* functions clones from the *true* block clones. A true function clone fragment spans the entire body of a function, whereas a true block clone must not constitute the entire body of a function.

Extended `gCad` is capable of differentiating true function clones from the true block clones. Any clone-group that is composed of only true function clones is categorized as a group of function clones, whereas, clone-groups consisting of only true block clones are categorized as groups of block clones. Separate genealogies are constructed for the clones at these two levels of granularity.

Over all releases of each of the subject systems, the total number and proportions of both the groups of function clones versus the block clones are presented in Table 5.9. The clone detection results for each of the systems identified clone-groups that contained both true function clones and true block clones. Therefore, it is not possible to categorize such a group as a group of only true function clones or only true block clones. This is why the total number of clone-groups reported in Table 5.9 is lower than that of Table 5.5. Addressing the research question *RQ*6, we now formulate our null hypothesis as follows.

H_0^6**:** *The granularity (entire function bodies or syntactic blocks) of clones does NOT make any difference in their removal in practice.*

Table 5.9: Removal of clone-groups at the granularities of function and block

Prog. Lang.	Subject System	Function Clones			Block Clones		
		Total	Removed	[%]	Total	Removed	[%]
Java	dnsjava	69	37	53.62	25	15	60.00
	JabRef	204	41	20.09	110	21	19.09
	ArgoUML	1183	97	8.19	305	20	6.55
C	ZABBIX	201	78	38.80	134	62	46.26
	Conky	115	35	30.43	59	30	50.84
	Claws Mail	510	40	7.84	337	29	8.60
C#	CruiseControl	889	354	39.82	1032	355	34.40
	iTextSharp	999	186	18.62	1687	173	10.25
	ZedGraph	229	162	70.74	337	218	64.69

A *MWW* test ($P = 0.93$) over the proportions of the removal of both true function and block clones *fails to reject* (as, $P > \alpha$) the null hypothesis.

Table 5.9 shows that developers remove both function and block clones as per their needs, as we do not see significant differences between the proportions of removal of function clones and block clones. For `ZABBIX` and `Conky`, the proportion of block clones removal is slightly higher. It seems that the clone removal rates for the two larger systems, `ArgoUML` and `Claws Mail` are far lower than the smaller systems. On the other hand, it appears that the developers of the relatively small systems `dnsjava`, `ZABBIX`, and `Conky` were more aware of the clones and were active in removing them through refactoring. From manual investigation, we found only one and two *Type-1* function clones in `dnsjava` and `Conky` respectively. Although as many as eight *Type-1* function clones were found in `ZABBIX`, seven of them were removed during the evolution of the system. Based on the above discussion, we now derive the answer to the research question *RQ6* as follows.

Ans. to RQ6: *In practice, the granularity (entire function bodies or syntactic blocks) of clones does not make any statistically significant difference in their removal.*

5.4.7 Textual Similarity

In Table 5.10, we present the average text similarities (actual and normalized) of the removed clones and the alive clones for each of the subject systems. Indeed, the degree of textual similarity among the clone fragments in a clone-group is important information as it corresponds to the differences between the clone fragments. Refactoring a clone-group with many variations can require more effort than refactoring a group of identical or very similar clones. Thus, the textual similarity for the clones in a clone-group can be expected

Table 5.10: Actual and normalized textual similarity of removed and alive clone-groups

Prog. Lang.	Subject System	Actual Textual Similarity				Normalized Textual Similarity			
		Removed Clones		Alive Clones		Removed Clones		Alive Clones	
		Average	SD	Average	SD	Average	SD	Average	SD
Java	dnsjava	0.60	0.20	0.67	0.18	0.80	0.12	0.81	0.10
	JabRef	0.76	0.18	0.68	0.18	0.85	0.13	0.82	0.11
	ArgoUML	0.76	0.20	0.66	0.17	0.85	0.14	0.80	0.14
C	ZABBIX	0.72	0.19	0.73	0.17	0.83	0.16	0.83	0.11
	Conky	0.76	0.16	0.69	0.15	0.88	0.09	0.84	0.09
	Claws Mail	0.73	0.20	0.65	0.22	0.87	0.12	0.82	0.15
C#	CruiseControl	0.67	0.19	0.68	0.18	0.83	0.09	0.84	0.09
	iTextSharp	0.72	0.22	0.66	0.19	0.86	0.11	0.84	0.12
	ZedGraph	0.80	0.22	0.70	0.22	0.91	0.14	0.87	0.14

SD = Standard Deviation

to be proportional to the necessary efforts for refactoring them. Taking these into consideration, we address the research question *RQ1*, and formulate our null hypothesis as follows:

H_0^{7a}**:** *Clone removal by the developers is NOT dictated by the similarity of program text (without normalization) in the clone fragments.*

From the table, we can see that the average *actual* textual similarity of removed clones for four systems (`JabRef, ArgoUML, Conky`, and `Claws Mail`) is higher than that of alive clones, while the other two subject systems (`dnsjava` and `ZABBIX`) exhibit slightly the opposite trend. A *MWW test* ($P = 0.041$), on the data of actual textual similarity in the alive and removed clones, *rejects* (as, $P < \alpha$) the null hypothesis H_0^{7a}, which indicates statistically significant relationship between the textual similarity of clones and their removal.

Sometimes the actual textual similarity does not estimate the actual effort for refactoring. For example, in case of different identifiers names in different clone fragments, textual similarity of a clone-group may be very low although they are easily refactorable, especially when there is refactoring support from the IDEs (Integrated Development Environments). That is why we also investigated the normalized textual similarity by removing the identifier differences. If we look at the normalized text similarities of removed and alive clones, we again see that the average *normalized* textual similarity for the removed clones is slightly higher than the alive clones in those four systems. The trend is slightly the opposite for `dnsjava`, while for `ZABBIX`, both the removed and alive clones exhibit equal average normalized textual similarity. With respect to the normalized text similarities between the remove and alive clones, we formulate another hypothesis as follows:

H_0^{7b}**:** *Clone removal by the developers is NOT influenced by the similarity of* ***normalized*** *program text in the clone fragments.*

A *MWW* test ($P = 0.091$) *fails to reject* (as, $P > \alpha$) the null hypothesis H_0^{7b}, suggesting that there is *no statistically significant difference* in the normalized text similarities between the removed and alive clones. However, the P value in the case of normalized textual similarity is much lower than that of the actual textual similarity, and this hints that there might be some influence of the differences in the names of variables/identifiers over clone removal.

Combining the observations for the actual and normalized text similarities over the removed and alive clones in the light of the two null hypotheses (H_0^{7a} and H_0^{7b}), we can now derive the answer to the research question *RQ7* as follows.

Ans. to RQ7: *The textual similarity in the source code of the clones does have statistically significant effect in the removal of clones, and the differences in the names of the variable/identifiers play the major role in this regard.*

This finding indicates that *Type-1* clones are most attractive to the developer for refactoring, and the developers, in practice, are more inclined in refactoring *Type-2* clones than refactoring *Type-3*.

5.4.8 Age

The information about the age (in terms of the number of releases the clone-groups remain alive before removal) of clone genealogies can indicate how quickly the developers act to remove clones. In order to examine this phenomenon, for each of the systems, we computed the age of each clone-group that was removed in any of the subsequent releases.

In Figure 5.2, we present the proportion of clone-groups found to have been removed in a subsequent releases of systems written in C#. We found similar trend in the systems written in C and Java. The majority of the dead clones in five of the subject systems (`ArgoUML, JabRef, ZABBIX, Conky,` and `ZedGraph`) were removed within the initial five to ten releases. This observation is consistent with that reported by Kim et al. [154], suggesting that many of the clones are possibly *volatile*.

However, in all the systems, a good number clones remained alive over a long sequence of releases before their removal. For example, 17% of the refactored clone-groups in `ArgoUML` remained alive in 43 subsequent releases, while 35% of the clone-groups in `Claws Mail` propagated over 27 subsequent releases, before their removal. Similar trends were found in other systems as well. From the above discussion, we answer the research question *RQ8* as follows.

Ans. to RQ8: *During the evolution of the software systems, a few early releases experience significant clone removal. Nevertheless, some clones propagated over a relatively long sequence of releases before they were finally removed.*

This finding is also in keeping with the answer to the research question *RQ5* (Section 5.4.5), which indicates that most of the clones do not undergo frequent changes before their removal. We suspect that once a developer comes to know of a clone during its first change, this awareness might drive the removal of the clone at a later release. This indicates an area where informed clone management can play a vital role.

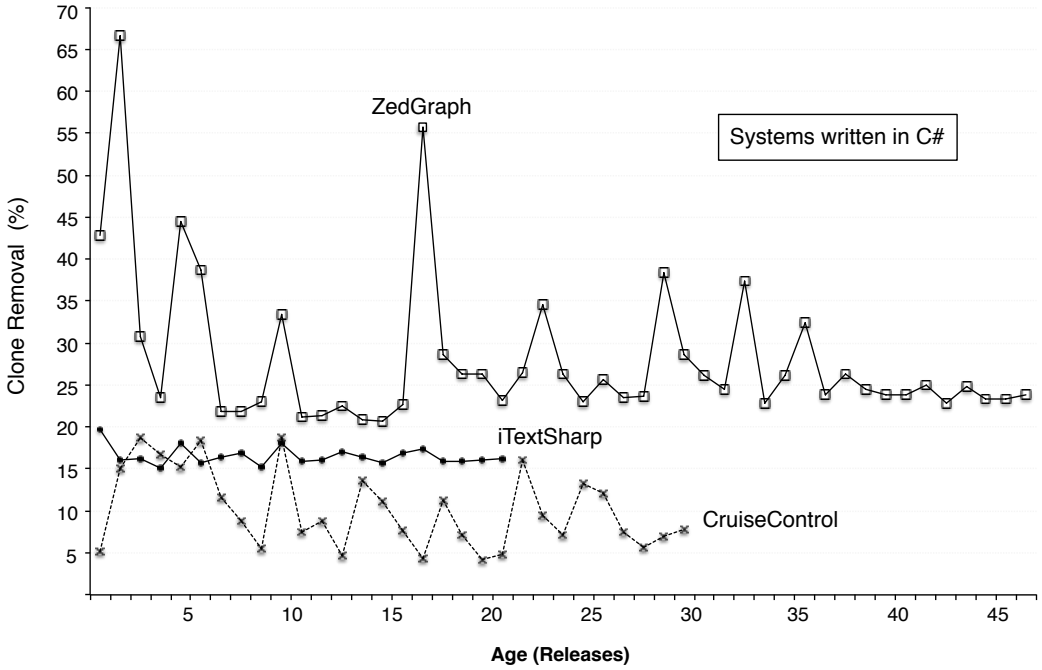


Figure 5.2: Clone Removal in Subsequent Releases of C# Systems

5.5 Threats to Validity

In this section, we discuss possible threats to the validity of our study and how we mitigated their effects.

Construct Validity: Perhaps the best way to investigate change and evolution of clones is to study of the individual clone fragments in terms of genealogies across versions of the system. As versions one might choose programmers' commit transactions or weekly/monthly snapshots of the code base, or the stable releases of the system. A number of the earlier studies [90, 154] used the programmers' commit transactions or weekly/monthly snapshots of the code base, while many other studies [248, 249, 313] used software releases as the versions.

Programmers often create clones for experimental purposes, which they remove shortly after creation [154]. Thus, daily, weekly or monthly snapshots can be too frequent to capture stable changes in the code base. Indeed, commit transactions are more susceptible to this issue, in addition to their sensitivity to the developers' commit styles [313]. However, when a version of a software is officially released, the source code is expected to be in a stable form. Moreover, even a large number of weekly/monthly revisions may correspond to only a few stable releases, whereas series of releases typically span a longer period of development time. Therefore, for our study, we selected stable releases of the systems instead of commit transactions or snapshots at certain time intervals.

Internal Validity: The internal validity of our study is subject to the accuracy in clone detection and genealogy extraction. The `NiCad` clone detector used in our study, is reported to be effective in detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones with high precision and recall [240, 238]. Moreover, our manual verification of random samples from the detected clones found no false positives. The genealogy extractor `gCad`, used in our study, is also reported to be accurate in the computation of near-miss clone genealogies [249]. Nevertheless, we carried out manual investigation to verify the correctness of the genealogies and to fix any inconsistencies. Indeed, the manual assessment can be subject to human errors. However, all the human participants of this study are faculty and graduate students carrying out research in the area of software clones, and thus we believe that they have affluent expertise to keep the probable human errors to the minimum.

External Validity: Our study is based on nine medium to fairly large open-source software systems, and thus one may question the *generalizability* of the findings. However, for each of the subject systems, we studied a significant number of releases, and we expect this to help minimize the threat to some extent. To further mitigate the threat, we carefully chose the subject systems from different application domains, and written in different programming languages.

Reliability: The methodology of this study including the procedure for data collection are documented in this chapter. The subject systems are open-source, while the `NiCad-2.6.3` clone detector and the extended version of `gCad` genealogy extractor are also available online[2]. Therefore, it should be possible to replicate the study.

5.6 Related Work

There has been considerable research in characterizing clone evolution and distinguishing clones of interest for refactoring. Genealogy-based studies on clone evolution in general are discussed in Section 4.6.3. In this section, we confine our discussion in those work that are particularly relevant to ours delineated in this chapter.

A number of attempts have been made to distinguish potential clones for refactoring and removal. From a manual analysis of 800 function/method level clones over six different open-source Java systems, Balazinska et al. [20] proposed a taxonomy of function clones based on the differences and similarities in the program elements. Based on the location of clones in the inheritance hierarchy, Koni-N'Sapu [158] proposed another clone taxonomy and a set of object-oriented refactoring patterns for refactoring each category of code clones. Later, Kapser and Godfrey [139] proposed a clone taxonomy based on the locations of clones in the file-system hierarchy and (dis)similarities in the code functionalities.

Schulze et al. [258] argued that aspect-oriented refactoring (AOR) can be more appropriate than object-oriented refactoring (OOR) in certain scenarios. They also proposed a code clone classification scheme to

[2]http://usask.ca/∼minhaz.zibran/pages/projects.html

support the decision of whether to use OOR or AOR for clone removal. Other techniques, such as design patterns [19] and traits [213] were also attempted to identify and refactor clones of interest. Torres [278] applied *formal concept analysis* and suggested that a classification of concepts containing duplicated code can provide hints about which refactoring can be suitable. He applied a concept-lattice based data mining approach to derive four categories of concepts containing duplicated code and suggested refactoring patterns suitable for refactoring clones in each of the categories.

Higo et al. [104] proposed a software-metrics-based approach to identify potential clones that can be easier to refactor using the *extract method* and *pull-up method* refactoring patterns. Variations of such metrics-based approaches are realized in tools namely `Gemini` [280] and `ARIES` [104]. Choi et al. [49] carried out a developer-centric study to determine the effectiveness of different combinations of metrics in distinguishing clones of interest for refactoring.

None of the aforementioned work was based on code clone genealogies as ours, where we examine the evolution of individual clone fragments to characterize the patterns of change and removal of clones. Based on the experience from an ethnographic study on copy and paste programming practices, Kim et al. [152] reported that "larger or frequently copied code fragments are good candidates for refactoring." From our study, we found evidence to partially support their conjecture. We found that clones that are larger in size, in terms of the number lines of code, appear to be attractive for refactoring, but the size of the clone-groups in terms of the number of clone fragments do not have significant impact on their removal, and most of the clones do not experience frequent changes before their removal.

Based on a case study with two open-source Java systems, Tairas and Gray [269] reported that in some cases clone refactorings were partially performed on only parts of the clones (i.e., sub-clones). However, their focus was only on the occurrences of refactorings composed of the *extract method* refactoring pattern. The objective of our work, as demonstrated in this chapter, was to investigate and characterize removal and refactoring of clones not only through the *extract method* refactoring patterns, but also by all other possible means.

Göde [90] conducted a case study over four systems, and investigated the extent clones were removed from the systems. He found many instances of deliberate clone removal, and the majority of those removals were performed by the *extract method* refactoring pattern. He further reported that the developers refactored mostly the closely located clones, which is also consistent with our findings as presented in this chapter. The study of Göde was based on only three metrics, and he concluded that more complex metrics such as change frequency of clones should be examined to better understand the phenomenon. Our study significantly differs from that of Göde. Based on clone genealogies over 228 releases of six software systems and using a wide range of metrics and characterization criteria, we capture a broader picture of clone removal and changes in open-source software systems.

5.7 Summary

This chapter presents a genealogy-based empirical study on the evolution of individual clone fragments to characterize the changes and removal of exact (*Type-1*) and near-miss (*Type-2* and *Type3*) code clones. We examined a total of 329 releases from nine open-source software systems written in Java, C, and C#.

In the study, we addressed eight research questions, and derived answers to those with a combination of qualitative and quantitative analyses as well as statistical tests of significance. The findings of our study shed light on the conventional wisdom about clone evolution, in particular, derive useful insights into the patterns of changes and removals of code clones in practice. We believe that the practical findings from this study make significant contributions to the existing wisdom about clone evolution, refactoring, and removal, which in turn, will be useful for devising effective tools and techniques for informed clone management.

From the study, we found that the sizes of the clone-groups (in terms of the number of member clone fragments), or the granularity (i.e., functions or blocks) of clones, or their dispersion in the file-system hierarchy do not have any significant effect on clone removal in practice. In terms of change patterns, we did not find any relationships between clone removal and any particular type (i.e., consistent or inconsistent) of changes. However, inconsistent changes are found to have dominated over consistent changes of code clones. We also found that the majority of clones that were removed, did not experience frequent changes before removal, and surprisingly, most of those clones underwent changes only once, before they were removed from their respective systems. A few early releases of the software systems are found to have experienced significantly more changes and removal of clones than the later releases.

During manual investigation, we discovered many instances of clones, which could be attractive for refactoring, but those were left alone, perhaps due to the lack of proper tool support for clone management and refactoring, or those clones escaped from the attention of the developers, or they were deliberately ignored by the developers considering the required effort to put within the tight development schedule. More importantly, we found that highly similar or larger clone fragments appeared to be attractive for removal.

Indeed, larger clone fragments can appear more visible to the developer while the refactoring of highly similar clone fragments can be easier, because refactoring of such clones deals with less variations in the clone fragments. However, we do not have any evidence suggesting that the removal of very similar or larger code clones can have higher positive impact on the software design than that from the removal of smaller code clones or clones with higher levels of dissimilarities. In this regard, a clone refactoring scheduler can play a vital role by suggesting a subset of most attractive clones for refactoring to maximize improvement in software design quality while keeping the refactoring effort within reach. In the next chapter (Chapter 6), we present our clone refactoring scheduler developed for this particular purpose.

Chapter 6

Refactoring Effort Estimation and Scheduling

"I believe there is no philosophical high-road in science, with epistemological signposts.
No, we are in a jungle and find our way by trial and error,
building our road behind us as we proceed."
– Max Born

In Chapter 3, we presented our IDE-based tool for the clone detection, which is an inseparable part of clone management. Once clones are identified, careful refactoring is what we need to remove unnecessary clones. From the analyses of code clones and their evolution, as described in Chapter 4 and Chapter 5, we found that some clones disappear during their evolution while many clones propagate through long series of subsequent releases of the software system. Moreover, clone removal in practice was not found to be affected by the sizes of the clone-groups, their granularity, or their distribution in the file-system hierarchy. Larger clone fragments can appear to be easily visible candidates for refactoring while clone fragments with less dissimilarities can be easier for refactoring. However, to make prudent decisions in choosing potential clones for refactoring one must take into account the required efforts and possible effects of refactoring on the overall quality of the source code. Addressing this issue, in this chapter, we present a new model for estimating refactoring efforts, and a novel scheduler for wisely choosing candidate clones for refactoring.

This chapter is organized as follows. In Section 6.1, we elaborate the problem description and its context. In Section 6.2, we identify the refactoring patterns that are suitable for code clone refactoring. In Section 6.3, we describe our clone refactoring effort model. Section 6.4 discusses how the effect of refactoring can be estimated by a developer. In Section 6.5, we describe the possible constraints on refactorings. In Section 6.6, we present our CSOP (Constraint Satisfaction Optimization Problem) formulation of the refactoring scheduling problem. In Section 6.8, we illustrate our empirical study to evaluate our refactoring scheduler and the effort model. In Section 6.9, we discuss related work. Finally, in Section 6.10, we conclude the chapter.

6.1 Introduction

Although cloning is a common code reuse mechanism, code clones may also be detrimental in many cases [133, 142, 154, 222, 227, 274]. Nevertheless, in many cases, code clones are unavoidable or even desirable. Yet, to prevent code inflation and reduce maintenance cost, the number of code clones should be minimized by

applying justified refactoring. However, refactoring is not free, rather it is often risky as it might introduce new bugs and hidden dependencies. Moreover, not all clones can always be feasible targets of refactoring [44, 248]. Therefore, it is important to have a prioritized refactoring schedule of the potential refactoring candidates (i.e., clones that are refactorable) so that the maintenance engineers can focus on a short list of refactoring candidates considering the existing constraints, potential benefits, risks, and available resources.

There are many patterns [81, 80] for refactoring source code in general. Given a context, a refactoring pattern describes a sequence of refactoring activities (i.e., modification operations) that can be performed to improve code/design quality. However, not all of the refactoring patterns are directly applicable to code clone refactoring. The applicability of a certain refactoring largely depends on the context (e.g., dependency of the code fragment under consideration with the rest of the source code). Therefore, for code clones, refactoring activities and the relevant contexts must to be identified in the first place. The consequences of clone refactoring (e.g., impact on the code/design quality) should also be taken into account. The effort required for applying certain refactoring on the code clones should also be minimized to keep the maintenance cost within reach. The application of a subset of refactorings from a set of applicable refactorings may result in distinguishable impact on the overall code quality. Moreover, there may be sequential dependencies and conflicts among the refactorings, which lead to the necessity that, from all refactoring candidates a subset of non-conflicting refactorings be selected and ordered for application, such that the quality of the code base is maximized while the required effort is minimized [309].

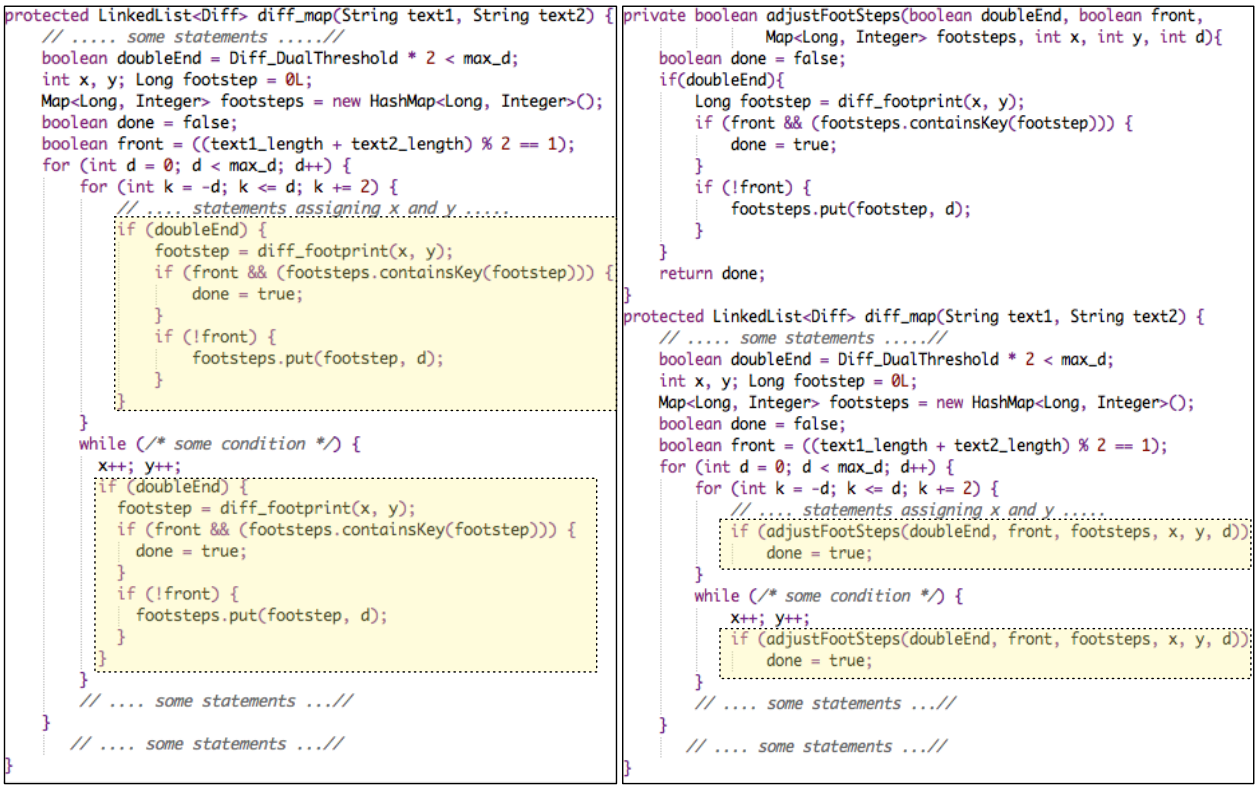
Automated software refactoring is often performed with the aid of graph transformation tools [223], where the available refactorings are applied without having been optimally scheduled [183]. The application order of the semi-automated (or manual) refactorings is usually determined implicitly by human developers. However, developers' efforts may be inefficient and error-prone, especially for large systems. While experienced developers may do it well, inexperienced ones may build a poor/infeasible schedule. The challenge is likely to be more severe when refactoring legacy systems or when a developer new to the code base must devise the refactoring schedule. Therefore, automated (or semi-automated) scheduling for performing selection and ordering of refactorings from a set of refactorings is a justified need. However, such a scheduling of code clone refactoring is a *NP-hard* problem [37, 176, 183] and, thus, the complexity of a problem instance grows exponentially for large systems having many code clones.

In this regard, this work makes two contributions. First, we introduce an *effort model* for estimating the developers' effort required to refactor code clones in procedural or object-oriented (OO) programs. Second, taking into account the *effort model* and a wide variety of possible hard and soft constraints, we formulate the scheduling of code clone refactorings as a constraint satisfaction optimization problem (CSOP) and solve it by applying Constraint Programming (CP) techniques that aims to maximize benefits (measured in terms of changes in the code/design quality metrics) while minimizing refactoring efforts.

To the best of our knowledge, ours is the first effort model for refactoring object-oriented source code and we are the first to apply CP techniques in the scheduling of code clone refactorings. We choose to adopt CP

for two main reasons. First, CP is a natural fit for solving CSOPs such as scheduling problems. Second, this technique integrates the strengths from both artificial intelligence (AI) and operations research (OR) and has been shown efficient in solving CSOPs [24, 306].

To evaluate the effectiveness of our scheduler and the code clone refactoring effort model, we also conduct an empirical study on six software systems written in Java. We find that our scheduler is capable of efficiently computing the optimal refactoring schedule and our refactoring effort model offers significant help in the estimation of the refactoring efforts.



```
protected LinkedList<Diff> diff_map(String text1, String text2) {
    // ..... some statements .....//
    boolean doubleEnd = Diff_DualThreshold * 2 < max_d;
    int x, y; Long footstep = 0L;
    Map<Long, Integer> footsteps = new HashMap<Long, Integer>();
    boolean done = false;
    boolean front = ((text1_length + text2_length) % 2 == 1);
    for (int d = 0; d < max_d; d++) {
        for (int k = -d; k <= d; k += 2) {
            // ..... statements assigning x and y .....
            if (doubleEnd) {
                footstep = diff_footprint(x, y);
                if (front && (footsteps.containsKey(footstep))) {
                    done = true;
                }
                if (!front) {
                    footsteps.put(footstep, d);
                }
            }
        }
        while (/* some condition */) {
          x++; y++;
          if (doubleEnd) {
            footstep = diff_footprint(x, y);
            if (front && (footsteps.containsKey(footstep))) {
              done = true;
            }
            if (!front) {
              footsteps.put(footstep, d);
            }
          }
        }
        // .... some statements ...//
    }
      // .... some statements ...//
}
```

```
private boolean adjustFootSteps(boolean doubleEnd, boolean front,
            Map<Long, Integer> footsteps, int x, int y, int d){
    boolean done = false;
    if(doubleEnd){
        Long footstep = diff_footprint(x, y);
        if (front && (footsteps.containsKey(footstep))) {
            done = true;
        }
        if (!front) {
            footsteps.put(footstep, d);
        }
    }
    return done;
}
protected LinkedList<Diff> diff_map(String text1, String text2) {
    // ..... some statements .....//
    boolean doubleEnd = Diff_DualThreshold * 2 < max_d;
    int x, y; Long footstep = 0L;
    Map<Long, Integer> footsteps = new HashMap<Long, Integer>();
    boolean done = false;
    boolean front = ((text1_length + text2_length) % 2 == 1);
    for (int d = 0; d < max_d; d++) {
        for (int k = -d; k <= d; k += 2) {
            // ..... statements assigning x and y .....
            if (adjustFootSteps(doubleEnd, front, footsteps, x, y, d))
                done = true;
        }
        while (/* some condition */) {
            x++; y++;
            if (adjustFootSteps(doubleEnd, front, footsteps, x, y, d))
                done = true;
        }
        // .... some statements ...//
    }
      // .... some statements ...//
}
```

Figure 6.1: Example of clone refactoring in VisCad: the method on the top-right corner is extracted by generalizing the clone pairs (shaded blocks on the left)

6.2 Clone Refactoring

There have been immense research in software refactoring in general. Fowler et al. [81] initially proposed a set of 72 software refactoring patterns and until recently the number has increased to 93 [80]. Those patterns of refactorings in general are meant to get rid of different types of code smells (including duplicated code) and to prevent software decay (i.e., loss of code/design quality) [4]. Based on a survey of existing literature [104, 105, 156, 176, 257, 300] and our experience, we find that among those general software refactoring patterns [81, 80], the following patterns are particularly suitable for code clone refactoring. Detail about these refactoring patterns can be found elsewhere [81, 80].

- **Extract method (EM)** extracts a block of code as a new method and replaces that block by a call to the newly introduced method. EM may cause splitting of a method into pieces. For code clone refactoring, similar blocks of code can be replaced by calls to an extracted generalized method. Figure 6.1 shows an example of the *extract method* refactoring.

- **Pull-up method (PM)** removes similar methods found in several classes by introducing a generalized method in their common superclass. Figure 6.2(a) demonstrates a *pull-up method* refactoring through a schematic diagram.

- **Extract superclass (ES)** introduces a new common superclass for two or more classes having similar methods and then applies *pull-up method*. *Extract superclass* refactoring may be necessary when those classes do not already have a common superclass and those classes can be brought under a common superclass. Figure 6.2(b) presents an a schematic diagram demonstrating the *extract superclass* refactoring pattern.

- **Extract utility-class (EU)** is applicable in situations where similar functions are found in different classes, but those classes do not conceptually fit to undergo a common superclass. A new class is introduced that accommodates a method generalizing the similar methods that must be removed from those classes. Figure 6.2(c) demonstrates *extract utility-class* refactoring through a schematic diagram.

A refactoring pattern is composed of a sequence of other refactoring patterns or low-level modification operations such as *identifier renaming*, *method parameter re-ordering*, *changes in type declarations*, *splitting of loops*, *substitution of conditionals*, *loops*, *algorithms*, and *relocation of method or field*, which may be necessary to produce generalized blocks of code from near-miss (similar, but not exact duplicate) clones. For the purpose of formulation, we use the term *refactoring operators* to denote both the composite refactoring patterns and low-level modification operations.

For code clone refactoring, these refactoring operators will operate on groups of clone fragments (i.e., code blocks that are clones of each other) having two or more members. We refer to such clone-groups as the refactoring *operands* or *candidates*. Thus, a *refactoring activity* (or simply, refactoring) r can be formalized as:

$$r = \langle op, g \rangle, \text{where } op \in \{EM, PM, ES, EU, ...\}$$

and g is the clone-group on which the refactoring operator op operates. More than one refactoring operators may be needed to refactor the same clone-group and, thus, a complete refactoring of a clone-group may require more than one refactoring activity.

Let us consider a clone-group, $g = \{c_1, c_2, c_3, \ldots, c_n\}$, where c_i $(1 \leq i \leq n)$ is a clone fragment inside method m_i, which is a member of class C_i hosted in file F_i contained in directory D_i. Mathematically,

$$c_i^\frown m_i^\frown C_i^\frown F_i^\frown D_i, \qquad \text{for object-oriented code.}$$

$$c_i^\frown m_i^\frown F_i^\frown D_i, \qquad \text{for procedural code.}$$

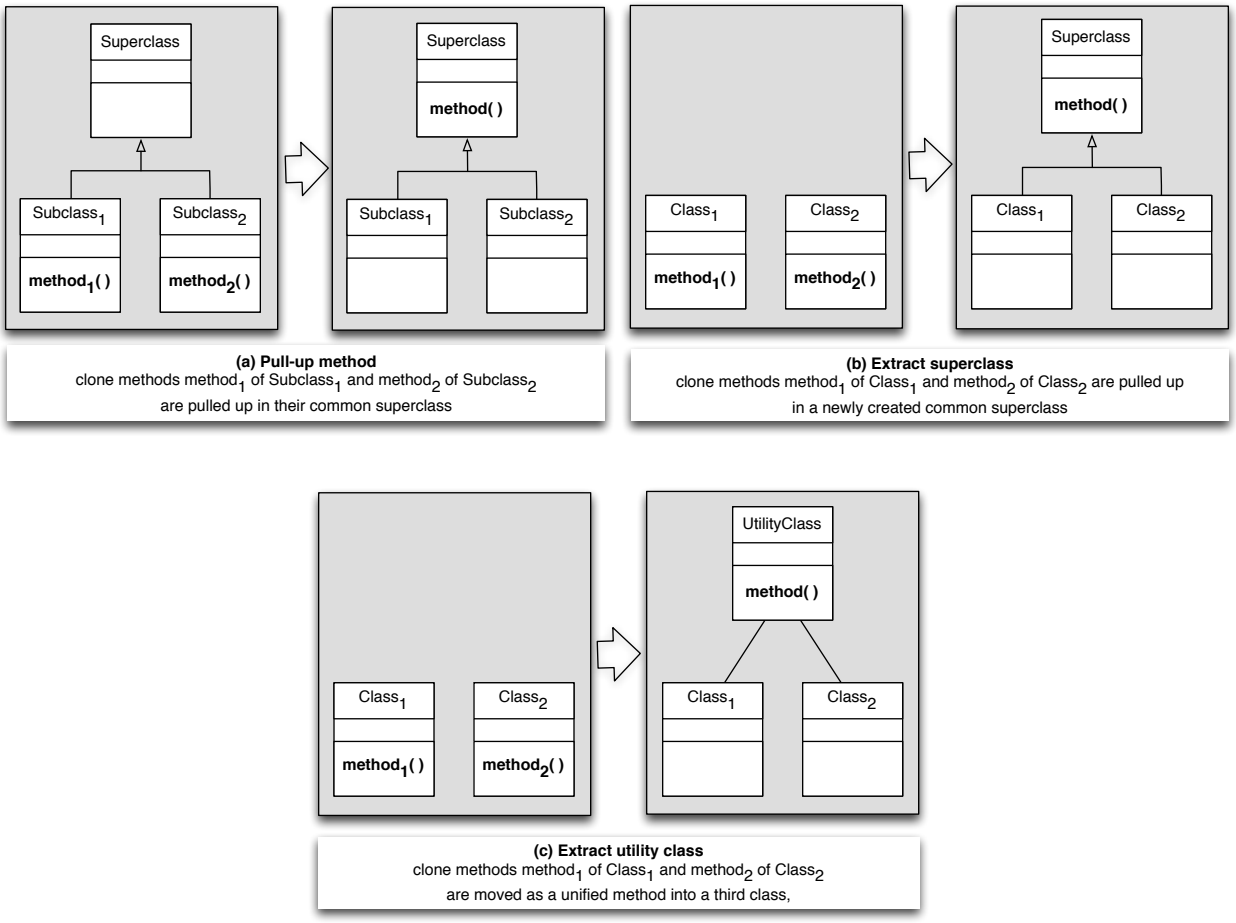


Figure 6.2: Different object-oriented patterns for code clone refactoring

Where, the symbol $\curvearrowright$ indicates a containment relationship. $x \curvearrowright y$ means that x is contained in y, in other words, y contains x. The relationship preserves transitive property, i.e., $x \curvearrowright y \curvearrowright z \Rightarrow x \curvearrowright z$. Thus, the set $C(g)$ of all classes hosting the clone fragments in g can be defined as:

$$C(g) = \{C_i \quad | \forall\ c_i \in g, c_i \curvearrowright C_i\} \tag{6.1}$$

We use this notation in subsequent sections of this chapter, in particular, in formalizing the navigation effort in Section 6.3.3.

