Efficient Code Clone Management based on Historical Analysis and Refactoring Support

Submitted to
Graduate School of Information Science and Technology
Osaka University

July 2013

Keisuke HOTTA

Abstract

Code clones have recieved great interests in recent years from many researchers, engineers, and practitioners in the field of software engineering. A code clone is defined as a group of code fragments that are identical or similar to one another. Code clones are introduced into source code of software systems by various reasons, and the most typical one is code cloning by copy-and-paste operations for reusing existing features. Typical software systems contain a certain amount of code clones because code cloning is a common practice for software developers.

The existence of code clones has been regarded as a bad smell for software evolution over a period of time because code clones require much attention to be maintained. Once code clones are introduced into source code, most of them should be consistently maintained. Unintended inconsistencies among code clones have a high risk for introducing bugs in software systems. However, it is not an easy task for developers or maintainers to be aware of all the code clones and maintain all of them consistently, specifically in the case of large software systems. This is a reason why code clones are regarded as bad factors for software evolution.

Many researchers have proposed a variety of techniques to cope with code clones based on this common wisdom. However, some of recent empirical studies have been against it. That is, these studies revealed that code clones do not highly affect software evolution. The discussion for harmfulness of code clones remains inconclusive, but it is widely accepted that not all but a part of code clones have negative impacts on software evolution.

For these reasons, it is not effective to prohibit software engineers from code cloning. Furthermore, prohibiting code cloning is also unrealistic because of advantages of it. Therefore, it is strongly required to manage code clones effectively.

The objective of the work described in this dissertation is to promote efficient software evolution through effective managements for code clones.

To achieve this objective, it is necessary to know state-of-the-art of research achievements. Therefore, we conducted a survey on research literatures on code clones. This survey categorized literatures into five categories, detection, removal, prevention, analysis, and bug detection. The survey told us current states of re-

search on code clones. It also told us limitations of previous research on code clone analysis, and unclear points of characteristics of them.

Based on the results of the survey, we conducted an empirical study to reveal how negative impacts code clones have. Although some existing studies investigated the impacts of code clones, they have some limitations to genealize their findings. To resolve these limitations, this study proposed a new metric and used multiple clone detectors. The metric proposed by this study captures modification frequencies on code clones, which is under an assumption that code clones have negative impacts if they are frequently modified. This study compared metric values on cloned code with non-cloned code to judge whether code clones affect software evolution on 15 open source software systems with four clone detectors. The experimental results indicate that code clones do not tend to be modified more frequently than non-cloned code, hence the study concluded that code clones do not have seriously negative impacts on software evolution.

Through the emprical study, we recognized a necessity of further analyses on impacts of code clones. This is because our first study lumped code clones all together, and so it did not consider how individual code clones affect on software evolution. Analyzing histories of individual code clones is helpful to investigate the research problem. To anlayze histories of individual code clones, it is necessary to track code clones across version histories of source code. There are some techniques to tackle this challenging task, but they still remain some issues. Therefore, we developed a new technique to track code clones by enhancing an existing clone tracking technique. The key idea of the enhancement is to link clones located in not only exactly the same regions but also similar ones. We confirmed an improvement of accuracy of clone tracking through an experiment on two open source software systems. Furthermore, we conducted an investigation for reasons why code clones are gone as an application of the proposed technique for clone tracking.

With the new tracking technique, we conducted another empirical study on evolution of code clones. This study detected clone genealogies, which represent how individual code clones evolved, and analyzed them. This investigation was interested in how long code clones survived and how many times they were modified throughout their lifetimes. The experimental results reaffirmed that most of clones had short lifetimes and were modified at most once, both of which were reported in previous research. Furthermore, the results revealed some characteritics that have not been revealed in previous research. One of the findings is that approximately 3% of code clones are long-lived and modified multiple times. In other words, approximately 97% of code clones do not require high costs to be maintained. This finding empirically supports an opinion that not all but a part of clones affect software evolution. Another finding is that code clones tend to be

modified more frequently in the former halves of lifetimes than in the latter ones. This finding suggests that it is necessary to start managing code clones in earlier stages of their lifetimes for an effective code clone management.

In the next, we proposed a way to cope with code clones, which is a support to remove them with a particular refactoring pattern. The refactoring pattern used in this study enables to remove code clones even if they include some gaps. Moreover, the proposed technique performs a fine-grained analysis on source code, which allows to handle instances that any of previous refactoring supports cannot handle. Furthermore, the proposed technique can handle all the code fragments included in a code clone as against to existing techniques that can only handle a pair of code fragments. We have developed a software tool as an implementation of the proposed technique, and validated the usefulness of the technique through two experiments, one is on open source software systems, and the other is with subjects.

This dissertation is organized as follows.

Chapter 1 gives the background of this work and an overview of this dissertation.

Chapter 2 presents the survey results on literatures that are related to code clones. This chapter explains a definition of code clones, and then it discusses some perspectives of them. The introductions of each research achievement follow the discussion, with categorized loosely into five categories.

In Chapter 3, we present the result of the empirical study on stabilities of code clones. The chapter describes how to perform the investigation, the experimental targets of this study, and the experimental results. In this study, we compared the experimental results of our experimental methodology with other methodologies proposed by other researchers. This chapter refers to the results of the comparisons, and then it discusses the experimental results.

Chapter 4 proposes the new clone tracking technique. This technque is an enhanced version of an existing technique. We fist explain the key idea of the enhamcement, and then discuss the effectiveness of the proposed technique with an experiment on open source software systems. Furthermore, we conduct another experiment to reveal why clones are gone, which is an application of the proposed technique.

Chapter 5 presents the result of the empirical study on genealogies of code clones. This chapter revisits two major findings on evolution of code clones, and then reveals some characteristics that any of previous reseach did not address.

Chapter 6 proposes the refactoring support technqiue for code clones including some gaps. This chapter formally describes the technique and introduces an implementation of the proposed technique. It also presents the experimental results to confirm the usefulness of the proposed technique.

Finally, Chapter 7 summalizes this dissertation and shows some future directions of this work.

List of Publications

Major Publications

[1-1] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji 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 11th International Workshop on Principles of Software Evolution (IWPSE-EVOL 2010), pp.73-82, Antwerp, Belgium, September 2010.

[1-2] Keisuke Hotta, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto: "An Empirical Study of Influence of Duplicate Code on Software Maintenance Based on Modification Frequency Comparison," IPSJ Journal, Vol.52, No.9, pp.2788-2798, September 2011 (in Japanese).

[1-3] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto: "Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph," in Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), pp.53-62, Szeged, Hungary, March 2012.

[1-4] Keisuke Hotta, Yui Sasaki, Yukiko Sano, Yoshiki Higo, and Shinji Kusumoto: "An Empirical Study on the Impact of Duplicate Code," Advances in Software Engineering, Hindawi Publishing Corporation, Vol.2012, May 2012.

[1-5] Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto: "CRat: A Refactoring Support Tool for Form Template Method," in Proceedings of the 20th International Conference on Program Comprehension (ICPC 2012), pp.250-252, Passau, Germany, June 2012.

[1-6] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto: "Supporting Template Methods Pattern Application on Code Clones by Program Dependence Graph," IEICE Journal, Vol.J95-D, No.7, pp.1439-1453, July 2012 (in Japanese).

[1-7] Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto: “A Refactoring Support with Form Template Method for Groups of Multiple Similar Methods,” IEICE Journal, Vol.J96-D, No.2, pp.362-364, February 2013 (in Japanese).

[1-8] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto: “A Survey on Code Clone Management Focusing on Prevention, Methodology for Efficient Analysis, and Bug Detection,” accepted by JSSST Computer Software, February 2014 (in Japanese).

Related Publications

[2-1] Yui Sasaki, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto: “Is Duplicate Code Good or Bad? An Empirical Study with Multiple Investigation Methods and Multiple Detection Tools,” in Proceedings of the 22nd International Symposium on Software Reliability Engineering (ISSRE 2011), Hiroshima, Japan, December 2011.

[2-2] Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto: “Enhancement of CRD-based Clone Tracking,” accepted by the 13th International Workshop on Principles on Software Evolution (IWPSE 2013), Saint Petersburg, Russia, August 2013.

Acknowledgements

During this work, I have been fortunate to have received assistance from many individuals.

First, I would like to express my heartfelt gratitude to my supervisor, Professor Shinji Kusumoto, for his considerate support, encouragement, and guidance throughout this work. He has spent much time and effort for this work. I do not believe this work would have been possible without his help.

I would like to express my sincere gratitude to Dean and Professor Katsuro Inoue for his helpful comments, valuable questions, and kind advice for this work.

I also would like to express my sincere appreciation to Professor Toshimitsu Masuzawa for his valuable comments and helpful suggestions on this dissertation.

I would like to express my sincere appreciation to Professor Ken-ichi Matsumoto, in Nara Institute of Science and Technology, for his valuable questions and discussions for this work.

Also, I would like to thank Associate Professor Kozo Okano for his valuable suggestions and discussions for this work.

I am deeply grateful to Associate Professor Hiroshi Igaki. I could not complete this work without his helpful comments, valuable suggestions, and kind encouragement.

I would like to express my heartfelt appreciation to Assistant Professor Yoshiki Higo for his great contributions throughout this work. His zealous coaching and support have strongly encouraged me, and his helpful comments and suggestions made it possible to complete this work.

Many of courses that I have taken during my graduate and undergraduate careers have been helpful in preparing this thesis. I would especially like to acknowledge the guidance of Professor Ken-ichi Hagihara and Professor Yasushi Yagi.

I also would like to express my appreciation to Assistant Professor Takeshi Kakimoto, in Kagawa National College of Technology, for his valuable suggestions and helpful advice.

I would like to thank Assistant Professor Norihiro Yoshida, in Nara Institute of Science and Technology, and Assistant Professor Hideaki Hata, in Nara Institute of

Science and Technology, for their valuable feedback for this work. I also would like to thank researchers on software engineering who have given me helpful comments and encouragement.

I would like to thank all the members and graduates of our laboratory, Kusumoto Laboratory.

I would like to express my sincere appreciations to the clerks of our laboratory, including Ms.Tomoko Kamiya and Ms.Kaori Fujino. Their kind support has been quite helpful for me to prepare this dissertation.

I owe deep debts of gratitude to my contemporaries at our laboratory, Mr.Kazuki Kobayashi, Mr.Tatsuya Fujikawa, and Ms.Yuko Muto, for their kind assistance and helpful comments. Loosing any of them must make it impossible to complete this work because their assistance has strongly helped and encouraged me.