6.3 Estimation of Refactoring Effort

The effort required for code clone refactoring is likely to depend on the type of refactoring operators and operands. For example, applying the *extract method* refactoring pattern on exact duplicate code clones will require less effort than that for applying it on near-miss code clones. Moreover, refactoring clone code fragments that are scattered across different locations of the source code with respect to the file system or inheritance hierarchy may require relatively more effort than that for refactoring clones residing cohesively at a certain location of the source code. To address these issues, we propose a code clone *refactoring effort model*, which, to the best of our knowledge, is the first model for the estimation efforts needed to refactor code clones in procedural and object-oriented source code. We have formulated this *effort model* based on our understandings, developed from a survey of existing literature [4, 65, 81, 80, 105, 198, 261] and our experience in clone refactoring.

6.3.1 Context Understanding Effort

The applicability of refactoring on certain code clones is largely dependent on the context. The context captures the relationship of a certain code fragment with the rest of the source code. Therefore, before refactoring, the developer must understand the context pertaining to the refactoring candidate at hand. Code clone refactoring may result in removal or relocation of code fragments that may span a block of code or an entire method/function or even an entire class. Such removal or relocation of code fragments may cause changes in the underlying inheritance hierarchy and method call-graph. Hence, for understanding the context and the possible impact of changes, the developer must examine the caller-callee delegation of methods and the inheritance hierarchy.

Effort for Understanding Method Delegation. A certain refactoring under consideration may cause the clone fragment to move to a different location (e.g., class, package). Such a relocation may hinder the visibility of any methods to which the clone fragment refers. Moreover, if a function clone is relocated, all references to the original function must be updated accordingly. To understand the delegation of methods involving the concerned code fragment $c_i \in g$, the developer must understand the chain of methods that can be reached from c_i via caller-callee relationships. Let $M_r(c_i)$ be the set of all such methods. The developer

must also comprehend the set $M_f(c_i)$ of all the methods from which c_i can be reached via caller-callee relationships. Then, the set of methods to be investigated for understanding the delegation effort concerning c_i is determined as:

$$delegation(c_i) = M_f(c_i) \cup M_r(c_i) \cup \{m_i\}. \tag{6.2}$$

Hence, for understanding delegation concerning all the clone fragments in g, the set of methods required to examine, becomes:

$$Delegation(g) = \bigcup_{c_i \in C(g)} delegation(c_i). \tag{6.3}$$

Thus, for the clone-group g, the total effort for understanding method delegation can be estimated as:

$$E_d(g) = \sum_{m \in Delegation(g)} LOC(m) \tag{6.4}$$

where, $LOC(m)$ computes the total lines of code in method m including the comments, but excluding all blank lines.

Effort for Understanding Inheritance Hierarchy. Suppose that $C_p(g)$ is the set of all lowest/closest common superclasses of all pairs of classes in $C(g)$[1]. The developer must also understand those classes in the inheritance hierarchy that have overridden (in subclasses) or referred to method m_i containing any code clone $c_i \in g$. Let $C_s(g)$ be the set of all such classes. Then, $C_h(g) = \{C_p(g) \cup C_s(g) \cup C(g)\}$ becomes the set of all classes required to be examined for understanding the inheritance hierarchy concerning the code clones in g and the effort $E_h(g)$ required for this can be estimated as:

$$E_h(g) = \sum_{C \in C_h(g)} LOC(C). \tag{6.5}$$

6.3.2 Effort for Code Modifications

To perform refactoring on the target clones, the developer must edit at different locations in the source code. The effort needed to perform such edits can be captured in terms of token modifications and code relocations.

Token Modification Effort. Developer's source code modification activities typically include modifications in the program tokens (e.g., identifier renaming). Let $T = \{t_1, t_2, t_3, \ldots, t_k\}$ be the set of tokens such that a token $t_i \in T$ is required to be modified to t'_i and the edit distance between t_i and t'_i is denoted as $\delta(t_i, t'_i)$. Then, the total effort $E_t(g)$ for token modifications can be estimated as:

$$E_t(g) = \sum_{i=1}^{k} \delta(t_i, t'_i). \tag{6.6}$$

[1]For any two Java classes C_i and C_j containing code clones c_i and c_j respectively, there may be at most one lowest common superclass, as Java does not support multiple inheritance. Any Java class is a subclass of the `Object` class. If this `Object` class is found to be the lowest common superclass of any pair of classes, this can be ignored, and those classes are considered to have no common superclass.

However, the state-of-the-art IDEs (Integrated Development Environments) such as Eclipse and NetBeans facilitate identifier/variable renaming simply by select-replace operations with the help of graphical user interfaces. Thus, with the availability of such tool support, the edit distance $\delta(t_i, t'_i)$ can simply be replaced by a constant μ. By default, $\mu = 1$, but the developer can set a different value to μ as appropriate.

Code Relocation Effort. When developers must move a piece of code from one place to another, they typically select a block of adjacent statements and relocate them all at a time. Hence, the code relocation effort $E_r(g)$ can be estimated as:

$$E_r(g) = |\beta|$$

where β is the set of all non-adjacent blocks of code that must be relocated to perform the refactoring. $|\beta|$ denotes the number of blocks in the set β.

6.3.3 Navigation Effort

Effort for source code comprehension, modification, and relocation is also dependent on the number of files and directories involved and their distributions in the file-system hierarchy. To capture this effort, our model includes the notion of navigation effort, $E_n(g)$, calculated as follows:

$$E_n(g) = |F_d(g) \cup F_h(g)| + |D_d(g) \cup D_h(g)| + DCH(g) + DFH(g) \tag{6.7}$$

where:

$$\begin{aligned}
F_d(g) &= \{F_i \mid m_i \frown F_i, m_i \in Delegation(g)\} \\
F_h(g) &= \{F_i \mid C_i \frown F_i, C_i \in C_h(g)\} \\
D_d(g) &= \{D_i \mid F_i \frown D_i, F_i \in F_d(g)\} \\
D_h(g) &= \{D_i \mid F_i \frown D_i, F_i \in F_h(g)\} \\
DCH(g) &= \max_{C_i, C_j \in C_h(g)} \{\partial(C_i, C_j)\} \\
DFH(g) &= \max_{F_i, F_j \in F_d(g) \cup F_h(g)} \{\eth(F_i, F_j)\}
\end{aligned}$$

Here, $DCH(g)$ refers to the *dispersion of class hierarchy* with $\partial(C_i, C_j)$ denoting the distance between class C_i and class C_j in the inheritance hierarchy. The distance between any two classes C_i and C_j is computed based on an abstract directed graph where each node represent a class and their exists an edge between each superclass and its subclass. Let C_p be the lowest common superclass of both C_i and C_j. Then, $\partial(C_i, C_j) = max\{pathLength(C_i, C_p), pathLength(C_j, C_p)\}$, where $pathLength(C_i, C_p)$ is measured as the number of edges in the shortest path from C_i to C_p. In Figure 6.3, the computation of distance between classes is illustrated with an example. More detail about $DCH(g)$ can be found elsewhere [104]. $DFH(g)$ is a similar metric that captures the *dispersion of files* with $\eth(F_i, F_j)$ denoting the distance between files F_i and F_j in the file-system hierarchy.

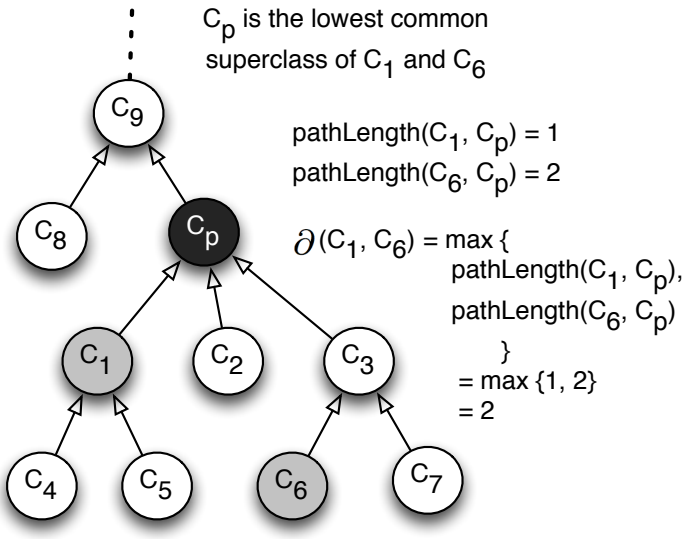


Figure 6.3: Computation of distance between classes

6.3.4 Effort Model

Based on the criteria discussed before, the total effort $E(g)$ needed to refactor clone-group g is estimated as:

$$E(g) = \ w_d \times E_d(g) + w_h \times E_h(g) + w_t \times E_t(g) + w_r \times E_r(g) + w_n \times E_n(g) \tag{6.8}$$

where w_d, w_h, w_t, w_r, and w_n are respectively the weights on the efforts for understanding method delegation, understanding inheritance hierarchy, token modification, code relocation, and navigation. By default, they are set to one, but the developer may assign different weights to penalize certain types of efforts.

6.4 Prediction of Refactoring Effects

The expected benefit from code clone refactoring is the structural improvement in the source code, which should also enhance the software design quality. Obvious expected benefits include reduced source lines of code (SLOC), less redundant code, to name a few. For procedural code, procedural metrics (e.g., SLOC, Cyclomatic Complexity) as well as structural metrics (e.g., fan-in, fan-out, and information flow) can be used to estimate software quality after refactoring. For object-oriented systems, these metrics can be supplemented by object-oriented design quality models, such as *QMOOD* [22] or design quality metrics, such as the *Chidamber-Kemerer* [47] metric suite. Quantitative or qualitative estimation of the effect of refactoring on the quality metrics can be possible before the actual application of the refactoring [37, 176, 252, 262, 266].

Having chosen a suitable set of quality attributes; let $Q = \{q_1, q_2, q_3, \ldots, q_\eta\}$ be the set of quality attribute values before refactoring and $Q_r = \{q'_1, q'_2, q'_3, \ldots, q'_\eta\}$ be the estimated values of those quality attributes after applying refactoring r. The impact of a certain refactoring r in code/design quality can be assessed by

Table 6.1: QMOOD formula for quality attributes (taken from [22])

Attribute		Formula
Reusability	=	$-0.25 \times \text{DCC} + 0.25 \times \text{CAM} + 0.5 \times \text{CIS}$ $+0.5 \times \text{DSC}$
Flexibility	=	$0.25 \times \text{DAM} - 0.25 \times \text{DCC} + 0.5 \times \text{MOA}$ $+0.5 \times \text{NOP}$
Understandability	=	$-0.33 \times \text{ANA} + 0.33 \times \text{DAM} - 0.33 \times \text{DCC}$ $+0.33 \times \text{CAM} - 0.33 \times \text{NOP} - 0.33 \times \text{NOM}$ $-0.33 \times \text{DSC}$
Functionality	=	$0.12 \times \text{CAM} + 0.22 \times \text{NOP} + 0.22 \times \text{CIS}$ $+0.22 \times \text{DSC} + 0.22 \times \text{NOH}$
Extendability	=	$0.5 \times \text{ANA} - 0.5 \times \text{DCC} + 0.5 \times \text{MFA}$ $+0.5 \times \text{NOP}$
Effectiveness	=	$0.2 \times \text{ANA} + 0.2 \times \text{DAM} + 0.2 \times \text{MOA}$ $+0.2 \times \text{MFA} + 0.2 \times \text{NOP}$

comparing the quality attributes before and after performing that particular refactoring. Hence, the total gain (or loss) in quality $\overline{Q}_r$ from refactoring r can be estimated as:

$$\overline{Q}_r = \sum_{j=1}^{\eta} \vartheta_j \times (q'_j - q_j) \tag{6.9}$$

where ϑ_j is the weight on the j^{th} quality attribute. By default $\vartheta_j = 1$, but the developers can assign different values to impose more or less emphasis on certain quality attributes.

In our work, we use the QMOOD design quality model for estimating the effect of refactoring on object-oriented source code. QMOOD is a prominent quality model for object-oriented systems, which is used by other researchers [37, 183, 176]. We choose QMOOD because this quality model has the advantage that it defines six high-level design quality attributes (Table 6.1) from the 11 lower level structural property metrics (Table 6.2). Indeed, the sum of differences in Equation 6.9 may not be able to utilize the fullest benefit of the quality model, but it serves our purpose.

6.5 Refactoring Constraints

Among the applicable refactorings, there may be conflicts and dependencies [198] besides their distinguishable impacts on the design quality. The application of one refactoring may cause the operands of other refactorings to disappear and thus may invalidate their applicability [37, 176, 198]. Besides such *mutual exclusion* on conflicting refactorings, the application of one refactoring may also reveal new refactoring opportunities, as suggested by Lee et al. [176]. We understand that these are largely due to the composite structure of

Table 6.2: QMOOD metrics for design properties (taken from [22])

Design Property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

certain refactoring patterns, where one larger refactoring is composed of several smaller core refactorings [4]. For example, when *extract superclass* refactoring is applied, *pull-up method* is also applied as a part of it (Figure 6.2(b)).

There may also be *sequential dependencies* between refactoring activities [176, 198]. Constraints of *mutual inclusion* may also arise when the application of one refactoring will necessitate the application of certain other refactorings [300]. Figure 6.4(a) presents an example of mutual inclusion constraint and Figure 6.4(b) demonstrates a mutual exclusion constraint with an example. The organization's management may also impose *priorities* on certain refactoring activities [37], for example, lower priorities on refactoring clones in the critical parts of the system. We identify such priorities as soft constraints in addition to the following three types of hard constraints.

Definition 7 (Sequential dependency) *Two refactorings r_i and r_j are said to have sequential dependency, if r_i cannot be applied after r_j. This is denoted as $r_j \nrightarrow r_i$.*

Definition 8 (Mutual exclusion) *Two refactorings r_i and r_j are said to be mutually exclusive, if both $r_i \nrightarrow r_j$ and $r_i \nleftarrow r_j$ holds. The mutual exclusion between r_i and r_j is denoted as $r_i \nleftrightarrow r_j$. Thus, $r_i \nleftrightarrow r_j$ implies both $r_j \nrightarrow r_i$ and $r_i \nrightarrow r_j$.*

Definition 9 (Mutual inclusion) *Two refactorings r_i and r_j are said to be mutually inclusive, if r_i is applied, r_j must also be applied before or after r_i, and vice versa. This is denoted as $r_i \leftrightarrow r_j$.*

The complete independence of r_i and r_j is expressed as $r_i \perp r_j$. For further detail about the refactoring constraints with concrete examples, interested readers are referred to elsewhere [37, 176, 198, 300].

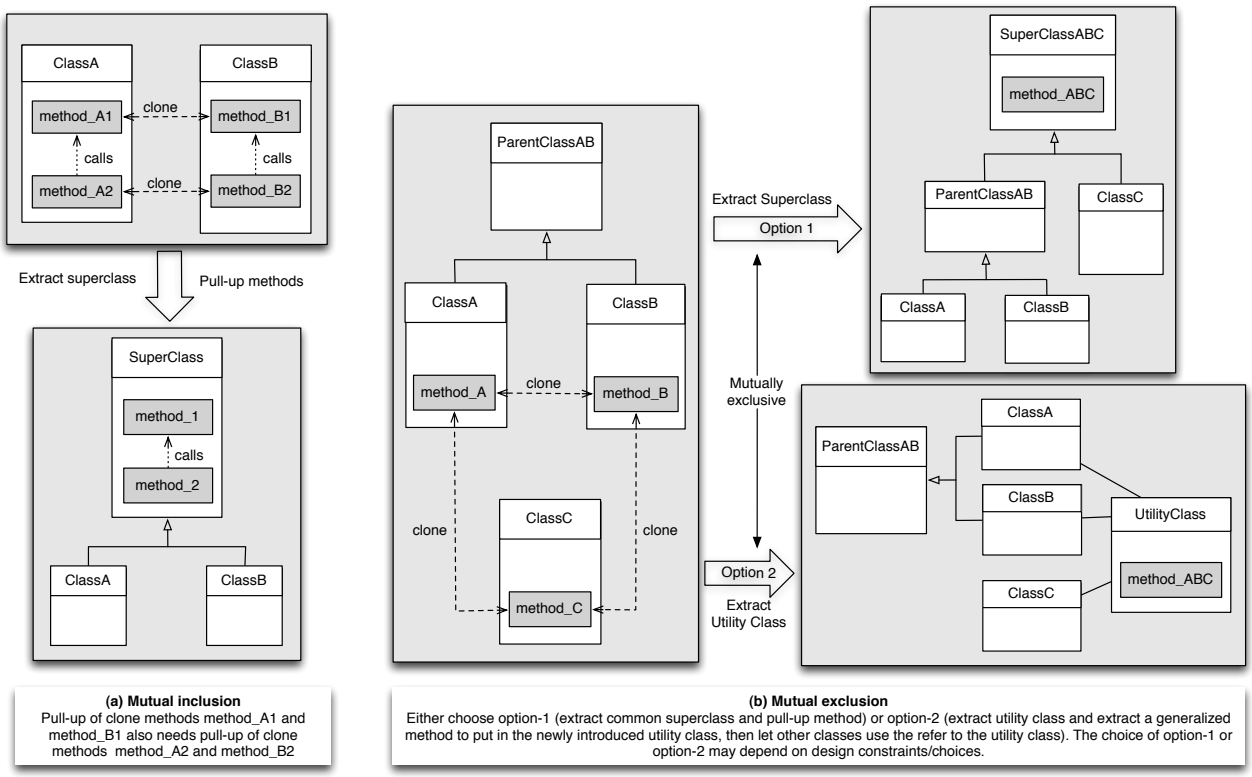


Figure 6.4: Mutual inclusion and mutual exclusion constraints on clone refactoring

6.6 Formulation of Refactoring Schedule

Upon identification of all the hard and soft constraints pertaining to a scheduling problem instance, computing an optimal refactoring schedule to maximize code quality while minimizing efforts is a NP-hard problem [37, 176, 183]. Finding the optimum solution for a large instance of such a problem becomes practically too expensive (time consuming) and, thus, a feasible optimal (near-optimum) solution is desired. However, the problem is by nature a Constraint Satisfaction Optimization Problem (CSOP). A CSOP is a kind of problem characterized by a set of constraints that must be satisfied and among all the feasible solutions, the best possible solution is desired. A solution is said to be feasible if it satisfies all the constraints. A solution is evaluated better than another based on an objective function, which a solver strives to optimize (i.e., maximize or minimize). We model the refactoring scheduling problem as a CSOP and solve it by applying a *constraint programming* technique, which no one reported to have applied before.

Having identified the set $\mathcal{R}$ of potential refactoring activities, we define two decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$:

$$\mathcal{X}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ 1 & \text{if } r \in \mathcal{R} \text{ is chosen} \end{cases}$$

$$\mathcal{Y}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ k & \text{if } r \in \mathcal{R} \text{ is chosen as the } k^{th} \text{ activity} \end{cases}$$

where $1 \leq k \leq |\mathcal{R}|$. Thus, $\mathcal{X}_r$ captures whether a refactoring r is included in the schedule and $\mathcal{Y}_r$ captures the order of refactoring r in the selected schedule of refactorings. We also define a $|\mathcal{R}| \times |\mathcal{R}|$ constraint matrix $\mathcal{Z}$ to capture the constraints and sequential dependencies between distinct refactorings r_i and r_j:

$$\mathcal{Z}_{i,j} = \begin{cases} 0 & \text{if } r_i \perp r_j \\ 1 & \text{if } r_i \nleftrightarrow r_j \\ +2 & \text{if } r_j \nrightarrow r_i \text{ and } r_i \leftrightarrow r_j \\ -2 & \text{if } r_i \nrightarrow r_j \text{ and } r_i \leftrightarrow r_j \\ +3 & \text{if } r_j \nrightarrow r_i, \text{ but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \\ -3 & \text{if } r_i \nrightarrow r_j, \text{ but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \end{cases}$$

$\mathcal{Z}_{i,j} = -\mathcal{Z}_{j,i}$ or $\mathcal{Z}_{i,j} = \mathcal{Z}_{j,i} = 1$, for all $\langle i, j \rangle$. Let ρ_r be the priority on the refactoring r that operates on clone-group g_r. The CSOP formulation of the refactoring scheduling problem can be defined as follows.

$$\text{maximize} \quad \sum_{r \in \mathcal{R}} \mathcal{X}_r \rho_r (\overline{Q}_r - E(g_r)) \tag{6.10}$$

subject to (with $i \neq j$),

$$\mathcal{X}_r + \mathcal{Y}_r \neq 1, \qquad \forall\ r \in \mathcal{R} \tag{6.11}$$

$$\mathcal{X}_{r_i} + \mathcal{X}_{r_j} = 2 \Rightarrow \mathcal{Y}_{r_i} \neq \mathcal{Y}_{r_j}, \qquad \forall\ r_i, r_j \in \mathcal{R} \tag{6.12}$$

$$\mathcal{Z}_{i,j} - \mathcal{Z}_{j,i} > 0 \Rightarrow \mathcal{Y}_{r_i} < \mathcal{Y}_{r_j}, \qquad \forall\ r_i, r_j \in \mathcal{R} \tag{6.13}$$

$$\mathcal{Z}_{i,j} - \mathcal{Z}_{j,i} < 0 \Rightarrow \mathcal{Y}_{r_i} > \mathcal{Y}_{r_j}, \qquad \forall\ r_i, r_j \in \mathcal{R} \tag{6.14}$$

$$|\mathcal{Z}_{i,j}| = 1 \Rightarrow \mathcal{X}_{r_i} + \mathcal{X}_{r_j} \leq 1, \qquad \forall\ r_i, r_j \in \mathcal{R} \tag{6.15}$$

$$|\mathcal{Z}_{i,j}| = 2 \Rightarrow (\mathcal{X}_{r_i} + \mathcal{X}_{r_j})\ \mathit{modulo}\ 2 = 0, \qquad \forall\ r_i, r_j \in \mathcal{R} \tag{6.16}$$

$$\sum_{r \in \mathcal{R}} \mathcal{X}_r \leq \mathcal{M} \tag{6.17}$$

Here, Equation 6.10 is the objective function for maximizing the code quality and minimizing the refactoring effort while rewarding refactoring activities having higher priorities. One of the product term in the objective function is $\mathcal{X}_r$, which is equal to 1 only for selected refactorings, and for all other refactorings $\mathcal{X}_r$ equals to 0. Thus, the objective function takes into account the priority, quality, and efforts pertaining to only the selected refactorings.

Equation 6.11 ensures that the decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$ are kept consistent as their values are assigned. If the refactoring r is not selected (i.e., $\mathcal{X}_r = 0$), then $\mathcal{Y}_r$ must also be 0, to denote that the refactoring r is not assigned any position in the sequence of the scheduled refactorings. If the refactoring r is selected (i.e., $\mathcal{X}_r = 1$), then $\mathcal{Y}_r$ must not be zero, i.e., $\mathcal{X}_r + \mathcal{Y}_r \neq 1$.

Equation 6.12 enforces that no two refactorings are scheduled at the same point in the sequence. Equation 6.13 and Equation 6.14 impose the sequential dependency constraints (i.e., $r_i \nrightarrow r_j$) on feasible schedules. Mutual exclusion (i.e., $r_i \nleftrightarrow r_j$) and mutual inclusion (i.e., $r_i \leftrightarrow r_j$) constraints are enforced by Equation 6.15 and Equation 6.16 respectively. Equation 6.17 specifies that maximum $\mathcal{M}$ number of refactorings can be chosen for scheduling. By default $\mathcal{M} = |\mathcal{R}|$ but $\mathcal{M}$ can be set to a lower integer when a schedule of a certain number of refactoring activities is desired, due to limitation of time, resource, or the like.

6.6.1 An Illustrative Example

Now, with an example, we further illustrate our formulation, especially the constraint matrix $\mathcal{Z}$. Consider a set of five refactorings $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ having constraints as follows:

(i) $r_2 \nleftrightarrow r_4$ (i.e., r_2 and r_4 are mutually exclusive)

(ii) $r_4 \nrightarrow r_1$ (i.e., r_1 cannot be applied after r_4)

(iii) $r_5 \nrightarrow r_3$ (i.e., r_3 cannot be applied after r_5)

(iv) $r_3 \leftrightarrow r_5$ (i.e., r_3 and r_5 are mutually inclusive)

Table 6.3: Constraint matrix $\mathcal{Z}$ representing the constraints among the refactorings in $\mathcal{R}$

	r_1	r_2	r_3	r_4	r_5
r_1				$+3$	
r_2				1	
r_3					$+2$
r_4	-3	1			
r_5			-2		

Other than the above mentioned constraints, any two refactorings $\langle r_i, r_j \rangle \in \mathcal{R}$ are independent (i.e., $r_i \perp r_j$). The constraint (iii) and constraint (iv) above jointly enforces that if either of r_3 and r_5 is selected, the other refactoring must also be selected and then r_3 must also be scheduled before r_5. According to the constraint-specifications, a valid constraint matrix $\mathcal{Z}$ is shown in Table 6.3. The empty cells in the table are filled up will zeros, which we omitted here for the purpose of better readability.

The constraint (i) is enforced by Equation 6.15. Here, $\mathcal{Z}_{2,4} = \mathcal{Z}_{4,2} = 1$. If both r_2 and r_4 are selected then $\mathcal{X}_{r_1} + \mathcal{X}_{r_2} = 1 + 1 = 2$, which violates the constraint in Equation 6.15.

The constraint (ii) is imposed by the Equation 6.13 and Equation 6.14. Here, $\mathcal{Z}_{1,4} = +3$ and $\mathcal{Z}_{4,1} = -3$ and thus, $\mathcal{Z}_{1,4} - \mathcal{Z}_{4,1} = 6$, which is higher than zero. Hence, Equation 6.13 imposes that $\mathcal{Y}_{r_1} < \mathcal{Y}_{r_4}$, and thus r_1 preceedes r_4 in the schedule (if both are selected). Again, with respect to Equation 6.14, $\mathcal{Z}_{4,1} - \mathcal{Z}_{1,4} = -6$, which is less than zero and hence $\mathcal{Y}_{r_4} > \mathcal{Y}_{r_1}$ is imposed. Thus, Equation 6.14, ensures that r_4 follows r_1 in the schedule (if both are selected).

The constraint (iii) is satisfied in the same way the constraint (ii) is satisfied. Although in this case, $\mathcal{Z}_{3,5} - \mathcal{Z}_{5,3} = 4$ and $\mathcal{Z}_{5,3} - \mathcal{Z}_{3,4} = -4$, the evaluation of negativity works the same way as does for satisfying the constraint(ii).

Finally, the mutual inclusion in constraint (iv) is enforced by Equation 6.16. According to our current example, $|\mathcal{Z}_{i,j}| = |\mathcal{Z}_{j,i}| = 2$. Hence, Equation 6.16 ensures that the remainder of $(\mathcal{X}_{r_3} + \mathcal{X}_{r_5})$ divided by 2 must be equal to zero, which is possible only if $\mathcal{X}_{r_3} = \mathcal{X}_{r_5} = 1$ or $\mathcal{X}_{r_3} = \mathcal{X}_{r_5} = 0$, that means if both or neither of r_3 and r_5 are selected. Thus, the constraint of mutual inclusion is satisfied.

6.7 Implementation

Based on the CSOP formulation of the scheduling problem, we developed a CP model using *OPL* (Optimization Programming Language)[2]. For OPL programming, we used the `IBM ILOG CPLEX Optimization Studio 12.2` IDE under an academic license. The IDE can be integrated with `CPLEX Solver` and `CP Optimizer`, which are IBM's optimization engines for solving optimization problems modeled in LP and CP respectively.

[2]OPL (Optimization Programming Language) is a relatively new modeling language for combinatorial optimization that simplifies the formulation and solution of optimization problems.

Constraint programming (CP) combines techniques from AI and OR and it has been shown to be effective in solving combinatorial optimization problems, especially in the area of scheduling and planning [24, 306]. Over the past decade, a separate conference series[3] is held to host research to integrate and combine AI and OR techniques in CP. CP allows a more natural and flexible way to express objective functions and constraints where the functions and equations do not necessarily have to be strictly linear. Based on the CSOP formulation of the scheduling problem, we also developed a CP model of the problem and for solving it, invoked the `CP Optimizer` from inside the `IBM ILOG CPLEX Optimization Studio 12.2` IDE.

The CP technique works as follows. Given a set of variables with their domains and a set of constraints on those variables, first the domains of the variables are identified. Then, based on the given constraints, the domains of the concerned variables are modified. When a variables domain is modified, the effects of this modification are propagated through *constraint propagation* to any constraint involving that variable. For each constraint, *domain reduction* detects inconsistencies among the domains of variables pertinent to that constraint and removes inconsistent values. When a particular variable's domain becomes empty, it may be determined that the constraint cannot be satisfied and backtracking may occur undoing an earlier choice. CP repeatedly applies constraint propagation and domain reduction algorithms to make the domain of each variable as small as possible while keeping the entire system consistent. To find the optimal solution, the CP technique may explore, in the worst case, all the feasible solutions and compare them based on the objective function's values.

For the purpose of evaluation, we also implemented Linear Programming (LP), genetic algorithm (GA), and three variants of greedy algorithms for optimizing the automated scheduling of code clone refactorings. Further details of about the LP, GA, and greedy scheduling techniques are presented in Section 6.8.4.

6.8 Empirical Evaluation

To evaluate our refactoring scheduler and the effort model, we conducted an empirical study on refactoring six software systems developed (or under development) in our software research lab[4]. The subject systems and their sizes in terms of source lines of code (SLOC) are described in Table 6.4. All the subject systems shown in Table 6.4 are written in Java and the sizes of the systems in terms of SLOC exclude the comments and blank lines. In particular, we designed the study to address the following two research questions:

RQ1: *Given a set of refactoring activities and a set of constraints for them, can our refactoring scheduler effectively compute conflict-free optimal scheduling of refactorings?* The effectiveness of the technique is measured by quantitative comparison with other techniques such as GA and LP. The conflict-freeness is verified by manually checking for any constraint violation in the schedules computed by the scheduler.

[3]International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR).

[4]Software Research Lab, Department of Computer Science, University of Saskatchewan, Canada, `http://www.cs.usask.ca/research/research_groups/selab`

Table 6.4: Software systems subject to the empirical study

Subject Systems	**SLOC**	**Description**
Mutation Framework	2,901	Ongoing extended implementation of the mutation framework proposed by Roy and Cordy [240]
LIME [309]	3,494	A source code comparison engine
SimCad [279]	3,771	A clone detection tool
gCad [248, 249]	4,563	A clone genealogy extractor
VisCad [10, 9]	9,323	A tool for analysis and visualization of code clones
DomClone [228]	2,239	A domain information-based clone analysis (prediction) tool

RQ2: *Is the code clone refactoring effort model (described in Section 6.3) useful in capturing and estimating the efforts required for performing the refactorings?* We address this exploratory research questions by qualitative analysis of the observation on developers during the study and the developers' feedback through post-study questionnaires.

Typically, it is difficult and risky for developers to refactor a source code with which they are not familiar [183]. However, it is the developers who are likely to know the best about the critical parts of the projects that they develop and, thus, can better assess both the efforts and effects of refactoring and prudently assign priorities on certain refactoring candidates. While the developers' ability to assess the refactoring efforts and effect for large and complex projects can still remain unreliable, for smaller systems, their ability should be fairly reliable. Therefore, to evaluate our refactoring scheduler, we chose small projects (Table 6.4) developed in our own research lab. The use of in-house software systems in the study not only facilitated manual verification for correctness but also reduced the evaluation cost.

At the beginning of the study, we described to the developers the objectives of the study and provided them with our refactoring effort model, as well as an initial list of refactoring operators that can be used for code clone refactoring. Then, we explained a catalog of common software refactoring patterns [80] to them and showed them how some of those can be applied for code clone refactoring. We also described the QMOOD quality attributes to them and, upon discussion, came to a consensus to use the first six metrics (Table 6.2) in our study. We all agreed that the rest of the metrics were too difficult to estimate through subjective investigation, and no automated tool exists to compute them. Hence, we ignored the effect of code clone refactoring on those metrics. To ignore them, the total gain in code/design quality (Section 6.4) was computed having values of changes in those metrics set to zero. All the developers were graduate students pursuing research in the area of software clones and thus possess some expertise in code clone analysis.

Table 6.5: Code clones in the systems under study

Subject Systems	**Number of**		
	Clone-groups	**Clone Fragments**	**Total Refactorings**
Mutation Framework	21	62	72
LIME	20	55	67
SimCad	16	42	64
gCad	28	91	93
VisCad	57	136	166
DomClone	21	56	77

Table 6.6: Demographic information about the developers used for refactoring effort/effect estimation

User	Age (in years)	Software development experience		Known Programming Languages
		in industry (months)	in academia (months)	
1	31	57	96	C, C++, Java, C#, Perl, Python, TXL, VB
2	34	30	156	C, C++, Java, Python, SQL
3	33	96	24	C, C++, Java, C#, Python
4	27	0	96	C, C++, Java, Visual C++, JavaScript
5	30	3	72	C, C++, Java, C#, TXL
6	35	0	144	C, C++, Java, VB, Scheme, Assembly, PHP, JavaScript

6.8.1 Clone Detection

The first and foremost activity towards code clone refactoring is the detection of code clones from the source code. We used NiCad-2.6.3 [238] for detecting near-miss *block* clones of at least five lines in pretty-printed format. We used the 'blind-rename' option of NiCad with UPIT (Unique Percentage of Items Threshold) set to 30%. The 'blind-rename' option instructs NiCad to ignore the differences in the names of identifies/variables during comparison of the code fragments. UPIT is a size-sensitive dissimilarity threshold that sets NiCad's sensitivity to differences in the code fragments during the detection of near-miss code clones. For example, if UPIT is set to 0% without the 'renaming' option, NiCad detects only exact clones (code clones that have identical program text but may have variations in layouts); if the UPIT is 30% having the 'renaming' option set, NiCad detects two code fragments as clones if at least 70% of the normalized pretty-printed text lines are identical (i.e., if they are at most 30% different). In the detection of code clones, NiCad also ignores the comments in the source code and reports code clones clustered into clone-groups based on their similarity.

6.8.2 Data Acquisition

The results of clone detection from the six subject systems were provided to the concerned developers, who then further analyzed the detected clones and re-arranged the groups when necessary, based on the suitability for refactoring within context according to their understanding. The demographic information about those developers are presented in Table 6.6. For the analysis, the developers used VisCad [9, 10], a code clone analysis and visualization tool developed in our research lab. For each of the systems, the number of clone-groups and the number of distinct clone blocks involved in those groups are presented in Table 6.5, which the developers identified as the potential candidates for refactoring.

Having the code clones organized into groups, the developers carried out further qualitative analysis to determine the strategies for refactoring each clone-group (refactoring candidate). The identification of a refactoring strategy, in particular, involved finding the appropriate refactoring operations, their order of application, and mutual dependencies (if any). For each of the clone-groups chosen for refactoring, the developers wrote down the sequence of operations that they would perform to refactor that clone-group. In determining the operations, the developers were free to choose any operations beyond the list of refactoring operators they were initially provided. The right-most column of Table 6.5 presents the total number of refactoring activities identified for each of the subject systems. The developers also noted down any restrictions in the ordering of the operations that must be followed to successfully refactor a clone-group. Any such ordering restrictions between clone-groups were recorded as well.

As an example, in Figure 6.1, we present a clone-pair (shaded blocks on the left) with partial context (surrounding code). The example is an excerpt from the source code of VisCad[5]. The developers chose

[5]The code is originally a part of `diff-match-patch`, an open-source library (available at http://code.google.com/p/google-diff-match-patch/) that VisCad uses internally. We deliberately chose to present this simple example, so that anyone can easily follow and verify.

Table 6.7: Example of operations and efforts for extract method

Operations for extract method	**Efforts**
Produce signature of the target method	15
Copy clone fragment to the body of target method	1
Perform necessary modifications in the body	5
Replace clone fragments by calls to the extracted method	2
Total efforts	23

to refactor them by applying the *extract method* operation. The developers recorded further fine grained operations and required efforts in order, as shown in Table 6.7. As explained by the developers, the effort for producing the method signature was estimated by twice (for type and name) the number of parameters to the method, plus three for method name, return type, and access modifier. Code modification effort was estimated by the number of words (tokens) added, deleted, or modified.

As far as we are concerned, there is no existing tool for calculating refactoring efforts and ours is the first conceptual model for this purpose, we relied on the developers' opinions and wanted to see to what extent our effort model was useful in estimating the effort of code clone refactoring. The developers were instructed to estimate efforts required for each refactoring activity that they identified or for each of the clone-groups as a whole they chose to refactor. Alhough they were provided the refactoring effort model, they were free to apply their own understanding and analytical evaluations for the efforts estimation. As the developers estimated refactoring efforts, at times, we observed and communicated with them to understand how they were estimating the efforts for refactoring. We used our observations and the developers' feedback for a subjective evaluation of our effort model.

In the estimation of quality gains expected from the refactorings of code clones, we again relied on the developers' judgements, which we feel is important in this context. Using the QMOOD design property metrics (Table 6.2), it was relatively easy for the developers to estimate the quality gain expected from the refactoring of a clone-group. For example, to estimate the change in *design size* or *complexity*, the developers did not compute the total number of classes or methods (before and after the refactoring) in the system, they just estimated the changes in the number of classes or methods. For example, the refactoring scenario presented in Figure 6.1 causes the complexity (number of methods) to increase by one and all other QMOOD design property metrics under consideration remain unaffected.

Next, the developers were instructed to assign non-zero priorities between -5 (the lowest priority) and $+5$ (the highest priority) to certain clone-groups that they considered important in terms of the necessity and risks involved in refactoring them. The priorities were set to $+1$ for clone-groups that were left unassigned by the developers. For each of the systems, the developers identified some intentional code clones in particular parts of the systems. They considered some of them to be critical and preferred not to take the risk of

refactoring them. Taking the developers' opinions into consideration, we could have excluded those from our study. Instead, we assigned the lowest priority to them for examining how our scheduler handles them in the scheduling process.

The developers' estimation of refactoring efforts, effects, and the assignment of priorities are then used to compute refactoring schedules in the evaluation of our code clone refactoring scheduler.

6.8.3 Data Normalization

As described before, both the estimation of expected change in code/design quality and the refactoring efforts, as well as the priorities were sometimes set on refactoring of clone-groups as a whole. Thus, in situations where the developers made those estimations for refactoring an entire clone-group, we equally distributed those estimations to all the refactoring operations involved in refactoring that particular clone-group.

Recall that the scheduling of code clone refactoring activities can be optimized towards three dimensions: minimizing the refactoring efforts, maximizing the refactoring benefits, and maximizing the satisfaction of priorities. However, the ranges of values obtained along those dimensions were different. For example, the priorities ranged between $+5$ and -5, whereas the values of total refactoring efforts varied between 4.0 and 47. To prevent our scheduler getting biased towards any of the individual dimensions, we first normalized the values obtained for all the three dimensions using the following procedure.

Let $S = \{v_1, v_2, v_3, \ldots, v_n\}$ be a set of values, then:

$$norm(v_i) = \frac{v_i}{max\{|v_1|, |v_2|, |v_3|, \ldots, |v_n|\}}, \quad \forall v_i \in S$$

where, $norm(v_i)$ denotes the normalized value of v_i. The set S can be the set of values for all the refactorings along any of the three dimensions. The normalization actually brings the magnitudes of all those values between zero and one (i.e., $0 < |norm(v_i)| \leq 1$) and thus minimizes the inter-dimension influence of the magnitudes, while still preserving the relative ratios of magnitudes within dimensions. The emphasis on the efforts compared to the qualities can be tweaked by setting higher or lower weights as described in Equation 6.8.

For priorities, before applying the aforementioned normalization, we carried out an additional normalization phase by adding $+6$ to each priority values. Thus, we removed any priority less than or equal to zero. This removal was necessary as in our objective function, the difference between quality and effort is multiplied by priorities and we must ensure that the multiplication negatives or multiplication by zero priority do not take place.

6.8.4 Schedule Generation

For each of the systems subject to our study, we enumerated all the refactorings, accumulated them with the normalized data, and organized them in an appropriate OPL format to feed to the schedulers for automated computation of the refactoring schedules.

We evaluated our CP scheduling approach in four phases. In the first phase, we compared our CP scheduler with three variants of a greedy algorithm. The second phase compared our CP approach with genetic algorithm (GA) that was used by other researchers [37, 176]. In the third phase, we compared the CP scheduling with a manual approach. Finally, the fourth phase compared the CP technique with the LP approach. In our study, we used the default settings in the estimation of total effort and quality gain, as described in Sections 6.3 and 6.4. For each of the subject systems, we first computed the refactoring schedule using our CP approach and then applied GA, greedy, manual, and LP approaches (described later) to compute schedules for the same set of refactorings.

The normalized data for each of the subject systems were separately fed to each of the schedulers. All the schedulers were executed on an Apple "MacBookPro5,5" computer with Intel Core 2 Duo (2.26 GHz) processor and 4 GB primary memory (RAM). The CP, LP, and greedy schedulers operated inside the `IBM ILOG CPLEX Optimization Studio 12.2` IDE running on Windows XP operating system. All the IDE parameters were set to the defaults. The GA scheduler operated on the same computer but on a Mac OS 10.6.8 operating environment.

CP Scheduling

For each of the subject systems, the normalized data for each of the subject system were fed to our scheduler as described in Section 6.6 and Section 6.7. The scheduler, upon obtaining the data in valid OPL format, applies constraint propagation and domain reduction techniques [306] to generate the optimal solution as instructed.

LP Scheduling

Linear programming (LP) is a mathematical programming technique for solving optimization problems. Over the past few decades, LP has been widely used in the operations research (OR) community for dealing with optimization problems. The basic idea is to formulate the problem as a *linear programming problem* and solve it using linear programming algorithms, such as the simplex method, ellipsoid method, and interior-point techniques [293]. A linear programming problem is a mathematical formulation of an optimization problem defined in terms of an objective function and a set of constraints. The objective function is a linear function of variables whose values are unknown and the set of constraints consists of linear equalities and linear inequalities. The requirement of the linearity of the objective function and the constraint equations as well as the solution technique are the most obvious traits that make LP distinct from CP. On the basis of the CSOP formulation described in Section 6.6, we implemented an LP model of the scheduling problem and invoked the `CPLEX Solver` for solving the LP model.

The CP implementation differs from the LP implementation in two ways: first, in the LP implementation, all the constraints were expressed in terms of strictly linear equations, whereas in CP, we used CP-specific OPL statements, all of which were not necessarily expressed in terms of linear equations. Second, the CP

and LP implementations included different instructions to explicitly specify whether to invoke the CP Solver or the LP Solver of the `IBM ILOG CPEX Optimization Studio 12.2`.

To compute scheduling of refactorings for each of the subject systems, we invoked our LP scheduler, which applied the *mixed integer linear programming (MILP)* technique for computing the schedules. MILP is a kind of LP, where the variables can hold integer or floating point values only. Our LP scheduler, in consultation with the `CPLEX Solver`, invokes the branch-and-cut algorithm, which in turn applies the simplex algorithm to solve a series of relaxed LP subproblems and gradually converge to a strictly optimal solution. The simplex algorithm operates by repeatedly applying linear algebraic techniques to solve systems of linear equations. Further detail about the branch-and-cut and simplex algorithms can be found elsewhere [293, 306].

GA Scheduling

Genetic algorithm (GA) is a kind of evolutionary algorithm from the field of artificial intelligence (AI) for solving optimization problems. In GA, a candidate solution is encoded as a sequence of values, called a *chromosome*; a set of candidate solutions is called a *population*. The algorithm iterates over generations to evolve a population towards better solutions through a number of operations such as *crossover* and *mutation*. A *fitness function* is used to guide the evolution towards optimality. In our study, the objective of the GA was set to select the best subset having maximum $\mathcal{M}$ members from the set $\mathcal{R}$ of all potential refactorings (recall from Section 6.6) for each of the subject systems.

Encoding: For each system subject to our study, all the candidate refactorings were enumerated with integers 1 through $|\mathcal{R}|$. Having such an enumeration, a common approach is to represent the problem as a binary Knapsack [37] problem and encode a solution as a binary string (Figure 6.5) of length $|\mathcal{R}|$. A bit in the string is 1, if and only if the corresponding refactoring is selected in the schedule. However, such an encoding scheme cannot deal with order dependencies among the refactorings.

Figure 6.5: Traditional encoding of a solution in a binary string

To capture the order dependencies, we devised a different encoding scheme. A candidate solution was encoded in a chromosome ζ as a sequence of $\mathcal{M}$ integers, having their positions indexed with 1 through $\mathcal{M}$, as shown in Figure 6.6.

Having $\zeta[i]$ denoting the chosen refactoring at index i, the encoding scheme is as follows.

- $\zeta[i] = 0$ implies that no refactoring is selected at the index i.
- $\zeta[i] = r_k$ implies that r_k is the i^{th} refactoring in the schedule represented by ζ.

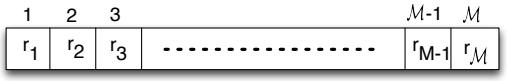


Figure 6.6: Our encoding of a solution in a chromosome

- $\zeta[i] < \zeta[j]$ and $\zeta[i] \neq 0 \neq \zeta[j]$ means that the refactoring at index i is scheduled before the refactoring at index j.
- A solution encoded by the chromosome ζ is *feasible*, if any two chosen refactorings $\zeta[i]$ and $\zeta[j]$ satisfy all the hard constraints. Otherwise, the solution is *infeasible*.
- A solution encoded by the chromosome ζ must not select the same refactoring more than once. That is, $\zeta[i] \neq \zeta[j]$ must hold if $\zeta[i] \neq 0$.

Crossover Operation: The crossover operation, as shown in the Figure 6.7, randomly selects two chromosomes ζ_{p_1} and ζ_{p_2} as parents, an index k as the point of crossover, and creates two offsprings ζ_{c_1} and ζ_{c_2} as follows,

$$\zeta_{c_1}[i] = \zeta_{p_1}[i], \text{ for } i \in \{1, 2, 3, \ldots, k-1\}$$
$$\zeta_{c_1}[j] = \zeta_{p_2}[j], \text{ for } j \in \{k, k+1, k+2, \ldots, \mathcal{M}\}$$
$$\zeta_{c_2}[i] = \zeta_{p_2}[i], \text{ for } i \in \{1, 2, 3, \ldots, k-1\}$$
$$\zeta_{c_2}[j] = \zeta_{p_1}[j], \text{ for } j \in \{k, k+1, k+2, \ldots, \mathcal{M}\}$$

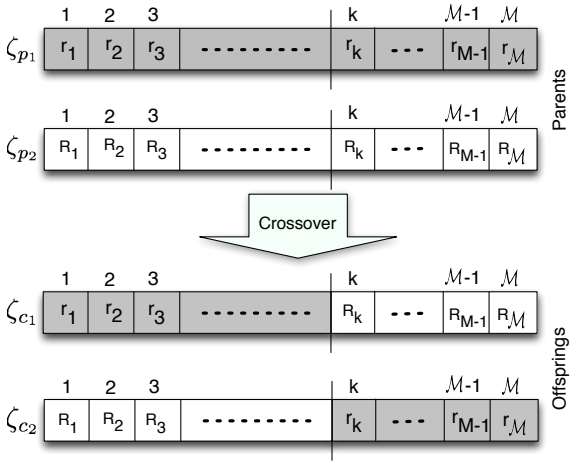


Figure 6.7: Crossover operation

A configurable parameter *crossover rate* defines what proportion of the population of chromosomes in a certain generation will participate in crossover operation to produce offsprings. A crossover rate of 80%

indicates that 80% of the population participate in crossover operation leaving 20% survivors (unchanged chromosomes).

Mutation Operation: The mutation operation on a chromosome ζ selects a random index k and replaces the refactoring $\zeta[k]$ by another randomly-selected refactoring $r \in \mathcal{R}$ that is not already in the chromosome. Mathematically, $\zeta[i] \neq r, \forall i \in \{1, 2, 3, \ldots, \mathcal{M}\}$ holds before the mutation.

Fitness Function: The fitness function is defined by the objective function described by the Equation 6.10 in Section 6.6. The fitness function determines how good (fit) a solution is, as represented by a chromosome.

Population Generation: The genetic algorithm begins with an initial population and evolves from generation to generation. A population of P distinct solutions is randomly created using the procedure described in Algorithm 2.

Algorithm 2 : Algorithm for Creating Population

Require: ▷ $\mathcal{M}$ is the length of a chromosome
  $R \leftarrow \{1, 2, 3, \ldots, |\mathcal{R}|\}$ ▷ enumerations over all refactorings $\mathcal{M} \leq |\mathcal{R}|$
  procedure CREATEPOPULATION(P)
    $S_p \leftarrow \{\}$ ▷ a set to accommodate all solutions
    while $|S_p| < P$ **do**
      $x \leftarrow$ RandomSolution( )
      $S_p \leftarrow S_p \cup \{x\}$
    end while
    return S_p
  end procedure
  procedure RANDOMSOLUTION
    $\zeta[\,] \leftarrow \{0, 0, \ldots, 0\}$ ▷ array of $\mathcal{M}$ zeros
    $S_r \leftarrow R \cup \{0\}$ ▷ $\mathcal{R}$ is the set of all refactorings
    for $k \leftarrow 0$ to $\mathcal{M} - 1$ **do**
      $r \leftarrow$ randomly chosen, $r \in S_r$
      $\zeta[k] \leftarrow r$
      if $r \neq 0$ **then**
        $S_r \leftarrow S_r - \{r\}$
      end if
    end for
    return ζ
  end procedure

Table 6.8: Parameters for genetic algorithm

Parameter		Value
Population size	≈	1.0 × choromosome-length
Mutation rate	≈	1.0%
Crossover rate	≈	80%
Elitism rate	≈	30%

Genetic Evolution: The algorithm evolves according to the description in Algorithm 3. The genetic algorithm and its evolution is characterized by a number of parameters. We executed the genetic algorithm several times with different combinations of parameters. After tuning with different combinations, we chose the combination that yielded the best performance (i.e., highest fitness) of the genetic algorithm. The chosen values for the parameters, as presented in Table 6.8, are consistent with general recommendations [219]. For each of the subject systems, we executed the GA scheduler five times. At each run, the scheduler executed for an evolution time of 10,000 seconds (i.e., 2.78 hour) and we kept the best (fittest) solution produced in the five runs. This run-time is much higher than those required for other scheduling approaches (e.g., CP, LP).

Greedy Scheduling

Recall that we identified three dimensions (i.e., effort, quality, and priority) for optimizing scheduling of code clone refactorings. Thus, we implemented (using OPL) three variants of a greedy algorithm, each aiming to optimize along one of the dimensions (i.e., optimization criteria) disregarding the other two. The prime objective of the $Greedy^e$ approach is to compute schedules by minimizing refactoring effort while the $Greedy^p$ and $Greedy^q$ approaches aim to maximize the satisfaction of priorities and quality gain, respectively. The general greedy scheduling algorithm can be described in terms of a few simple steps. First, all the refactorings are sorted in the descending order of the optimization criteria. Then, refactorings are chosen one by one from the top of the sorted list as long as the new candidate does not conflict with any of the already chosen refactorings.

Intuitively, the minimum refactoring effort (i.e., zero effort) can be achieved by scheduling no refactoring at all. Therefore, in the application of the approach greedy towards refactoring efforts, we must set a minimum number of refactorings that must be scheduled. To keep the approach greedy towards refactoring efforts comparable with our CP technique, the minimum number of refactorings was set equal to the number of refactorings scheduled by our CP scheduler. The values along all the three dimensions obtained from these scheduling approaches are presented in Table 6.9.

Algorithm 3 : Genetic Algorithm

$g \leftarrow 0$ ▷ generation set to 0

$\zeta_{best} \leftarrow \emptyset$ ▷ null array

$S_p \leftarrow \text{CreatePopulation}(max_{population})$

repeat

  $g \leftarrow g + 1$

  $S_p \leftarrow \text{FilterInfeasibles}(S_p)$ ▷ drop infeasible solutions

  $S_p \leftarrow \text{Sort}(S_p)$ ▷ sort in descending order of fitness

  $\zeta_{best} \leftarrow \text{First}(S_p)$ ▷ record the best solution

  $S_p \leftarrow \text{DropPoors}(S_p)$ ▷ drop poor solutions w.r.t. elitism rate

  $d \leftarrow max_{population} - |S_p|$

  if d is odd **then**

    $d \leftarrow d + 1$

  end if

  $S_n \leftarrow \{\}$

  for $i \leftarrow 1$ to $\frac{d}{2}$ **do**

    $\{\zeta_{p_1}, \zeta_{p_2}\} \leftarrow$ random chromosomes from S_p

    $\{\zeta_{c_1}, \zeta_{c_2}\} \leftarrow \text{CrossOver}(\zeta_{p_1}, \zeta_{p_2})$

    $S_n \leftarrow S_n \cup \{\zeta_{c_1}, \zeta_{c_2}\}$

  end for

  for $\zeta \in S_n$ **do**

    $m \leftarrow \text{MutateChance}(\zeta)$ ▷ chance of ζ to be mutated

    if $m \geq mutationRate$ **then**

      $\zeta \leftarrow \text{Mutation}(\zeta)$ ▷ mutate ζ

    end if

  end for

  $S_p \leftarrow S_p \cup S_n$

  $t \leftarrow \text{elapsedTime}()$ ▷ time duration of evolution

until $(g \geq max_{generation})$ or $(t \geq max_{time})$

return ζ_{best}

Table 6.9: Comparison of automated scheduling approaches

Subject systems	Scheduling approaches	Values at dimensions			Quality −Effort	P × (Q − E)	Refac. chosen
		Prior.	Effort	Quality			
Mutation Framework	Greedy^{p}	20.06	21.94	18.53	-3.41	-68.40	40
	Greedy^{e}	9.63	6.06	10.04	3.98	38.33	20
	Greedy^{q}	18.16	21.82	19.64	-2.18	-39.59	42
	GA*	21.27	19.99	18.46	-1.53	-32.54	36
	LP	9.34	7.86	11.48	3.62	33.81	20
	CP	9.34	7.86	11.48	3.62	33.81	20
LIME	Greedy^{p}	22.42	21.12	19.93	-1.19	-26.68	47
	Greedy^{e}	13.00	8.28	13.61	5.33	69.29	33
	Greedy^{q}	16.29	23.49	26.07	2.58	42.03	51
	GA	10.17	15.71	15.21	-0.50	-5.09	33
	LP	11.04	12.32	16.12	3.80	41.95	33
	CP	11.04	12.32	16.12	3.80	41.95	33
SimCad	Greedy^{p}	27.42	25.23	16.82	-8.41	-230.60	52
	Greedy^{e}	13.23	7.12	13.7	6.58	87.05	25
	Greedy^{q}	23.57	24.64	30.18	5.54	130.58	51
	GA*	19.33	17.18	20.95	3.77	72.87	32
	LP	12.78	8.99	18.96	9.97	127.42	25
	CP	12.78	8.99	18.96	9.97	127.42	25
gCad	Greedy^{p}	19.65	21.62	20.00	-1.62	-31.83	41
	Greedy^{e}	9.61	9.53	11.57	2.04	19.60	28
	Greedy^{q}	12.05	23.48	25.98	2.50	30.13	44
	GA*	25.18	26.12	20.86	-5.26	-132.45	45
	LP	6.70	15.19	17.99	2.80	18.73	28
	CP	6.70	15.19	17.99	2.80	18.73	28
VisCad	Greedy^{p}	36.14	32.57	25.71	-6.86	-247.92	66
	Greedy^{e}	16.12	18.63	13.20	-5.43	-87.53	40
	Greedy^{q}	29.02	33.81	34.32	0.51	14.80	72
	GA*	45.03	42.57	38.09	-4.48	-201.73	83
	LP	15.02	16.20	22.32	6.12	91.92	41
	CP	15.33	15.78	21.90	6.12	93.82	40
DomClone	Greedy^{p}	37.64	28.06	23.77	-4.29	-161.48	62
	Greedy^{e}	18.79	7.54	10.98	3.44	64.64	33
	Greedy^{q}	33.12	25.62	28.93	3.31	109.63	56
	GA*	26.64	24.02	23.23	-0.79	-21.05	36
	LP	19.14	13.33	23.35	10.02	191.78	35
	CP	19.49	12.57	22.41	9.84	191.78	33

Here, Greedy^{p} = approach greedy towards priority satisfaction,
Greedy^{e} = approach greedy towards effort minimization,
Greedy^{q} = approach greedy towards quality gain,
*the computed schedule was infeasible
$P \times (Q - E)$ = Priotity × (Quality − Effort)

Table 6.10: Time and memory comparison of CP and LP scheduling

Subject systems	Scheduling approaches	Resource Consumpsion	
		Time (sec.)	Memory
Mutation Framework	LP	14.33	7.39 MB
	CP	0.14	5.9 Mb
LIME	LP	185.69	6.45 MB
	CP	0.20	5.2 Mb
SimCad	LP	54.09	12.40 MB
	CP	0.97	9.4 Mb
gCad	LP	58.08	12.40 MB
	CP	0.94	9.4 Mb
VisCad	LP	313.78	37.81 MB
	CP	6.02	27.6 Mb
DomClone	LP	396.02	8.29 MB
	CP	0.80	6.7 Mb

Here, MB = Mega**byte**, Mb = Mega**bit**

Manual Scheduling

In the third phase of the evaluation, our goal was set to schedule roughly 25% of the total number of refactorings for each of the subject systems. The developers of the concerned systems were instructed to manually (or, in the way they would do it without help from any automated scheduler) produce a schedule as best as they could. Manually solving a CSOP such as scheduling of code clone refactoring is a time-consuming and difficult task, especially for medium to large problem instances. Therefore, we chose to schedule 25% of the total number of refactorings to keep the problem instance small enough to be handled by the manual approach. With the same goal (i.e., to schedule roughly 25% of all the refactorings), we executed our CP scheduler.

The purpose of the manual approach was to confirm that manually solving a CSOP can be difficult and the solution obtained from manual scheduling can be worse than an automated technique, such as CP. The objective was to compare the produced schedules, given a set of constraints, estimation of refactoring efforts, effects, and priorities. It was not necessary to actually carry out the those refactorings in the subject systems.

6.8.5 Findings

The values along the three optimization dimensions namely the satisfaction of *priorities* ($\sum_{r\in\mathcal{R}} \mathcal{X}_r\rho_r$), required *effort* ($\sum_{r\in\mathcal{R}} \mathcal{X}_r E(g_r)$), and expected gain in software *quality* ($\sum_{r\in\mathcal{R}} \mathcal{X}_r\overline{Q}_r$), obtained from our CP

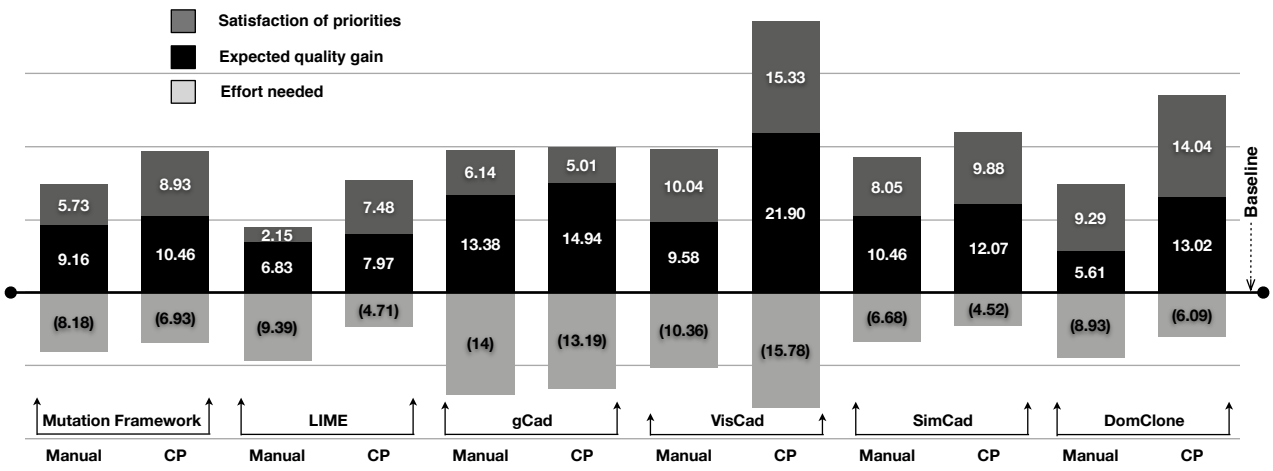


Figure 6.8: Automated CP vs. manual scheduling

scheduling and manual scheduling, are presented in Figure 6.8. For an effective schedule, the values for expected quality gain and satisfaction of priorities are expected to be high while the values for the efforts needed are expected to be low. Hence, in the figure the heights of the bars above the baseline (round-ended line) are expected to be high while the heights of the bars below the baseline are expected to be as low as possible.

Table 6.9 presents values along all the three optimization dimensions obtained by separately running all the automated schedulers for each of the subject systems in our study. Recall that the objective function as stated in Equation 6.10 optimizes along these three dimensions. From our observations during the study and the developers' feedback, as well as the results presented in Table 6.9 and Figure 6.8, we can answer the two research questions formulated before.

> **Answer to RQ1:** Yes, given a set of refactoring activities and constraints among them, our refactoring scheduler can effectively compute a conflict-free optimal schedule of refactorings.

As seen from Figure 6.8, for four of the subject systems (i.e., Mutation Framework, LIME, SimCad, and DomClone), the refactoring schedules generated by our CP approach are consistently lower while the expected quality gain and satisfaction of priorities are consistently higher compared to those for the manually computed schedules. For gCad, both the CP and manual approaches performed almost equally well. In the case of VisCad, the schedule computed by CP demands higher refactoring effort compared to the manually computed schedule; however, the expected quality gain and satisfaction of priorities for the CP schedule are much higher than those for the schedule computed by manual approach. Overall, it can be said that the CP approach outperforms the manual approach.

For all the subject systems, as seen in Table 6.9, our CP scheduler and the LP scheduler compute the optimal refactoring schedule by efficiently balancing the three optimization dimensions (i.e., effort, quality,

and priority). Again, the efforts are expected to be low while the quality gain and priorities are expected to be as high as possible. For some of the smaller systems (Mutation Framework, LIME, and gCad), the greedy approaches, especially the approach greedy towards refactoring efforts, closely competes with our CP approach. For Mutation Framework and LIME, the approach greedy towards efforts can be perceived (according to the third column from the right) to have performed even slightly better than our CP scheduler.

Again, for all the systems, the priorities and quality gain for the schedules computed by the approach greedy towards priorities are consistently higher than those for the schedules computed by CP approach. According to the second column from the right, the approach greedy towards priorities closely competes with our CP scheduler for smaller systems such as LIME and SimCad, whereas for gCad, the approach greedy towards priorities is found to have performed better than CP. However, the required efforts for those schedules (computed by the approach greedy towards priorities) are also consistently much higher (two or three times for most of the systems) than those for the schedules computed by the CP approach. These much lower efforts can make the CP scheduler preferable.

Other than those few cases discussed above, for all the systems the CP and LP schedulers are found to have significantly outperformed the other techniques. We also found that the schedules generated by the CP approach exhibited higher values of the objective-function (i.e., Equation 6.10) compared to those computed by the greedy approaches. As the sizes of the systems in terms of SLOC and the number of candidate refactorings increases, the CP and LP schedulers outperform the greedy schedulers, which is visible for the largest systems, VisCad and DomClone. Overall, the CP and LP schedulers perform better than the greedy schedulers or at least as good as those schedulers.

The risks of refactorings can be best estimated through subjective analysis by the individuals who are familiar with the underlying source code. Quantitative measurement of such risks would be very difficult, if not impossible. However, the risks of refactorings can be expected to be positively proportional to the number of refactorings. In this sense, the CP and LP schedulers also minimize the risks of refactorings, as seen in the right-most column of Table 6.9, the optimal schedule obtained from our scheduler always includes the least number of refactorings, compared to those from the GA and greedy schedulings.

As expected, our CP scheduler always outperformed manual scheduling for all the six subject systems (Figure 6.8). The superiority in the optimality of the schedules (in terms of efforts, quality gain, and priorities) obtained from our CP and LP schedulers, compared to manual scheduling, gradually increased as the sizes of the systems and the number of candidate refactorings increased. Our CP scheduler took no more than seven seconds in computing any of the refactoring schedules presented in this chapter, whereas, for manual scheduling, the developers had to spend several hours depending on the number of refactoring candidates and the constraints involved. Recall that at each run the GA scheduler executed for more than two hours. Thus, in terms of run-time, the CP approach outperforms the manual, greedy, and GA approaches.

Special Note on GA

The performance of GA scheduling is found to be worse than all other automated scheduling techniques in our study. Within the 2.75 hours evolution time, GA was able to produce feasible solution for only LIME. For all other systems, the results of GA scheduling presented in the Table 6.9 correspond to infeasible solutions. Given the refactoring scheduling problem instances for those subject systems, during our study, we found that the GA technique executed for hours and found solutions, which were not *feasible* solution. An explanation to this observation can be the fact that optimization problems with many constrains can easily become *GA-hard* [59, 72], because *crossover* and *mutation*, the core operations of GA are based on random selections, which do not guarantee for constraint satisfaction or optimization. Thus, traditional GA may work for optimization problems with a *few* constraints, but GA approaches do not seem to work well for CSOPs [72]. Above all, GA approaches are by nature time consuming and memory intensive.

CP vs. LP

As can be seen in Table 6.9, the results for both CP and LP are identical for all the subject systems, except for VisCad and DomClone. This observation means that, for each of those systems, both CP and LP produced equally optimal solutions and, thus, in terms of the quality (e.g., optimality) of the solutions both CP and LP performed equally well.

For VisCad and DomClone, although the optimal schedules computed by CP and LP are different, the values of the objective functions were found to be equal: 8.364 for VisCad and 6.03 for DomClone. There were more than one equally optimal solutions, which allowed the CP and LP schedulers to choose different solutions with the same objective values. However, the CP scheduler picked the solutions with the number of chosen refactoring less than that of the LP scheduler. Thus, our CP scheduler mitigates the risk of refactoring better than the LP scheduler. However, the subtle difference may not be statistically significant and a larger study can verify this phenomenon with statistical confidence.

In Table 6.10, we present the time and memory consumption of both the CP and LP schedulers in the computation of optimal scheduling of clone refactorings for each of the subject systems. As can be observed in the table, the time and memory consumption of our CP scheduler is significantly less than that of the LP scheduler. Therefore, we can conclude that the CP approach outperforms the LP approach in terms of both run-time and memory consumption.

Addressing RQ2 for Evaluation of Effort Model

During the study, we observed the developers as they were manually estimating the efforts required to refactor the code clones at hand and assigning priorities to the candidate clones. We encouraged them to *think aloud* so that we could capture information about what and how they were thinking as well as what kind of difficulties they were facing. As they completed their part, we collected feedback from them using

Table 6.11: Developers feedback on the Likert scale questions

Ques.	Choice of answers and number of developers' responses in brackets				
IQ_1	very easy (0)	easy (0)	somewhat difficult (0)	difficult (2)	very difficult (4)
IQ_2	provided no assistance (0)	somewhat useful (0)	quite useful (1)	very useful (3)	it was a necessity (2)

questionnaire with a fixed set of both close and open-ended questions. Other than the questions about the developers' background, the questionnaire included the following questions:

IQ_1**:** How difficult was it to use the effort model in manually estimating the required efforts for refactoring the clones?

IQ_2**:** To what extent the effort model appeared useful to you in the estimation of the refactoring efforts?

IQ_3**:** Is there anything that you think is missing and should be included in the effort model?

IQ_4**:** Is there anything that you suggest to exclude from the effort model?

IQ_5**:** Any other comments about the effort model?

The questions IQ_1 and IQ_2 were Likert scale questions. The possible answers to these questions and the developers' feedbacks are presented in Table 6.11. The questions IQ_3, IQ_4, and IQ_5 were open-ended questions. None of the developers responded to IQ_3 and IQ_4, which hints the completeness of our effort model, at least from those developers' perspective. Only one of them responded to IQ_5 with a concise comment saying, "Tool [support] needed [for calculation of such effort estimation]."

Upon collection of the developers' feedback through the questionnaire, we further conducted a *focus group* discussion session with all the developers to obtain their opinions about usefulness and potential improvements of our effort model. During the *focus group* session, all the developers indicated that the model was useful and it guided them in the estimation of the efforts. One of the developers further expressed that he would not have any clues about how to estimate efforts without the help of the effort model. All the developers proposed that an automated tool, offering accurate calculations according to the model, would be necessary to use the effort model more accurately. Our observations of the developers (while they were estimating the refactoring efforts) also support this proposition. Some of the developers argued that the effort model was useful for quantitative estimation of refactoring efforts but it alone could not capture the risks involved in code clone refactorings. However, everyone agreed that the effort model and the priority scheme in combination were effective in capturing both the efforts and the risks.

Based on the subjective evaluation of the participant developers and their feedback, we can now answer the RQ2 as follows:

> **Answer to RQ2:** Yes, the code clone refactoring effort model (described in Section 6.3) is useful in capturing and estimating the efforts required for performing the refactorings.

6.8.6 Threats to Validity

In this section, we point to the possible threats to the validity of our work and how we addressed those threats to minimize their effects. Recall that the objective of our empirical study was twofolds: first, to evaluate our refactoring effort model and, second, to evaluate our CP scheduler. Hence, we organize the discussion of the threats along these two perspectives.

Construct Validity

Construct validity questions the correctness of the design of the study in terms of whether the data collection and operational measures are used correctly to reflect the concepts studied. Ensuring construct validity is typically challenging for studies involving human developers [235].

In our study, we relied on the developers' qualitative evaluations in the estimation of both refactoring efforts and effects. There is a possibility to question the individual developer's ability to correctly estimate those following our effort model. This work is based on our initial proof-of-concept proposal [307], where the reviewers suggested to evaluate our refactoring effort model from the developers' perspective. We also understand that manual scheduling of refactorings may be too difficult for large systems but, for smaller systems, the developers can be expected to do a fair job in estimating the efforts and risks involved in refactoring the system. Hence, we intentionally chose in-house systems and their respective developers for our study, which may appear as a bias.

In practical settings, it is often likely that refactorings, especially during the development phase, will be performed by the concerned developers who are familiar with the source code. Thus, our choice of in-house systems and their developers rather imitates the practical settings. Moreover, the subject systems, being in-house and fairly small, allowed the developers to estimate the refactoring efforts and effects with a higher probability of accuracy compared to that if we had used large open-source subject systems. Nevertheless, we manually verified each developer's refactoring solutions by performing a detailed inspection of the code clones and surrounding source code in the subject systems. We did not rely only on observing the developers while they were estimating the refactoring efforts. Through a questionnaire, we also collected the developers' feedback about the effort model and further verified their feedback in a *focus group* session. As such, we have a high confidence in the validity of the evaluation.

Our refactoring scheduler (the primary contribution of this work) is independent of how the refactoring data are obtained. Given a set of refactorings along with their mutual constraints and priorities, as well as the estimation of refactoring efforts and changes in code/design quality, how effectively our scheduler can compute the optimal schedule is the question for evaluating the scheduler. Due to the unavailability of any baseline approach or benchmark data, it was not possible to evaluate our scheduler in terms of precision (specificity) and recall (sensitivity). Therefore, we chose to compare our CP scheduler with other approaches (i.e., greedy, GA, LP, and manual) in terms of optimization values along the three dimensions (i.e., effort, quality, and

priority satisfaction) as well as in terms of runtime and memory consumption. We manually investigated all the refactoring schedules obtained from our CP scheduler and confirmed correctness (i.e., feasiblity) in terms of constraint satisfaction. The choice of in-house systems and their developers also facilitated manual investigation of constraints satisfaction and optimality of the refactoring schedules computed by our scheduler.

Manual verification of optimality was difficult as it was difficult to manually produce an optimal schedule, because the code clone refactoring scheduling problem is NP-hard [37, 183, 176]. However, we made efforts to challenge the optimality of the produced solutions by attempting to further increase the objective values by means of pseudo-random replacement of refactorings in the computed schedule. As we were not successful in that endeavour, we became convinced that our CP scheduler indeed produced optimal solution.