My heartfelt thanks go to Ms.Yui Sasaki for her valuable comments, helpful suggestions, and close cooperation throughout this work, especially for the work described in Chapter 3. It must be impossible to accomplish this work without her kind assistance and great contributions.

I would like to express my appreciation to Dr.Takeshi Nagaoka for his help to prepare this dissertation. I also would like to thank Ms.Yukiko Sano for her coaching and assistance, especially for the work described in Chapter 3. I would like to appropriate Mr.Masayuki Owashi, Mr.Akihiko Ito, Mr.Masataka Ugumori, Mr.Kensuke Tanaka, Mr.Tetsuaki Nakamura, and Mr.Keizo Miyamoto for their helpful suggestions and valuable advice. I learned many things from them, including how to work on research problems and how to enjoy the work.

I would like to express my gratitude to Mr.Shinya Yamada for his coaching, warm encouragement, and helpful advice. Moreover, I am deeply grateful to Mr.Katsuma Ubukata, Ms.Tomoko Kanemitsu, and Mr.Minoru Nishino for their valuable discussions for this work. I also would like to thank Mr.Toshiaki Tanaka and Mr.Kiyoyuki Miyazawa for their valuable technical advice. I could not finish this work without kind support from the above individuals.

I am grateful to the members of our research group, including Mr.Tomoya Ishihara, Mr.Shuhei Kimura, Mr.Hiroaki Murakami, Mr.Jiachen Yang, Ms.Ayaka Imazato, and Mr.Noa Kusunoki for their valuable comments and kind support. Their technical advice has been very helpful to prepare this dissertation, and their assistance has encouraged me many times.

Moreover, I would like to appreciate Mr.Yoshihiro Nagase, Mr.Kentaro Hanada, Mr.Kazuki Yoshioka, Mr.Yoshitomo Okada, Mr.Akiyoshi Taguchi, Mr.Takuya Yasunaga, Mr.Koichi Umekawa, Mr.Yukihiro Sasaki, and Mr.Hiroaki Shimba for their encouragement. I could have had cheerful days in our laboratory because of their delightful breezinesses.

I feel deep appreciations for all the other members and graduates of our labora-

tory. I think I am very fortunate because I could work with such great colleagues.

I also wish to thank the members and graduates of Inoue Laboratory for their assistance, including Ms.Eunjong Choi, Mr.Yu Kashima, Mr.Masayuki Tokunaga, Mr.Tomoo Masai, Mr.Masakazu Ioka, Mr.Pei Xia, Mr.Akira Goto, and Mr.Yuki Yamanaka. Their technical comments have been very valuable for this work.

Finally, I would like to thank my famirly and all of my friends in or graduated from Osaka University for their kind support and encouragement. I could complete this work because I could have really enjoyed my days in Osaka University, which is owing to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Software evolution refers to the process of developing software initially and repeatedly updating it for various reasons. The term lacks a general definition, but it is used as a substitute of software maintenance [17, 123]. One of the reasons why it refers to specifically the maintenance phase of software development is that the phase consumes a large amount of costs for developing typical software systems [19].

Software maintenance is one of the software life cycle processes [1]. It is a set of activities associated with changes to a software product after it has been delivered to end users. ISO/IEC 14764 presents the following four categories of software maintenance activities [2].

Corrective: reactive modification to correct discovered problems. This category also includes **emergency** maintenance which is defined as unsheduled and temporary maintenance to keep a software system operational.

Adaptive: modification to keep a software product usable in a changed/changing enviroment.

Perfective: modification to improve some aspect of software quality, such as performance, maintainability, or reliability.

Preventive: modification to detect and correct latent faults in a software product before they appear as actual faults.

The role of software systems in social activities has become more and more important, which also inidicates the high importance of software maintenance activities. However, the growth of size and complexity of software products makes

software maintenance more difficult and more burdensome. Czerwonka et al. reported common characteristics of software maintenance as follows [25].

- Software maintenance phase comsumes the majority of resources of software life cycle processes. Vliet said that maintenance phase requires at least 50% of total costs in his book [149].
- It often happens that maintenance tasks are done by people who had not created the software product.
- The maintenance team is typically much smaller than the development team.
- Changes on deployed software sometimes introduce unwanted behavior, which indicates that maintenance activities have a high risk.
- Creating and verifying a fix for deployed software frequently have to be done in a limited time frame.

These characteristics indicate how difficult and how severe software maintenance is. Because of these factors, technologies that make software maintenance more efficient are strongly required in society.

This dissertation focuses on code clones to meet such a challenging requirement. A code clone, or simply a clone, is defined as '*a code fragment that has identical or similar code fragments to one another*'. The presence of code clones is pointed out as a *bad smell* for software maintenance [32]. The reason is that: if we need to make a change in one place, we will probably need to change the others as well, but we sometimes might miss it [98].

The harmfulness of code clones could cause collapses of software projects at the worst case. One of such cases is the collapse of the software project of the Japan Patent Office in the Ministry of Economy, Trade and Industry, Japan. Reuses of existing code that was not well-tested introduced many negative code clones, which was pointed out as one of the causes of the collapse [52]. As just described, the harmfulness of code clones appears in real software projects. For these reasons, code clones have a high level of interest, which makes code clone analysis a hot topic in the research area of software engineering.

In spite of the potential harmfulness of code clones, software systems contain a certain amount of code clones [7]. One of the typical situations where code clones are introduced in software systems is cloning existing code by copy-and-paste operations. Code cloning by copy-and-paste operations should result in creating new code clones, but it has a strong advantage that it allows developers to implement new functions quickly. Developers, therefore, often perform such instant reuses of code [76, 163]. This is a typical reason why code clones exist in software systems

even though the presence of code clones is regarded as a bad smell for software maintenance.

For these reasons, managing code clones is necessary to avoid or reduce negative impacts of code clones and to take advantage of code cloning. Herein, the term 'code clone management' means the whole activities on code clones, including locating, monitoring, tracking or removing them [84]. The eventual goal of this research is to achieve an effective management of code clones for efficient software maintenance.

1.2 Overview of the Research

This dissertation presents the results of four studies, all of which are closely related to management of code clones. The first one is an empirical study on the influences of code clones. The second one improves a conventional technique to track code clones across version histories of source code. The third one is another empirical study on clone evolution, which analyzes genealogies of code clones. The last one is a technique to support removing code clones that cannot be removed easily.

An Empirical Study on the Influences of Code Clones

For efficient management of code clones, it is necessary to know how harmful code clones are. If code clones are not harmful, it should be reasonable for efficient software maintenance to deal with other factors instead of code clones. On the other hand, if code clones have negative impacts on software maintenance, it should be valid to pay attention to code clones. Although some research efforts have been done to reveal the impact of code clones on software maintenance, the argumentation on the harmfulness of code clones is still open.

In this study, we conducted an empirical study on how harmful code clones are. The key idea of this study is that code clones have a negative impact *if code fragments included in at least one code clone are modified more frequently than ones that are not included in any code clones*. We proposed a new metric named Modification Frequency to capture our key idea, which is calculated based on the number of modifications, not the number of changed lines of code. We conducted an experiment on open source software systems with multiple code clone detectors. Our experimental results showed the following findings:

- Code fragments included in code clones are not modified more frequently than ones not included in code clones totally, but there exists some code clones modified frequently.

- Modification frequencies on code clones differ from code clone detectors.
- Modification frequencies on code clones are variable throughout their lifetime.

To summarize our experimental results, it is not always true that code clones have a negative impact on software maintenance, but there exists some instances that have a negative impact. This finding indicates the importance to detect and focus on code clones that affect software maintenance negatively.

Tracking Clones' Evolution across Version Histories

Tracking clones allows us to know how they evolved. Detecting clone evolution, which represents how a code clone evolves, should contribute to a successful clone management. The bases of this are as follows.

- Analyzing clone evolution provides us with phenomena and characteristics of clones that cannot be revealed by analyzing clones at the latest revision of code.
- Clone evolution tells us when and how it was modified, which enables us to find unintended inconsistencies of modifications on code clones. In other words, clone evolution can be used to detect bugs related to code clones.

There exists some techniques to track clones in version histories. Clone Region Descriptor [27] (refered as **CRD**) is one of the well known techniques for clone tracking. CRD tracks clones based on their locations, which allows CRD to surpass the other techniques in change-tolerance on clones. In other words, CRD can track clones that the other techniques cannot.

Although the original CRD-based tracking has good change-tolerance, it has a room for improvement. This study enhances the original CRD-based tracking and realizes better change-tolerance than the original one. Also, this study reveals the reason of clone disappearances with the enhanced CRD. As well as the investigation of clone disappearances, the enhanced tracking will be able to be a basis of the further research on clone evolution.

Analyzing Genealogies of Code Clones with the Enhanced Clone Tracking Technique

Clone tracking allows us to analyze the influences of clones more detail because it can tell how an individual clone evolves. However, because our previous study on the influcences of clones does not consider evolution of them, it cannot

reveal the influcences of individual clones. Although our manual inspections found some instances of clones having negative impacts, our previous study did not reveal how many clones have negative impacts on software evolution. To know the influences of clones more detail, this study analyzes the influences of clones with the clone tracking technique that enhances the CRD-based tracking.

This fine-grained analysis revealed that there are a few code clones that should require much attention of developers or maintainers. This finding empirically supports the results of the first study, which is that not all but a part of clones have negative impacts. Our another finding through this study is that clones tend to be modified more frequently in former halves than in latter halves of their lifetimes. Hence, we can say that it is important to find negative clones in their earlier stages and start dealing with them as soon as possible.

A Support for Removing Code Clones

The two previous studies revealed that *not all but a part of clones has negative impacts on software maintenance*. To achieve an intelligent clone management, it is necessary to handle negative clones.

Removing code clones is one of the effective ways to handle negative clones. However, the activity has not just positive aspects but also negative ones. The negative aspects of removing code clones are as follows.

Risky: To remove code clones from source code, of cource, it is necessary to make some modifications on the source code. Careless removal of code clones may introduce new bugs into the source code.

Costly: Removing code clones requires much effort for software maintainers. They need to locate code clones to be removed, consider how to remove them, and decide whether they should remove the code clones or not.

These characteristics become more noticeble in the case that code clones to be removed have some gaps. Gaps included in code clones make the removal process of the clones more complicated. This means that removing such clones is more risky and more costly. Therefore, techniques to assist removing clones having some gaps are required for software maintainers to achieve such a challenging task.

This study proposed a novel technique to support removing code clones having some gaps. This technique regards wholes of methods having code clones as its target, instead of targetting code clones themselves. Focusing on wholes of methods enables to remove code clones having some gaps easily. By applying the proposed technique to similar methods, code clones existing between them are merged with unique processes of each method remained preserved.

We have implemented the proposed technique as a tool named *CRat*, and confirmed the effectiveness of it through case studies.

1.3 Overview of the Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 refers to current states of research on code clones. It privides a definition of code clones, and introduces some discussions around them. Furthermore, it explores literatures of research on code clones with five categories, including detection, removal, prevention, analysis, and finding bugs related to clones.

Chapters 3, 4, 5, and 6 present the four studies described above. In Chapter 3, we describe the study on stability of code clones. This chapter presents the methodology, the experimental targets, and the results of this empirical experiment, and then it compares the experimental results with other experimental methodologies proposed by other researchers.

Chapter 4 explains a new clone tracking technique based on CRD. It describes our key idea to track code clones more accurately, and comfirms improvements of accuracy of clone tracking through an experiment on two open source software systems. Moreover, it presents an application of the clone tracking technique, which reveals why and how frequently clones are gone.

Chapter 6 presents a refactoring support for code clones. It formally describes the clone removal technique, and then it explains a tool implementation of the proposed method. In addition, it validates the usefulness of the proposed method through two experiments, one of which is on open source software systems, and the other one is with some subjects.

Finally, Chapter 7 concludes this dissertation with a summary and directions for future work.

Chapter 2

A Survey on Code Clone Management

2.1 Code Clones

2.1.1 Definition

A code clone is a set of code fragments that are similar to each other according to some definitions of similarity [15]. As this definition expresses, there exists a vagueness on the definition of code clones. That is, there is no formal or generic definition of them. Therefore, every clone detector has its own definition of code clones [43,54,131]. Establishing a suitable definition of code clones is still an open issue for all the researchers who are intereseted in them [84].

Here, we introduce some terms around code clones; clone relationship, clone pair, cloned fragment, and clone set. Let α and β be code fragments, and assume that they are *similar* to each other. Then, it is defined that there is a clone relationship between α and β. In addition, a pair of the two code fragments (α, β) is called a clone pair, and each code fragment of which a clone pair consists is called a cloned fragment. Moreover, a set of code fragments is called a clone set if every code fragment included in the set is *similar* to all the other fragments in the set. Note that clone set has alternative names such as clone class or clone group. This dissertation, however, uses clone set among its alternatives.

Figure 2.1 shows a simple example of clone pairs and clone sets. This example has three code fragments, α, β, and γ, and they have clone relationships to each other. Hence, there are three clone pairs (α, β), (β, γ), and (α, γ) in this example. In addition, there is a clone set that consists of the three cloned fragments α, β, and γ.

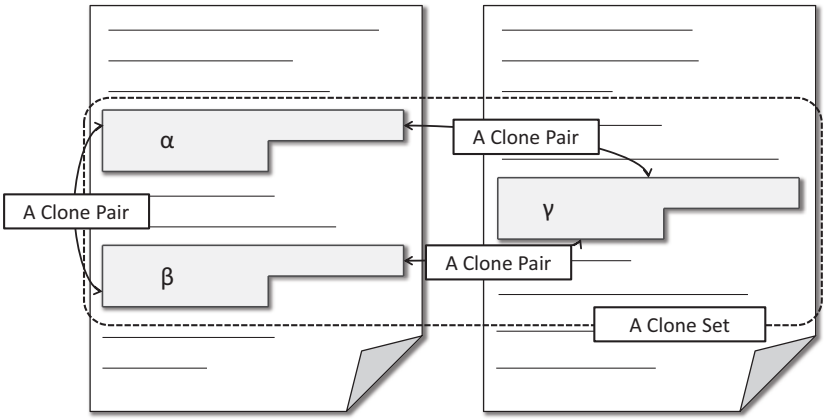


Figure 2.1: Clone Pair and Clone Set

2.1.2 Types

Code clones can be categorized into the following three types by the degree of their similarities [16].

Type-1: An exact copy except for white spaces and comments.

Type-2: Syntactically identical copy; only identifiers of variables, types, literals, or functions were changed.

Type-3: A copy with further modifications in Type-2; statements were changed, added, or removed.

Adding to the traditional three types, some researchers have proposed another type of clones, which is referred as Type-4 clones. The following is the definition of Type-4 clones [131].

Type-4: Fragments perform the same computation but implemented through different syntactic variants.

Type-1, Type-2, and Type-3 clones can be said as *syntactically* similar, whereas Type-4 clones can be said as *semantically* similar.

Figure 2.2 describes each type of clones. Figure 2.2(a) represents the original code fragment, and the other fragments shown in Figures 2.2(b), 2.2(c), 2.2(d), and 2.2(e) are clones of the original fragment.

```
1: public static int sum(int[] a) {
2:   int result = 0;
3:   for (int i = 0; i < a.length; i++) {
4:     result += a[i];
5:   }
6:   return result;
7: }
```

(a) Original Fragment

```
1: public static int sum(int[] a) {
2:     int result = 0;
3:
4:     for (int i = 0; i < a.length; i++) {
5:         // adding a[i] into result
6:         result += a[i];
7:     }
8:
9:     return result;
10: }
```

(b) Type-1 Clone

```
1: public static int sum(int[] a) {
2:   int s = 0;
3:   for (int j = 0; j < a.length; j++) {
4:     s += a[j]; // adding a[j] into s
5:   }
6:   return s;
7: }
```

(c) Type-2 Clone

```
1: public static int sum(int[] a) {
2:   int s = 0;
3:   for (int i = 0; i < a.length; i++) {
4:     s += a[i];
5:   }
6:   // printing the result
7:   System.out.println("the sum is " + s);
8:   return s;
9: }
```

(d) Type-3 Clone

```
1: public static int sum(int[] a) {
2:   return calc(a, 0);
3: }
4:
5: public static int calc(int[] a, int i) {
6:   if (i < a.length) {
7:     return a[i] + calc(a, i + 1);
8:   } else {
9:     return 0;
10:  }
11: }
```

(e) Type-4 Clone

Figure 2.2: Types of Clones

The code fragment in Figure 2.2(b) is an exact copy of the original one except for the different layout and the presence of a comment. This means that the code fragment in Figure 2.2(b) and the original one consists of a Type-1 clone. Figure 2.2(c) shows another code fragment having some differences compared to the one in Figure 2.2(a). In addition to the presence of a comment, this code fragment has different names of variables from the original one. Hence, the code fragment in Figure 2.2(c) is a Type-2 clone of the original one. Figure 2.2(d) presents a code fragment having more gaps than the one in Figure 2.2(c). This code fragment has not only differences of variable names and comments but also a statement that the original fragment does not have. Therefore, the clone shown in 2.2(d) is a Type-3 clone.

As these figures show, code fragments shown in Figures 2.2(b), 2.2(c), and 2.2(d) are textually similar. In contrast, the one in Figure 2.2(e) is no longer similar to the original in the text level. However, the method `sum` in Figure 2.2(e) and the original method `sum` in Figure 2.2(a) compute the same values from the same

arrays. In other words, these two methods functionally equal to each other. This is an example of Type-4 clones that are *semantically* similar to each other.

2.2 Discussions on Code Clones

2.2.1 Causes of Creation

Code clones are created into software systems by a variety of reasons. Baxter et al. listed the followings as the reasons [15].

Code reuse by copying pre-existing idioms: Code cloning by developers will introduce code clones into software systems. This is the most popular situation that code clones are created. Code reuse by copy-and-paste operations is a common practice in software development, because it is quite easy, and it enables to make software development faster.

Coding styles: Some functions should be coded in accordance with standard styles. Error reporting or user interface displays will be typical instances of such functions. These styles sometimes create clones, because functions that are coded in the same style have a high possibility that they are similar to each other.

Instantiations of definitional computations: Many of simple and frequently used functions will be repeated in software systems even when copying is not used, which results in the presence of clones. Payroll tax, queue insertion, or data structure access are instances of such functions.

Failure to identify/use abstract data types: Some programming languages do not support abstract data types. If software systems are written in such languages, functions intended for use on another data structure of the same type will be duplicated. Baxter et al. reported that "*we have founded many systems with poor copies of insertion sort on different arrays scattered around the code*", which is an instance of clones created by this reason.

Performance enhancement: Some clones are introduced intentionally for performance reasons. Code abstraction, such as extracting existing features as new methods, will reduce code duplications, but may introduce overheads of execution. Hence, developers of systems having tight time constraints sometimes repeat the same functions without abstracting them to avoid occurences of such overheads.

Accidents: Different developers may write similar code accidentally. However, it is rare that the amount of similar code generated accidentally becomes high.

In addition, using code generation tools, plagiarism of code, and padding the amount of code by intentional cloning are pointed out as causes of clone creation [71].

Among a variety of reasons, code cloning by developers is the most typical cause of clone creation.

Kapser and Godfrey categorized patterns of cloning into the following four groups [74].

Forking: cloning the existing code as a 'springboard' of development of similar solutions. A case of this group of cloning is that developers create a new driver for a hardware family by cloning the existing driver and tailor it to new one. In this cloning group, cloned fragments will evolve independently from the original code.

Templating: cloning when the desired behavior is already known and an existing solution closely satisfies this need. One simple example is to achieve the same behavior for `float` and `short` in the C programming language. Cloned fragments created by patterns of cloning categorized into this group will evolve closely to the original code.

Customization: cloning when existing code solves a very similar problem to the current development problem, but it cannot be used to solve the current problem as is. In such a case, developers clone the existing code, and make some customizations on it to handle the current problem.

Exact Matches: cloning to solve a particular problem repeated with in a software system. Cloning categorized into this group occurs in the case that the cloned code is too small or there is little worthwhile to abstract the solution of the problem. One of the examples is cross-cutting concerns such as logging or debugging.

Some researchers revealed why code cloning occurs in development of experimental software systems.

Kim et al. investigated the reasons why developers clone code with researchers in IBM Corporation [76]. Their research showed that some clones were created by unavoidable reasons including limitations of programming languages.

As well as Kim et al.'s investigation, Zhang et al. analyzed the situations where code cloning occurs in a commercial software system [163]. Their interviews for developers of the system revealed that close to half of developers claimed that they "often" clone code by copy-and-paste, and almost all of the remaining subjects claimed that they "sometimes" do code cloning. Furthermore, they found that developers copy existing code by not only technical reasons, including "Avoiding

breaking existing features" and "Difficult to be reused", but also the presence of time limitations or some organizational reasons, including "Issues of code ownership".