Given that the problem is well-defined, the mathematical foundation behind the LP technique guarantees to identify the optimum solution and so a head-to-head comparison between the schedules produced by the CP and LP techniques enables an automated approach for mathematical verification of the optimality of the schedules computed by the CP scheduler. As the schedules obtained from the CP scheduler were identical (or having same objective values) to those computed by the LP scheduler, we can confidently conclude that our CP scheduler indeed produced the optimum refactoring schedules for each of the subject systems in our study.

Internal Validity

Internal validity is mostly concerned with the "possible errors in our algorithm implementations and measurement tools that could affect outcomes" [221].

Our refactoring effort model requires some fine grained computations (e.g., token modification efforts in terms of edit distances), which were not possible for the developers to perform by hand. For this reason, during estimation of the refactoring efforts, the developers used our effort model as a guideline and followed that as much as it was feasible. However, this use does not affect the functionality of our refactoring scheduler. Rather, our observations and the developers' expressions towards the need for realization of the effort model in a software tool further indicate the necessity and effectiveness of our effort model.

In the estimation of the impact of refactoring on code/design quality, we used the six of the QMOOD design property metrics and ignored the rest. Moreover, the impact of clone refactoring was estimated in accordance with Equation 6.9. The weighted sum of differences in Equation 6.9 might have not been able to capture the full benefits of the quality model. These are also threats to the study.

While the choice of the design property metrics does affect the estimation of refactoring *effects* it does not affect the estimation of refactoring *efforts* based on our effort model. Indeed, the inclusion of all the metrics may affect the scheduling approaches and produce different schedules, but we see no reason why this may degrade the performance of our CP scheduler compared to the others. Nevertheless, carrying out a follow-up study including all the QMOOD design property metrics can be worthwhile, which we plan to do in the future.

In our study, we found that the GA approach did not perform well and produced infeasible solutions. Our choice and implementation of the mutation and crossover operators may seem to be responsible. To minimize this threat, we tweaked those operators in several ways and tuned the parameters to the GA algorithm in separate runs. Then, we chose the best combination of parameters to use in our study. We believe that the reason to the poor performance of the GA approach was that the large set of constraints actually made the problem GA-hard [59, 72], as discussed in Section 6.8.5.

External Validity

External validity questions the generalizability of the results of a study across different experimental settings with larger population not considered in the study.

The six subject systems used in our study are in-house and small to medium in size. All the six respective developers are graduate students; among them two are Ph.D. students and the rest are M.Sc. students at the end of their program. It is arguable that the population is not large enough and subject systems of the study are not representatives of industrial or open-source systems while the developers may not represent the industrial practitioners. Thus, our study may be subject to threats to external validity. However, the choice of in-house systems and their developers helped us to minimize the threats to construct validity. Three (50%) of the participants of the study had years of experience of working as developers in software industry. Thus, the group of developers participated in our study represents a sample of programmers with different levels of expertise. Therefore, we believe that our study achieves an acceptable level of external validity. The threats to external validity can be further minimized by increasing the number and sizes of the subject systems, choosing both industrial and open-source software for study, and involving developers with diverse levels of expertise (i.e., beginner, intermediate, expert).

Reliability

The methodology of the study including the procedure for data collection are documented in this chapter. The NiCad clone detector as well as the in-house software systems used in our study are available online[6]. The data, the OPL implementation of our CP and LP scheduler, as well as the Java implementation of genetic algorithm are also made available online[7] for the interested parties. Therefore, it should be possible to replicate the study.

6.9 Related Work

In this section, we discuss the previous research found in the literature that are relevant to our work on the scheduling of clone clone refactoring as presented in this chapter. Other than ours, there are only

[6] http://www.cs.usask.ca/faculty/croy/
[7] http://usask.ca/~minhaz.zibran/pages/projects.html

three works [37, 176, 183] found in the literature that attempted to deal with the scheduling of code clone refactoring. Bouktif et. al. [37] formulated the refactoring problem as a constrained *Knapsack problem* and applied a metaheuristic *genetic algorithm* (GA) to obtain an optimal solution. However, they ignored the constraints that might exist among the refactorings. Lee et. al. [176] applied *ordering messy GA (OmeGA)*, whereas Liu et. al. [183] used a heuristic algorithm to schedule refactoring of bad smells in general. Both of those works took into account the conflict and sequential dependencies among the refactoring activities, but missed the constraints of mutual inclusion and refactoring effort. In Table 6.12, we summarize all the possible constraints among clone refactorings and those that are taken into account in the aforementioned three works as compared to ours.

Our work, as presented in this Chapter, significantly differs from all those work in two ways. First, for computing the refactoring schedule, we applied constraint programming approach, which is different from theirs. Second, we took into account a wide category of refactoring constraints and dimensions of optimizations, some of which they ignored, as summarized in Table 6.12. Although Bouktif et. al. [37] proposed a brief effort model for code clone refactoring, their model was for procedural code only, whereas, our effort model is applicable to not only procedural but also to object-oriented source code, as it takes into account diverse categories of efforts covering the constructs of an object-oriented system.

Table 6.12: Comparison of code clone refactoring schedulers

	Bouktif et al. [37]	Lee et al. [176]	Liu et al. [183]	Our Scheduler [312]
Approach	GA	OmeGA	Heuristic	CP
Refactoring effort	√			√
Quality gain	√	√	√	√
Sequential dependency		√	√	√
Mutual exclusion		√	√	√
Mutual inclusion				√
Priorities satisfaction				√

O'Keeffe et. al. [220] conducted an empirical comparison of *simulated annealing* (SA), *GA* and *multiple ascent hill-climbing* techniques in scheduling refactoring activities in five software systems written in Java. However, we used CP, which combines the strengths of both AI and OR techniques [24], and thus led to our belief that CP would be a better choice for solving such scheduling problems. Indeed, from our empirical study, we found that the CP approach outperformed both GA and Linear Programming (LP) techniques in the scheduling of code clone refactorings. In our case, GA did not perform well because the refactoring scheduling problem that we have addressed is much stricter with a wide range of hard constraints that might have made the problem *GA-hard* [59, 72], as discussed in Section 6.8.5.

A number of methodologies [71, 156, 257, 258, 300] and metric based tools such as `CCShaper` [105] and `Aries` [104] were proposed for semi-automated extraction of code clones as refactoring candidates. Several tools, such as `Libra` [110] and `CnP` [114], were developed for providing support for simultaneous modification of code clones. Clearly, an efficient scheduling of those refactoring candidates is missing in those tools.

6.10 Summary

In this chapter, we presented have our work towards conflict-aware optimal scheduling of code clone refactorings. To estimate the refactoring effort, we proposed an effort model for refactoring code clones in object-oriented and procedural source code. Our clone refactoring effort model takes into account the efforts required for understand the context, navigating across different parts of the code base, and altering different source code components. Moreover, the risks of refactoring are captured in a priority scheme. To estimate the impact of clone refactoring on the overall software quality we adopt the `QMOOD` [22] design quality model.

Considering a diverse category of refactoring constraints as well as the refactoring efforts, effects, and priorities, we modelled the scheduling of code clone refactoring as a CSOP and implemented the model using the CP technique. To the best of our knowledge, ours is the first effort model for refactoring object-oriented source code and our CP approach is a technique that no one else in the past reported to have applied in this context. Combining the strengths from both AI and OR, the CP approach has been shown to be effective in solving scheduling problems [24, 306]. Our CP scheduler computes the conflict-free schedule making optimal balance among the three optimization dimensions: minimized refactoring effort, maximized quality gain, and satisfaction of higher priorities.

To evaluate our approach, we conducted an empirical study with six in-house software systems and their developers. Through comparison with greedy, genetic algorithm (GA), linear programming, and manual approaches, we showed that our CP scheduler outperformed those techniques. From the user-centric study, our refactoring effort model was also found by the developers to be useful for estimating the efforts required for code clone refactoring. Although our effort model is designed primarily for refactoring code clones in object-oriented source code, with minor tuning, the model can be applied to clone refactoring in procedural source code, or source code refactoring in general.

Chapter 7

Vision of Software Clone Management

"If you show people the problems and
you show people the solutions they will be moved to act"
– Bill Gates

Clone management is a vast area of research and contributions. The process of clone management can be initiated with the detection of clones and then careful refactoring of wisely chosen clones can help minimizing the number of clones and maintain code quality. The insights obtained from clone analyses can help making wise choices and devising techniques and strategies for clone management. In the earlier chapters of this thesis, we presented our contributions in these three most important areas of clone management. However, we do not expect that this thesis brings "the end" of clone management research. Instead, based on an extensive survey of the literature and our experience in the area, in this chapter, we present possibilities for improvements that can be achieved by long-term research efforts from the community.

This chapter is organized as follows. In Section 7.1, we rehearse the motivation behind clone management and expose the dearth of extensive surveys in the area. In Section 7.2, we describe the vagueness in the definition of clones and indicate possibilities to address this issue. Section 7.3 introduces different aspects of clone management activities along with the current state and future possibilities towards a versatile clone management system. Section 7.4, presents the design space for an effective clone management tool. In Section 7.5, we speculate whether clone management in the test code can help in managing clones in production code as well. Section 7.6, we discuss the present state and possibilities towards the adoption of clone management in industrial context. Finally, we conclude the chapter in Section 7.7 with a summary of achievements and scopes along different dimensions of clone management activities.

7.1 Introduction

Since the emergence of software clones as a research area in early 1990s, significant contributions over years made the field grow and become quite a mature area of research. Over the entire course of software clone research there have been only two notable general surveys on clones. Koschke [162], in 2007, presented a brief summary of the important findings about different aspects of software clones including cause-effect of cloning, clone avoidance, detection, and evolution along with a set of open questions. In the same year, Roy

and Cordy [236] also published another survey containing a thorough review on those same areas with specific focus on clone detection tools and techniques. A few recent surveys either focus on detection [232, 243] or evolution of clones [222]. In this chapter, we provide an extensive survey on code clone research with strong emphasis on clone management and point to future research directions in the area. We believe that years of further research efforts from the community is necessary to reach a satisfactory state in clone management.

7.2 How Much is Clone?

Recall from chapter 2 (Section 2.1) that duplicate or similar code fragments are roughly known to be code clones, but the definition of clone has remained more or less vague over the last decade. The vagueness is reflected in the definition given by Ira Baxter, "Clones are segments of code that are similar according to some definition of similarity (Ira Baxter, 2002) [162]." Such a definition is dependent on how similarity is defined, and also raises question on how much of code can be regarded as a code segment. Clone research over the past decade somewhat addressed these issues, and agreed on the categorizing definitions of code clone presented in chapter 1. A number of similarity-based alternative definitions were also proposed by Mayrand et al. [197], Balazinska et al. [20], Bellon et al. [35, 166], and Davey et al. [58], and Kontogiannis [159], but they are not widely used.

While the definition of *Type-1* and *Type-2* clones are somewhat precise, the definition of *Type-3* clones still remains vague [162]. The definition does not precisely indicate how much differences in terms of addition, modification, or deletion of statements are allowed in code segments to be regarded as *Type-3* clones. The practitioners commonly consider code segments as *Type-3* clones when the difference in the statements remain below a (dis)similarity threshold [15, 182, 238, 310]. However, a consensus on an appropriate value for such a threshold is yet to be established [162].

Currently, a researcher's definition of similarity is typically constrained by the program representation and detection mechanism of his or her particular clone detector and, hence, varies from tool to tool and also from parameter settings controlling a tool. A common definition is needed when empirical results are to be compared, for instance, on effects of clones or on accuracy of clone detectors. The difficulty to reach a consensus on a suitable definition, however, inevitably depends also on the purpose of the clone detection. A definition of similarity will include the "value" of a clone for the given task (e.g., bug fixing or refactoring). We do not foresee the advent of a unified definition, we rather expect that task-specific taxonomies of code similarity will emerge in the future and studies will further differentiate contexts and purposes of clones.

7.2.1 Choice of Clone Granularity

According to the definition of clone (as presented in chapter 1) that is currently accepted in the community, a pair of very small portion of code such as two identical identifiers, two similar statements, functions or blocks each having only one statement can also be valid candidates for clones. However, those tiny code

segments cannot be real clones of pragmatic significance. Thus, the practitioners typically disregard those code segments that are smaller than a given threshold. Again, due to the differences in the varying contexts and techniques for clone detection, there has been no consensus in the community on such a threshold, although minimum 20 to 30 tokens [154, 248] or three to five lines of code [242, 313, 308, 310] is a common threshold used in practice.

We believe that the chosen granularity of the code segments should exhibit some characteristics so that the detected clones at that granularity actually becomes useful from the maintenance perspective. In this regard, we propose the following desired characteristics as inspired by our experience and the criteria proposed by Giesecke [86].

Coverage: The set of all code segments should cover maximal behavioural aspects of the software.

Significance: Each code segment should possess implementation of significant functionality.

Intelligibility: Each code segment should constitute sufficient amount of code such that a developer can understand its purpose with little effort.

Reusability: The code segments should feature a high probability for informal reuse.

A source file typically contains a large bulk of code (a Java source file may contain multiple classes, interfaces, and so on). Alhough the set of all source files cover all behavioural aspects of the software, informal reuse at file level is unlikely in a software system. Classes in object-oriented systems also satisfy the *coverage* criterion, but they are also unlikely to be informally reused, as more elegant concepts such as inheritance and delegation are there for their formal reuse.

Methods or blocks (a method body itself forms a block) cover almost all behavioural aspects, and those that are missed (e.g., library/package inclusion, declaration and initializations) can be neglected [86]. Methods and blocks contain implementation of significant functionality that can be understood with less effort than for an entire source file or class. Methods isolate functionality and so often do the blocks, and thus they are likely to be informally reused, as there is no easy rigorous way to reuse methods from another context [86], other than formally composing a library including the commonly accessed methods.

Code segments at the arbitrary statement level satisfy the *coverage* criterion, but not the other three criteria. A sole statement or a short sequence of statements typically does not cover a unit of significant functionality. The meaning of a statement, in general, is likely to be incomprehensible without considering its context, and the context of the host block or function. Moreover, sequences of import/include statements, declaration/initialization statements, or the sequences of statements spanning code boundaries such as boundaries of two functions, classes, or blocks do not exhibit significant potential for reuse, rather those should be ignored in many cases while dealing with code clones.

Thus, we suggest that code segments at the level of functions or blocks can be the most suitable granularity for dealing with code clones, specially for maintenance in terms of reengineering and refactoring opportunities.

Giesecke [86] also proposed in favour of function as the most adequate level of clone granularity. Indeed, block level granularity also covers functions, although it does not distinguish a block that constitutes an entire function body from a block which does not. Driven by the same understanding, Higo et al. developed `CCShaper` [109, 105] to post-process the clone detection result from `CCFinder` to extract block level clones as the potential candidates for refactoring.

7.3 State and Vision of Clone Management

Indeed, clone management is a wide area of research and development. Current state of the art has to go a long way to achieve an integrated, versatile, and flexible tool support for effective clone management. Based on extensive literature survey [311] and our experience in the area, we envision a set of clone management activities and workflow as presented in Figure 7.1.

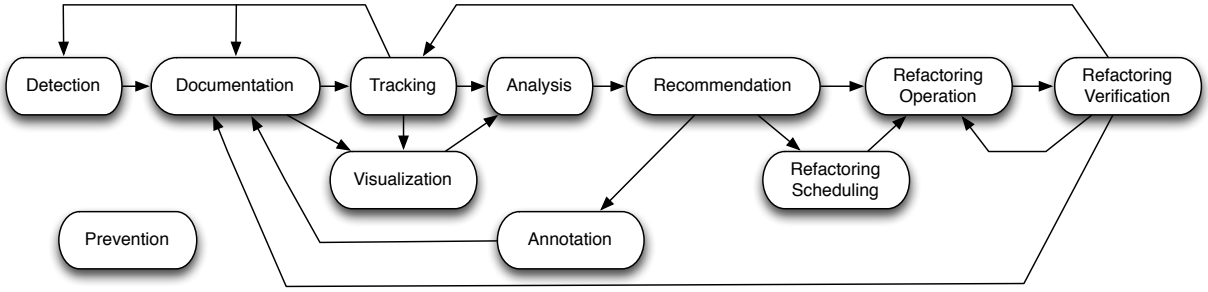


Figure 7.1: Clone management workflow

To manage clones, first they have to be identified. The result of clone detection forms clone documentation that records the location of code segments and their clone relationships. If the code base changes due to ongoing development, the changes and locations of the clones need to be tracked, and the documentation needs to be updated accordingly. The clone documentation may be analyzed to determine justification of clones or to find potential clones for removal. Clone visualization techniques can aid such analysis. Clones that are found to have justified reasons to exist may be further documented and/or annotated. The candidates for refactoring can be scheduled for modification and/or removal. Upon the application of refactoring operations, a follow up verification step may examine if the refactoring caused any change in program behaviour, and in accordance, may initiate roll-back and re-refactoring. Upon completion of refactoring, the clone documentation needs to be updated for consistency. In the following sections, we describe the state of the art and scope for further development in support for each of the clone management activities.

7.3.1 Clone Prevention

Clone-aware development practice can help avoiding creation of unnecessary clone clones. Programmers and developers can be informed the impacts of clones and trained starting from the institutional course-work level

so that they can adopt a programming practice with minimal copy-paste. However, as described in chapter 1, there may be unintentional clones without the awareness of the developers, and at times, one may have to deliberately create clones for several reasons such as for keeping program components decoupled.

Ongoing research also reveals the notion of clones in software artifacts other than the source code [130], such as clones in higher level code structure [25, 26] or high level concepts [195], clones in the models of formal model based development [63, 64], in UML domain models [263], UML sequence diagrams [184], in the graph based Matlab/Simulink models [224], and duplication in requirement specification documents [130, 131].

Clones in requirements documents may lead to duplicate implementation of very similar features if, for example, similar use cases are given to different development teams to implement. In turn, this may result in semantic code clones, or even code segments with very similar structure. Therefore, the detection of clones in requirements specification artifacts could help avoiding clones in code, or to identify semantic clones, which in turn could help to differentiate features from sub-features.

Similarly, clones in design models, especially in model-driven development, may also lead to clones in the source code. Thus, the upfront detection of clones in design models might help to reconsider architectural choices, and result in a leaner, more abstract and essential design resulting minimal code clones.

Early detection and removal of clones from software artifacts such as requirements documents and domain models can prevent creation of clones in the source code, which otherwise could have been introduced if similar or duplicate requirements or models are translated to source code. Large scale empirical studies are necessary to further explore these possibilities.

7.3.2 Clone Detection

Until recently, various techniques for code clone detection have been proposed. The strength and weaknesses of different clone detection techniques are discussed in Chapter 2 (Section 2.5). As mentioned before, many clone detectors and prototype tools are available today. Still, little is known about the usefulness of the code clones detected by different clone detectors. Furthermore, the comparative evaluation of the clone detectors is still an open challenge [236, 243] as different tools often use different definitions of clones and we do not have reliable reference benchmark to compare clone detection results. The variation in parameters for tuning the clone detectors is another difficulty in head-to-head comparison of clone detectors [289]. Not to mention, handing big data from large scale clone detection is a growing challenge for clone management and for many other related applications [264].

Challenges in the Empirical Evaluation of Clone Detection Tools

Typically, the effectiveness (i.e., specificity and sensitivity) of a clone detection tool or technique is measured in terms of precision (p) and recall (r),

$$p = \frac{|C_s \cap C_d|}{|C_d|}, \qquad r = \frac{|C_s \cap C_d|}{|C_s|}$$

where C_d denotes the set of clones that a particular clone detection tool identifies, and C_s refers to the set of clones actually existing in the source code. To empirically evaluate a clone detection tool, one needs to compute these quantities. But how can one accurately find all the clones in a system? A reference corpus is necessary to serve as the set of all clones in a system. To obtain this, manual investigation can be a solution for small systems, but the time and effort needed for such a subjective analysis is likely to be impractical for a fairly large system. However, the reference corpus needs to be large to be able to evaluate the scalability of the clone detector under consideration. Therefore, in practice [279, 310], one or more clone detectors that are already known to be effective are applied to detect clones from a system, and the union of all the sets of clones reported by those tools is used as a reference corpus to examine the precision and recall of a clone detection tool to be evaluated.

Using a similar idea, Bellon et al. [35] carried out a case study to compare the effectiveness of six clone detectors (`CCFinder, CloneDr, Dup, Duploc, Duplix` and `CLAN`) representing different clone detection techniques. Bellon's study [35], which was performed more than six years ago, is still the most comprehensive quantitative study on the comparative evaluation of clone detectors, and the reference benchmark produced from the study is the only notable benchmark data available to date [163]. A similar but smaller scale study was also conducted earlier by Bailey and Burd [13].

Bellon's benchmark data was constructed as the union of the sets of clones reported by the clone detectors subject to the study. Then the precision and recall for each tool were computed based on how many of the benchmark clones a particular tool detected. Thus, the result was skewed in favour of those tools that contributed more in the construction of the benchmark data. Moreover, only 2% clones of the reference set were manually verified. Hence, there is a possibility that the benchmark data might have included significant amount of false positives that could have been reported by those participating tools.

The Need for Reference Benchmark

The above discussion suggests that, a reliable large set of benchmark data[1] is still necessary, and new tools should be evaluated with the standard benchmark data before publication [162]. Both precision and recall should be reported, since precision or recall alone draws a partial picture only. Moreover, both the runtime complexity and memory efficiency should be reported, since these two criteria affect the scalability of a certain technique. Theoretical runtime complexity only can be misleading, because high memory requirement may cause time-consuming page swapping at the operating system level, and increase the execution time in practice [245].

The creation of a reliable large benchmark data set with manually verified clones can be a mammoth task. The tedious work of manual investigation and verification can be accomplished by collaborative efforts from the community. The *mutation framework* of Roy, Cordy, and Svajlenko [240, 264] can also be used for the

[1] Similar to what available at `http://math.nist.gov/MatrixMarket/` for use in comparative studies of algorithms for numerical linear algebra.

purpose by injecting a set of artificially created (and manually verified) clone fragments in various locations of a certain code base, and the set of synthetic clones can serve the purpose of a reference benchmark.

Indeed, the vagueness in the definitions of clones may lead to disagreement in different human evaluators' perception of clones, as found in the study of Walenstein et al. [286]. Therefore, the purpose of the benchmark data should be defined first, and the definition of similarity should be formalized accordingly, as well as the level of clone granularity. A purpose could be, for example, the evaluation of clone detection techniques for clone maintenance and removal.

7.3.3 Clone Documentation

Most of the clone detectors record a code fragment (as well as a clone fragment) using the host file's name along with the range of line numbers [238], or the file-name along with character offset and length of the code fragment [114]. However, different clone detectors report the results of clone detection in different formats such as XML, HTML, and plain text. There are variations in the reported information as well. Some clone detectors (e.g., `CCFinder`) record clone information in terms of clone pairs while some other clone detection tools (e.g., `NiCad`) record information in terms of clone-groups consisting of two or more cloned code segments. Such variations make it difficult for data exchange between clone detectors, which also adds to the challenges in head-to-head empirical comparison of clone detectors. To minimize the differences in the presentation of clone information, Harder and Göde [100] recently proposed the *Rich Clone Format (RCF)*, an extendible schema based data format for storage, persistence, and exchange of clone data.

Note that the description of code regions in terms of absolute locations in the host file can be invalidated when changes in the file alters the line numbers, even if the changes are not inside the code/clone regions of interest. To minimize this threat, in `Wrangler`, relative locations of program entities are used instead of absolute locations [180]. The starting line number of each function in its host file is recorded, and with the relative location, every function is considered to have started from line 1 at column 1. This was expected to save clone regions from being invalidated due to the changes in the absolute locations.

Duala-Ekoko and Robillard [67, 69] proposed clone region descriptor (CRD) to describe clone regions within methods in a way that is independent of the exact text of the clone region or its absolute location in a file. The definition of CRD in extended BNF form is as follows:

$$
\begin{aligned}
&< \text{CRD} > \quad ::= \quad < file >< class >< \text{CM} > [< method >] \\
&< method > \quad ::= \quad < signature >< \text{CM} >< block > * \\
&< block > \quad ::= \quad < btype >< anchor >< \text{CM} > \\
&< btype > \quad ::= \quad \text{‘for’} \mid \text{‘while’} \mid \text{‘do’} \mid \text{‘if’} \mid \text{‘switch’} \mid \text{‘try’} \mid \text{‘catch’}
\end{aligned}
$$

Thus, CRD describes a code/clone region in terms of the distinguishing descriptions of the enclosing file, class, method, block, and a corroboration metric (CM). The corroboration metric is simply the addition

of cyclomatic complexity and fan-out of the block. The $< anchor >$ for a block is a distinguishing string description for the block, and the derivation of the description is dependent on the type of the block. The termination statement for a 'loop', the branching predicate for an 'if' block, and the switch expression for a 'switch' statement is used as $< anchor >$. For a 'try' block, CRD uses the list of exception types caught in 'catch' clauses associated with the block. For a 'catch' block, simply the type of the exception caught is used as the $< anchor >$. Using the CRD scheme, Duala-Ekoko and Robillard developed `CloneTracker` [68] for tracking the evolution of code clones detected by the underlying clone detector, `SimScan`.

However, such a scheme like CRD [67] has a number of limitations. First, small changes in the code corresponding to the $< anchor >$ (e.g., termination condition of loop, branching predicate of conditional statements) will invalidate the CRD. Second, the scheme is vulnerable to nesting levels, and thus a simple addition or removal of nesting level will invalidate the CRD. Third, the association of 'else' blocks with the closest 'if' block prevents the CRD scheme differentiating between the two types of blocks. Most importantly, the use of the CRD scheme did not save `CloneTracker` [68] from re-invoking the underlying clone detector to identify possible changes in the clones, although the computational expense of re-detection was indicated as one of the motivations behind the design of CRD.

What Needs to be Documented

The above discussion indicates that the line and column information, or the abstract level CRD-based documentation of clone regions are more or less vulnerable to changes in the source code. To overcome such sensitivity to code change, marker based tagging support in IDEs such as Eclipse can be used for clone documentation. Such tagging of clones can provide built-in support for accommodating changes in the source files [48]. Further investigation may be required to verify this possibility.

Note that all the clone detectors document clone regions with respect to their locations in the source files only. Only the CRD scheme captures information about the classes hosting the clones, and only `CloneTracker` [68] makes use of CRDs. However, `CloneTracker` itself is not a clone detector, rather it computes CRDs from the clone detection results obtained from the `SimScan` clone detector.

We believe, in the documentation of clone regions, information about the location of clones in the program elements (e.g., classes in object-oriented code) can be useful for the analysis and visualization of clones with respect to the inheritance hierarchy. More discussion about this need is presented in Section 7.3.6.

Due to the diversity in the information and format of clone documentations adopted by different clone detectors, there is a practical difficulty in data exchange and interoperability between different tools. In Table 7.1, we identify a set of information that need to be included in clone documentation. Interoperability among different tools can be attained if all clone detection/management tools include in clone documentation these information in a common format. The set of information is chosen carefully not only to achieve interoperability, other clone management activities are also taken into account. For example, the information about the clone-groups can serve the developers in their decision for clone refactoring, while the evolutionary

Table 7.1: Information need to be recorded with clone documentation

Information	Purpose
Core Information	
File name and path	To identify source file that contains the clone
line numbers	To identify the start and end lines of the clone region
Class name	To identify the host class containing the clone
Complementary Information	
Character offsets	For more precise identification of beginning and end of clone region
Tag/comment placeholder	Justification about creation/preservation of the clone
Information About Clone-Group	
Group size	Number clones in the group
List of clones	List of clones in the group
Parameters	To identify where variability among the members occur
Evolutionary Information	
Ancestor group	Link between clone-groups in current and previous version of the system
Evolutionary group parameters	To identify group level differences between a clone-group and its ancestor
Ancestor clone	Link between clone fragments in current and previous version of the system
Evolutionary clone parameters	To identify code fragment level differences between a clone and its ancestor
Tool Information	
Tool name	To identify the tool used for the creation or update of this clone documentation
Timestamp	To identify the time when the clone documentation was created or last updated

information should help in tracing clone genealogies and changes in evolving clones. It is encouraging that a similar initiative has been taken to develop a common model [143] for documenting clone detection results, and a wiki[2] has recently been created for open discussion on the topic.

7.3.4 Clone Tracking

During the development of an evolving software system frequent changes take place in the code base. Such changes may introduce new code segments that might form new clones. Moreover, changes in source files may invalidate the clone regions and clone relationships necessitating corresponding updates in the recording of clone information. Such updates can be accomplished in two ways: Re-detection and incremental detection.

Re-detection

The detection of clones from the entire system may be invoked every time the code changes. This approach may incur too much overhead as the detection of code clones in a fairly large system can be computationally expensive. Hence, the approach is unlikely to be suitable for proactive clone management.

Incremental Detection

A better approach can be incremental detection, where only the source code in the modified portion of the code base is examined for any clones and the outcome is accumulated with the previously preserved clone detection results.

[2] `http://www.softwareclones.org/ucm/`

Not many attempts were made towards incremental clone detection. The first attempt was made by Göde and Koschke [88, 92]. They proposed a suffix tree based algorithm `iClone` [88, 92] for incremental detection of clones in subsequent versions of a given system. As input, `iClone` needs sequences of revisions of a program that is to be analyzed. In addition to the source code for each revision, `iClones` also needs a separate file as input that summarizes, in `Subversion`[3]'s format, all the changes in the source files for every two consecutive revisions.

In `iClone`, each source file is represented as a sequence of tokens extracted using lexical analysis, and tokens are stored in a token table. For the detection of clone in the initial revision, a generalized suffix tree (GST) is constructed for a large sequence of tokens obtained by concatenating token sequences from all the source files. The paths from the root all the non-leaf internal nodes point to clones in the code [310]. The GST is preserved and updated during the detection of clones in the remaining revisions. For the detection of clones in revision i, only the files that are changed, added or deleted between revision $i-1$ and revision i are examined, and in accordance, the GST is updated.

`iClone`'s approach for clone tracking appears to be memory intensive [92, 116], due to the use and maintenance of a large GST constructed from all tokens of the entire program. Moreover, the tokens of each source file are stored in separate token table, and thus a large number of token tables are also needed to be stored, which may also consume a significant amount of memory. Indeed, the suffix tree based clone detection technique of `iClone` can detect *Type-1* and *Type-2* clones only, but not *Type-3* [92, 310]. Besides, the need for input a file summarizing changes between subsequent versions further limits the applicability of the technique in the implementation of a clone management system having *decentralized* architecture. However, the approach can be a suitable for clone tracking in a *centralized* clone management system, where the source code is kept in a central `Subversion` repository and the code change information can easily be made available.

Hummel et al. [116] proposed an index-based incremental clone detection approach. They demonstrated their technique as a pipeline of three phases: preprocessing, detection, and post-processing. The preprocessing and post-processing components were reused from `ConQAT`. The detection component introduces *clone index*, the central data structure of their approach. The *clone index* is a list of tuples *(file, statement index, sequence hash, info)*, where, *file* is the name of the source file, *statement index* is the position in the sequence of normalized statements for the *file*, *sequence hash* is an MD5 hash code computed over the chunk of n normalized statements from the *statement index*, and *info* contains additional data such as the start and end lines of the chunk of consecutive statements. The heart of the technique lies in the *sequence hash*; tuples with the same sequence hash indicate possible clones containing at least n statements. Consecutive such clones are further merged to report only maximal clones.

Similar to `iClone`, the technique of Hummel et al. is also limited in detecting *Type-1* and *Type-2* clones, but not *Type-3*. The first step of the detection algorithm is to create, for each file, a list of duplicated chunks.

[3] `http://subversion.tigris.org/`

For a large software system, preserving such lists can cause significant memory consumption and updating those with the changes in the source code can incur significant computational overhead. The MD5 hashing algorithm, specially for small n, might produce same hash value for different chunks of statement, and thus may cause false positives in the clone detection [116]. The technique can also appear inefficient, when the list of tuples are not maintained as a sorted list, while such maintenance may frequently invoke sorting overhead whenever the source code changes.

Higo et al. [111] proposed a PDG-based incremental clone detection technique, where PDGs are generated from the analysis of control and data dependencies in the program code. The PDGs are preserved in the database, and clone detection is performed by approximate comparison of PDGs. The PDGs in the database are kept in sync with the evolving source code by examining only the updated and newly created source files. As mentioned in Section 2.5.1, PDG based techniques are computationally expensive and they often report *non-contiguous* clones that may not be perceived as clones by a human evaluator.

Li and Thompson [180] enhanced the clone detection technique of `Wrangler` by introducing incremental detection. The initial clone detection is performed in two steps. First, the source code is normalized and parsed to produced AST (Abstract Syntax Tree). The AST is then annotated and serialized. Then a suffix tree based approach is applied to the serialized AST for detecting the initial clone candidates. The second step applies *anti-unification* technique to get rid of the false positives. The annotated AST is preserved, which is updated whenever changes in the source code take place. The approach maintains a table, where the annotated AST representation of each expression statement is preserved. The storage and update of the table might cause significant memory consumption and computational cost. Moreover, the approach is limited in the detection of only *Type-1* and *Type-2* clones in source code written in the functional programming language Erlang.

The clone tracking approach of `JSync` [215, 216] appears to be computationally elegant. `JSync` preserves the clone-groups and N buckets obtained from the initial clone detection. Since `JSync` is implemented as a plugin to SVN, the change information of the source files are readily available, and based on that information `JSync` can determine the fragments modified, added to, or deleted from the source code repository. `JSync` then removes from the clone-groups those fragments that were changed or deleted. Then the LSH (Locality Sensitive Hashing) technique is applied to the newly added and modified code fragments to place them in the buckets. Then the fragments in each bucket are compared pair-wise to update the clone-groups. Thus, the clone detection technique of `JSync` appears to be inherently incremental and consequently computationally efficient for tracking clones.

Recently, Zhang et al. [303] proposed an approach to provide timely notification about relevant code cloning events for different stakeholders through continuous monitoring of code repositories. They adopted the incremental clone detection technique based on the `CCFinderX` clone detector. A similar system for notifying the creation and changes of code clones was also proposed by Yamanaka et al. [296].

Table 7.2: Summary of tool support for incremental clone detection

Tool	Technique	Integration with IDE/VCS	Supported clone types		
			Type-1	Type-2	Type-3
`iClone` [92]	Preservation of suffix tree	Separate tool	●	●	○
Tool of Higo et al. [111]	Preservation of PDGs	Separate tool	◐	◐	◐
Tool of Hummel et al. [116]	MD5 Hashing and indexing	Integrated with ConQAT	●	◐	○
`Wrangler` [180]	AST, suffix-tree, anti-unification	Integrated with Eclipse, Emacs	●	●	○
`JSync`[215] [216]	AST, LSH, feature vector	Integrated with subversion	●	●	◐

Legends: ●= supported, ◐= partially supported for a limited subset, ○= not supported

In Table 7.2, we summarize the techniques and tools proposed for incremental clone detection. As can be noted, all the tools have vivid limitations in dealing with *Type-3* clones, and the storage of a high volume of data has been a common issue with all the techniques. Further research in the area my inform techniques for integrated incremental detection of clones including *Type-3* with better storage efficiency. We envision that other classes of techniques (cf., Chapter 2) will also have incremental variants as there is a clear need for scalable, fast and near-miss incremental detection techniques for efficient clone management.

7.3.5 Clone Refactoring/Reengineering

The investigations of opportunities for clone based reengineering and refactoring of clones for their removal have suggested techniques such as generics, design patterns, software refactoring patterns, and synchronized modifications of code clones.

Generics and Templates

Basit el al. [28] investigated the potential of generics in removing code clones. They carried out two case studies on the *Java Buffer Library* and the *C++ Standard Template Library (STL)*. The *Java Buffer Library* was found to have 68% redundant code, and using generics they were able to remove only 40% of them. Although they performed little better for the *C++ STL*, they concluded that the constraints of language constructs limit the applicability of generics in clone removal. They further hypothesized that meta level parameterizations might perform better as they are relatively less restrictive than generics or templates.

The hypothesis on the potential of meta level parameterizations was addressed by Jarzabek and Li [121] in a later study. They also used the *Java Buffer Library* for their case study. They applied a generative programming technique using *XVCL (XML-based Variant Configuration Language)*[4] to represent similar (but not necessarily identical) classes and methods in generic and adaptable form. Using the technique they were able to eliminate 68% of the code from the original *Java Buffer Library*.