Summarizing these findings, ad hoc reuse of the existing code by code cloning is not only a common practice but also sometimes unavoidable one. Hence, it is impossible to avoid creating clones in software systems completely.

2.2.2 Harmfulness

At present, there is a huge body of work on empirical evidence on code clones, starting with Kim et al.'s report on clone genealogies [78]. They have conducted an empirical study on two open source software systems, and found that 38% or 36% of groups of code clones were consistently changed at least once. On the other hand, they observed that there were groups of code clones that existed only for short periods (five or ten revisions) because each instance of the groups was modified inconsistently. Their work is the first empirical evidence that a part of code clones, not all of them, increases costs required for modifications on source code.

However, Kasper and Godfrey have different opinions regarding code clones. They reported that code clones can be a reasonable design decision based on an empirical study on two large-scale open source systems [74]. They built several patterns of code cloning in the target systems, and they discussed the pros and cons of code cloning using the patterns. Bettenburg et al. also reported that code clones do not seriously affect software quality [18]. They investigated inconsistent changes to code clones at the release level on two open software systems, and they found that only 1.26% to 3.23% of inconsistent changes introduced software errors into the target systems.

Monden et al. investigated the relation between software quality and code clones on the file level [115]. They used the number of revisions of every file as a barometer of quality: the larger the number of revisions of a file is, the lower its quality is. They selected a large scale legacy system, which had been being operated in a public institution, as the target of their experiment. The result showed that, modules including code clones were 40% lower quality than modules not including code clones. Moreover, they reported that the more code clones a source file included, the lower quality it has.

Lozano et al. investigated whether the presence of code clones was harmful or not [100]. They developed a tool that traces which methods include code clones (in short, duplicate method) and which methods are modified in each revision. They conducted a pilot study, and found that: duplicate methods tended to be more frequently modified than non-duplicate methods; however, duplicate methods tended

to be modified less simultaneously than non-duplicate methods. Their findings imply that the presence of code clones increased costs for modifications, and programmers were not aware of code duplications, so that they sometimes overlooked code fragments that had to be modified simultaneously.

Also, Lozano and Wermelinger investigated the impact of code clones on software maintenance [101]. Three barometers were used in the investigation. The first one is *likelihood*, which indicates the possibility with which a method is modified in a revision. The second one is *impact*, which indicates the number of methods that are simultaneously modified with a method. The third one is *work*, which can be represented as a product of *likelihood* and *impact* ($work = likelihood \times impact$). They conducted a case study on four open source systems for comparing the three barometers of methods including and not including code clones. The result showed that: *likelihood* of methods including code clones was not so different from one of methods not including code clones; there were some instances that *impact* of methods including code clones were greater than one of methods not including code clones; if code clones existed in methods for a long time, their *work* tended to increase greatly.

Moreover, Lozano et al. investigated the relation between code clones, features of methods, and their changeability [103]. Changeability means the ease of modifications. If changeability decreased, it will be a bottleneck of software maintenance. The result showed that the presence of code clones can decrease changeability. However, they found that changeability was more greatly affected by other properties such as length, fan-out, and complexity of methods. Consequently, they concluded that it was not necessary to consider code clones as a primary option.

Krinke hypothesized that if code clones are less stable than code fragments that are not included in any code clones, maintenance costs for cloned code are greater than those for non-cloned code. He conducted a case study in order to investigate whether the hypothesis is true or not [88]. The targets were 200 revisions (a version per week) of source code of five large scale open source systems. He measured *added*, *deleted*, and *changed* LOCs on cloned code and non-cloned code, and compared them. He reported that non-cloned code was more frequently *added*, *deleted*, and *changed* than cloned code. In other words, code clones are more stable than non-cloned code. Consequently, he concluded that the presence of code clones did not necessarily make it more difficult to maintain source code.

Göde et al. replicated Krinke's experiment [34]. Krinke's original experiment adopted line-based approach whereas their experiment adopted token-based approach. The experimental result was basically the same as Krinke's one: cloned code was more stable than non-cloned code in the viewpoint of addition and change. On the other hand, from the viewpoint of deletion, non-cloned code is more stable than cloned code.

Also, Krinke conducted an empirical study to investigate ages of code clones [89]. In this study, he calculated and compared average ages of cloned code and non-cloned code on four large-scale Java software systems. He found that the average age of cloned code is older than non-cloned code, which implies cloned code is more stable than non-cloned code.

Rahman et al. investigated the relationship between code clones and bugs [128]. They analyzed four software systems written in C language with bug information stored in Bugzilla. They used Deckard, which is an AST-based clone detector, and reported that only a small part of bugs located on code clones, and the presence of code clones did not dominate bug appearances.

Göde modeled how Type-1 code clones were generated and how they evolved [33]. He investigated how code clones evolved with nine open source software systems. The result showed that: the ratio of code duplications was decreasing as time passed; the average lifetime of code clones was over a year; in the case that code clones were modified inconsistently, there were a few instances that additional modifications were performed to restore their consistency.

Also, Göde et al. conducted an empirical study on clone evolution and performed a detailed tracking to detect when and how clones had been changed [35]. In their study, they traced clone evolution and counted the number of changes on each clone genealogy. They manually inspected the result in one of the target systems, and categorized all the modifications on clones into consistent or inconsistent. In addition, they carefully categorized inconsistent changes into intentional or unintentional. They reported that almost all clones were never changed or changed only once during their lifetimes, and only 3% of the modifications had high severity. Therefore, they concluded that many of clones do not cause additional change effort, and it is important to identify clones having high threat potential to manage code clones effectively.

To summarize these experimental results, some empirical studies reported that code clones should have a negative impact on software evolution whereas the others reported the opposite result.

The former, which claims that clones are harmful, is summalized as follows.

- There exists many clone related bugs [65, 98].
- Methods having clones are more frequently modified than those not having clones [100, 102].
- Methods having clones require more maintenance costs than those not having clones [101, 114].
- Modules having clones have less maintainability than those not having clones [115].

On the other hand, the latter, which states clones are not harmful, is summalized as follows.

- Many clones live for only a short time period [78].
- Lack of modifing correspondents of cloned fragments rarely occur [5, 18, 35, 135, 143].
- Most of code clones have no relation to bugs [128].
- Clones are modified less frequently than non-cloned code [34, 38, 88, 89].
- There exists little clones that affect software maintenance (such as having the same bugs) [74].

At present, there is no consensus on the impact of the presence of code clones on software evolution. However, these variabilities of investigational results indicate that not all of clones are harmful, but a part of them negatively affect software maintenance.

2.3 Management of Code Clones

2.3.1 Needs

As described in 2.2.1, code cloning is a common practice in actual software development. Moreover, in some cases, it is unavoidable because of limitations of time, programming languages, or organizations. In addition, some researchers claimed that most of clones cannot be easily removed [77, 78].

These facts indicate that it is no longer possible to completely prohibit clones from existing in software systems.

Furthermore, as described in 2.2.2, not all but a part of clones has negative impacts on software maintenance. This indicates that it is not effective to spend much effort to prevent or remove clones that are not harmful [150].

Therefore, software maintainers should pay their attention to harmful clones by preventing or removing them, or by assessing the presence of such harmful clones. In other words, managing code clones is necessary to achieve effective software evolution.

2.3.2 Definition

Because of the abstractiveness of the term “clone management”, there can be a variety of its definitions. This dissertation gets into line with the definition that

Rainer Koschke stated in the literature [84]. He defined clone management as follows.

> "*Software clone management* comprises all activities of looking after and making decisions about consequences of copying and pasting."

This definition includes not only detetion or removal of clones but also all the other activities that are concerened about code clones.

This dissertation categorizes all the activities of clone management into the following five categories.

- Detection
- Removal
- Prevention
- Analytic methodology
- Detection and prevension of clone related bugs

Research achievements in each category are introduced in the following subsections.

2.4 Detection

2.4.1 Overview

Clone detection is the most hot topic in the research area of code clones. Clone detection reports the information where code clones are in target software systems. The role of clone detection is quite important because many of research or techniques of clone management stand on it. Hence, great effort has been spent on clone detection, which has yielded many techniques and tools to automatically detect code clones from source code [43, 54, 71, 83, 129, 131].

Detectors of code clones can be classified into the following categories [16,43].

- Text-based
- Token-based
- Tree-based
- Graph-based
- Others

The remainder of this section describes each category in detail.

2.4.2 Text-based Techniques

Text-based detection techniques detect code clones by comparing every line of code as a string. They detect multiple consecutive lines that match in specified threshold or more lines as code clones. The biggest advantage of this technique is that it can detect code clones quickly compared with other detection techniques. This technique requires no pre-processing on source code, which realizes the fast detection. However, detectors based on this technique cannot detect code clones including differences of coding styles (e.g. whether long lines are divided into multiple lines or not).

The method proposed by Johnson [64] and the method proposed by Ducasse et al. [28] are instances of text-based clone detectors. In these methods, every line of code is compared after white spaces and tabs are removed. These methods are language-independent because they compare lines of code textually.

Simian is one of the well-used text-based clone detectors [140], which can handle many programming languages. *DuDe* is another text-based clone detector that detects chains of smaller extract clones [154]. *SDD* is a scalable detector, which generates index and inverted index for code fragments and their positions and finds similar fragments with an n-neighbor distance algorithm [95].

2.4.3 Token-based Techniques

In the token-based approach, source code is lexed/parsed/transformed to a sequence of tokens. This technique detects common subsequences of tokens as code clones. Compared to the text-based approach, the token-based approach is usually robuster against code changes such as formatting and spacing. Detection speed is inferior as compared with text-based techniques, meanwhile superior as tree- or graph-based approaches. This is because, in token-based approach, source code need not to be transformed into intermediate representations.

One of the famous clone detectors categorized into the token-based detectors is *CCFinder*, developed by Kamiya et al. [72]. It has been widely used among researchers or engineers. A variety of enhancements has been proposed on *CCFinder*, including an extended version named *CCFinderX* [21] and a distributed version named *D-CCFinder* [99].

CP-Miner is also a token-based detector. *CP-Miner* has been developed by Li et al. [98]. *CP-Miner* calculates a hash value from every statement, and then it applies a frequent pattern mining algorithm to detect code clones [4]. Frequent patterns do not have to be consecutive, which means that *CP-Miner* can detect Type-3 clones.