Recently, Hotta et al. [112] proposed an approach to identify prospective pairs of function clones for refactoring using the *Form Template Method* refactoring pattern. Their approach is based on the analysis of clones detected using a PDG-based technique. However, the technique is applicable in a certain scenarios where a number of constraints can be satisfied. For example, the pair of functions must have the same return type and they must have been defined in different classes sharing a common base class [112].

Design Patterns

Balazinska et al. [21] attempted to replace code clones by applying the *strategy design pattern* for partial redesign of software systems written in Java. The idea was to factorize commonalities in the cloned methods and parameterize their differences to preserve the original behaviours, and then weave them through the *strategy design pattern*. Their approach was realized in a tool named `CloRT`, and the reengineering technique was applied to the source code of JDK 1.1.7 for empirical evaluation. However, the *synchronized* (thread safe) methods were kept out of the study. The reengineering process merged the source code of 28 methods, but created 84 new methods, and thus actually increased the line of code (LOC). Indeed, LOC can be a fair estimation for the size of the code base, but not for design quality. Use of a design quality metric suite could have been used to reflect the actual impact of the reengineering technique on the quality of the source code. The use of other design patterns, such as the *factory pattern* may also produce similar result. Further investigation is needed towards this possibility.

Traits

Traits [256] are a modularity mechanism that complements inheritance to facilitate an orthogonal means of sharing functionality in object oriented classes. Traits can be considered as a language extension, and essentially, a trait is a set of pure methods (methods that do not directly refer to any instance variable), which can be used in a class or another trait simply by name. Traits can also be a potential mechanism for removing duplicated code, when it becomes difficult due to restrictions from inheritance hierarchy. Murphy-Hill et al. [213] applied the traits mechanism to remove duplicated code from the *java.io* library. Using 14 traits, they were able to get rid of 30 duplicated methods by refactoring 12 classes.

[4] `http://xvcl.comp.nus.edu.sg/`

Aspects

Aspects [150] and aspect-oriented techniques (AOT) support the modularization of features that cross-cut the class hierarchy. AOT can be promising in improving modularization, which consequently may reduce code clones. The behaviours that Murphy-Hill et al. [213] parcelled up into traits could also be put into different aspects, and introduced into the target by a mechanism like intra-class declarations of AspectJ [213]. On the contrary, Jarzabek and Li [121] argues that due to the lack of parameterization mechanism and constrained composition rules, AOT in its pure form is not meant for elimination of redundancies. More investigation in this regard is necessary, which might reveal interesting results and opportunities.

Synchronized Modification

Simultaneous editing has been a popular approach for applying the same editing operations to more than one identical text-snippets.

An early work [200] on simultaneous editing incorporated the feature in a text processing system called `LAPIS` [199]. `LAPIS` offers a library of built-in parsers and patterns for various kinds of text structure, including HTML and Java source code. The user is enabled to select multiple regions of text (manually or by using patterns) and perform simultaneous editing.

Toomim et al. [277] proposed to apply simultaneous editing to simultaneously edit duplicated code. They called such editing as "linked editing", and to support this inside editor, they developed `CodeLink` as an extension to the XEmacs editor. The "linked editing" of `CodeLink` is functionally same as the "simultaneous editing" of `LAPIS`. However, `CodeLink` saves the user from explicitly select multiple regions for edit as to be done with `LAPIS`. Instead, in `CodeLink` the user manually selects two code segments to link them. Then `CodeLink` applies a LCS (longest common subsequence) algorithm to map between the differences and similarities in the code segments, and enables simultaneous editing in the duplicated regions. However, due their dynamic programming implementation of the LCS algorithm, their approach appears to be memory intensive and computationally expensive. For k clones, each of n tokens, the dynamic programming implementation of LCS algorithm runs in $O(n^k)$ time. Moreover, `CodeLink` "does not always report the most intuitive set of differences between any two code fragments" [277]. The support for linked/simultaneous editing is also available in tools such as `CPC` [292] and `CloneBoard` [62].

Consistent Renaming

Programmers often perform modifications after copy-pasting a code fragment. Such modification typically include renaming of identifiers according to the new context of the cloned code. IDEs like Eclipse provide necessary support for consistently renaming an identifier and all its references within scope. Jablonski and Hou developed `CReN` [117] as a plugin to Eclipse that can check for any inconsistencies in the renaming of identifiers within a code fragment and suggest modifications for making the renaming consistent. They further extended `CReN` and developed `LexId` [118], which supports consistent modification of the same parts

of different identifiers in a code segment. However, the consistent renaming support from these tools are limited to within a single code fragment, unlike the linked/simultaneous editing between clone pairs [114]. Jacob et al. developed `CSeR` [119] by extending the Java editor of Eclipse to visualize the similarities and differences while a programmer edits a copy-pasted code. Hou et al. combined `CReN` and `CSeR` into a single toolkit named `CnP` [114], which they are developing towards support for proactive clone management based on copy-paste clipboard activities.

Since `JSync` [215] is developed as a plugin to the SVN version control system, it can exploit the change information between versions of Java source files to determine whether any changes occurred in cloned code regions. For such a clone pair it maps between the nodes of ASTs of the corresponding code using a *treed* algorithm, detects inconsistencies in the identifier renaming, and suggests for consistently renaming them. This feature of `JSync` is very similar to that of the consistent renaming support of `CReN` [117]. `JSync` also supports *clone synchronization* between clone pairs when one of the clones changed between version while the other remained unchanged. Clone synchronization of `JSync` simply accommodates the changes from the modified code fragments into the unchanged code. In case both the clone pairs changed between versions, `JSync` suggests *clone merging*. For clone merging, `JSync` suggests to accommodate the changes from both the clone fragments to each other, and any conflict is simply left to the user's resolution. Moreover, the clone synchronization support of `JSync` is limited to simple changes in the identifiers, control structures, literals, and method calls. However, `JSync` also allows annotating clone pairs in case the developer wants to retain the inconsistencies. Through an empirical evaluation, Nguyen et al. [215] reported that `JSync` can attain 83% precision in recommending change propagation between clone pairs. However, they did not report what fraction of the clones were of *Type-1, Type-2,* or *Type-3*.

Refactoring Patterns

Fowler in his book [81], presented 72 patterns for refactoring object-oriented source code in general for the removal of code smells. Over time the number of refactoring patterns has increased to 93, and a *refactoring catalog*[5] is maintained, which lists and describes them all. Earlier research [104, 105, 135, 156, 176, 257, 300, 307, 308] suggest that a subset of those general software refactoring patterns [81] are suitable for clone refactoring, as we described in Chapter 6. Further detail about these refactoring patterns can be found at the *refactoring catalog* and elsewhere [81].

Kerievsky [149] also proposed a *chained constructor* refactoring pattern[6], which can also eliminate duplicated code from the constructors of the same class [214]. Other refactoring patterns that can be found in the literature are some sort of variants or compositions of the aforementioned object-oriented refactoring (OOR) patterns. Other than the OOR patterns, Schulze et al. [258] proposed three aspect oriented patterns described as *extract feature into aspect, extract fragment into advice*, and *move method from class to interface*.

[5]Catalog of OO refactoring patterns: `http://refactoring.com/catalog/`
[6]Catalog of 27 refactoring patterns from J. Kerievsky's book: `http://industriallogic.com/xp/refactoring/catalog.html`

Tool Support for Refactoring Patterns: Although a variety of potential refactoring patterns have been identified for code clone refactoring, tool support for the automated or semi-automated application of those refactoring operations has been quite limited. A number of semi-automated analytical techniques have been proposed for finding candidate clones that are easier and suitable for refactoring. Such techniques are described in Section 7.3.6. In this section, we focus on the techniques realized in tools for the modification of the candidate clones in actual refactoring operations.

Limited support for the *rename refactoring, extract method, extract superclass* and *pull-up method* refactoring can be available from IDEs like Eclipse, when the necessary program element are manually identified by human efforts. To aid refactoring of clones in object-oriented code written in C++, Fanta and Rajlich [75] developed tool support for five high level restructuring features namely *function insertion, function expulsion, function encapsulation, renaming*, and *argument reordering*. However, the implementation of those features are very limited in operational scope. For example, the inserted function cannot be a member of any class, cannot be overloaded, cannot be a template function, and cannot be called using pointer.

`Wrangler` [179] supports clone removal by *folding* expressions against a function definition in Erlang programs. Folding searches in the program for instances of the right-hand side of the selected function clause, and under the user's control, it replaces them with applications of the function to the actual parameters. The *folding* operation in functional languages is similar to the *extract method* refactoring in imperative languages.

`CloneDR` provides facility to automatically or interactively perform *extract method* refactoring by identifying identical blocks of code and producing a parameterized function/method that generalizes the blocks. Recall that `CloneDR` has limitation in detecting *Type-3* clones [243], and the application of extract method refactoring may be simpler for *Type-1* and *Type-2* clones, which is likely to be much more cumbersome for *Type-3* clones due to addition or deletion of statements at arbitrary locations of the clone fragments.

Verification of Clone Modification/Refactoring

Human effort in source code modification can be error prone. For example, while renaming identifiers in a copied code, the programmer may mistakenly leave a name unchanged. The work of Jablonski and Hou can be considered as a contribution in the verification of clone modification. Their tool `CReN` [117] can check for any inconsistencies in the renaming of identifiers within a code fragment and suggest modifications for making the renaming consistent.

`JSync` [215] also offers a similar facility. While `CRen` supports this validation right in the editor inside Eclipse IDE, `JSync` performs the check at the server side, when the code is checked-in to the central repository. Any identified inconsistency in the renaming of identifiers are reported back to the developer with suggestions for attaining consistency.

By definition, refactoring should only alter the structure of the program without changing its behaviour. Therefore, the automatic or semi-automatic refactoring of code clones should be followed by verification to ensure that the refactoring did not change the program behaviour [309]. Automated test case generation and

automated adaptation of test cases to the refactored code (clones) may help in this regard. However, to the best of our knowledge, no significant attempt is made towards this possibility.

7.3.6 Clone Analysis

Many studies have been conducted to explore the characteristics of existence and evolution of code clones in software systems. While most of the work found in the literature are based on open-source software systems, there is lack of studies in the industrial context. Clone analysis helps to find opportunities for reengineering/refactoring code clones. Moreover, in depth studies on clone evolution can reveal the types of changes and their impacts that different categories of clones experience during their evolution, and thus can inform clone management by indicating what type of tool support is required for effective clone management in practical settings.

Analysis of Clone Evolution

Software development and maintenance in practice follow a dynamic process. With the growth of the program source, code clones also experience evolution from version to version. Many studies have been conducted to date for understanding the overall evolution [7, 8, 89, 185, 313], stability of cloned code [18, 91, 113, 169, 170, 201, 202], the relation of clone evolution with software faults [23, 94, 259, 294, 295], and other characteristics of clone evolution. While such high level studies inform the characteristic and impact of code cloning, more lower level analyses that investigate the change patterns in the evolution of individual clone fragments can suggest techniques for optimizing clone management including refactoring and removal. There is a recent survey on clone evolution [222]. Hence, we keep this section brief with specific focus on the evolution of individual clone fragments and their change patterns.

Recall from Chapter 2 (Section 2.6) that Kim et al. [153, 154] first introduced *"clone genealogy"* to study the evolution of individual clone fragments. The categorization of clone genealogy and clone change patterns mostly used in the literature are described in Chapter 2. Aversano et al. [11] further caracterized the inconsistent change pattern into two categories:

Independent Evolution: In independent evolution, the clones of a clone-group, once changed inconsistently, evolve independently across versions. Thus, independent evolution may cause branching with multiple lineages in a genealogy. Figure 2.3 shows independent evolution of clone lineages originating from version k of the software system.

Late Propagation: In late propagation, one or more clones in a clone-group may experience inconsistent changes and may disappear from the clone-group in the next one or more versions, but at a later version those clones re-appear in the clone-group. Figure 7.2 demonstrates late propagation, where the clone fragment D disappears between transition from version V_{k+1} to version V_{k+2} and after two later versions the clone fragment re-appears in the clone-group at version V_{k+4} of the software system.

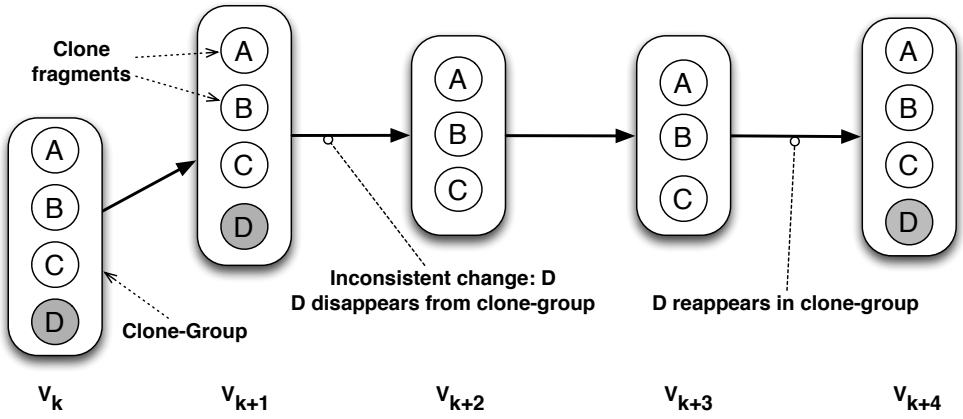


Figure 7.2: Late Propagation

Studies [18, 36] on clone change patterns revealed that inconsistent changes in clones sometimes caused program faults. Moreover, late propagation is reported to have even more significant correlation with software defects and thus concluded to be "more risky than other clone genealogies" [23].

Using the clone genealogy model, recently Saha [247] studied the change patterns in the evolution of clones in five open-source software systems. Major findings from the study report some common properties of frequently changed clone-groups, such as, cohesive clone-groups having small number of clones experienced more changes than others. Intuitively, such clone-groups can be easier for modification/refactoring and it is unlikely that the developers of those open-source systems used any tool support dedicated for clone management. Thus the clone changes could have exhibited a very different pattern, had the developers used proper clone management tool. The frequent changes to small cohesive clone-groups rather suggest the necessity of tool support for dealing with especially the dispersed (a.k.a., diffused [82]) large clone-groups.

Scope for Improvements: Studies on clone change patterns using the genealogy-based model described above can suffer from a number of issues. First, due to the threshold based similarity measure used in practice for *Type-3* clones, there remains an open question on the appropriate value for the threshold. Moreover, for *Type-3* clones, is it an appropriate practice to group *Type-3* clones into disjoint sets? If not, the traditional notion of genealogy cannot apply to *Type-3* clones. Can we devise a more appropriate alternative?

Second, according to the clone genealogy characterization scheme described in Chapter 2 (Section 2.6), a genealogy can be characterized as inconsistently changed if only a single clone over the entire length of the genealogy experiences even a very minor inconsistent change. To draw a better picture, we may capture formation such as, what portion of clones in clone-groups change in how many versions, and how large the changes are.

Third, the definitions of dead/alive genealogies in terms of the *last versions* are subject to the choice of versions. A different choice in the number of versions would have different last version. Moreover, consider

a genealogy originated at the second last version and extended to the last version, while another genealogy originated from one of the very early versions and extended to the second last version and disappears at the last version. The current genealogy model will categorize the first genealogy as alive, while the second as a dead genealogy. Thus, we are missing important information about how long the genealogies propagate. Hence, instead of characterizing as alive/dead genealogies, we might characterize genealogies in terms of what portion of the series of versions the genealogies span, similar to the notion of *succession-length* of Zibran et al. [313].

Finally, from the correlation between late propagation and software defects, can we really derive a causal relationship concluding that late propagation is riskier than other clone genealogies? Inconsistent changes are believed to often cause defects, and clones may disappear from a genealogy due to inconsistent changes. Later modifications, which could be even bug fixing activities, may cause changes in a disappeared clone to sync it back to its original clone-group. In such a scenario, late propagation actually contributes in repairing the defect introduced from inconsistent changes. Thus, we believe, late propagation can really play a dual role, and more studies are necessary to distinguish them.

Analysis to Find Clone Based Reengineering Opportunities

Early work [14, 30] proposed macro extraction by exploiting duplications in source code written in C and C++. While those approaches pioneered clone refactoring research, they were focused on simple refactoring in procedural code and did not tackle the issues that might raise in refactoring clones in object-oriented context.

Ducasse et al. [71] proposed to use clone detection tools for guiding the refactoring of code clones. For the detection of code clones in SmallTalk code, they used `DUPLOC`, a text-based clone detector that performs basic string matching over the lines of source code. From the analysis of the identified duplicated code, they found two cases where clone refactoring operations could be applied. Those two cases were simply two variations of the well-known *extract method* [81] refactoring pattern.

For the identification of code clones as suitable candidates for refactoring, Balazinska et al. [19] proposed clone analysis based on similarities and differences in the candidate code clones. They also suggested a number of measurements for context analysis, which capture the relationships between a class and its member methods, between a method and its member variables, as well as the caller-callee relationships among the methods. Although their work focused on the refactoring of code clones in object-oriented systems, they completely ignored the horizontal and vertical dependencies due to the inheritance hierarchy. Based on a similar analysis, Zibran and Roy proposed a code clone refactoring effort model [308], which takes into account all the relationships including the inheritance hierarchy.

Ueda et al. developed `Gemini` [280, 281], a graphical tool to aid clone analysis for corrective maintenance of code clones. `Gemini` works on top of the `CCFinder` clone detector, and facilitates the visualization of clones using scatter plot. It also computes a number of metrics (i.e., RAD, LEN, POP, DFL, RNR) to

capture various properties of the detected clones [106]. A metric-graph view enables the user to set upper and lower bounds on each metric to filter out those clones that falls beyond the boundaries according to the user's interest [49, 105]. Since `CCFinder` cannot detect *Type-3* clones, the functionality of `Gemini` stayed limited in *Type-1* and *Type-2* clones only. However, they demonstrated that the Scatter Plot of `Gemini` enabled the users' subjective perception to visualize *Type-3* (gapped) clones, without actually detecting them automatically [282].

As `CCFinder` detects clones that can be any arbitrary lines of code in the code base, those often lack the contextual cohesion necessary for effective refactoring. To minimize this issue with `Gemini`, Higo et al. added to `Gemini` a feature called `CCShaper` [105, 109] that applies a parser-based technique (using JavaCC) to extract the block clones from `CCFinder`'s clone detection result. The objective was to extract block clones which might be easily merged by applying the *extract method* refactoring pattern. A tool named `ARIES` [104] was also published, which combined the metric view of `Gemini` and the clone filtering feature of `CCShaper` as well as introduced two additional clone metrics (i.e., NRV and DCH) beyond those previously existed in `Gemini`. Method/function level code clones belonging to different clone-groups, at times, may have dependency in terms of caller-callee or data sharing (reference or assignment) relationships. Yoshida et al. [300] called such clones "chained clones", and argued that all of such clones should be refactored at once. Thus, they extended `ARIES` with a PDG based technique to *identify* such chained clones from the initially detected clones. Indeed, due to the use of `CCFinder` as the backend clone detector, their automated technique can handle only *Type-1* and *Type-2* clones, but not *Type-3*.

Tairas and Gray [269], through an empirical study on two open source Java systems (JBoss and ArgoUML), reported that in some cases clone refactoring was partially performed on only parts of the clones, and in most cases those refactored clones were still further refactorable. However, they focused to investigate only the occurrences of refactorings composed of the *extract method* refactoring pattern. Indeed, there are other types of refactorings including small modifications (e.g., identifier renaming, addition/deletion of statement) in the code fragments, which might have caused partial changes in the clones, as they found. Most importantly, they observed changes by comparing clones between pairs of consecutive releases, and simply *assumed* that those changes occurred due to refactoring. However, such an assumption may not hold true, as those small changes may be caused by many reasons such as corrective or adaptive maintenance, or to change functionalities of those code fragments. Moreover, "it is unlikely that developers of JBoss and ArgoUML used a clone detection tool to identify clones" [269], or used any tool support for clone refactoring. This might be a plausible reason why there were small changes in the clones that still remained refactorable from the perspective of clone removal. This rather emphasizes the need for effective clone management tool support for software maintenance.

Higo et al. developed `Libra` [110], a tool on top of the `CCFinder` clone detector to facilitate the detection of clones of only a given code fragment, instead of detecting all clones from the entire given code base. They argued that such a clone detection may be useful to find potential clones where simultaneous modifications

can be applied. `Libra` was not integrated with any IDE. Moreover, since it internally invokes `CCFinder`, it cannot handle *Tyep-3* clones. Lee at al. [175] used an algorithm based on feature-vector computation over AST that finds the k most similar clones of a given code segment. But, their tool was not reported to have integration with IDE. Zibran and Roy [310] developed a similar tool which is integrated with the Eclipse IDE, and can find all the three types (*Type-1, Type-2*, and *Type-3*) of clones of a chosen code fragment.

Clone Taxonomies Based On Reengineering Opportunities

Analysis of diverse characteristics and properties of code clones in quest of reengineering opportunities led to different taxonomies of code clones, and refactoring strategies for different categories of clones.

From a manual analysis of 800 function/method level clones over six different open-source Java systems, Balazinska et al. [20] proposed a taxonomy of function clones based on the differences and similarities in the program elements. Taking into account the location of clones in the inheritance hierarchy, Koni-N'Sapu [158] proposed a clone taxonomy and a set of object-oriented refactoring patterns for refactoring each categories of code clones. Tokunaga et al. [275] also suggested to develop a collection of code clone refactoring patterns based on precise clone categorization.

Schulze et al. [258] argued that *aspect-oriented refactoring (AOR)* can be more appropriate than *object-oriented refactoring (OOR)* in certain scenarios. They proposed a code clone classification scheme to support the decision whether to use OOR or AOR for clone removal. However, any empirical evaluation of the effectiveness of their clone classification scheme or the clone removal procedure is not available. A similar workflow for the detection and refactoring of code clones was proposed by Kodhai et al. [156], the implementation of which is not available, let alone any empirical evaluation. Torres [278] argued that classification of concepts containing duplicated code can provide hints about which refactoring can be suitable. The applied a concept-lattice based data mining approach to derive four categories of concepts containing duplicated code and suggested refactoring patterns suitable for refactoring clones in each of those categories.

Taking into account both the locations of clones in the file-system hierarchy and (dis)similarities in the functionalities, Kapser and Godfrey [139] proposed a taxonomy, which is often considered as the most extensive clone taxonomy to date [162]. To classify clones from non-cloned code, recently Yang et al. [297] proposed a machine-learning approach which take into account the developers judgment about a particular code is really a true clone or not. Despite the number of proposed clone taxonomies for reengineering opportunities, the state of the art still demands more investigation in this regard suggesting open questions [162] for instance, can other properties such as cost, benefit, risk of refactoring be incorporated in a taxonomy?

7.3.7 Clone Visualization

For the purpose of finding and characterizing code clones suitable for refactoring, reengineering, or removal, in depth analysis of the various properties of the clones and their context is required. Clone visualization has been proven to be effective in aiding such analysis.

Visualization of Distribution and Properties of Clones

A major challenge in identifying useful cloning information is to handle the large volume of textual data returned by the clone detectors. To mitigate the problem, a number of visualization techniques, filtering mechanisms and support environments are proposed in the literature. Jiang et al. [125] categorized the proposed clone presentation techniques based on two dimensions. The first dimension refers to the level at which the entities are visualized (such as at the code segment level or file level or subsystem level). The second dimension refers to the type of clone relation addressed by the presentation, that is, whether clones are showed at the clone pair level or grouped into clone sets or super clones. A *super clone* is an aggregated representation of multiple clone-groups residing in the same source code entity (e.g., file).

Johnson [128] used the popular Hasse diagram to represent textual similarity between files. Later, he also proposed hyper-linked web pages to explore the files and clone-groups [129]. Cordy et al. [53] used HTML for interactive presentation of clones where overview of the clone-groups is presented in a web page with hyperlinks and users can browse the details of each clone-group by clicking on those links. Although such representations offer quick navigation, they cannot reveal the high level cloning relations. A set of polymetric views [234] were also proposed in the literature that permit encoding of a number of code clone metrics to the visual elements.

Among various visualizations, scatter plot is quite popular and capable of visualizing inter-system and intra-system cloning characteristics [52, 186]. However, the size of the scatter plot depends on the size of input (e.g., size of the source code) rather than the amount of cloning. Thus, using a scatter plot for visualizing cloning information of a large software system may become challenging due to the large size of the plot. Moreover, non-contiguous sections that contain the same clone cannot be grouped together in scatter plot. To overcome this, Tairas et al. [271, 272] proposed a graphical view of clones (also known as Visualizer view) that represents each source file as a bar and clones within the files are represented with stripes. Clones belonging to the same clone-group are encoded with same color.

Jiang et al. [124] extended the idea of cohesion and coupling to code clones and proposed a visualization technique that uses shape and color to encode the metric values. They also developed a framework [124] for large scale clone analysis and proposed another visualization, called a clone system hierarchical graph that shows the distribution of clones in different parts (with respect to the file-system hierarchy) of a system. Fukushima et al. [82] developed another visualization using graph drawing technique to identify diffused (scattered) clones, where clone fragments are represented as nodes and the nodes corresponding to clone fragments that are located in the same file are connected with edges to form a clone cluster. Nodes that connect different clone clusters are called diffused clones, which have cloning relationships across different files implementing different functions.

`Gemini` [281] is an example of a clone support environment that uses `CCFinder` for clone detection and can visualize cloning relationship using scatter plots and metric graphs. Kapser and Godfrey developed `CLICS` [140, 144], another tool for clone analysis. `CLICS` can categorize clones based on a clone taxonomy [139]

(previously proposed by Kapser and Godfrey) and can support query based filtering. However, the operability of CLICS is limited to source code written in C/C++ and Java only.

Tairas et al. [272] developed an Eclipse plug-in that works with the clone detector CloneDR, and implements the visualizer view along with a list view for presenting the detected clones. Clone Visualizer [305] is a similar tool developed as an eclipse plug-in that works on top of the Clone Miner clone detection tool. In addition to supporting clone visualization through stacked bar chart and line graph, Clone Visualizer supports query based filtering. CYCLONE[7] [100] is another tool that uses RCF (Rich Clone Format) [100] file as input and supports single and multi-version program analysis and visualization. RCF is a data exchange format capable of storing clone detection results. A separate viewer application named RCFVIEWER[8] is also developed for the visualization of clone information stored in RCF format. Haupmann et al. [101] proposed *Edge Bundle View* for the visualization of distribution of clones across subsystems and directories of the software system. They implemented a prototype of the view on top of the quality assessment toolkit ConQAT. Another implementation of the same view is found in a commercial clone detector, SolidSDD[9].

Table 7.3 summarizes the various clone visualization techniques realized in different tools. As can be noted, all the visualization techniques focus on visualization of clone pairs or clone-groups with respect to their dispersion in the file-system hierarchy only. However, the cost-benefit analysis of code clone refactoring must take into account the distribution of clones in the inheritance hierarchy [307, 308, 312]. Therefore, from the perspective of clone removal or refactoring, the visualization of the clones with respect to the inheritance hierarchy can offer useful insights, and future work in clone visualization should address this possibility.

Visualization of Clone Evolution

Visualization support can aid analysis of clone evolution, and thus different techniques and tools have been proposed for visualizing properties of clone evolution including the genealogy model.

Adar and Kim [3] developed SoftGuess, a system for clone evolution exploration that supports three different views. SoftGUESS is developed on top of GUESS [2], the graph exploration system, which models the evolution of a software system using graphs. The genealogy browser of SoftGuess offers a simple visualization of clone evolution where nodes represent clones, arranged from left to right, and the those that belong to the same class are arranged vertically in the same position. Thus, each column represents a version. A link between a pair of node reflects the predecessor and successor relationship during the evolution of the software. The encapsulation browser shows how clones within a clone-group are distributed in different parts of a system and how they fit in the hierarchical organization of the software system by visualizing the containment relationship through a tree structure. Finally, the dependency graph describes how the nodes (package, class or method) within a version are evolved from other nodes and how they evolve in the next version. In addition, SoftGUESS also supports charting and filtering mechanisms based on Gython, a SQL

[7] http://softwareclones.org/cyclone.php
[8] http://www.softwareclones.org/
[9] http://www.solidsourceit.com/products/SolidSDD-code-duplication-cloning-analysis.html

Table 7.3: Summary of clone visualization techniques

Visualization Technique	Clone Granularity	Clone Relation	Realized in Tool	Integrated IDE
Tree Map [9, 10, 234]	File, Subsystem	Clone Group	VisCad, ConQAT, CyClone	None
Scatter/Dot Plot [9, 10, 102, 234, 281]	File, Subsystem	Clone Pair	VisCad, Gemini	None
System Model View [234]	File, Subsystem	Clone Pair	None	None
Clone System Hierarchical Tree [124]	File, Subsystem	Clone Pair, Clone Group	Yes	None
Hasse Diagram [128]	File	Clone group	No	None
Clone Group Family Enum. [234]	File	Clone group	No	None
Duplication Web [234]	File	Clone pair	No	None
Dependency Graph [140]	Subsystem	Clone Pair	CLICS	None
Hierarchical Dependency Graph [9, 10]	Subsystem	Clone Pair	VisCad	None
Clone Coupling and Cohesion [125]	Subsystem	Super Clone	Yes	None
Metric Graph [281]	Code Segment	Clone Group	Gemini	None
Clone Cluster View [82]	Code Segment	Clone Group	No	None
Hyper-Linked Web Page [53, 129]	Code Segment	Clone Group	NiCad	None
Clone Visualizer View [271, 272]	File, Code Segment	Clone Group	AJDT Visualizer plug-in	Eclipse IDE
Stacked Bar Chart [305]	File, Code Segment	Clone Group	Eclipse plug-in	Eclipse IDE
Line Chart [305]	File, Code Segment	Clone Group	Yes	None
Clone Scatter Plot [234]	File, Code Segment	Clone Group	No	None
Edge Bundle View [101]	File, Subsystem	Clone Pair	ConQAT, SolidSDD	None

type query language. However, SoftGUESS lacks an 'overview' feature and requires users' interaction for data reduction through queries. Although a query is a powerful mechanism to identify important patterns of cloning, the formulation of queries could be difficult due to the cognitive efforts required from the developers.

The clone evolution analysis tool CYCLONE [100] offers five different views to analyze clone data stored in a RCF file, where RCF is a binary format to encode clone data including the evolutionary characteristics. The evolution view in CYCLONE visualizes clone genealogies that uses simple rectangles and circles to denote software entities. Each circle represents a clone fragment arranged in a set of rows where each row represents a particular version of the software. The clones that belong to the same clone-group are packed within a rectangle. Finally, the evolution of a clone fragment across versions is presented with a connecting line. In addition, the view employs colors to distinguish types and the changes in the clones. Although the view highlights many important evolutionary characteristics, the volume of data produced by the genealogy extractor still limits its usefulness, and thus call for aggregation and filtering mechanisms. A similar visualization support is available in VisCad [9], with additional flexibility of metric based filtering of genealogies.

Saha et al. [250] presented an idea for clone evolution visualization using the popular scatter plot. In their proposed approach, scatter plots show the clone pairs associated within a pair of software units (file, directory or package). The clone pairs are rendered with different colors depending on the type of clone genealogies they are associated with. Selecting a clone pair through the users' interaction (double clicking on a clone pair in the scatter plot) shows the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone-groups in a particular version and then provide a way to call for genealogy browser to dig deeper. However, the approach does not provide overall characteristics of the genealogies, and neither an implementation of the proposal nor an empirical evaluation of the technique is available yet. Moreover, due to the large number of clone pairs, the identification and selection of useful patterns in such a scatter plot can be difficult, which is why different variants of the traditional scatter plot appeared in the literature [52].

7.4 Design Space for a Clone Management System

In this section, we discuss different criteria that must be taken into account in making strategic and design decisions for a versatile clone management system we envision.

7.4.1 Clone Management Strategies

For dealing with code clones, Mayrand et al. [196] proposed two concrete activities namely "problem mining" and "preventive control", which were further supported by a later study of Lague et al. [172]. Giesecke [86] categorized them into compensatory and preventive clone management, respectively. Giesecke [86] suggested that all clone management activities can be associated with on or more of the three categories: corrective, preventive, and compensatory management.

Corrective clone management aims for *removal* of existing clones from the system. The objective of *Preventive clone management* is to prevent creation of new clones in the system. *Compensatory clone management* deals with applying techniques (such as annotation, documentation) for avoiding the negative impacts of clones that are not removed from the system for some valid reasons. In practical settings, avoiding clones may be impossible at times, and the expectation of a clone-free system can be unrealistic. Thus, *preventive* clone management actually refers to *proactive* management [114, 115] that aims to deal with the clones during their creation or soon after they are introduced. An opposite strategy, *retroactive clone management* [48] adopts the *post-portem approach* [310], where clone management activities initiate after the development process is complete up to a milestone.

Clone management in legacy systems can be the most appropriate for the *post-mortem* strategy. Indeed, prevention is easier to cure. Therefore, *proactive* clone management is preferable to *post-mortem* approach. While, ideally, all clones should be managed proactively, in practical settings, proactive treatment for all clones may not be feasible or possible. Therefore, a versatile clone management system should focus on support for proactive management, while at the same time, should also facilitate retroactive clone management [48].

7.4.2 Design Choices

Most clone detectors [137, 238, 122, 107] are implemented as stand-alone tools separate from IDEs (Integrated Development Environments) and typically search for all clone in a given code base. While clone detection from such tools can help clone management in post-mortem approach, researchers and practitioners [86, 99, 114, 115, 172, 215, 309, 310] believe that clone management activities should be integrated with the development process to enable proactive management.

Hou et al. [115], during the on-going development of their clone management tool `CnP` [114], explored the design space towards tool support for clone management. However, their work was tightly coupled with the clone detection technique based on the programmers' copy-paste operations. Thus, their findings are limited in scope to the management of copy-pasted code, and most of the findings are not applicable to clone management based on similarity based clone detection.

We identify three major dimensions and some sub-dimensions in the design space for a versatile clone management system. These dimensions are inspired by our experience and the different clone management scenarios reported by Giesecke [86].

Architectural Centrality

The need for the integration of clone management activities with the development process suggests that the IDEs should include features to support clone management activities during the actual development phase. While a programmer typically works inside an IDE running on her individual workstation, for fairly large projects, specially in industrial settings, a team of developers collaboratively work on a shared code base kept in a version control system (e.g., SVN, CVS) set up on a central server. Hence, the supports for clone

management activities can be implemented as features augmenting the local IDEs, or the functionalities can be implemented at the central repository.

Decentralized Architecture: The clone management functionalities, when augment the features in local IDEs, can enable the individual programmers to exploit the benefit of clone management. In the decentralized scenario different developers can use different tools, and some programmers can get the flexibility to completely or partially disregard clone management at their respective situations. Apparently, such a decentralized implementation may completely disregard the existence of a central server, and enforces proactive clone management before check-in to the shared repository. However, this necessitates additional requirement for establishing means for communication between distributed developers, as well as combining and synchronizing clone information across all the developers.

Centralized Architecture: The centralized architecture inherently aims to support clone management in distributed development process. The functionalities can be implemented as a client-server application on top of central version control systems. Such a centralized clone management system may require greater effort and offer less flexibility than a decentralized implementation [86]. Indeed, a client-server implementation cannot support those individual programmers who work alone on their stand-alone local machines [310]. But, the centralized architecture may facilitate the integration of clone detection feature with the continuous or periodic (e.g., diurnal) build process.

Triggering of Clone Management Activity

A clone management activity may be initiated by the practitioner, or such an activity may be triggered in response to certain actions in the system. Thus, an activity can be human triggered or system triggered.

Human Triggered Initiative: A developer, after writing or modifying a piece of code, may invoke search for its clones in the system, upon finding the clones, she may analyze and decide how to deal with them. In such an ad-hoc triggering scenario, the developer, at times, may forget to perform the necessary clone management. An instance of clone management activity may also be periodically scheduled in advance as part of a larger plan of process activities, and clone management activities can be carried out following the post-mortem approach on the current status of the code base.

System Triggered Initiative: The development environment can trigger clone management activities in response to certain events, such as saving changes in the code, or check-in of modified code to the central repository having the clone detection capability integrated with the build process. Such events may notify and suggest the developer to perform the required clone management operations. However, care must be taken so that those auto-generated notifications and suggestions do not irritate the developer or hinder her normal flow of work.

Scope of Clone Management Activity

An instance of clone management activity may be *clone-focused* or *system-focused*. A clone-focused activity deals with a narrow set of clones of a particular code segment of interest. On the contrary, a system-focused clone management activity aims to deal with a broad collection of clones in the entire code base, or particular portions of the system.