RTF, a clone detector proposed by Basit et al, uses suffix arrays on tokens to

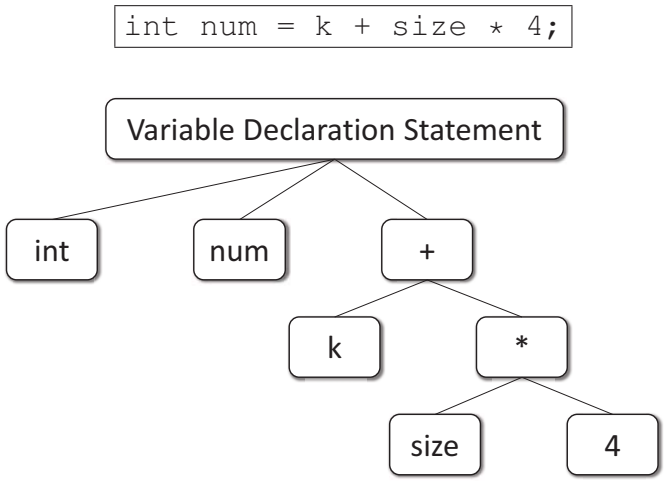


Figure 2.3: An Example of ASTs

reduce costs for clone detection [13].

Kawaguchi et al. developed *SHINOBI*, a token-based clone detector on an IDE [75]. It works as an add-in of Microsoft Visual Studio, and it applies a token-based clone detection engine, which is similar to *CCFinder*.

2.4.4 Tree-based Techniques

In the tree-based detection, a program is parsed to a parse tree or an abstract syntax tree (in short, AST) with a parser of the language in interest. An AST is one of the intermediate representations that capture structures of source code. Figure 2.3 shows an example of ASTs. Tree-based techniques regard common subtrees as code clones. This approach considers the structure of source code, therefore tree-based detectors do not detect code clones ignoring the structure of source code such as code clones including a part of a method and a part of another method. However, a disadvantage of this approach compared with text- and token-based approaches is that it requires more costs including both of time and memories for its clone detection because of the additional costs required to transform source code to parse trees or ASTs.

One of the pioneers of AST-based clone detectors is *CloneDR* developed by Baxter et al. [15, 23]. *CloneDR* compares subtrees of ASTs by characterization metrics based on a hash function through tree matching, instead of comparing subtrees of ASTs directly. This processing allows *CloneDR* to detect code clones quickly from large software systems. It can handle many programming languages.

```
  fp3 = lookaheadset + tokensetsize;
  for (i = lookaheads(state); i < k; i++) {
%   fp1 = LA + I * tokensetsize;
%   fp2 = lookaheadset;
%   while (fp2 < fp3)
%       *fp2++ |= fp1++;
  }
```

(a) Code Fragment 1

```
  fp3 = base + tokensetsize;
  …
  if (rp) {
    while ((j = *rp++) >= 0) {
      …
#     fp1 = lookaheadset;
#     fp2 = LA + j * tokensetsize;
#     while (fp1 < fp3)
#         *fp1++ |= *fp2++;
    }
  }
```

(b) Code Fragment 2

Figure 2.4: A Clone Pair with a Differnt Order of Statements

Moreover, it has a function to assist clone removal.

DECKARD is another well-used clone detector based on tree comparisons [61]. It applies a locality sensitive hashing algorithm [26] to detect code clones, which allows *DECKARD* to detect Type-3 code clones.

Koschke et al. developed *clones* [85], which is also a tree-based approach as well as *CloneDR* and *DECKARD*. It compares ASTs with a suffix tree algorithm to have an increase of detection speed.

2.4.5 Graph-based Techniques

In the graph-based approach, code clones are detected by comparing graphs created from source code. Isomorphic subgraphs are regarded as code clones in this approach. Program dependence graph (in short, PDG) is one of the well-used graphs to detect code clones, which is a directed graph that represents dependencies between elements of programs [31, 153].

These graphs require semantic analyses for their creation, therefore this approach requires much more costs including both of time and memories than the other detection techniques. However, this technique can detect code clones with additions/deletions/changes in statements or code clones including some differences that have no impact on the behavior of program. This is because graph-based techniques can consider the meanings of program.

Figure 2.4 shows one of the code clones that include some differences that have no impact on the behavior of programs. Other techniques cannot detect these two code fragments as a code clone because there is a different order of statements.

One of the leading graph-based clone detection methods is the one proposed by Komondoor and Horwitz [81]. Their method detects isomorphic subgraphs of PDGs with program slicing. Krinke's method [86], and Higo et al.'s method [42, 139] are also included in graph-based techniques. Each detection method is optimized to reduce detection cost. Krinke sets a limit of search ranges of PDGs

with a threshold. By contrast, Higo et al. confine nodes to be bases of subgraphs with some conditions. Moreover, Higo et al. use execution dependence, which enables detecting code clones that other graph-based methods could not detect.

2.4.6 Other Detection Techniques

One of the detection techniques that can be categorized into this category is a metrics-based approach. First, metrics-based detectors calculate metrics on every program module (such as files, classes, or methods), then detect code clones by comparing the coincidence or the similarity of these values. *CLAN* is a clone detector categorized into the metrics-based approach [109]. *CLAN* finds duplications by comparing metrics obtained from ASTs created from source code. Kontogiannis et al. developed another metrics-based detector [82], which uses ASTs to calculations of metric values as well as *CLAN*. Ottenstein proposed an approach to detect plagiarisms of source code with metrics [126]. Lanubile and Mallard developed a metrics-based approach to find functional clones in web applications [91]. Kodhai et al. combined metrics-based approach and texual comparison of source code to detect Type-1 and Type-2 functional clones [79].

Beside this, there are some file-based detection methods [125, 138]. This detection technique detects code clones by comparing every file instead of statements or tokens, which let it be one of the most scalable approach to detect clones. However, this technique cannot find code clones that exist in a part of a file. In addition, there exists method-based techniques that are more fine-grained compared with file-based techniques [56]. Although these techniques are inferior to file-based ones in the point of detection speed, they can detect code clones that file-based techniques miss.

Moreover, incremental detection techniques are under intense studies [36, 47, 51, 120]. The incremental clone detection keeps results of clone detection in previous revisions and their intermediate products, and it uses them in the next detection of code clones. Reusing results and intermediate products of analyses on previous revisions can reduce cost of the clone detection on the current revision substantially.

There exists some techniques not classified into the above categories. The followings briefly describe them. *NICAD* is able to detect Type-3 clones on the levels of methods or blocks by using pretty printing and an algorithm to detect longest common subsequences [122, 132]. Basit and Jarzabek proposed a technique to detect higher-level clones than code fragments, which means structual clone detection [11, 12]. Murakami et al. developed a clone detector named *FRISC*, which uses hash sequences created from statements to detect code clones [116]. *FRISC* holds repeated instructions as a pre-processing of clone detetion to eliminate false

positives and detect clones that other detectors cannot detect. Murakami et al. also developed another detector that is also based on comparison of hash sequences, which is named *CDSW*. [117] *CDSW* uses a well-known algorithm, the Smith-Waterman algorithm [141], that finds similar alignments between two sequences. They optimized the Smith-Waterman algorithm for the purpose of clone detection. Li and Thompson combined tokens and ASTs to detect and remove code clones [96]. Maeda proposed a technique for clone detection using PALEX [105]. PALEX is a representation of source code that contains recorded parsing actions and lexical formatting information, and using it enables his technique to be language independent.

2.4.7 Comparison and Evaluation of Clone Detectors

Comparing and evaluating clone detectors plays an important role to identify an efficient clone detector. However, lacking generic or strict definitions of code clones makes it challenging to compare and to evaluate clone detectors. Furthermore, suitable detectors should vary with individual purposes or individual users. Therefore, it is impossible to decide what is the best detector among all the clone detectors.

There are some empirical studies for comparison and evaluation of clone detectors [16, 20, 30, 85, 133, 134, 144]. Rattan et al. reported some general remarks among the above studies through their systematic review as follows [129].

- Token-based detectors have high recall and reasonable precision because they detect a large number of clones.
- Tree-based detectors detect less number of clones than token-based detectors, which results high precision and low recall compared to token-based detectors.
- Metrics-based detectors have good precision but low recall because less number of clone candidates are detected.
- Graph-based tools is robust for Type-3 clone detection, but they suffer from high time complexity.

2.5 Removal

Clone removal is also an active research area as well as clone detection, and so many researchers proposed techniques to remove clones from source code [48]. Basically, clone removal stands on *refactoring*. Refactoring means changing the

internal structures of software systems without changing their external behavior. Because clone removal must not change the behavior of its target software, it can be said that clone removal is a kind of refactoring.

First, this section describes what is refactoring, and then refers how to remove clones, and finally introduces research achievements on clone removal after that.

2.5.1 Refactoring

Definition

Refactoring is one of the frequently performed activities in the maintenance phase. Fowler, a pioneer of refactoring, defined refactoring as "*the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [32]. It has been reported that the maintainability of software systems decays over time [29]. A set of suitable refactorings could be beneficial to prevent decays of software maintainability. However, applying refactorings requires much effort for software maintainers, and it is quite difficult for maintainers to apply refactorings manually without introducing any human errors because of the difficulty of applying refactorings [118].

Because of these factors, techniques to assist refactoring activities are required. This fact makes refactoring as one of the hot topics in the research area of software engineering [111].

Process

The refactoring process consists of several activities as follows [111]:

1. Identify places that should be refactored,
2. Determine which refactroing(s) should be applied to the places,
3. Guarantee that the behavior of the program is preserved by the selected refactoring(s),
4. Apply the refactoring(s),
5. Assess the effect of the refactoring(s) on quality of the software or the process and
6. Maintain the consistency between the refactored program and other software artifacts (e.g. documentation, design documents, requirements specification, tests).

Each of these activities can be supported by different tools, techniques or formalisms.

Behavior Preservation

Refactoring must not change the behavior of program accoring to its definition.

The original definition of behavior preservation is suggested by Opdyke [124]. The definition states that, for the same set of input values, the resulting set of output values should be the same before and after the refactoring. However, requiring the preservation of input-output behavior is insufficient, since many other aspects of the behavior may be relevant as well. For example, in the case of *real-time software*, an essential aspect of the behavior is the execution time of certain operations. Thus, refactoring must preserve all the kinds of temporal constraints. For *embedded software*, memory constraints and power consumption are also important aspects.