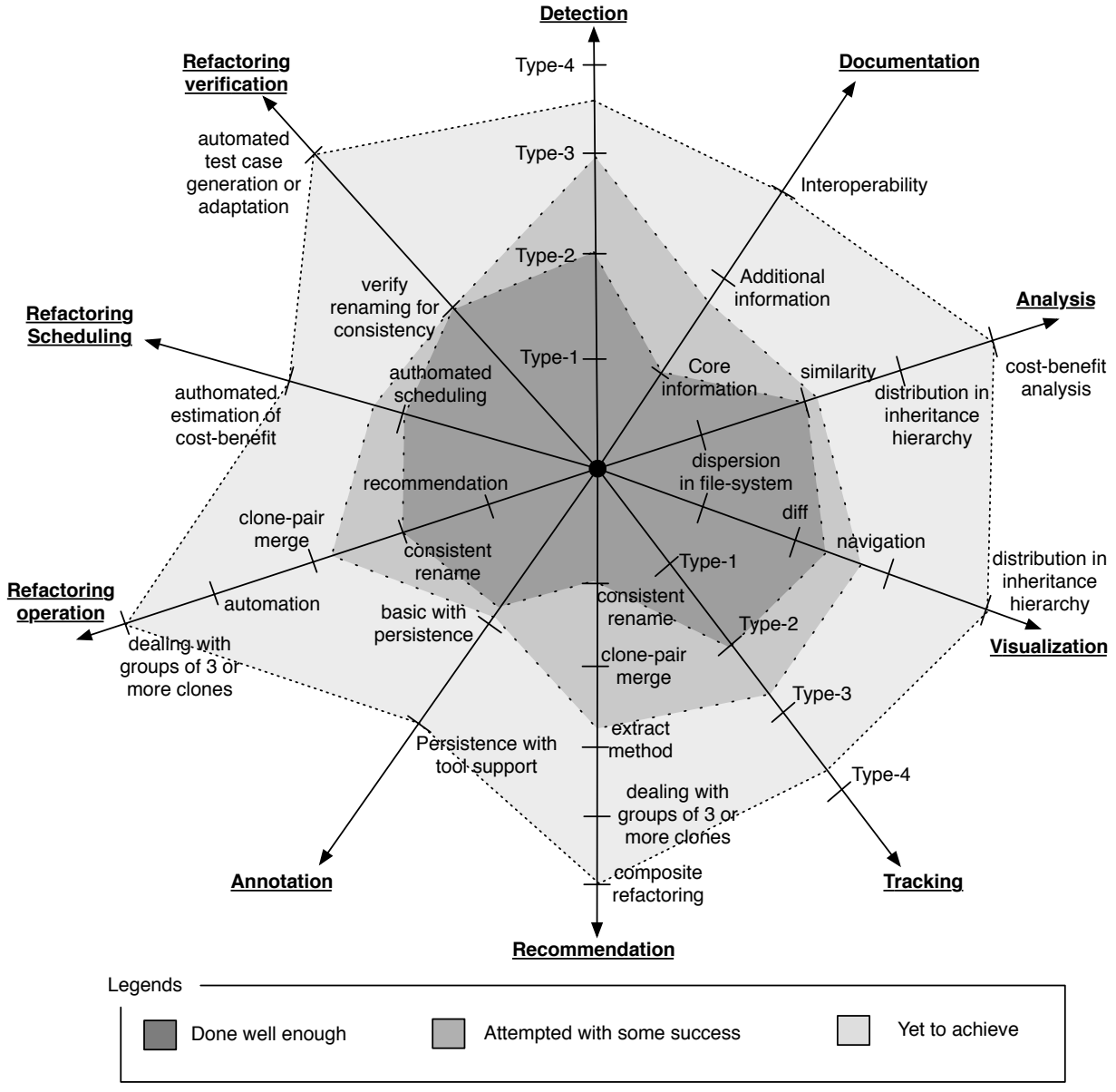


Figure 7.3: Achievements and scopes along different dimensions of clone management activities

7.5 Clones in Test Code

Until recently, code clone research remained focused on production code (i.e., source code for implementation of features according to requirement). However, for program verification and validation, test code is also produced in parallel to the production code, and both of those type of source code constitute a code base. This poses a number of open questions:

- Can it be beneficial to detect and manage clones in the test code as well?
- Is there any relationship between cloned code and their test code? Can the test code for the cloned production code be expected to be clones as well?
- Can clone detection in the test code prevent creation of clones in the production code, specially in test-driven development?

7.6 Industrial Adoption of Clone Management

Despite the active research on software clones and their impact on the development and maintenance of software system, management of code clones is still far from wide industrial adoption. A reason to this could be the unavailability of integrated tool support for versatile clone management. Or may be, the industry is not well aware of the problem. Typically, the organizations in the software industry operate on a limited budget and often in tight schedule, when the major objective becomes to be able to deliver the product to the client in time. Cloning becomes an advantage in such scenarios; the immediate effect of code cloning is rewarding, since cloning offers a reuse mechanism for low risk faster development process.

The possible negative impacts of code clones are generally deferred at later stage in the maintenance phase, for instance, until a fault in the system is discovered, which might have been caused due to inconsistent changes in the cloned code, or when it becomes difficult to manage the code base due its very large size. Many software organizations have separate business agreement with the client to provide maintenance support for their product over a defined period of time, for which the client makes additional payments.

Thus, from the business perspective, a software organization may have two phases of business: business on product delivery and product maintenance. Indeed, code cloning is beneficial at the first phase, although at the second phase there remains a possibility of increase in necessary maintenance effort due to code clones. Since, the software company may consider the maintenance phase a separate cash-inflow business, they might become apathetic in clone management, specially during the active development process. Therefore, the adoption of clone management in the industry largely depends on their realization in the fact that some initial effort in proactive clone management may significantly save later maintenance efforts.

What we as researchers need to show first is sufficient empirical evidence of real problems caused by clones. We have made good progress in recent years here. Then we need to pro- vide usable working solutions. We

need to demonstrate their benefits in real case studies. Because benefits are expected to show up only in the long run, we need long-term studies in realistic industrial settings. Such long-term industrial studies are difficult to conduct, however.

Despite these difficulties, we see signs that clone management is gathering momentum in industry. There are several clone detectors available as Eclipse plug-ins and only recently Microsoft introduced a clone management feature in Microsoft Visual Studio [57, 291]. There are several other industrial attempts as well [284, 296] including a recent Dagstuhl seminar on the topic [29].

7.7 Summary

In Table 7.4, we present a number of research questions relevant to clone management. In the right-most column, we present our opinion about the resolution status of each of the questions. We formulate this presentation having been inspired by the outcome of the brainstorming session of the Second International Workshop on Detection of Software Clones (IWDSC'03) in November 2003 [288]. On the basis of the outcome from the brainstorming session, in the second column from the right, we present the resolution status of each questions at that time (i.e., November 2003).

In Figure 7.3, we summarize the state of the art along the different dimensions code clone management and scopes for further improvements. Although software clone research has gained quite some maturity over the last decade, the majority of the work focused on the detection and analysis of code clones. Compared to those, clone management has earned recent interest due to its pragmatic importance. Notably three surveys [162, 222, 236] appeared in the literature, none of which focused on clone management, and thus a survey on clone management was a timely necessity. This paper presents a comprehensive survey on clone management and pin-points research achievements and scopes for further work towards a versatile clone management system.

At the fundamental level, the vagueness in the definition of clones at times causes difficulties in formalization, generalization, creation of benchmark data-set, as well as comparison of techniques and tools. A set of task oriented definitions or taxonomies can address these issues. Most of the integrated tools have limitations in detecting *Type-3* clones, and the detection of *Type-4* clones has still remained an open problem. Moreover, most of the research on software clones so far emphasized clone analysis at different levels of granularity. A variety of techniques for the visualization of clones and the evolution have been proposed. Surprisingly, while clone analysis points to the importance of considering inheritance hierarchy for extracting clone reengineering candidates, there is still not enough visualization support to analyze clones with respect to their existence in the inheritance hierarchy.

Research on clone management beyond detection has mostly been limited to devising techniques to identify clones that are easier to deal with. In ideal case, simple things should be made easy, while difficult things possible. The state of the art demands more research in semi-automated tool support for clone refactoring

Table 7.4: Research questions and current solvable status in clone research

#	Issues (asked as questions about knowledge)	Cat.	Solved? 2003	Solved? Now
1	What is the definition of a "clone" in source code and beyond?	definition	partly	maybe
2	What are the categories of clones?		no	partly
3	Do we understand clones in source code written in diverse programming paradigms?		no	partly
4	Clones at which granularity can be appropriate for what purpose?		no	partly
5	Can we define clones at a level of abstraction higher than the source code?		no	partly
6	Can we define clones in artifacts (e.g., models, requirements) other than source code?		no	partly
7	Do we have standard difference measures for near-miss (especially for *Type-3*) clones?		partly	partly
8	Can we perform basic clone detection on procedural systems?	detection	yes	yes
9	Can we describe the differences between the detected clone fragments?		no	partly
10	Do we have IDE-integrated clone detection tools?		maybe	partly
11	Can/should we detect clones at an abstraction higher than the source code?		no	maybe
12	Can we detect syntactically similar clones?		partly	yes
13	Can we detect semantic (*Type-4*) clones?		no	no
14	Does the size of the system affect the rate of clone occurrence?	phenomena	no	no
15	Does the programming language/paradigm affect the creation of clones?		no	partly
16	Do we know the reasons/psychology behind the creation for clones?		no	partly
17	Does the development process affect the creation of clones?		no	partly
18	Does XP produce fewer clones?		no	partly
19	Which clone detection technique is good for what purpose?	evaluation	no	no
20	How to evaluate and compare the tools and techniques for clone detection and management?		no	partly
21	Do we have benchmark to compare tools and techniques for clone detection and management?		partly	partly
22	Do we have context-sensitive categorization of good and bad clones?	implication	no	partly
23	What proportion of clones contribute in the creation and propagation of bugs?		no	partly
24	What is the stability of cloned code compared to non-cloned code?		no	no
25	Do we know what type of changes to the clones typically cause vulnerabilities?		no	partly
26	Can near-miss clones be traced through system evolution?	tracking	no	partly
27	Can we construct clone derivation histories?		no	no
28	Can we investigate the changes that a clone fragment experience during evolution?		no	partly
29	What can be done for effective clone management?	clone-based actions	no	partly
30	Do we understand the costs/benefits of clone detection and removal?		no	partly
31	Should we care about clones in test code and auto-generated code?		no	no
32	Should we aggressively remove clones?		no	partly
33	Which clones should be refactored, kept, or encouraged?		no	partly
34	How can we refactor a group of more than two near-miss clones?		no	no
35	Can we automate refactoring well?		no	no
36	Can we annotate clones with justification, should we must keep them?		no	partly
37	What kind of tool support can help to minimize/prevent the creation of clones?	prevention	no	partly
38	Can support from programming language/paradigm help to prevent clones?		no	no
39	Can customization of development process help in minimizing clone creation?		no	no
40	Do we know the business cases of clone management?	economics	no	partly
41	Can we *quantify* the end-user value of clone management?		no	no
42	Do we have adequate economic models of clone management costs?		no	no

and cost-benefit analysis of clone removal/refactoring. For integrated clone management, `JSync` [215] offers a relatively wider set of features compared to others. But, we see that the state of the art is still far from *integrated* tool support, and more to be done towards a versatile clone management system. Perhaps, due to the unavailability of such tools, there is not much developer-centric ethnographic studies on the patterns of clone management in practice, as well as on the usability and effectiveness of tool support. This chapter exposes such potential avenues where years of further research is necessary to create a better impact in the community.

Chapter 8

Conclusion

"Nothing is impossible. Not if you can imagine it.
That's what being a scientist is all about"
– Hubert Farnsworth

Duplication of source code is a common phenomenon in every software project in practice. Cloning indeed helps to speed-up development process specially when project schedule remains tight. Moreover, code cloning enables reuse of existing piece of code instead of risking to introduce program faults due to human errors while writing the code from the scratch. Code cloning can also facilitate decoupling program units for their independent evolution. Sometimes code clones can appear in the source code without the awareness of the developers especially in distributed development environments. At times it can even be difficult (if not impossible) to avoid clones due to limitations of programming language, underlying framework, APIs, or the programming problem at hand.

Despite the advantages of code cloning, code clones can have detrimental impacts on the maintenance of the software system. Redundant code can unnecessarily inflate the source code and cause increase in resource consumption, which can be crucial for embedded systems and hand-held devices. Copying a piece of code that contains any unknown bugs can propagate the bug to all those locations where the code is pasted. Upon copy-paste the programmer may accidentally forget to make a necessary change and thus can introduce program faults. During maintenance, the existence of code clones can impose an extra overhead to make sure that a change in a piece of code is followed by consistent updates to all its copies.

Due to the dual role of code clones, we must detect and manage clones effectively to minimize the negative impacts while obtaining the best out of code cloning. Earlier research in software clones primarily focused on the detection of *Type-1* and *Type-2* clones and investigation of cloning characterstics for those types of clones. Indeed, the detection of *Type3* clones and large scale analysis of near-miss (*Type-2* and *Type-3*) clones from the management perspective were largely ignored or limited. This thesis focuses on the detection and analysis of clones from the management perspective, and we deal with all three (*Type-1*, *Type-2*, and *Type-3*) types of clones.

The remaining of the chapter is organized as follows. In Section 8.1, we present a brief summary of the entire thesis. Section 8.2 points to the major contributions of this thesis. Section 8.3 discusses the limitations of this thesis. Finally, Section 8.4 concludes the chapter with future research directions.

8.1 Summary

Clone management can start by first identifying the clones in the source code. Most of the existing clone detectors are developed as tools separate from IDEs. However, for proactive clone management, clone detection must be possible from inside IDE. Those few tools that are integrated with IDEs have limitations in detecting *Type-3* clones. Therefore, we have developed an IDE-integrated clone detector that can effectively detect both exact (*Type-1*) and near-miss (*Type-2* and *Type3*) clones. Our novel approach for clone detection and the evaluation of our prototype tool implementation are presented in Chapter 3.

To understand the existence, evolution, and management implications of code clones in evolving software systems, we then carried out a number of in-depth empirical studies in large scale. In terms of clone-density, first we studied the proportion and evolutionary characteristics of clones in a large number of official releases of diverse open-source software systems. In a follow-up study, we examined the cloning patterns in two industrial web applications. Then, in a separate study, we looked deeper into the evolution of individual clone-group using the clone genealogy model. From these studies, as presented in Chapter 4, we revealed different patterns in the changes of code clone densities in the evolving software systems. While an increasing trend in the number of clones was found to be dominant, there were variations, irregularities, and influence of underlying programming languages or paradigms.

To further investigate the management implications of evolving clones, we carried out another in-depth study as presented in Chapter 5. Again using the clone genealogy model, in this study, we examined how the individual clone-groups change or disappear during their evolution over subsequent releases of the software systems. Our study shed light in the existing wisdom and contradictions in the community about clone management, and revealed the characteristics that make a certain type of code clones attractive candidates for removal in practice. We also found many long-lived clones that could have been removed by refactoring.

Indeed, clone removal by refactoring can help to minimize the number of clones in the source code. While refactoring involves efforts and risks, it also have impacts on the code quality. Moreover, refactorings may have conflicts and dependencies. Considering all the constraints, the cost-effective selection of an optimal set of candidate clones for refactoring is not a trivial task. This refactoring scheduling problem is addressed in Chapter 6. For the estimation of clone refactoring efforts, we introduced a new effort model, and then we presented our novel approach for conflict-aware optimal scheduling of code clone refactorings.

Although the work presented in Chapter 3 through Chapter 6 address the most important areas (i.e. clone detection, analysis, and refactoring) of clone management, based on an extensive literature survey, in Chapter 7, we have exposed scopes for improvements and further contributions in the area. In those avenues, long-term research efforts are needed from the community to produce a versatile clone management solution for use in the industry and research.

8.2 Contributions

While we have provided the details of our research in the earlier chapters, and presented an abridged summary of the entire thesis in Section 8.1, here we outline the major contributions of this thesis as follows:

Integrated Support for Focused Search for Near-miss Clones: We have developed an IDE-integrated tool for searching exact (*Type-1*) and *near-miss (Type-2 and Type-3)* clones as described in Chapter 3. In the detection of *Type-3* clones, we used a suffix-tree-based *k-difference hybrid algorithm*, which no one before us applied in this context. Our clone detection approach combines the advantages of the AST-based and text-based techniques, yet we avoid expensive string matching by incorporating Suffix-tree and hash-based fingerprinting. To enable proactive clone management, we implemented our clone search tool as a plug-in to the popular Eclipse IDE. Our clone search tool detects only the clones of a particular code fragment of the developer's choice, and thus avoids computations that otherwise would have been involved had it detected all the clones from the entire code base. This focused clone search is suitable for the developer's active interaction with the source code during the development or maintenance phase, and also saves the developer from too many reported clones detected from the entire code base that a traditional clone detector would report otherwise. This particular work of ours is published in ACM-SAC'2012 [310].

Understanding how Clones Are Managed in Practice: We have conducted exploratory studies that reveal useful insights into the characteristics of clones, their changes and removal during evolution. The first study as presented in Chapter 4 (Section 4.2) was the largest empirical study in on clone evolution at the time it was published [313]. The follow-up empirical studies also appeared in peer-reviewed refereed conferences [211, 248]. Our empirical study (Chapter 5) on the evolution and change patterns of individual clones informs clone management by deriving useful insights into what type of changes the individual clone fragments experience during their evolution. At the time the work was published in ACM-SAC'2013 [314], this was the first genealogy-based study on clone evolution that included *Type-3* clones. An extension of the work was also invited and published in ACM Applied Computing Review [315].

Our empirical studies on clone evolution generally remains unique from most other similar studies in two ways. First, instead of studying versions (i.e., periodic snapshots or commit transactions) of the software systems, we studied the official releases and thus captured stable changes in the clones during their evolution over a longer period of time. Second, in our studies, we included subject systems of diverse sizes from a variety of application domains and written in different programming languages.

Clone Refactoring Effort Model and Scheduler: We have developed a parameterized effort model and a refactoring scheduler for cost-effective scheduling of code clone refactorings (Chapter 6). Ours is the first effort model for the estimation of efforts needed to refactor clones in object-oriented source code.

The proof of concept of the effort model appeared in ICPC'2011 [307]. Our refactoring scheduler adopts a constraint programming approach that no one reported to have applied before in this context. The work is published in SCAM'2011 [308]. As one of the best papers, an extended version of our work was invited and published in an special issue of IET Software Journal [312].

Based on our in-depth analysis of the existing literature and our experience, we have also identified (Chapter 7) scopes for further research and development towards a versatile clone management system. A part of the chapter appeared as a Vision Keynote in the IEEE CSMR-18/WCRE-21 Software Evolution Week (SEW'14) [244].

8.3 Limitations

The limitations of the individual parts of this dissertation research are pointed out in the respective chapters. Here, we further discuss the limitations of the work as a whole.

As mentioned in Section 1.1.2, the scope of this thesis is limited to *Type-1*, *Type-2*, and *Type-3* clones in the source code only. We have not dealt with semantic (*Type-4*) clones or clones in other software artifacts such as models and requirement documents. Moreover, our work has remained limited to code clones at the granularity of functions/methods and syntactic blocks only, although much of this work can easily be made applicable for clones in other levels of granularities.

In the implementation of our clone search tool, we adopted a novel hybrid algorithm that exploits the advantages of both AST-based and textual comparison based techniques. Nevertheless, like any other state-of-the-art clone detectors, our clone search tool also has its strengths and weaknesses. The technique for clone detection used in our tool is sensitive to reordering of statements (e.g., variable declaration statements). AST-based tree matching techniques could have been considered as an alternative to avoid such sensitivity to statement ordering. But tree matching can be computationally expensive. Moreover, minor differences in the source code can result in large variations in the corresponding abstract syntax trees. Thus, clone detection using AST-based tree matching suffers from low recall [13, 34, 35]. The "code preprocessing" part of our clone detection technique is language-dependent, and the current implementation of our Eclipse plug-in supports source code written in Java only. However, we also have a TXL-based command line version of our tool that can deal with any programming language, if the TXL grammar for the language can be provided. Although we have developed our clone search tool as a plug-in to the popular Eclipse IDE, the tool is still in its prototype stage. There are scopes for investing more programming efforts for further fine tuning and inclusion of additional features to make it an industry standard finished product.

In general, the empirical studies presented in this thesis might have suffered from the limitations of the underlying tools and algorithms used for data collection and analysis. Although in the studies, we included a large number of subject systems picked from diverse application domains and written in different programming languages, those subject systems were mostly open-source except for the two web applications

used in the study presented in Section 4.4. This can be regarded as a limitation of this thesis. One may question, to what extent the findings and clone management implications derived from those studies can be expected to hold for industrial software systems. Although we made immense attempt, it was not possible to get access to the source code of industrial (i.e., closed-source) software systems due to the proprietary and security concerns from those organizations' side.

The participants of the user-centric studies are collected using our acquantance and relationships with them. Thus, there is a threat of bias that the participants might have deliberately or unintentionally performed in favour of our tool or technique under evaluation. In addition, the manual verifications at all the necessary scenarios are carried out by ourselves, which is also a threat to a possible bias. However, at every manual verification stage, we strived to make neutral judgements and we also advised the participants of the user studies to keep their performance unaffected from any possible bias.

8.4 Future Research Directions

Clone management is a wide area of research, which has recently drawn interested in the community. A fairly rich clone management tool with necessary features is still not available and we believe, years of research and development is needed towards this goal. We see two streams of future work. The first stream encompasses further research to device tools and techniques for better clone management. In Chapter 7, we have identified clone management activities, workflow, strategies, and pointed to avenues for further development with respect to the state of the art. Other than those, our clone search tool can be improved by incorporating different clone detection techniques and thus allowing the users to customize by choosing detection techniques and tuning parameters according to their need. Our work can also be extended by integrating our clone search tool with the refactoring scheduler, features for clone annotation, metaphor-based clone visualization (with respect to the file system hierarchy as well as inheritance hierarchy), and support for semi-automated clone refactoring beyond those refactoring supports that are already available in a typical IDE. Integrated tool support can be more effective in supporting clone management [244, 270, 290].

The second stream of future work includes the application of clone detection and management techniques in other areas and exploration of additional business values of clone management. The techniques for clone detection, in particular, clone tracking and annotation can be applied for code provenance and origin analysis. Clone detection in source code and binary executables can help in malware detection. The techniques for detecting near-miss code clones can be applied for plagiarism detection, copy-right infringement, software version merging, product line engineering, feature extraction, aspect mining, and resource allocation in job schedulers. Tracking of code clones and their changes can contribute in recommending program co-changes. Near-miss clone detection techniques have the prospect of being used in example-based code recommenders to offer code completion in a scale larger (e.g., method or block completion) than simple API suggestions that are generally available from IDEs. More research and prototyping are required to explore all these possibilities.

References

[1] S. Abd-El-Hafiz. A metrics-based data mining approach for software clone detection. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, pages 35–41, 2012.

[2] E. Adar. GUESS: a language and interface for graph exploration. In *Proceedings of the International Conference on Computer Human Interactions (CHI)*, pages 791–800. ACM, 2006.

[3] E. Adar and M. Kim. SoftGUESS: Visualization and exploration of code clones in context. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 762 –766, 2007.

[4] D. Advani, Y. Hassoun, and S. Counsell. Understanding the complexity of refactoring in software systems: a tool-based approach. *Int. J. Gen. Sys.*, 35(3):329–346, 2006.

[5] F. Al-omari, I. Keivanloo, C. Roy, and J. Rilling. Detecting clones across microsoft .net programming languages. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 405–414, 2012.

[6] D. Anderson, D. Sweeney, and T. Williams. *Statistics for Business and Economics*. Thomson Higher Education, 10th edition, 2009.

[7] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 273–280, 2001.

[8] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755 – 765, 2002.

[9] M. Asaduzzaman. Visualization and analysis of software clones. M.Sc. thesis, University of Saskatchewan, Canada, 2011.

[10] M. Asaduzzaman, C. Roy, and K. Schneider. VisCad: flexible code clone analysis support for NiCad. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 77–78, 2011.

[11] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 81 –90, 2007.

[12] M. Bahtiyar. JClone : Syntax tree based clone detection for java. Master's thesis, Linnaeus University, 2010.

[13] J. Bailey and E. Burd. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 36 – 43, 2002.

[14] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

[15] B. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 86 –95, 1995.

[16] B. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28 – 42, 1996.

[17] B. Baker and U. Manber. Deducing similarities in java sources from bytecodes. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATEC)*, pages 15–15, 1998.

[18] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 24–33, 2007.

[19] M. Balazinska, E. Merlo, M. Dagenais, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 98–107, 2000.

[20] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the IEEE Symposium on Software Metrics (METRICS)*, pages 292 –303, 1999.

[21] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 326–336, 1999.

[22] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. on Softw. Engg.*, 28(1):4–17, 2002.

[23] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 273 –282, 2011.

[24] R. Barták. Constraint programming: In pursuit of the holy grail. In *Proceedings of the 8th Annual Conference of Doctoral Students (WDS), invited lecture*, pages 1–10, 1999.

[25] H. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes*, 30:156–165, 2005.

[26] H. Basit and S. Jarzabek. *Towards Structural Clones: Analysis and semi-automated detection of design-level similarities in software*. VDM Verlag Dr. Müller, 2010.

[27] H. Basit, S. Puglisi, W. Smyth, A. Turpin, and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the Joint Meeting on European software engineering conference (ESEC) and the ACM SIGSOFT symposium on the foundations of software engineering (FSE): companion papers*, pages 513–516, 2007.

[28] H. Basit, D. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 109–114, 2005.

[29] I. Baxter, M. Conradt, J. Cordy, and R. Koschke. Software clone management towards industrial application (dagstuhl seminar 12071). *DagStuhl Report*, 2(2):21–57, 2012.

[30] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377, 1998.

[31] S. Bazrafshan. Evolution of near-miss clones. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 74–83, 2012.

[32] S. Bazrafshan, R. Koschke, and N. Göde. Approximate code search in program histories. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 109–118, 2011.

[33] M. Beard. Extending bug localization using information retrieval and code clone location techniques. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 425 –428, 2011.

[34] S. Bellon. Vergleich von techniken zur erkennung duplizierten quellcodes. Diploma thesis, Universität Stuttgart, 2002.

[35] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. on Softw. Engg.*, 33(9):577–591, 2007.

[36] 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. *Science of Computer Programming*, pages 1–17, 2010.

[37] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A novel approach to optimize clone refactoring activity. In *Proceedings on the annual conference on Genetic and evolutionary computation (GECCO)*, pages 1885–1892, 2006.

[38] G. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[39] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 77 –86, 2010.

[40] M. Bruntink. Aspect mining using clone class metrics. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE)*, pages 1–6, 2004.

[41] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31:804–818, 2005.

[42] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5:46–54, 2007.

[43] P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proceedings of the International Workshop on Software Clones (IWSC)*, 2009.

[44] D. Cai and M. Kim. An empirical study of long-lived code clones. In *Proceedings of the International conference on Fundamental approaches to software engineering (FASE): part of the joint European conferences on theory and practice of software (ETAPS)*, pages 432–446, 2011.

[45] D. Chatterji, J. Carver, and N. Kraft. Claims and beliefs about code clones: Do we agree as a community? a survey. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 15–21, 2012.

[46] D. Chatterji, J. Carver, B. Massengil, J. Oslin, and N. Kraft. Measuring the efficacy of code clone information in a bug localization task: An empirical study. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 20–29, 2011.

[47] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Trans. Softw. Eng.*, 25(5):476–493, 1994.

[48] A. Chiu and D. Hirtle. Beyond clone detection. CS846 Course Project Report, University of Waterloo, 2007.

[49] E. Choi, N. Yoshida, T. Ishio, K. Inoue, and T. Sano. Extracting code clones for refactoring using combinations of clone metrics. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 7–13. ACM, 2011.

[50] J. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the International International Workshop on Program Comprehension (IWPC)*, pages 196–206, 2003.

[51] J. Cordy. Source transformation, analysis and generation in TXL. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 1–11, 2006.

[52] J. Cordy. Live scatterplots. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 79–80. ACM, 2011.

[53] J. Cordy, T. Dean, and N. Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 1–12, 2004.

[54] J. Cordy and C. Roy. The nicad clone detector. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 219–220, 2011.

[55] J. Cordy and C. Roy. Tuning research tools for scalability and performance: the NiCad experience. *Science of Computer Programming*, 79(1):158–171, 2014.

[56] A. Cuomo, A. Santone, and U. Villano. A novel approach based on formal methods for clone detection. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 8–14, 2012.

[57] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. XIAO: tuning code clones at hands of engineers in practice. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 369–378, 2012.

[58] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3/4):219 – 236, 1995.

[59] Y. Davidor. Epistasis variance: A viewpoint on GA-hardness. In *Proceedings of the Foundations of Genetic Algorithms (FOGA)*, pages 23–35, 1990.

[60] I. Davis and M. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 77–78. ACM, 2010.

[61] I. Davis and M. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 242 –246, 2010.

[62] M. de Wit. *Managing Clones Using Dynamic Change Tracking and Resolution.* M.Sc. thesis, Delft University of Technology, 2008.

[63] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 57–64. ACM, 2010.

[64] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 603–612. ACM, 2008.

[65] R. DeLine, G. Venolia, and K. Rowan. Software development with code maps. *ACM Commun.*, 53(8):48–54, 2010.

[66] G. Di Lucca, M. Di Penta, and A. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, pages 481 – 486, 2002.

[67] E. Duala-Ekoko and M. Robillard. Tracking code clones in evolving software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 158–167, 2007.

[68] E. Duala-Ekoko and M. Robillard. CloneTracker: tool support for code clone management. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 843–846, 2008.

[69] E. Duala-Ekoko and M. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Trans. Softw. Eng. Methodol.*, 20:3:1–3:31, 2010.

[70] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 109 –118, 1999.

[71] S. Ducasse, M. Rieger, G. Golomingi, and B. Bym. Tool support for refactoring duplicated OO code. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Workshop Reader*, number 1743 in LNCS, pages 2–6. Springer-Verlag, 1999.

[72] A. Eiben, P. Raue, and Z. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In *Proceedings of the First IEEE Conference on Evolutionary Computation (CEC), IEEE World Congress on Computational Intelligence*, pages 542–547, 1994.

[73] W. Evans, C. Fraser, and F. Ma. Clone detection via structural abstraction. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 150–159. IEEE Computer Society, 2007.

[74] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Softw. Engg.*, 13:601–643, 2008.

[75] R. Fanta and V. Rajlich. Removing clones from the code. *Journal of Software Maintenance: Research and Practice*, 11(4):223–243, 1999.

[76] M. Farhadi. Assembly code clone detection for malware binaries. Master's thesis, Concordia University, Canada, 2013.

[77] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9:319–349, 1987.

[78] F. Fontana, M. Zanoni, A. Ranchetti, and D. Ranchetti. Software clone detection and refactoring. *ISRN Software Engineering*, 2013:1–8, 2013.

[79] C. Forbes, I. Keivanloo, and J. Rilling. Doppel-code: A clone visualization tool for prioritizing global and local clone. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, pages 366–367, 2012.

[80] M. Fowler. Refactoring catalog, http://refactoring.com/catalog/, last access: Dec 2012.

[81] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[82] Y. Fukushima, R. Kula, S. Kawaguchi, K. Fushida, M. Nagura, and H. Iida. Code clone graph metrics for detecting diffused code clones. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 373 –380, 2009.

[83] M. Funaro, D. Braga, A. Campi, and C. Ghezzi. Combining syntactic and textual approach in clone detection. In *Proceedings of the International Workshop on Software Clones (IWSC)*, 2010.

[84] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 321–330, 2008.

[85] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Proceedings of the international conference on Fundamental Approaches to Software Engineering (FASE)*, pages 411–425, 2006.

[86] S. Giesecke. Generic modelling of code clones. In *Proceedings of the Dagstuhl Seminar Report on Duplication, Redundancy, and Similarity in Software (DRSS)*, pages 1–23, 2007.

[87] S. Giesecke. Dupman - eclipse duplication management framework, http://sourceforge.net/projects/dupman/, last access: Dec 2011.

[88] N. Göde. Incremental clone detection. Diploma thesis, University of Bremen, 2008.

[89] N. Göde. Evolution of type-1 clones. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 77 –86, 2009.

[90] N. Göde. Clone removal: Fact or fiction? In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 33–40, 2010.

[91] N. Göde and J. Harder. Clone stability. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 65 –74, 2011.

[92] N. Göde and R. Koschke. Incremental clone detection. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 219–228, 2009.

[93] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–28, 2010.

[94] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320, 2011.

[95] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 131–142, 2000.

[96] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. on Softw. Engg.*, 31(2):166–181, 2005.

[97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer and Computational Biology. Cambridge University Press, 1st edition, 1997.

[98] J. Harder and N. Göde. Modeling clone evolution. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 17–21, 2009.

[99] J. Harder and N. Göde. Quo vadis, clone management? In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 85–86, 2010.

[100] J. Harder and N. Göde. Efficiently handling clone data: Rcf and cyclone. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 81–82, 2011.

[101] B. Hauptmann, V. Bauer, and M. Junker. Using edge bundle views for clone visualization. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 86–87, 2012.

[102] J. Helfman. Dotplot patterns: a literal look at pattern languages. *Theor. Pract. Object Syst.*, 2:31–41, 1996.

[103] A. Hemel, K. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 63–72, 2011.

[104] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proceedings of the IASTED (International Association of Science and Technology for Development) International Conference on Software Engineering and Applications (SEA)*, pages 222–229, 2004.

[105] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. *Product Focused Software Process Improvement (PROFES)*, (LNCS 3009):220–233, 2004.

[106] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Inf. Softw. Technol.*, 49:985–998, 2007.

[107] Y. Higo and S. Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 315 –316, 2009.

[108] Y. Higo, K. Tanaka, and S. Kusumoto. Toward identifying inter-project clone sets for building useful libraries. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 87–88, 2010.

[109] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *Product Focused Software Process Improvement (PROFES)*, pages 185–197, 2002.

[110] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous modification support based on code clone analysis. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 262–269, 2007.

[111] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental code clone detection: A PDG-based approach. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 3 –12, 2011.

[112] K. Hotta, Y. Higo, and S. Kusumoto. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 53–62, 2012.

[113] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 73–82, 2010.

[114] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 238–242, 2009.

[115] D. Hou, F. Jacob, and P. Jablonski. Exploring the design space of proactive tool support for copy-and-paste programming. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 188–202, 2009.

[116] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1 –9, 2010.

[117] P. Jablonski and D. Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the OOPSLA (annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications) workshop on Eclipse Technology Exchange (ETX)*, pages 16–20, 2007.

[118] P. Jablonski and D. Hou. Renaming parts of identifiers consistently within code clones. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 38–39, 2010.

[119] F. Jacob, D. Hou, and P. Jablonski. Actively comparing clones inside the code editor. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 9–16, 2010.

[120] K. Jalbert and J. Bradbury. Using clone detection to identify bugs in concurrent software. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1 –5, 2010.

[121] S. Jarzabek and S. Li. Unifying clones with a generative programming technique: a case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(4):267–292, 2006.

[122] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 96–105, 2007.

[123] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the joint meeting of the European software engineering conference (ESEC) and the ACM SIGSOFT symposium on The foundations of software engineering (FSE)*, pages 55–64, 2007.

[124] Z. Jiang and A. Hassan. A framework for studying clones in large software systems. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 203 –212, 2007.

[125] Z. Jiang, A. Hassan, and R. Holt. Visualizing clone cohesion and coupling. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 467 –476, 2006.

[126] J. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 171–183, 1993.

[127] J. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 120 –126, 1994.

[128] J. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 32–41, 1994.

[129] J. Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 16–25, 1996.

[130] E. Juergens. Research in cloning beyond code: a first roadmap. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 67–68, 2011.

[131] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *Proceedings of the International Conference on Software Engineering (ICSE)*, volume 2, pages 79 –88, 2010.

[132] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - a workbench for clone detection research. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 603–606, 2009.

[133] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 485–495, 2009.

[134] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *Workshopband SE Konferenz*, pages 443–446, 2008.