Another pragmatic way to guarantee the behavior preservation is using test suites. This means that if all the test suites are passed before and after refactorings, it is regarded that the refactorings do not affect the behavior of the program. If sufficient test suites are prepared, the fact that all the test suites still pass after the refactorings will be a good evidence that the behavior of the program is preserved.

Another approach is to formally prove that refactorings preserve the full program semantics. The bahavior preservation can be formally proved if a language with a simple and formally defined semantics is used in the target software systems. However, it is difficult to prove the behavior preservation for more complex languages such as C++.

2.5.2 Refactoring Patterns Used for Code Clone Removals

This subsection describes refactoring patterns that can be used for clone removal, all of which were proposed by Fowler [32, 44].

Extract Class/SuperClass

Extract Class indicates extracting a part of a class as a new class. If there is a large and/or complex class, the class requires much costs for its maintainance. Extract Class is useful in such a case. If there is a class-level duplication, Extract Class will remove code clones included in the duplicated classes. Figure 2.5 shows an example of refactoring with Extract Class. In this case, there are duplicate fields `officeAreaCode` and `officeNumber`, and duplicate operation about them. By applying Extract Class to this example, duplicate fields and duplicate operation are extracted as a new class `TelephoneNumber`, and the classes

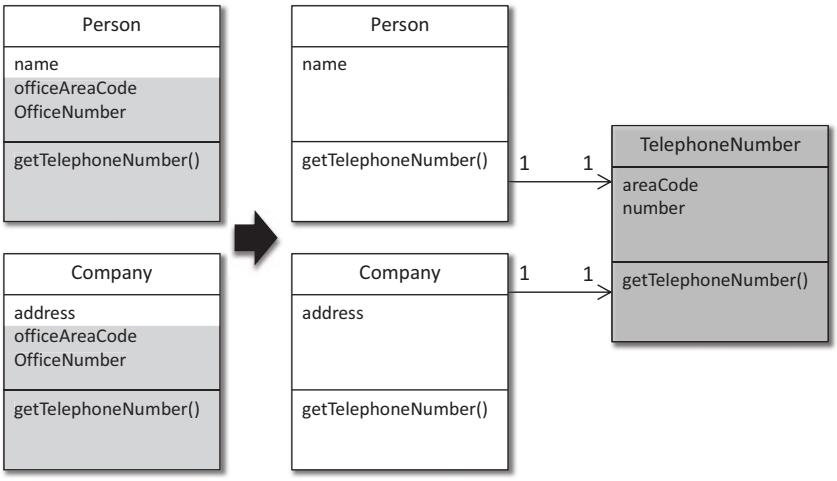


Figure 2.5: An Example of Extract Class

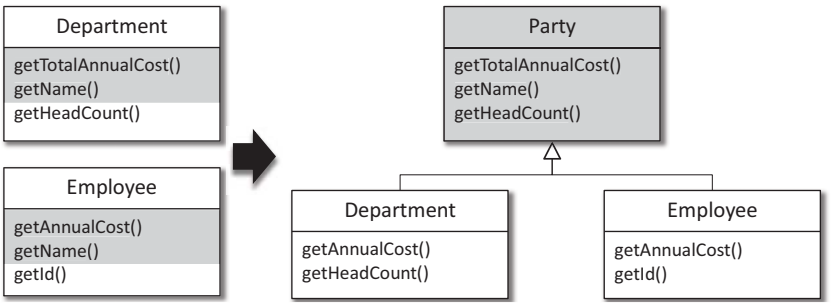


Figure 2.6: An Example of Extract SuperClass

`Person` and `Company` uses the class. By this modification, the duplicate code is removed from the two classes.

If duplicate classes do not extend different base classes, Extract SuperClass may be a better solution for clone removal. Extract SuperClass is similar to Extract Class. The difference is that Extract SuperClass uses inheritance; meanwhile Extract Class uses delegation. In Extract SuperClass, duplications between two (or more) classes are extracted as a new class and all the original classes are changed to extend the new class. Figure 2.6 shows an example of the application of Extract SuperClass. In this example, a new class `Party` is created by extracting the duplication of two classes `Department` and `Emproyee`, then the two classes are changed to extend the class `Party`.

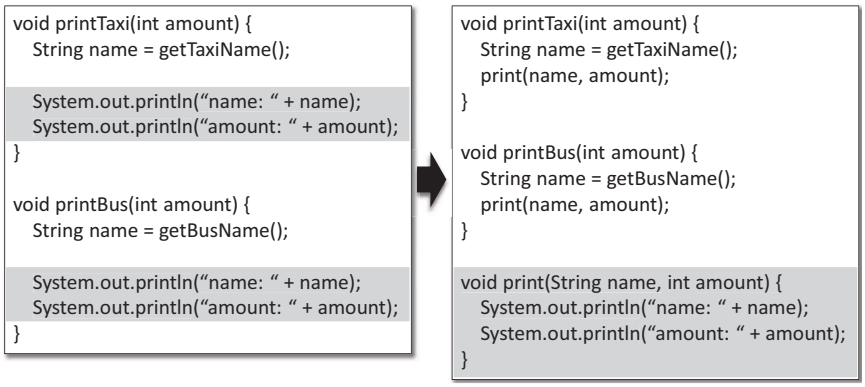


Figure 2.7: An Example of Extract Method

Extract Method

Extract Method indicates extracting a part of a method as a new method. This refactoring pattern is often used for improving reusability by segmentalizing too long and/or too complex methods into short and simple methods. Extract Method can remove code clones by extracting them as a new method and replace them by method call instructions for the method. Figure 2.7 shows an example of clone removal with the application of Extract Method. In this example, there are the same statements between two methods `printTaxi` and `printBus`. By applying Extract Method, these duplicate statements are extracted as a new method `print`, and the original statements are replaced by the method call. As a result, code clones between the two methods are merged into a single method. An advantage of this pattern as clone removal technique is that it can be applied if a part of a method contain code clones and the other part does not contain code clones. In addition, this pattern is capable of wide applications because it does not use class hierarchies. Therefore, this pattern is useful in such a case that versatile processes, which can be merged as a library, are scattered across source code as code clones. However, this pattern introduces many methods if multiple cloned fragments exist in a single method and there are some non-cloned fragments between every two fragments.

Pull Up Method

Pull Up Method indicates pulling up identical methods existing in derived classes into their common base class as a new method. This pattern is effective if there are some methods that behave the same way in all the derived classes. By applying this pattern, duplicate methods are merged into a base class, which

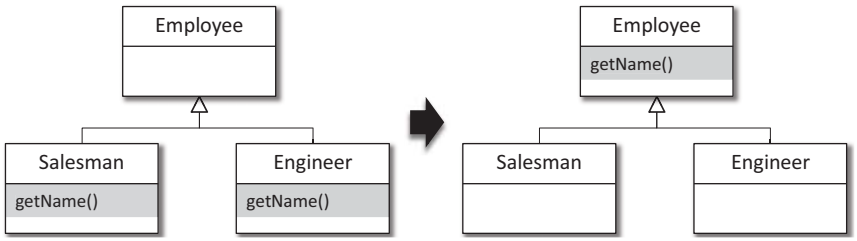


Figure 2.8: An Example of Pull Up Method

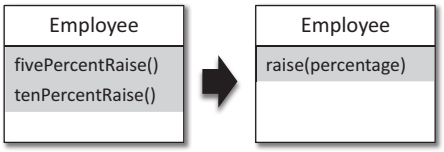


Figure 2.9: An Example of Parameterize Method

means that code clones existing in derived classes are removed. Figure 2.8 shows an example of the application of Pull Up Method. In this case, two duplicate methods `getName` in class `Salesman` and `Engineer` are pulled up into the same base class `Employee`. This pattern can be applied if and only if target methods are exactly the same. Moreover, this pattern uses inheritance relationships among classes. Therefore, the range of application of this pattern is narrower than that of Extract Method refactoring pattern.

Parameterize Method

If there are similar methods in a single class, the duplication may be removed by Parameterize Method. Parameterize Method is used in a case that several methods do similar things but with different values contained in the method body. In this pattern, a new method that uses a parameter for different values is created. Figure 2.9 shows an example of Parameterize Method refactoring. The class `Employee` before refactored in this example has two similar methods, named `fivePercentRaise` and `tenPercentRaise`. A Parameterize Method refactoring creates a new method that uses a parameter for different values between the two methods, which removes clones between them.

Pull Up Constructor

This pattern is very similar to the Pull Up Mehtod. The only difference is the target of this pattern is not a method but a constructor.

Replace Method with Method Object

This pattern is a hybrid of Extract Class and Extract Method refactoring patterns. This pattern is used in the case that there exists long and similar methods that use local variables, which make it difficult to extract these methods by Extract Method refactoring pattern. This patten applies Extract Class refactoring pattern to such methods. Extract Class replaces local variables into fileds of the extracted class, and so Extract Method can be applied easily after Extract Class refactoring is applied.

Form Template Method

Form Template Method refactroing pattern is a hybrid of Extract Method and Pull Up Method refactoring patterns. This pattern targets similar methods existing in derived classes that have a same base class. In this pattern, processes that are common in all the target methods are pulled up into the base class with Pull Up Method refactoring pattern. On the other hand, the processes that are not common in the target methods remain in each derived class. The remaining processes are unique in each derived classes. These unique processes are extracted as a new method with Extract Method refactoring pattern.

The steps for applying Form Template Method are as follows:

1. Detect common processes in all the target methods,
2. Extract unique processes as new methods with Extract Method refactoring pattern,
3. Rename methods to make correspondence of signatures. The targets of renaming are methods that created in 2. and called in the same point of the common processes and
4. Pull up common processes as a new method in the base class with Pull Up Method refactoring pattern.

Figure 2.10 shows an example of refactorings with this pattern. There are two classes that have the same base class, `Site`, and these two classes have the methods that are similar to each other, `getBillableAmount`.

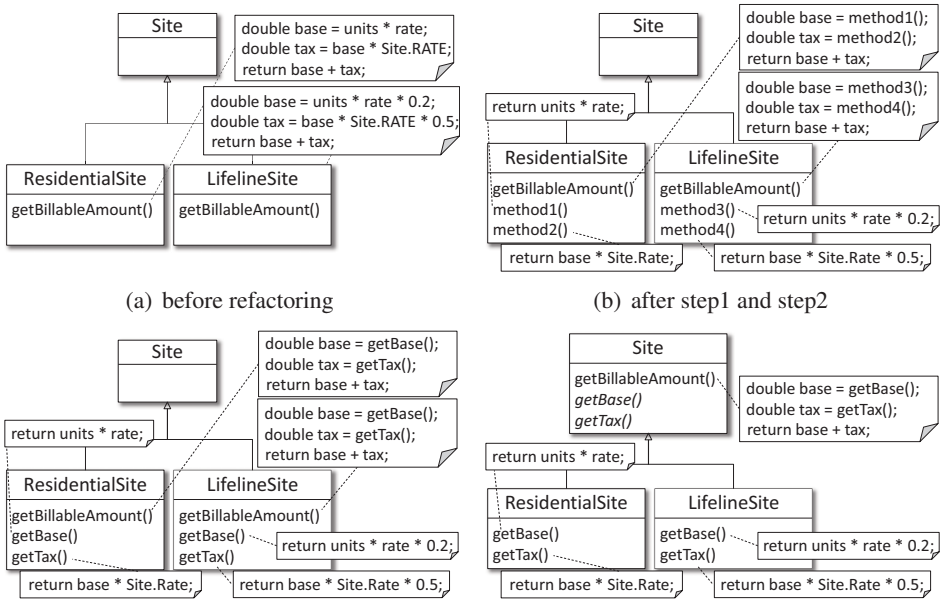


Figure 2.10: An Example of Refactorings with Form Template Method

To apply Form Template Method to this target, at first, it is necessary to distinguish the common and unique processes in the two methods. In this example, the differences of the two methods are in the calculation ways of variables `base` and `tax`.

The next step extracts each of the calculations of `base` and `tax` as new methods (shown in Figure 2.10(b)), which results in four methods (currently, they are named as `method1`, `method2`, `method3`, and `method4`).

In the next step, the four new methods are renamed to make correspondence of signatures (shown in Figure 2.10(c)). In this example, `method1` in `ResidentialSite` and `method3` in `LifelineSite` are called as the first processing of the original methods. Also, `method2` and `method4` are called as the second processing of the original methods. Therefore, `method1` and `method3` are renamed to make their signatures correspondent. In this example, `method1` and `method3` are renamed as `getBaseAmount`. Similary, `method2` and `method4` are renamed as `getTaxAmount`.

Finally, the common processes are pulled up as a new method in the base class `Site`. Note that this step defines `getBaseAmount` and `getTaxAmount` as abstract methods in the base class. Figure 2.10(d) shows the code after the refactoring

has finished.

By applying this refactoring pattern to similar methods, code clones existing between these methods are merged into a base class. An advantage of clone removal with this pattern compared with Pull Up Method is that this pattern can be more widely used than Pull Up Method because this pattern can be applied to methods that are not exactly the same. Compared with clone removal with Extract Method refactoring pattern, the application range of this pattern is narrower. However, this pattern is effective in such a case that common processes are segmentalized by unique processes. This is because separated common processes can be merged as a single method with Form Template Method refactoring pattern, meanwhile each fragment of common processes is extracted as a method with Extract Method refactoring pattern.

The rest of this dissertation calls a method created in base classes by pulling up the common processes **template method**.

2.5.3 Research on Clone Removal

Fowler, a pioneer in the field of refactoring, mentioned that "*the number one in the stink parade is duplicate code*" [32]. He also presented some sets of operations for merging code clones. However, because it is quite difficult for maintainers to apply refactorings manually without introducing any human errors, much research effort has been performed on refactoring assistance [111].

Choi et al. found from open source software systems that most of refactorings applied to code clones are either Extract Method or Replace Method with Method Object [22].

Higo et al. proposed a method for merging code clones [44]. Their method consists of two phases. The first phase is quick detection of *refactoring-oriented code clones* from source code. The second phase is measurement of metrics indicating how the refactoring-oriented code clones should be merged. They implemented their method as a tool named *ARIES*. Using *ARIES* in the refactoring process, maintainers of software systems can readily know which and how code clones can be merged. They conducted a case study with *ARIES*, and they confirmed that *ARIES* performs the process successfully.

CloneDR, which is an implementation of the AST-based technique of clone detection, offers not only the locations of code clones but also forms of merged code fragments [15]. The forms help users understand what operations are required to merge code clones. However, the tool does not care about the positional relationship between code clones in the class hierarchy.

Bakazinska et al. proposed a refactoring technique for duplicate methods [9]. Their technique provides the differences between code clones, which help users

to determine whether code clones can be merged or not. Also, their technique measures the coupling between a duplicated method and its surrounding code. In their method, code clones are removed by using two design pattern Strategy and Template Method.

Cottrell et al. implemented a tool that visualizes the detailed correspondences between a pair of classes [24]. The classes are generalized to form an intermediate, AST-like structure that distinguishes between what is common and what is specific to each class. The specific instructions will influence the degree of relatively between the classes. The tool works after users identify two classes that should be merged.

Komondoor et al. proposed an algorithm for procedure extraction [80]. The inputs to the algorithm are the CFG (control-flow graph) of a procedure and a set of nodes in the CFG. The goal of the algorithm is to revise the CFG with the following conditions:

- The set of nodes that are extractable from the revised CFG;
- The revised CFG is semantically equivalent to the original CFG.

The implementation of this algorithm adopts heuristics for enhancing scalability. Although the algorithm has a worst-case exponential time complexity, their experimental results indicated that it may work well in practice. However, the algorithm can be applied only to a single clone pair. Different techniques are needed to determine how two or more code clones can be extracted as a single procedure with preserving semantics.

The majority of clone removal techniques is based on Extract Method or Pull-Up Method refactorings, and there are few techniques based on Form Template Method refactoring.

Juillerat et al. proposed a method to automatically apply Form Template Method to a pair of similar methods with ASTs [66]. Their method can show source code after the application of the pattern, and the execution time and memory space required to the calculation are not so high.

Masai et al. proposed a method to support refactorings with Form Template Method with ASTs likewise Juillerate et al. [107]. Their method considers the structural information of ASTs to detect unique processing, meanwhile Juillerat et al. compare ASTs with token sequences that are made from ASTs. Also, they implemented a function to suggest suitable divisions between common- and different-parts on the specified method pair to users with a cohesion metric COB [37, 55].

Although many reseachers have spent their effort on research of clone removal, as described in 2.2.2, it cannot be said that all the clones are “bad smells” for

software maintenance. In addition, clone removal has been countered by some reseachers [77, 78]. The bases of such counter opinions are as follows.

- Many code clones live in a short time.
- Most of code clones that live in a long time cannot be easily refactored.

Hence, it is important before removing clones to select targets of removal carefully, with pros and cons of it (including benefits that the removal will offer, costs that the removal will require, and risks that the removal might bring) being considered.

2.6 Prevention

As 2.2.1 described, code cloning by copy-and-paste operations should result in creation of code clones. Although clones created by the operations may affect software maintenance adversely, it has a big advantage that is hard to be abandoned. That is to say, it can rapidly provide functions that are similar to the existing ones. Almost all of recent software development has a strictly-limited time and resources, which forms a hotbed of cloning. In addition, cloning is sometimes unavoidable because of technical or organizational limitations [163].

However, it is quite obvious that unlimited cloning wreaks enormous damage on software evolution because of a large amount of negative clones. Hence, it is important for effective software evolutaion to prohibit creating negative clones and to allow creating only positive ones.

Laguë et al. proposed a strategy of clone prevention named "Preventive Control" [90]. It allows code cloning only when the cloning has a valid reason. When developers want to clone any code fragment, preventive control forces them to explain the reason of the cloning that they want to do. In the case that the explained reason is regarded as valid, developers can perform the cloning. If it is not the case, the cloning will be rejected. Therefore, preventive control can prevent introducing negative clones in software systems.

Another way of clone prevention stands on watching behavior of developers on IDEs. Venkatasubramanyam et al. proposed a framework to observe developers' behavior to prevent clone creation [148]. When developers are just about to clone any code, the framework checks whether the cloning satisfies constraints that all the cloning is imposed. If the cloning violates any constraints, the framework rejects the cloning. A big feature of this framework is that its users can define the constraints of cloning as they like. Venkatasubramanyam et al. provided an instance of constraints that "Do not clone code fragments whose cyclomatic complexity [110] values are larger than or equal to five".

Furthermore, there exists a techique to predict harmfulness of clones that are created by copy-and-paste operations at the point of these operations, which was proposed by Wang et al. [151]. They regarded code cloning as harmful if clones created by it need any consistent modifications in the future. On the other hand, code cloning is regarded as not harmful in the case that clones created by the cloning no longer need any modifications, or they do not need any consistent modifications. Their prediction uses 21 metrics that are collected based on features or locations of code. They predicted harmfulness of clones with Bayesian Networks [127], a well-known machine learning technique, and collected metrics. This technique can be used in the following two strategies.

- Prohibiting cloning that is predicted to produce any harmful clones.
- Allowing cloning only if it is predicted to produce non-harmful clones only.

Wang et al. evaluated their technique on both of the two strategies with commercial software products of Microsoft, and they stated that their technique is enough effective on both of the two strategies.

2.7 Analytic Methodology

As described in 2.4, much effort has been spent on clone detection, which produces a variety of clone detectors. Coupled with advanced hardware, clone detectors are now substainable for use on very large scale software in industry.

Of course, detecting clones is not enough to promote effective software evolution. Clone detectors come into their own when their outputs are used in some way. A particular way to make use of detected clones is to analyze them. However, clone detectors tend to report a long list of clones from a large scale software system. Hence, it becomes a challenging task for developers or maintainers of large scale software systems to treat such a large amount of clones [63]. This indicates that there is a strong demand for techniques to analyze a large amount of clones effectively.

This section introduces some techniques that can respond to such a request.

2.7.1 Filtering and Categorizing

One way to promote efficient analysis of clones is to reduce the amount of clones by filtering out uninteresting clones. Similarly, categorizing or clustering clones should allow developers or maintainers to analyze clones in a shorter time frame. Table 2.1 summalizes these techniques.

Table 2.1: Overview of Methods for Filtering/Categorizing/Clustering Clones

Proposed by	Basis	Overview
Jiang et al. [63]	Specified by users	Omitting clones not satisfying specified conditions
Yang et al. [158, 159]	Machine learning with TF-IDF	Learning useful clones specified by individual user
Zhang et al. [164]	Specified as the SQL format	Getting clones that satisfy conditions specified with the SQL format
Tairas & Gray [142]	Latent semantic analysis based on identifier names	Clustering clones sharing many identifiers having the same names into the same cluster.
Kapser & Godfrey [73]	Locations of clones	Categorizing clones with their locations such as *Same File*, *Same Dir*, *Different File*.
Merlo & Lavoie [112]	Owner blocks of clones	Categorizing clones like *method-method*, *block-block*, or *method-block*.
Balazinska et al. [10]	Comparison of methods shairing cloned fragments	Categorizing clones into 18 types based on differences of methods.
Xing et al. [156]	Comparison of PDGs	Categorizing clones into 7 types based on differences of PDGs
Kamiya [69]	Configurations such as *Makefile*s	Classifying clones based on configuration information such as *appearing in a configuration, but not in another configuration*
Kamiya [70]	Caller-callee relationships	Classifying cloned functions that call the same funciton as *content clone*, and those that are called by the same function as *context clone*.