[135] N. Juillerat and B. Hirsbrunner. An algorithm for detecting and removing clones in java code. In *Proceedings of the 3rd Workshop on Software Evolution through Transformation (SeTra)*, pages 63–74, 2006.

[136] T. Kamiya. Agec: An execution-semantic clone detection tool. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 227–229, 2013.

[137] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[138] C. Kapser. *Toward an Understanding of Software Code Cloning as a Development Practice*. PhD thesis, University of Waterloo, Canada, 2009.

[139] C. Kapser and M. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 85–94, 2004.

[140] C. Kapser and M. Godfrey. Improved tool support for the investigation of duplication in software. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 305–314, 2005.

[141] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 19–28, 2006.

[142] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008.

[143] C. Kapser, J. Harder, and I. Baxter. A common conceptual model for clone detection results. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 72–73, 2012.

[144] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: A case study. *J. Softw. Maint. Evol.*, 18:61–82, 2006.

[145] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A tool for automatic code clone detection in the IDE. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 313–314, 2009.

[146] I. Keivanloo, C. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 36–42, 2012.

[147] I. Keivanloo, C. Roy, and J. Rilling. SeByte: A semantic clone detection tool for intermediate languages. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 247–249, 2012.

[148] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. In A. Rashid and M. Aksit, editors, *Transactions on aspect-oriented software development IV*, pages 143–162. Springer-Verlag, 2007.

[149] J. Kerievsky. *Refactoring to Patterns.* Addison Wesley, 2004.

[150] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, 1997.

[151] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 301–310, 2011.

[152] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*, pages 83–92, 2004.

[153] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 1–5, 2005.

[154] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[155] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. Saranya. CloneManager: A tool for detection of type1 and type2 code clones. *Information Processing and Management*, 70:568–570, 2010.

[156] E. Kodhai, V. Vijayakumar, G. Balabaskaran, T. Stalin, and B. Kanagaraj. Method level detection and removal of code clones in C and Java programs using refactoring. *International Journal of Computer Communication and Information System (IJCCIS)*, 2(1):93–95, 2010.

[157] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS)*, pages 40–56, 2001.

[158] G. Koni-N'Sapu. A scenario based approach for refactoring duplicated code in object oriented systems. Diploma thesis, University of Bern, 2001.

[159] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 44 –54, 1997.

[160] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern matching for design concept localization. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 96 –103, 1995.

[161] K. Kontogiannis, R. Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Softw. Eng.*, 3(1/2):77–108, 1996.

[162] R. Koschke. Survey of research on software clones. In *Proceedings of the Dagstuhl Seminar Report on Duplication, Redundancy, and Similarity in Software (DRSS)*, pages 1–24, 2006.

[163] R. Koschke. Frontiers of software clone management. In *Frontiers of Software Maintenance (FoSM)*, pages 119 –128, 2008.

[164] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 309–318, 2012.

[165] R. Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *J. of Software: Evolution and Process*, pages 1–23, 2013.

[166] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 253–262, 2006.

[167] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 301 –309, 2001.

[168] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 170 –178, 2007.

[169] J. Krinke. Is cloned code more stable than non-cloned code? *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, 0:57–66, 2008.

[170] Jens Krinke. Is cloned code older than non-cloned code? In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 28–33, 2011.

[171] D. Krutz and E. Shihab. CCCD: Concolic code clone detection. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 489–490, 2013.

[172] B. Lague, D. Proulx, J. Mayrand, E. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 314–321, 1997.

[173] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, 1989.

[174] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–286, 2003.

[175] M. Lee, J. Roh, S. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the ACM SIGSOFT international symposium on Foundations of software engineering (FSE)*, pages 167–176, 2010.

[176] S. Lee, G. Bae, H. Chae, D. Bae, and Y. Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Software - Practice and Experience*, 41(5):521–550, 2010.

[177] S. Lee and I. Jeong. SDD: high performance code clone detection system for large scale source code. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 140–141, 2005.

[178] A. Leitão. Detection of redundant code using r2d2. *Software Quality Control*, 12:361–382, 2004.

[179] H. Li and S. Thompson. Similar code detection and elimination for Erlang programs. *Practical Aspects of Declarative Languages*, 5937:104–118, 2010.

[180] H. Li and S. Thompson. Incremental clone detection and elimination for erlang programs. In *Proceedings of the International conference on Fundamental approaches to software engineering (FASE): part of the joint European conferences on theory and practice of software (ETAPS)*, pages 356–370, 2011.

[181] J. Li and M. Ernst. CBCD: Cloned buggy code detector. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 310–320, 2012.

[182] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176 – 192, 2006.

[183] H. Liu, G. Li, Z. Ma, , and W. Shao. Conflict-aware schedule of software refactorings. *IET Software*, 2(5):446–460, 2008.

[184] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 269 –276, 2006.

[185] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, page 22, 2007.

[186] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 106–115, 2007.

[187] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 227–236, 2008.

[188] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 65–72, 2010.

[189] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 18 –21, 2007.

[190] G. Lucca, M. Di Penta, and A. Fasolino. Clone analysis in the web era: An approach to identify cloned web. In *Proceedings of the IEEE Workshop on Empirical Studies on Software Maintenance (WESS)*, pages 107–113, 2001.

[191] A. Lucia, R. Francese, G. Scanniello, and G. Tortora. Reengineering web applications based on cloned pattern analysis. In *Proceedings of the International International Workshop on Program Comprehension (IWPC)*, pages 132 – 141, 2004.

[192] A. Lucia, R. Francese, G. Scanniello, and G. Tortora. Understanding cloned patterns in web applications. In *Proceedings of the International International Workshop on Program Comprehension (IWPC)*, pages 333–336, 2005.

[193] A. Lucia, G. Scanniello, and G. Tortora. Identifying clones in dynamic web sites using similarity thresholds. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*, pages 391–396, 2004.

[194] A. Mao, J. Cordy, and T. Dean. Automated conversion of table-based websites to structured stylesheets using table recognition and clone detection. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 12–26, 2007.

[195] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 107 – 114, 2001.

[196] J. Mayrand, B. Lague, and J. Hudepohl. Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. In *Proceedings of the IEEE Workshop on Empirical Studies on Software Maintenance (WESS)*, 1996.

[197] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 244 –253, 1996.

[198] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *J. Softw. and Syst. Modeling*, 6(3):269–285, 2007.

[199] R. Miller. *Lightweight Structured Text Processing.* PhD thesis, Carnegie Mellon University, 2001.

[200] R. Miller and B. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track of USENIX Annual Technical Conference (USENIX)*, pages 161–174, 2001.

[201] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the ACM Symposium on Applied Computing (SAC), SE Track*, pages 1227–1234, 2012.

[202] M. Mondal, C. Roy, and K. Schneider. An empirical study on clone stability. *ACM Applied Computing Review*, 12(3):20–36, 2012.

[203] M. Mondal, C. Roy, and K. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Proceeding of the IEEE CSMR-18/WCRE-21 Software Evolution Week (SEW'14)*, pages 114–123, 2014.

[204] M. Mondal, C. Roy, and K. Schneider. A fine-grained analysis on the evolutionary coupling of cloned code. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014)*, pages 1–10 (to appear), 2014.

[205] M. Mondal, C. Roy, and K. Schneider. An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study. *Science of Computer Programming*, pages 1–48, 2014.

[206] M. Mondal, C. Roy, and K. Schneider. Late propagation in near-miss clones: An empirical study. In *Proceedings of the 8th International Workshop on Software Clones (IWSC 2014)*, pages 1–15 (to appear), 2014.

[207] M. Mondal, C. Roy, and K. Schneider. Prediction and ranking of co-change candidates for clones. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*, pages 32–41, 2014.

[208] D. Mongomery, E. Peck, and G. Vinning. *Introduction to Linear Regression Analysis.* John Wiley & Sons, Inc., 5th edition, 2012.

[209] D. Montgomery, C. Jennings, and M. Kulahci. *Introduction to Time Series Analysis and Forecasting.* Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 2008.

[210] B. Muddu, A. Asadullah, and V. Bhat. CPDP - a robust technique for plagiarism detection in source code. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 39–45, 2013.

[211] T. Muhammad, M. Zibran, Y. Yamamoto, and C. Roy. Near-miss clone patterns in web applications: An empirical study with industrial systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–6 (to appear), 2013.

[212] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 93–102, 2013.

[213] E. Murphy-Hill, P. Quitslund, and A. Black. Removing duplication from java.io: a case study using traits. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 282–291, 2005.

[214] S. Nasehi, G. Sotudeh, and M. Gomrokchi. Source code enhancement using reduction of duplicated code. In *Proceedings of the IASTED International Multi-Conference on Software Engineering*, pages 192–197. ACTA Press, 2007.

[215] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone management for evolving software. *IEEE Trans. on Softw. Engg.*, 1(1):1–19, 2011.

[216] T. Nguyen, H. Nguyen, J. Al-Kofahi, N. Pham, and T. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 491 –494, 2009.

[217] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Cleman: Comprehensive clone group evolution management. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 451–454, 2008.

[218] T. Nguyen, H. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone-aware configuration management. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 123–134, 2009.

[219] M. Obitko. Introduction to genetic algorithms, http://www.obitko.com/tutorials/genetic-algorithms/recommendations.php, last access: January 2014.

[220] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.: Res. Pract.*, 20:345–364, 2008.

[221] A. Orso, N. Shi, and M. Harrold. Scaling regression testing to large software systems. *SIGSOFT Softw. Eng. Notes*, 29(6):241–251, 2004.

[222] J. Pate, R. Tairas, and N. Kraft. Clone evolution: a systematic review. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 1–23, 2011.

[223] J. Pérez, Y. Crespo, B. Hoffmann, and T. Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *Int. J. Softw. Tools Tech. Transfer*, pages 183–199, 2010.

[224] N. Pham, H. Nguyen, T. Nguyen, J. Al-Kofahi, and T. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 276–286, 2009.

[225] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with JPlag. *Journal OF Universal Computer Science*, 8:1016–1038, 2000.

[226] M. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[227] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 72–81, 2010.

[228] M. Rahman, A. Aryani, C. Roy, and F. Perin. On the relationships between domain-based coupling and code clones: An exploratory study. In *Proceedings of the International Conference on Software Engineering (ICSE), NIER Track*, pages 1265–1268, 2013.

[229] D. Rajapakse. An investigation of cloning in web applications. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 924–925, 2005.

[230] D. Rajapakse. *Exploiting similarity patterns in web applications for enhanced genericity and maintainability.* PhD thesis, School of Computing, National University of Singapore, 2006.

[231] D. Rajapakse and S. Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 116–126, 2007.

[232] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Infor. and Soft. Tech.*, 55(7):1165–1199, 2013.

[233] M. Rieger. *Effective Clone Detection Without Language Barriers*. PhD thesis, Institut fuï Informatik und angewandte Mathematik, Germany, 2005.

[234] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 100–109, 2004.

[235] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

[236] C. Roy and J. Cordy. A survey on software clone detection research. Tech Report TR 2007-541, School of Computing, Queens University, Canada, 2007.

[237] C. Roy and J. Cordy. An empirical study of function clones in open source software systems. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 81–90, 2008.

[238] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 172–181, 2008.

[239] C. Roy and J. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 153–162, 2008.

[240] C. Roy and J. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 157–166, 2009.

[241] C. Roy and J. Cordy. Are scripting languages really different? In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 17– 24, 2010.

[242] C. Roy and J. Cordy. Near-miss function clones in open source software: an empirical study. *J. of Softw. Maintenance and Evolution: Research and Practice*, 22(3):165–189, 2010.

[243] C. Roy, J. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, 2009.

[244] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future. In *CSMR-18/WCRE-21 Software Evolution Week (SEW'14)*, pages 18–33, 2014.

[245] V. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 336 – 339, 2004.

[246] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the 18th international symposium on Software testing and analysis (ISSTA)*, pages 117–128, 2009.

[247] R. Saha. Detection and analysis of near-miss clone genealogies. M.Sc. thesis, University of Saskatchewan, 2011.

[248] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 87–96, 2010.

[249] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 293 –302, 2011.

[250] R. Saha, C. Roy, and K. Schneider. Visualizing the evolution of code clones. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 71–72, 2011.

[251] R. Saha, C. Roy, K. Schneider, and D. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 139–148, 2013.

[252] H. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of OO design quality and its automation? In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 154–162, 2000.

[253] H. Sajnani and C. Lopes. A parallel and efficient approach to large scale clone detection. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 46–52, 2013.

[254] H. Sajnani, J. Ossher, and C. Lopes. Parallel code clone detection using mapreduce. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 261–262, 2012.

[255] A. Santone. Clone detection through process algebras and java bytecode. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 73–74, 2011.

[256] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 327–339. Springer Berlin / Heidelberg, 2003.

[257] S. Schulze and M. Kuhlemann. Advanced analysis for code clone removal. In *Proceedings of the Workshop on Software Reengineering (WSR)*, pages 1–2, 2009.

[258] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a refactoring guideline using code clone classification. In *Proceedings of the 2nd Workshop on Refactoring Tools (WRT)*, pages 6:1–6:4, 2008.

[259] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 13 –21, 2010.

[260] D. Shawky and A. Ali. Modeling clones evolution in open source systems through chaos theory. In *Proceedings of the International Conference on Software Technology and Engineering (ICSTE)*, pages V1–159 – V1–164, 2010.

[261] J. Sillito, G. Murphy, and K. Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, 2004.

[262] F. Simon, F. Steinbrucker, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 30–38, 2001.

[263] H. Störrle. Towards clone detection in UML domain models. In *Proceedings of the 4th European Conference on Software Architecture (ECSA): Companion Volume*, pages 285–293, 2010.

[264] J. Svajlenko, C. Roy, and J. Cordy. A mutation analysis based benchmarking framework for clone detectors. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 8–9, 2013.

[265] N. Synytskyy and J. Cordy. Robust multilingual parsing using island grammars. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, pages 266–278, 2003.

[266] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 183–192, 2003.

[267] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE 44)*, pages 679–684, 2006.

[268] R. Tairas and J. Gray. Get to know your clones with CeDAR. In *Proceedings of the annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 817–818, 2009.

[269] R. Tairas and J. Gray. Sub-clone refactoring in open source software artifacts. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 2373–2374, 2010.

[270] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *J. of Info. and Softw. Tech.*, 54(12):1297–1307, 2012.

[271] R. Tairas, J. Gray, and I. Baxter. Visualization of clone detection results. In *Proceedings of the OOPSLA (annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications) workshop on Eclipse Technology Exchange (ETX)*, pages 50–54, 2006.

[272] R. Tairas, J. Gray, and I. Baxter. Visualizing clone detection results. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 549–550, 2007.

[273] M. Thomsen and F. Henglein. Clone detection using rolling hashing, suffix trees and dagification: A case study. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 22–28, 2012.

[274] S Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15:1–34, 2010.

[275] M. Tokunaga, N. Yoshida, K. Yoshioka, M. Matsushita, and K. Inoue. Towards collection of refactoring patterns based on code clone classification. In *Proceedings of the Asian Conference on Pattern Languages of Programs (AsianPLoP)*, pages 86–91, 2011.

[276] W. Toomey. Ctcompare: Code clone detection using hashed token sequences. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 92–93, 2012.

[277] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC)*, pages 173–180, 2004.

[278] R. Torres. Source code mining for code duplication refactorings with formal concept analysis. M.Sc. thesis, Vrije Universiteit Brussel, Belgium, 2004.

[279] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 13 –22, 2011.

[280] Y. Ueda, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Code clone analysis tool. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE)*, volume 2, pages 31–32, 2002.

[281] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the IEEE Symposium on Software Metrics (METRICS)*, pages 67–76, 2002.

[282] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 327 – 336, 2002.

[283] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

[284] R. Venkatasubramanyam, S. Gupta, and H. Sing. Prioritizing code clone detection results for clone management. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 30–36, 2013.

[285] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 128 –135, 2004.

[286] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the International Working Conference on Reverse Engineering (WCRE)*, pages 285–294, 2003.

[287] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In *Proceedings of the Dagstuhl Seminar Report on Duplication, Redundancy, and Similarity in Software (DRSS)*, pages 1–10, 2006.

[288] A. Walenstein, A. Lakhotia, and R. Koschke. The second international workshop on detection of software clones (IWDSC'03), workshop report. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 1–5, 2004.

[289] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: a rigorous approach to clone evaluation. In *Proceedings on the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 455–465, 2013.

[290] W. Wang and M. Godfrey. We have all of the clones, now what? toward integrating clone analysis into software quality. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 88–89, 2012.

[291] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 170–179, 2012.

[292] V. Weckerle. CPC: an eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, Germany, 2008.

[293] W. Winston. *Operations Research Applications and Algorithms*. Duxbury Press, Belmont, California, USA, 3rd edition, 1994.

[294] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *Proceedings of the International Conference on Mining Software Repository (MSR)*, pages 149–158, 2013.

[295] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo. An empirical study on the fault-proneness of clone migration in clone genealogies. In *CSMR-18/WCRE-21 Software Evolution Week*, page 10, CSMR-18/WCRE-21 Software Evolution Week.

[296] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Applying clone change notification system into an industrial development process. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 199–206, 2013.

[297] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Classification model for code clones based on machine learning. *Empirical Software Engineering*, pages 1–31, 2014.

[298] W. Yang. Identifying syntactic differences between two programs. *Softw. Pract. Exper.*, 21:739–755, 1991.

[299] N. Yoshida, T. Hattori, and K. Inoue. Finding similar defects using synonymous identifier retrieval. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 49–56, 2010.

[300] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *Proceedings of the IEEE Symposium on Software Metrics (METRICS)*, pages 16–25, 2005.

[301] Y. Yuan and Y. Guo. Cmcd: Count matrix based code clone detection. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 250–257, 2011.

[302] Y. Yuan and Y. Guo. Boreas: an accurate and scalable token-based approach to code clone detection. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 286–289, 2012.

[303] G. Zhang, X. Peng, Z. Xing, S. Jiang, H. Wang, and W. Zhao. Towards contextual and on-demand code clone management by continuous monitoring. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 497–507, 2013.

[304] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 285–294, 2012.

[305] Y. Zhang, H. Basit, S. Jarzabek, D Anh, and M. Low. Query-based filtering and graphical view generation for clone analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 376 –385, 2008.

[306] M. Zibran. *A Multiphase Approach to University Course Timetabling.* M.Sc. thesis, University of Lethbridge, Lethbridge, AB, Canada, 2007.

[307] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 266 – 269, 2011.

[308] M. Zibran and C. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 105–114, 2011.

[309] M. Zibran and C. Roy. Towards flexible code clone detection, management, and refactoring in IDE. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 75–76, 2011.

[310] M. Zibran and C. Roy. IDE-based real-time focused search for near-miss clones. In *Proceedings of the ACM Symposium on Applied Computing (SAC), SE Track*, pages 1235 – 1242, 2012.

[311] M. Zibran and C. Roy. The road to software clone management. Technical Report 2012-03, Department of Computer Science, University of Saskatchewan, Canada, http://www.cs.usask.ca/documents/techreports/2012/TR-2012-03.pdf, 2012.

[312] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring. *IET Software*, 7(3):167–186, 2013.

[313] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 295–304, 2011.

[314] M. Zibran, R. Saha, C. Roy, and K. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proceedings of the ACM Symposium on Applied Computing (SAC), SE Track*, pages 1123–1130, 2013.

[315] M. Zibran, R. Saha, C. Roy, and K. Schneider. Genealogical insights into the facts and fictions of clone removal. *ACM Applied Computing Review*, 13(4):30–42, 2013.

Appendix A

Publications Out of This Dissertation Research

What follows is a list of publications that are outcome of the research presented in this thesis. I am the prime contributor in all the publications except for the two ([b5], [b11]), where I am not the primary author, but I made significant contributions in those works as well. For the joint work, the contributions made by myself and the co-authors are clarified in Section A.1.

a. Refereed Journal Contributions:

[a1] **M. F. Zibran** and C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring. Journal of IET Software, 7 (3): 167–186, 2013.

[a2] **M. F. Zibran**, R. K. Saha, C. K. Roy, and K. A. Schneider. Genealogical Insights into the Facts and Fictions of Clone Removal. ACM Applied Computing Review, 13 (4): 30–42, 2013.

b. Refereed Conference and Workshop Contributions:

[b3] C. K. Roy, **M. F. Zibran**, and R. Koschke. The Vision of Software Clone Management: Past, Present, and Future. In IEEE CSMR-18/WCRE-21 Software Evolution Week (SEW'14), Vision Keynote, pp. 18–33, Belgium, 2014.

[b4] **M. F. Zibran**, R. K. Saha, C. K. Roy, and K. A. Schneider. Evaluating the Conventional Wisdom in Clone Removal: A Genealogy-based Empirical Study. In the 28th ACM Symposium On Applied Computing (ACM-SAC 2013), pp. 1123–1130, Portugal, 2013 (invited at the Journal of ACM Applied Computing Review).

[b5] T. Muhammad, **M. F. Zibran**, Y. Yamamoto, C. K. Roy. Near-miss Clone Patterns in Web Applications: An Empirical Study with Industrial Systems. In the 26th Annual Canadian Conference on Electrical and Computer Engineering (CCECE 2013), pp. 1–6, 2013, Canada.

[b6] **M. F. Zibran** and C. K. Roy. IDE-based Focused Search for Near-miss Clones. In the 27th ACM Symposium On Applied Computing (ACM-SAC 2012), pp. 1235–1242, Italy, 2012.

[b7] **M. F. Zibran** and C. K. Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In the 11th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011), pp. 105–114, USA, 2011 (one of the seven best papers invited at a special issue of the IET Software Journal).

[b8] **M. F. Zibran** and C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach. In the student symposium of the 19th IEEE International Conference on Program Comprehension (ICPC 2011), pp. 266–269, Canada, 2011.

[b9] **M. F. Zibran**, R. K. Saha, M. Asaduzzaman, and C. K. Roy. Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study. In the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2011), pp. 295–304, USA, 2011.

[b10] **M. F. Zibran** and C. K. Roy. Towards Flexible Code Clone Detection, Management and Refactoring in IDE. In the 5th ACM International Workshop on Software Clones (IWSC 2011), pp. 75–76, USA, 2011.

[b11] R. K. Saha, M. Asaduzzaman, **M. F. Zibran**, C. Roy, and K. A. Schneider. Evaluating Code Clone Genealogies at Release level: An Empirical Study. In the 10th IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2010), pp. 87–96, Romania, 2010.

c. Technical Report, Posters, Tool Demonstrations, and Presentations:

[c12] **M. F. Zibran** and Chanchal K. Roy. The Road to Software Clone Management: A Survey, Technical Report 2012-03, pp. 1–62, Department of Computer Science, University of Saskatchewan, Canada, 2012.

[c13] **M. F. Zibran**. Diagnosis and Treatment of Code Clones. In poster symposium of the 27th IEEE International Conference on Software Maintenance (ICSM 2011), USA, 2011.

[c14] **M. F. Zibran** and C. K. Roy. Code Clones: Etiology, Effects, and Treatment. In the CSER (Consortium for Software Engineering Research) 2011 Spring Meeting (poster session), Ontario, Canada, 2011.

[c15] **M. F. Zibran** and C. K. Roy. Cloning in Software: Why, When, and How?. In the College of Arts and Science Graduate Students Poster Symposium, University of Saskatchewan, Canada, 2011.

[c16] **M. F. Zibran** and C. K. Roy. Flexible Code Clone Detection and Management in IDE. In the Technology showcase of the 20th Annual Conference (CASCON 2010), Centre for Advanced Studies Research, IBM Canada Software Laboratory, Canada, 2010.

The publications [b6] and [c16] constitute the majority of Chapter 3. The empirical studies appeared in [b5], [b9], and [b11] are compiled in Chapter 4. Chapter 5 corresponds to the publications [a2] and [b4]. Chapter 6 compiles the work appeared in [b8], [b7], and [a1]. A proof-of-concept of the work (Chapter 6) was published in [b8], a matured state of the same work appeared in [b7], and an extended version of the work was published in [a1]. Chapter 7 is composed of an analytical survey [c12] of the existing literature, part of which also appeared in [b3]. The research idea of the entire thesis appeared in the doctoral symposium of ICSM' 2011 [c13], which helped us in validating significance the problem addressed in this thesis. The publications [b10], [c14], and [c15] are also related to this thesis as a whole.

A.1 Co-authorship

The research presented in this thesis is my own, conducted under the supervision of Chanchal K. Roy. Ideas and techniques that are not products of my own work are duly cited, or, in cases where citations are not available, are presented using language that indicates they existed prior to this work. Small parts of the research presented in this thesis involved joint work with other researchers to make co-authored publications.

The majority of the clone-density based empirical study, as presented in Chapter 4, was published in ICECCS'2011 [b9] in co-authorship with Ripon K. Saha and Muhammad Asaduzzaman. I am the sole contributor in initial formulation of the idea, data collection, and analysis involved in the work. While preparing the paper, the co-authors helped in organizing the write-up and correcting presentation flaws. The studies briefly presented in Section 4.3 and Section 4.4 appeared in SCAM'2010 [b11] and CCECE'2013 [b5] respectively. In both these works, I contributed in the analysis, writing, and shaping the results into publishable research papers.

In the genealogy-based study as described in Chapter 5, we used *extended* version of `gCad` [249] for the extraction and characterization of evolving clones. Ripon K. Saha was the original author of the *basic* `gCad` [249], and he contributed by providing the source code of his tool, which I further extended significantly for the purpose of the study. I carried out the empirical study including study design, data collection, analysis as well as organization and presentation of the write-up for the published papers [b4], [a2]. Kevin A. Schneider contributed by proof-reading the papers and providing suggestions for improvements.

Chapter 7 of the thesis is compiled on the basis of my extensive survey on the existing literature [c12]. As mentioned before, a part of the chapter appeared as a Vision Keynote paper in the IEEE CSMR-18/WCRE-21 Software Evolution Week (SEW'14) [b3]. Rainer Koschke helped in preparing the paper out of the work by providing suggestions for the improvement of the organization of the paper.

Other than the aforementioned contributions of the co-authors of my papers, the entire work presented in this thesis is the outcome of my own research carried out under the supervision of Chanchal K. Roy.

Appendix B

User Manual for Our Clone Search Tool

As described in Chapter 3, our clone search tool is capable to detecting both exact (*Type-1*) and near-miss (*Type-2* and *Type-3*) clones of a chosen code fragment that the user selects from the IDE's editor. Details about the different types of code clones are presented in Chapter 2. We named our tool as `CloMan`. The name is formed by concatenating the prefix (i.e., clo) of the word 'clone' and the prefix (i.e., man) of the word 'management'. Note that the main objective of our tool is to support proactive clone management by facilitating *focused* clone search. `CloMan` is developed as a plugin to the popular Eclipse IDE. Here, we describe how to install and use our tool. In Section B.1, we present step-by-step procedure for the installation of our tool. Section B.2 presents how the user can tune the tool by setting different configuration parameters. Finally in Section B.3, we describe how this tool can be effectively used for searching clones in the source code.

B.1 Installation

Our Eclipse plugin (`CloMan`), is available for installation from its *Update Site* at http://www.usask.ca/~minhaz.zibran/tools.

For installation, please follow the following steps. Each step is demonstrated with the help of screenshots where the locations of the user's input/choice are marked with red rectangles.

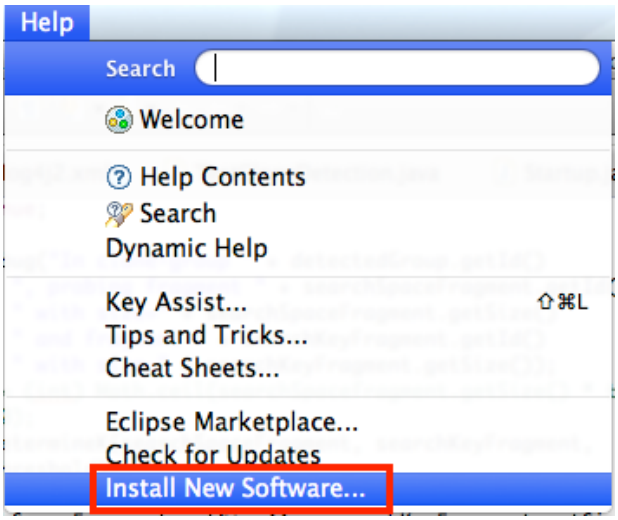


Figure B.1: Step 1 of installation

Step 1: Choose Help from Eclipse's menu and then select Install New Software, as shown in Figure B.1.

Step 2: The completion of step 1 brings the installation window as shown in Figure B.2. Press the Add... button to add a new Update Site.

Step 3: The completion of step 2 brings up a dialog box asking for the *Name* and *Location* of the Update Site to add. Fill-up the text boxes according to the following specification and press the Ok button.

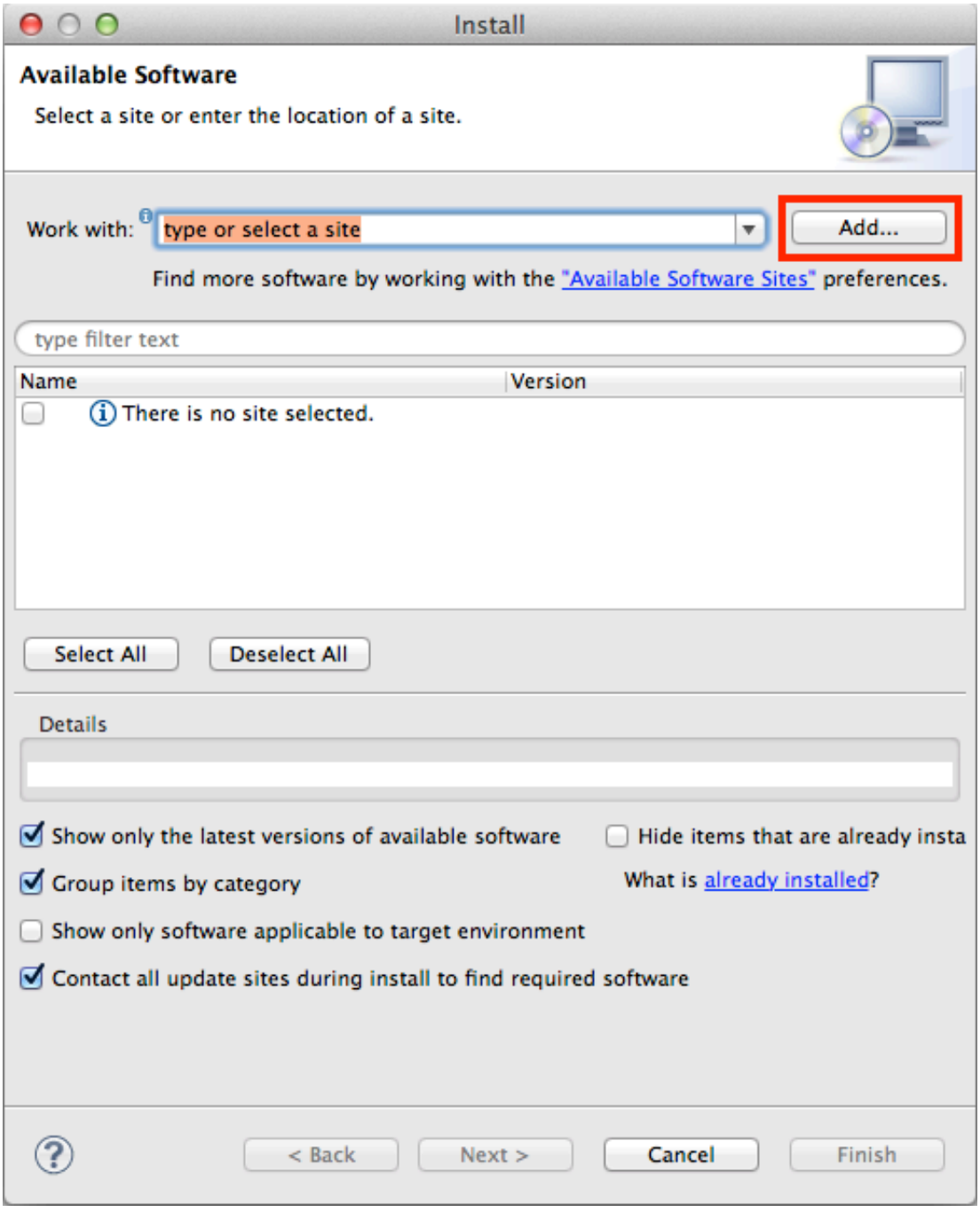


Figure B.2: Step 2 of installation: add a new *Update Site*

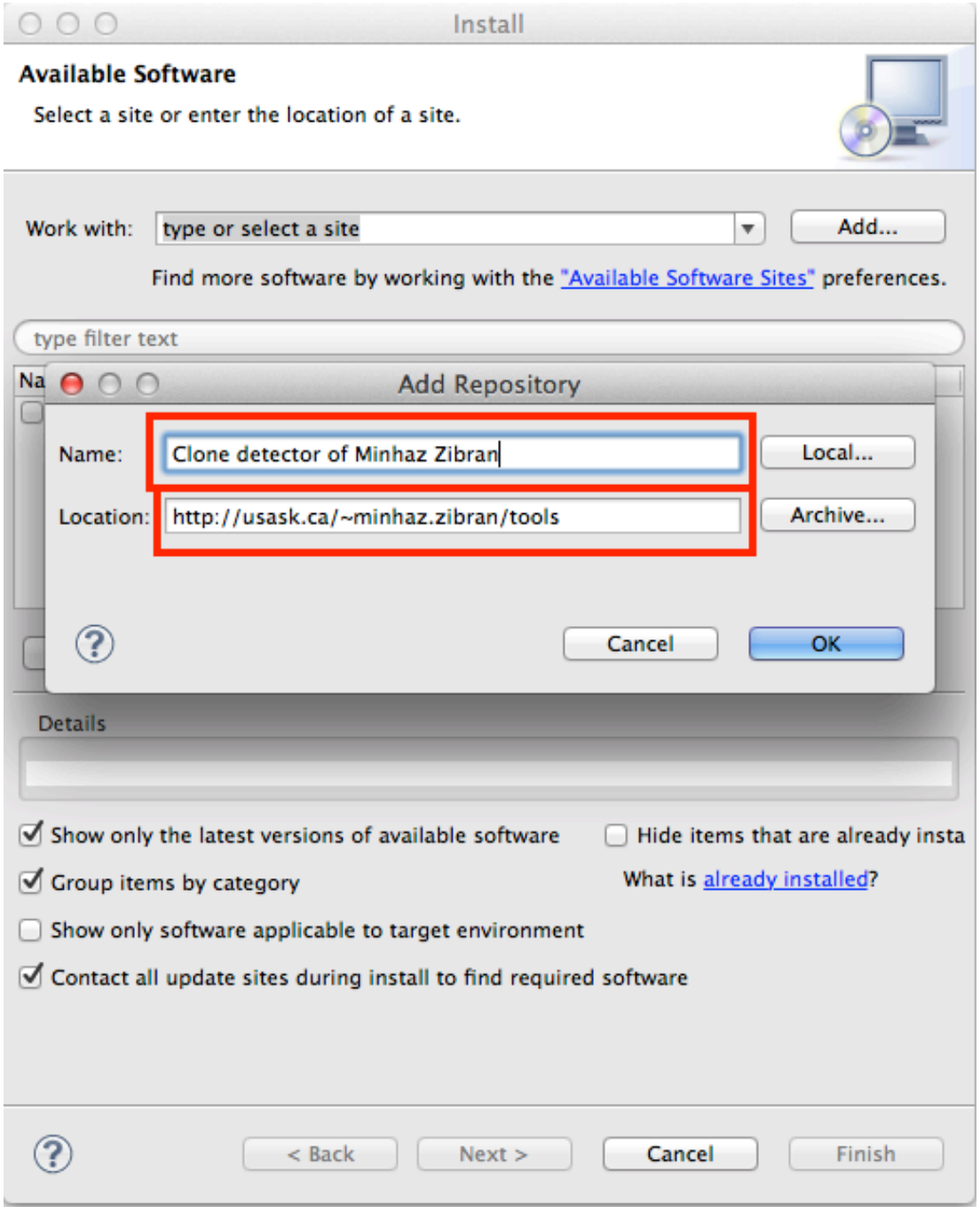


Figure B.3: Step 3 of installation: filling in information for new *Update Site*

Name: Anything to describe the Update Site (e.g., Clone Detector of Minhaz Zibran)

Location: http://www.usask.ca/∼minhaz.zibran/tools

Step 4: Upon completion of step 3, the installation window populates with available tools for installation, as shown in Figure B.4.

4(a) Select the check-box by the tool's name `CloMan`

4(b) Press the Next > button at the bottom of the installation window

Step 5: Upon completion of step 4, the installation window switches to the next step (Figure B.5) for you to review items and finalize installation.

5(a) Make sure the tool's name (i.e., `CloMan`), its *Version* and *Id* are displayed. Select the tool's name.

5(b) Review the details of the plugin to be installed shown in the *'Details'* text-area.

5(c) Click on *'More..'* below the *'Details'* text-area. This will bring up another dialog box for reviewing the *Copyright* (Figure B.6), *General Information* (Figure B.7), and *License Agreement* (Figure B.8). Review the information and press the Ok button. The dialog box disappears.

5(d) Press the Next > button at the bottom of the installation window

Step 6: Upon completion of step 5, the installation window switches to the new window (Figure B.9) to allow you review (and accept/reject) the license agreement.

6(a) Please carefully read the license agreement and select the radio-button with label "I accept the terms of the license agreement"

6(b) Press the Finish button and wait for the installation to complete.

B.2 Customization

Our tool `CloMan` allows a set of customization options to the user through Eclipse's *Preferences* page.

1. From Eclipse's main menu, choose Eclipse. It brings a drop down menu as shown in Figure B.10. Open Eclipse's *Preferences* page by selecting Preferences... from the drop down menu.

2. Find and select out tool's name (i.e., `CloMan`) from the left side of the Eclipse's *Preferences* page, as shown in Figure B.11. All the available customization options are displayed on the right side of the page. Set values of the different tuning parameters according to your choice, and press the Ok button to make the customization in effect.

B.3 Usage

The usage of our tool is fairly intuitive. To invoke focused clone search, take the following steps:

1. Select a code fragment from the editor as the *seed* for search and click the right mouse-button. This will bring a popup menu as shown in Figure B.12.

2. Choose the Find Clones menu item from the popup menu (Figure B.12) to initiate clone detection.

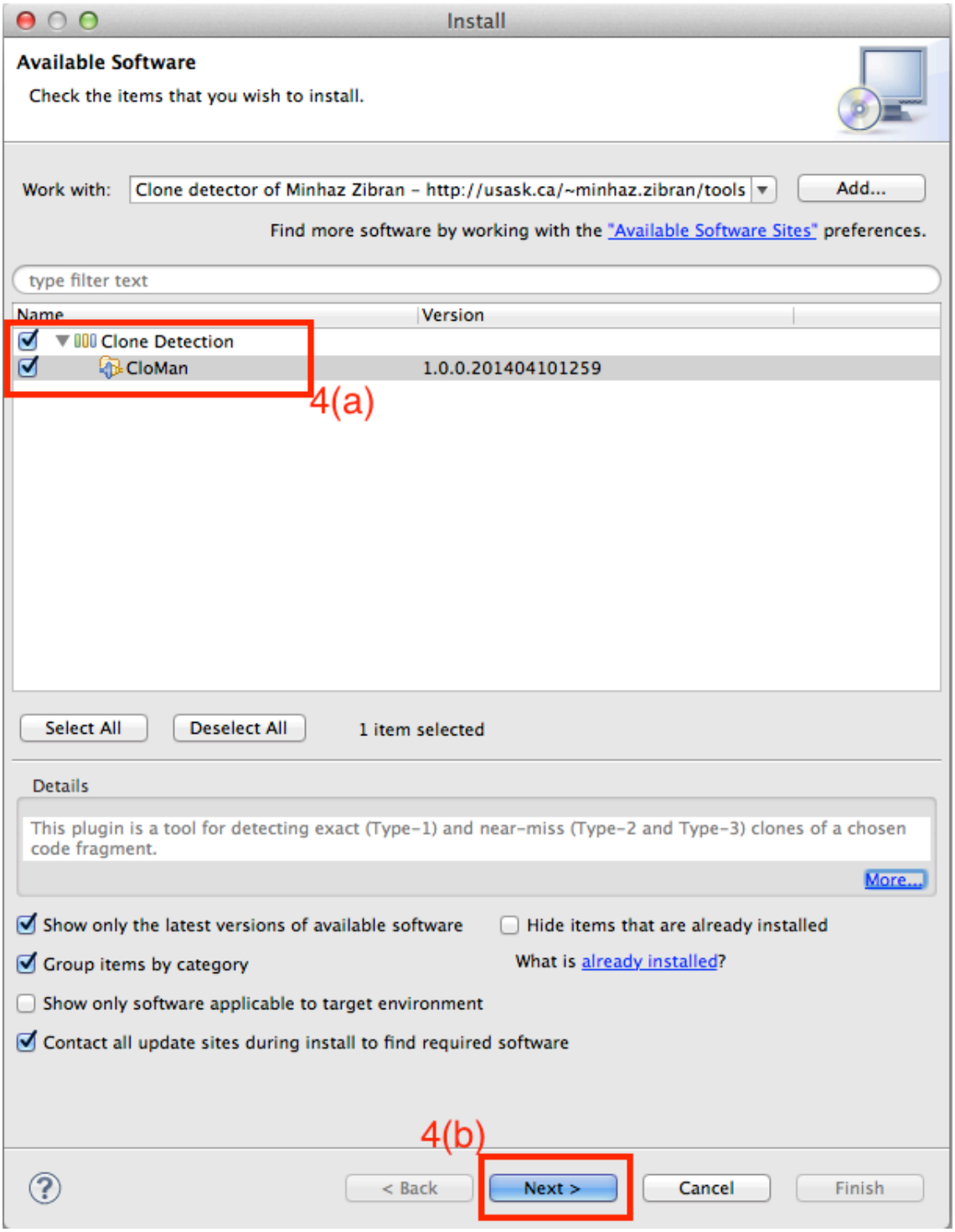


Figure B.4: Step 4 of installation: tool selection

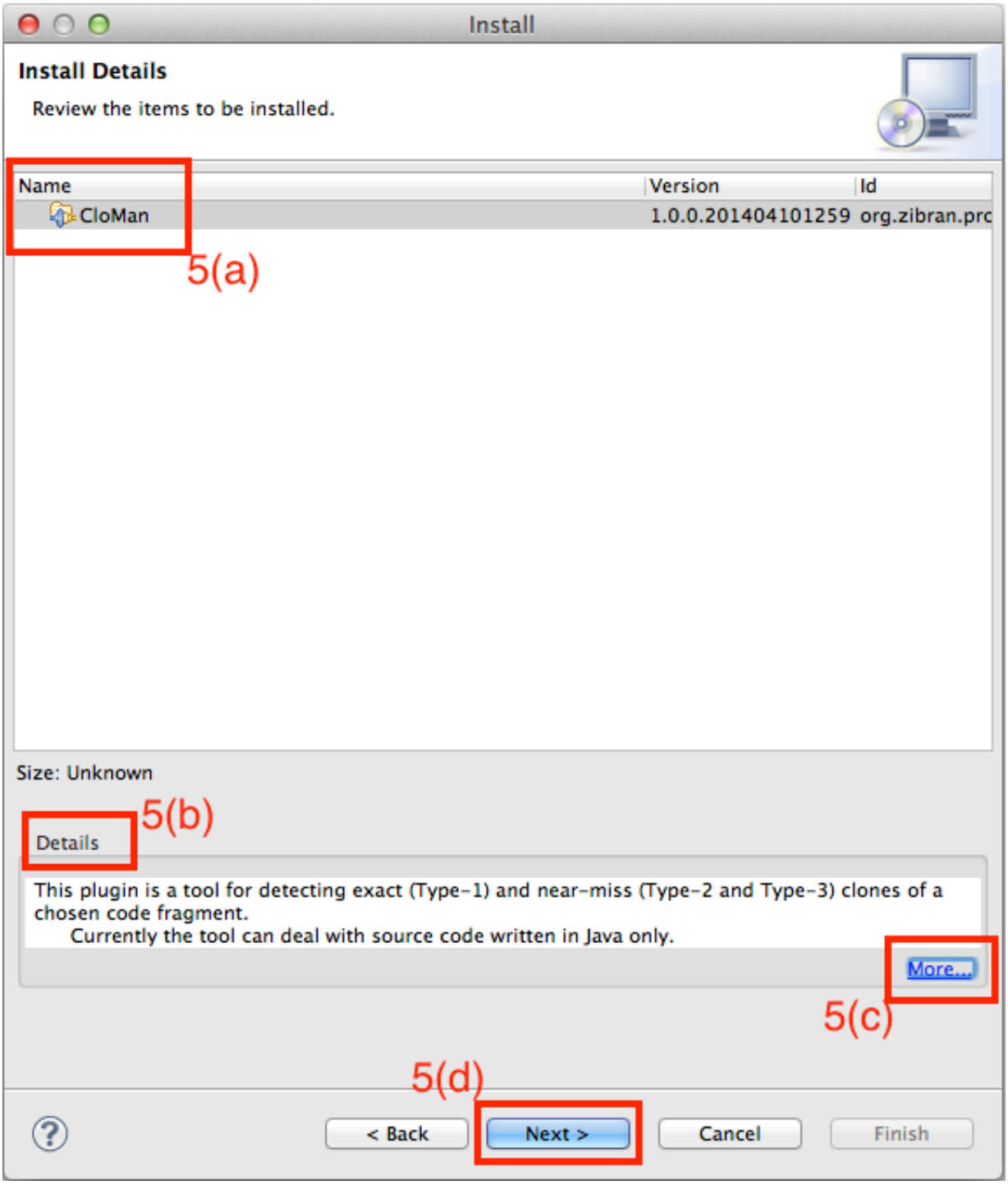


Figure B.5: Step 5 of installation: choice review

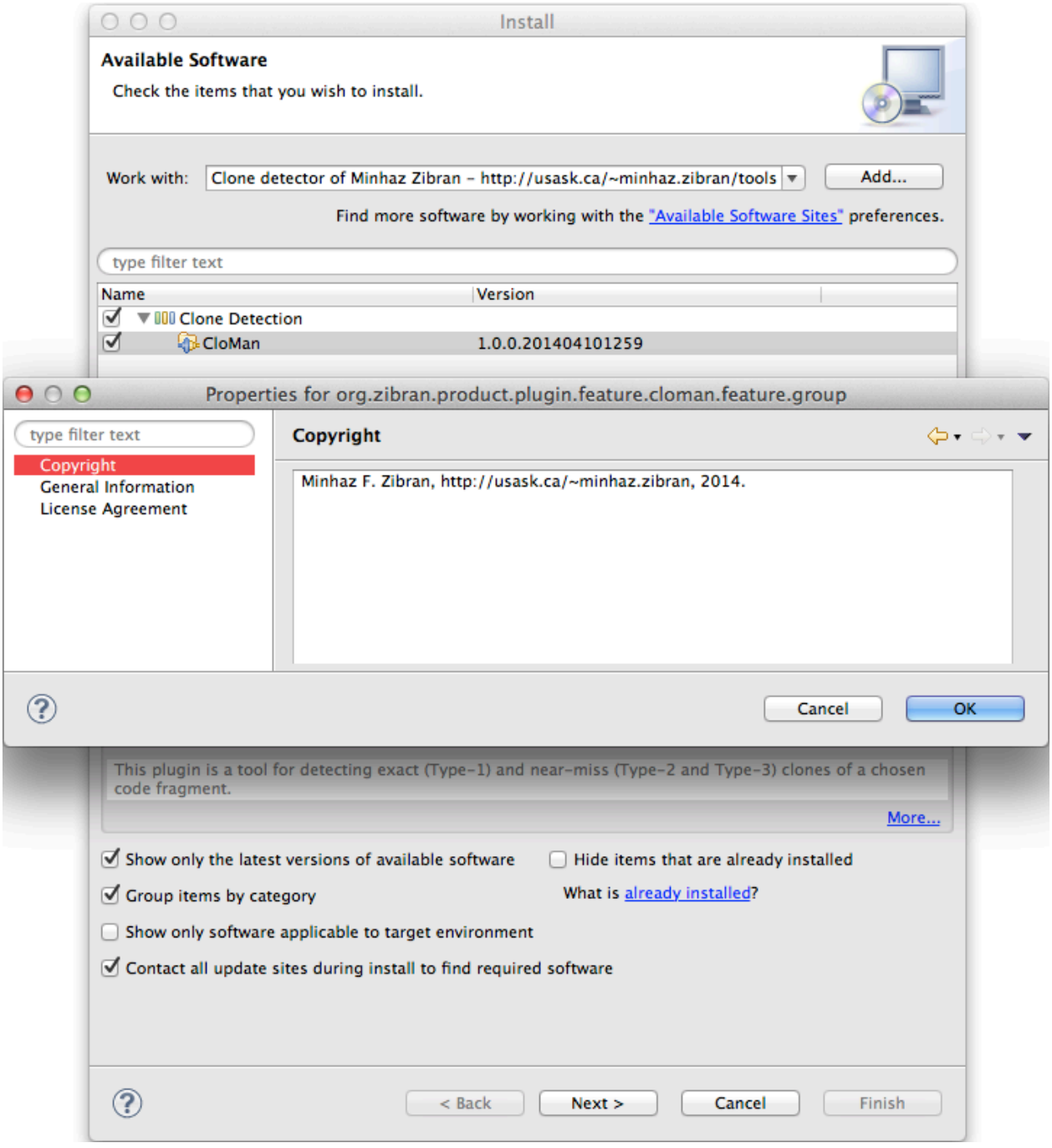


Figure B.6: Step 5(c) of installation: reviewing the *Copyright* information

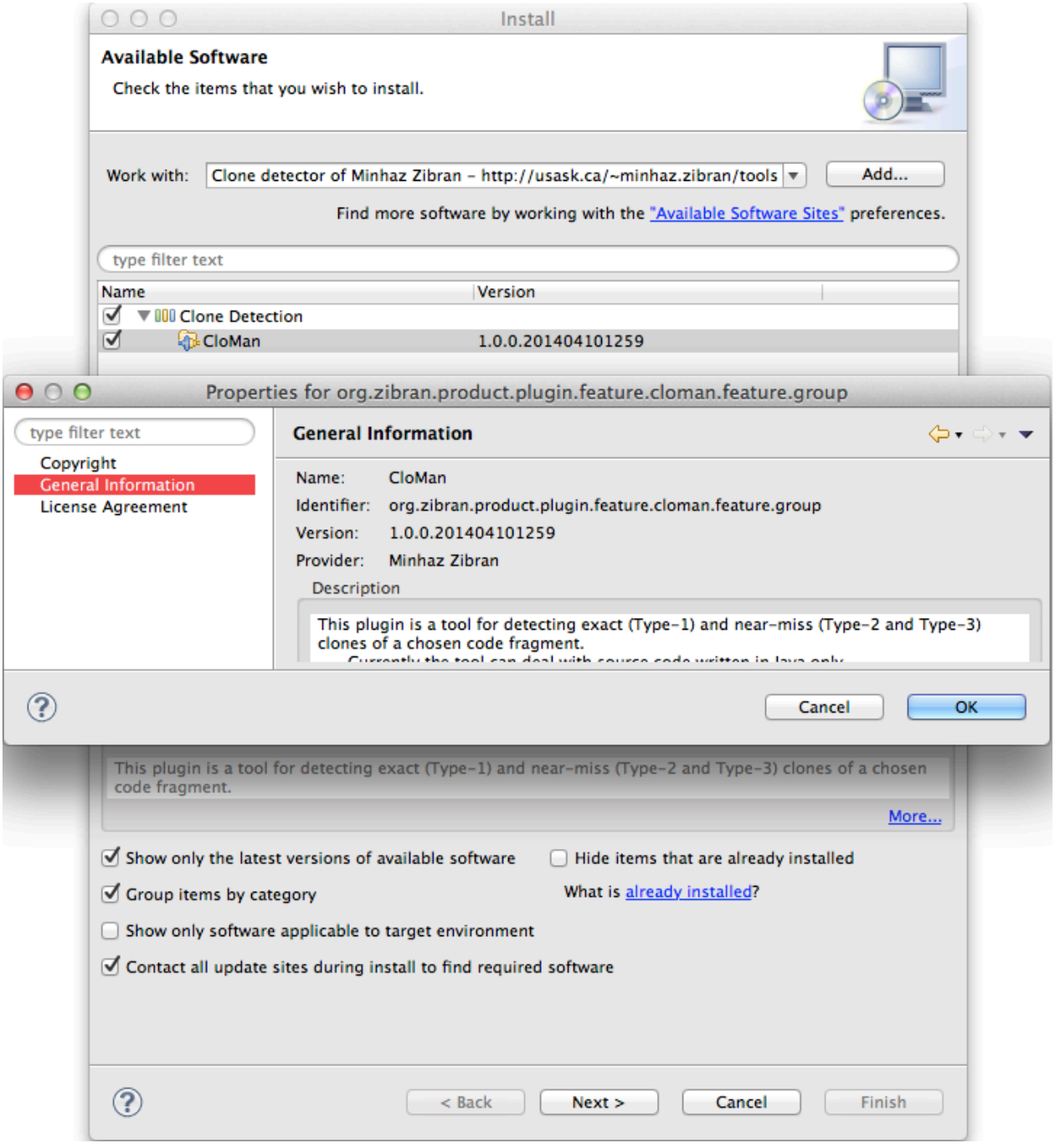


Figure B.7: Step 5(c) of installation: reviewing the *General Information*

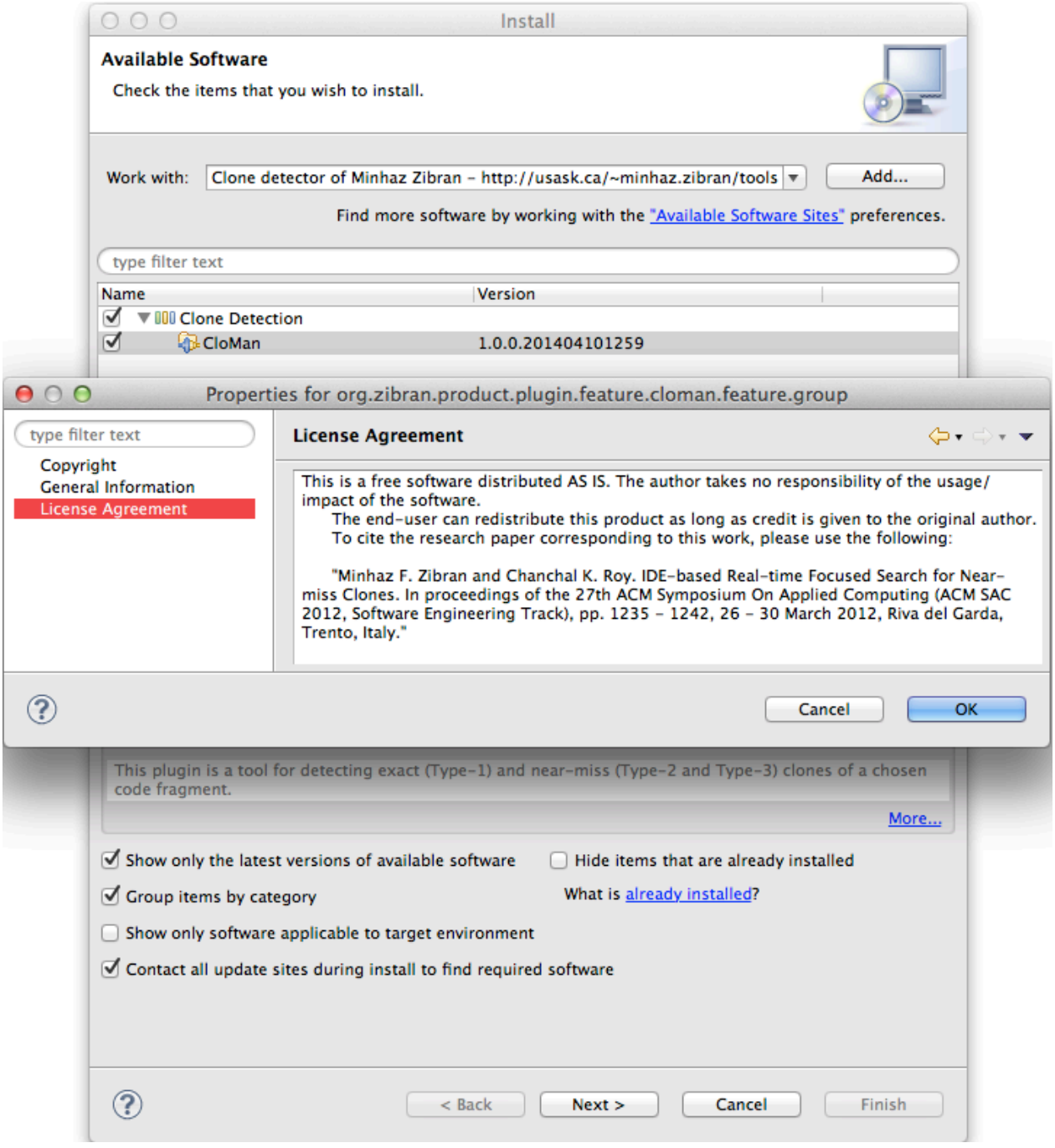


Figure B.8: Step 5(c) of installation: reviewing the *License Agreement*

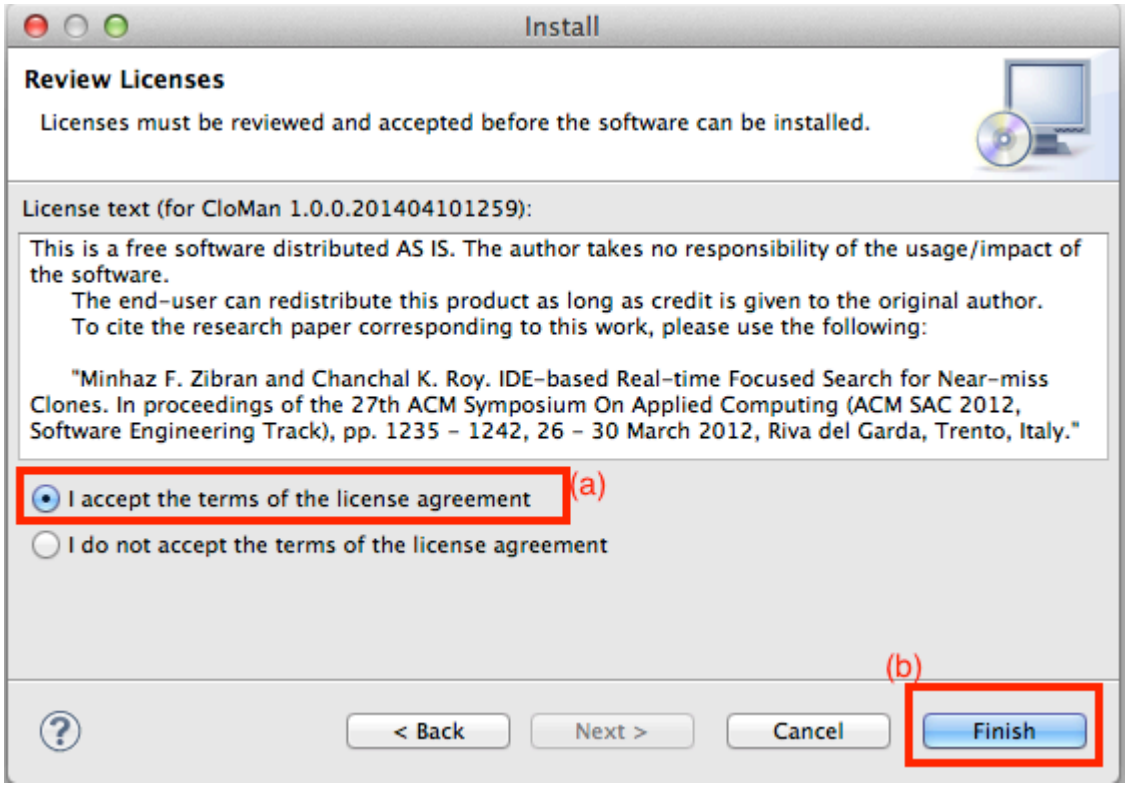


Figure B.9: Step 6 of installation: acceptance of license agreement

Figure B.10: Step 1 of customization: opening Eclipse's *Preference* page

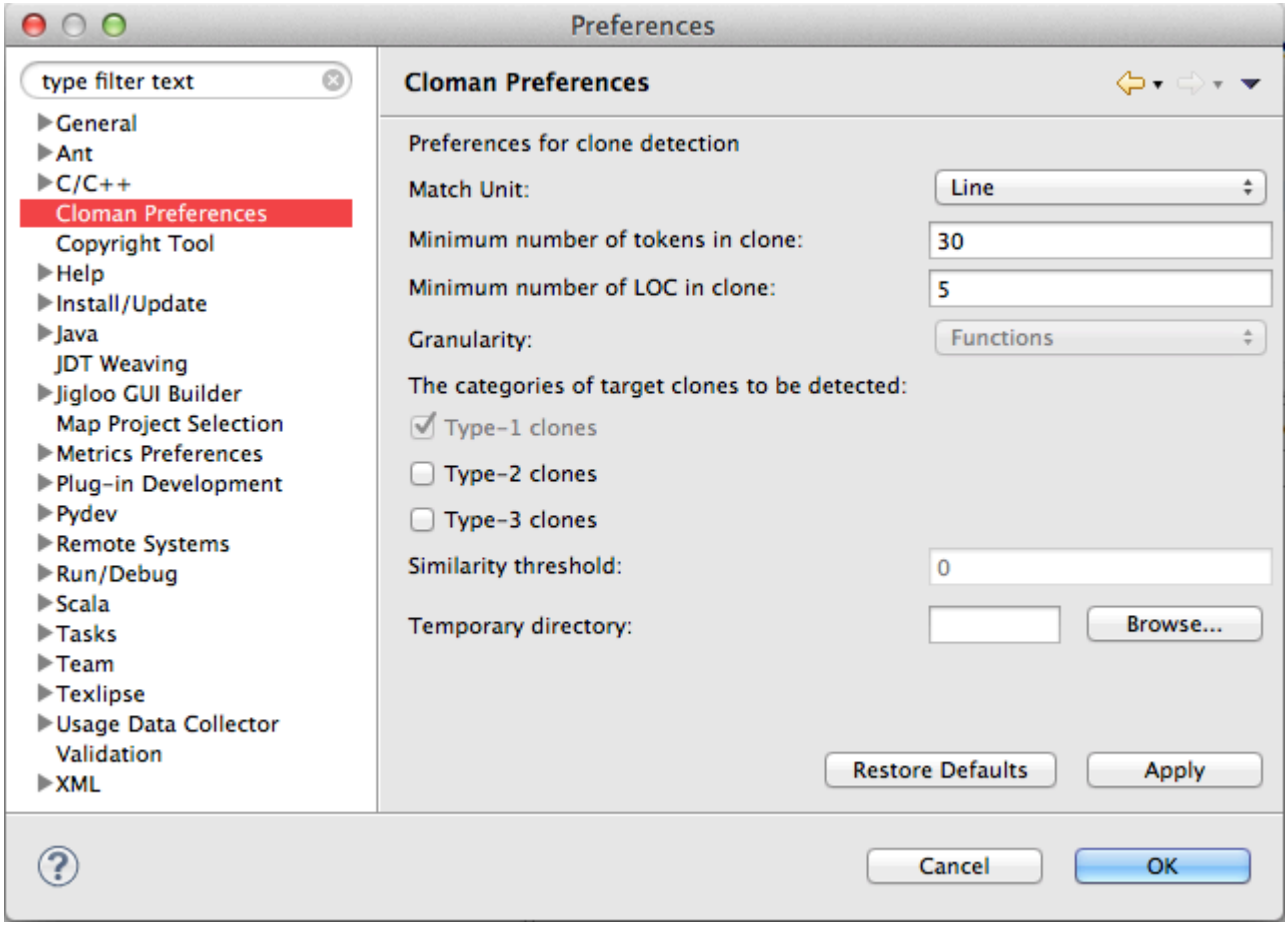


Figure B.11: Step 2 of customization: tuning `CloMan`'s parameters

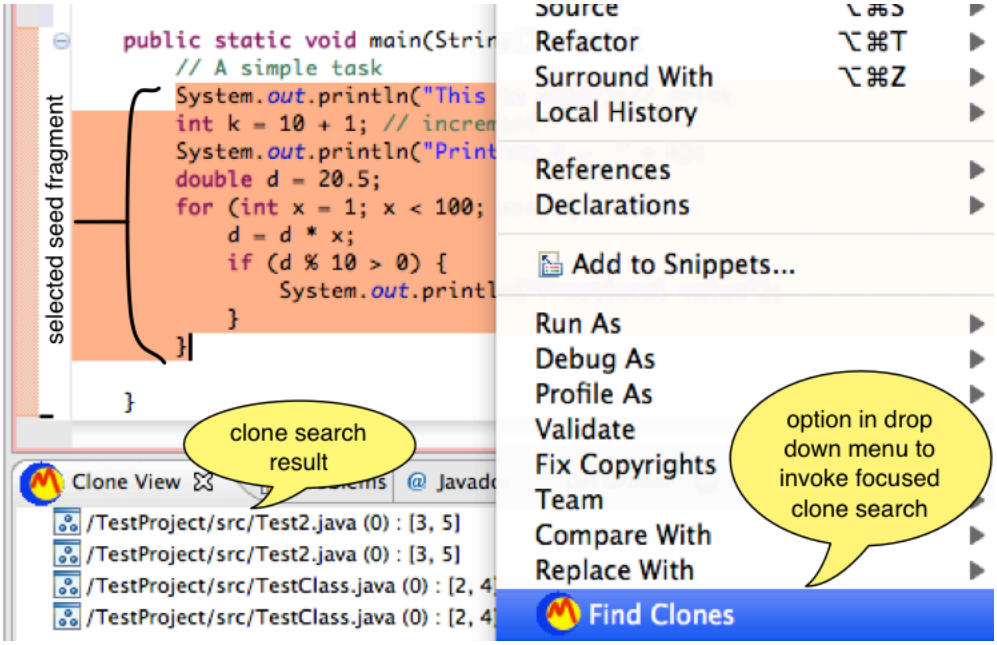


Figure B.12: Usage of our clone search tool `CloMan`

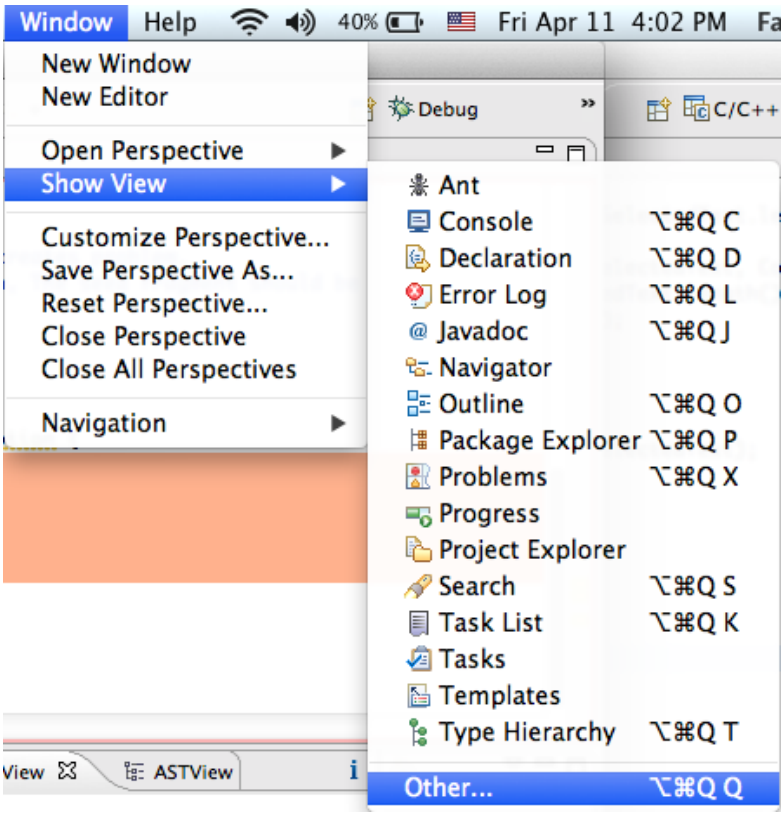


Figure B.13: Step 1 of bringing up the *Clone View*

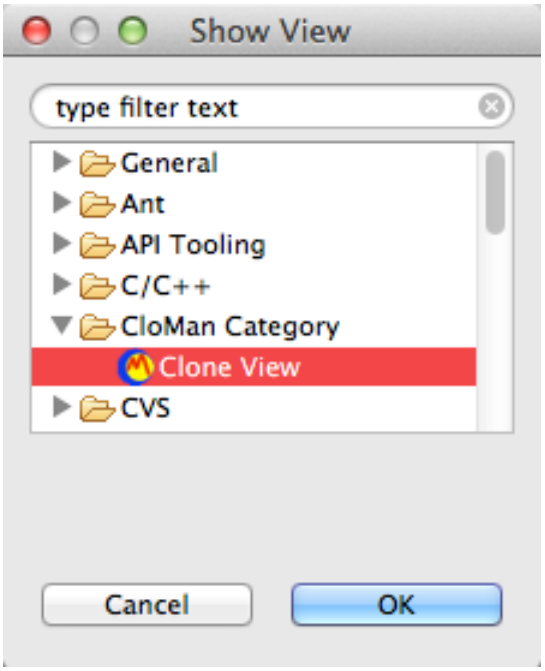


Figure B.14: Step 2 of bringing up the *Clone View*

The result of clone search is presented in *Clone View* as shown in Figure B.12. The detected clones are presented as a list sorted in the descending order of their similarity with the fragment that was selected as the seed for the focused search. Double click on any item of the list to view the corresponding clone fragment in the editor.

In case you don't find the *Clone View*, make it appear by doing the followings.

1. Choose the Window, then from the appearing drop down sub-menu, choose Show View. This brings a second drop-down sub-menu, from which you should select Other, as shown in Figure B.13. A *Tree View* window appears as shown in Figure B.14.

2. Find and expand the CloMan Category node in the *Tree View* window. This should expose a child node *Clone View*, which you should select.

3. Press the Ok button at the bottom of the *Tree View* window.

Appendix C

OPL Model of The Refactoring Scheduler

The constraint programming (CP) model of our clone refactoring scheduler as written in OPL (Optimization Programming Language) is presented below.

```
using CP;

// data variables
int nRefactor = ...;
int nTarget = ...;
range myRange = 1..nRefactor;
float efforts[myRange] = ...;
float qualityGain[myRange] = ...;
float priorities[myRange] = ...;
int dependencies[myRange][myRange] = ...;


// decision variables
dvar int+ x[myRange]in 0..1;
dvar int+ y[myRange] in 0..nTarget;


// impose time-limit if required
// execute{cp.param.timeLimit=2;}


// objective function
maximize //find optimal
        sum(i in myRange) x[i] * priorities[i] * (qualityGain[i] - efforts[i]);


// constraints
subject to {
        // if x[i] = 1, y[i] > 0, and vice versa
        forall (i in myRange)
                 x[i] * y[i] >= x[i];

        // y[i] != y[j], unless y[i]=y[j]=0
        forall(i, j in myRange: i != j)
                y[i] != y[j] || y[i]==0;

        // sequential dependencies
        forall(i, j in myRange: i != j)
                if (dependencies[i][j] - dependencies[j][i] > 0)
                        y[i] < y[j] || y[j] == 0;

        // mutual exclusion
        forall(i, j in myRange: i != j)
                   if(abs(dependencies[i][j]) == 1)
                           x[i] + x[j] <= 1;
```

```
        // mutual inclusion
        forall(i, j in myRange: i != j)
                   if(abs(dependencies[i][j]) == 2)
                           x[i] + x[j] == 2 || x[i] + x[j] == 0;

        // max number of refactoring that can be selected
        // sum(i in myRange) x[i] <= nTarget;
};

// post-processing
execute{
    var totalEffort = 0.0;
    var totalGain = 0.0;
    var totalSelect = 0;
    var priorSat = 0.0;
    for(var i in myRange) totalEffort = totalEffort + x[i]*efforts[i];
    for(var j in myRange) totalGain = totalGain + x[j]*qualityGain[j];
    for(var k in myRange) totalSelect = totalSelect + x[k];
    for(var m in myRange) priorSat = priorSat + x[m]*priorities[m];
    writeln("Satisfied priority = ", priorSat);
    writeln("total effort = ", totalEffort);
    writeln("total gain = ", totalGain);
    writeln("total selection = ", totalSelect);
};
```