Jiang et al. proposed a filtering technique for code clones based on data mining techniques [63]. This filtering ommits "uninteresting" clones from the results of clone detectors. Their method allows its users to decide criterion of "interesting" clones because it strongly depends on the purpose of clone analysis or the analyzers' notions whether a code clone is "interesting" or "uninteresting". In other words, a code clone can be "interesting" even though it is "uninteresting" in another situation or for another analyzer.

Yang et al. proposed another technique to filter code clones [158,159], which is based on a machine learning technique. In similar to Jiang et al., Yang et al. stand on the basis that usefulness of a code clone should differ in according to situations or users. To be flexible to such variability of usefulness of clones, they developed a web-base system named *Fica* for filtering of clones. *Fica* internally uses a machine learning technique to predict whether a given code clone is "interesting" or not in a particular situation and/or for a particular user. *Fica* learns judgements of usefulness of clones from training data sets, and predicts whether clones not included in the training data sets are "interesting" or not. Each user of *Fica* has her/his own training data set, thus *Fica* can be flexible to different notions of different users. In addition, every user of *Fica* can have multiple training data sets, which offers *Fica* a great frexibility for different usages.

They performed an open survey, with 33 participants contributed. The survey revealed a high accuracy of *Fica*'s prediction. Furthermore, it found the following observations.

- "Uninteresting" clones tend to fall into several categories, thus they are likely to form clusters.
- "Interesting" clones tend to be unique comparing to uninteresting ones.
- Users having more experience on code clones are more likely to agree with each other compared to users having less experience. In other words, experts on code clones share common opinions on interests of clones.

Zhang et al. developed a filtering of clones based on SQL instructions [164]. Their method stores results of a clone detector, named *CloneMiner* [12], which allows users to select clones with any SQL instructions. In addition, they have implemented their method as a plugin of Eclipse, the de-facto standard of IDEs for Java, which is named *CloneVisualizer*.

Tairas and Gray proposed a technique to cluster clones based on a latent semantic analysis [142]. This method regards an identifier as a *word*, thus clones sharing many identifiers with the same names are clustered into a single group.

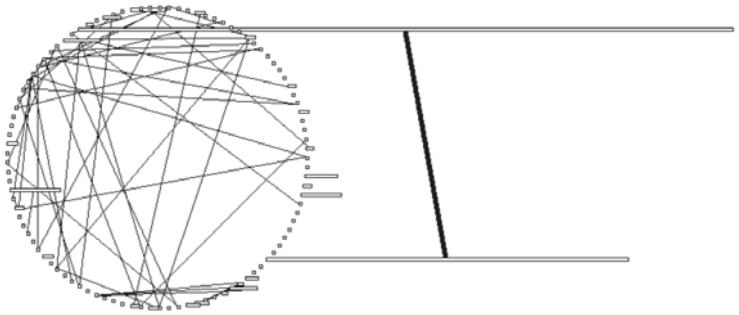


Figure 2.11: Rieger et al.'s Visualization (Duplication Web) (Cited from the literature [130])

Many categorizations of clones have been proposed in recent years, including techniques based on locations [73], syntactical information [112], differences [10, 156], configurations [69], or caller-callee relationships [70]. These categorizations will be helpful to assess or understand code clones in software systems.

2.7.2 Visualizing

Software visualization is quite useful to understand software systems [14]. Many researchers successifully have made use of visualization techniques to understand circumstances of code clones.

Rieger et al. applied polymetric views to clone visualization [130]. Polymetric view is a visualization technique that represents mulitple characteristics in a single view [92]. They proposed six types of clone visualization, one of which is shown in Figure 2.11. Figure 2.11 shows an instance of duplication web. Duplication web places all the source files as boxes in a concentric fashion, and draws edges between source files if they share any code clones. The width of each edge represents the amount of shared clones between two source files. This means that there is a heavy edge between two source files if they share a large amount of clones. Furthermore, the length of each box indicates the amount of code clones whose elements are included only in the file represented by the box without any exceptions. Duplication web shows which files share a lot of clones, which will be helpful for software maintainers.

Wattenberg proposed Arc Diagram that visualizes patterns of strings that frequently occur [152]. Figure 2.12 describes an example of an arc diagram. Arc diagrams can show patterns frequently observed in not only source code but also other sequences including byte code, DNA arrangements or musical scores.

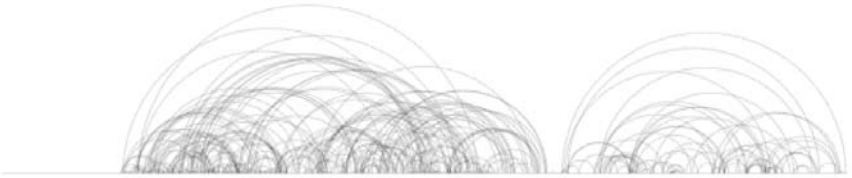


Figure 2.12: An Arc Diagram (Cited from the literature [152])

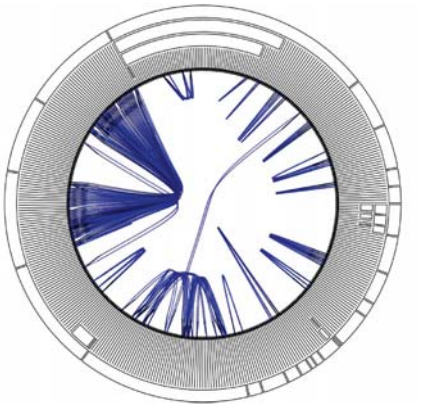


Figure 2.13: Clone Visualization with Bundle Edge View (Cited from the literature [40])

Hauptmann et al.'s technique uses bundle edge views [40]. Bundle edge view is a kind of graph visualizing techniques. To reduce the number of edges of graphs, bundle edge view bundles edges with similar origins and destinations together [49]. Figure 2.13 shows an example of bundle edge views for clone visualization. Each edge of the graph indicates that two source files connedted by the edge shares clones. This view shows distribution of clones across the whole of a given software system in a concise way.

Yoshimura and Mibe proposed a technique to visualize analogous relationships between source files [162], which is shown in Figure 2.14. Their goal is to explain how clones distribute for stakeholders who are not experts of software development. This visualization technique can show how many source files share clones in one glance.

Ueda el al. developed *Gemini* [147], which is a GUI frontend of *CCFinder* [72]. *Gemini* uses scatter plots to look down at distributions of clones. In addition, it has a function to filter out clones with some metrics. It is equipped a graph named Metric Graph to support the filtering. Users of *Gemini* can filter clones in a

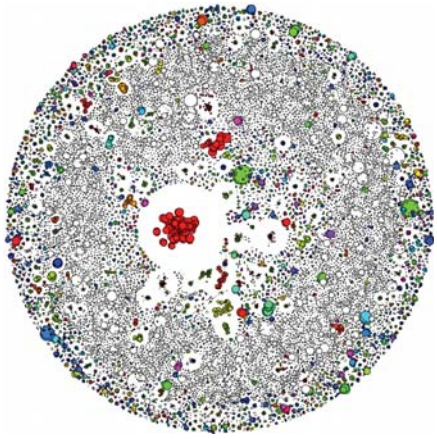


Figure 2.14: Clustered Source Files based on Their Similarities (Cited from the literature [162])

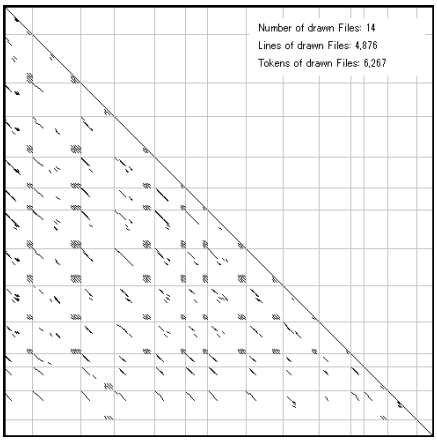


Figure 2.15: An Exapmle of Scatter Plots Shown by Gemini [147]

intuitive way with the graphs. As another research related to *Gemini*, Miyazaki et al. proposed a technique to support users of *Gemini* to decide where they should start analysis of clones [113].

Adar and Kim have offered a library for clone visualization named *SoftGUESS* [3]. Their library includes visualizations for clone genealogies, for hierarchical structures of clones, and dependencies of clones at the package level.

Recent research has high interests on clone genealogies. As described above, *SoftGUESS* can visualize clone genealogies. Another technique to visualize clone genealogies is proposed by Saha et al., which uses scatter plots [136].

2.8 Detection and Prevention of Clone Related Bugs

The presence of code clones can be a cause of bug spreadings. That is to say, if a bug hides in one of cloned fragments, its correspondents have high probabilities that they also have the same bug. Hence, developers or maintainers may neglect to fix all the cloned fragments in a clone set if they are unaware of the presence of the clone. Moreover, it is quite natural that similar bugs will be exposed in similar situations. This means that, if a bug in a system was fixed and the fixed bug was one of clone related bugs, the system should still fail to work in the similar situations that the fixed bug occured. In this case, users of the system will think that "the bug has not been fixed even though the provider of the system said the bug was already fixed". They might even say that "the provider and the developers have low technical capabilities because they could not fix a bug". Therefore, clone related bugs may force providers, developers or maintainers of software systems to lose their users' trust.

Unfortunately, it is sometimes not enough for finding clone related bugs to run clone detectors, specifically in large scale software systems. Many researchers have spent great effort to detect clone related bugs efficiently from large scale software systems for this reason. This section simply describes their achievements.

2.8.1 Preventing Clone Related Bugs

There is a variety of approaches to prevent introducing clone related bugs. The approaches include supporting simultaneous modifications, suggesting cloned fragments of the code fragments that a developer is about to modify, or finding oversights of simultaneous modifications.

Toomim et al. proposed a technique to support simultaneous modifications on clones, which is named "Linked Editing" [145]. Users of linked editing prescribe it to link two code fragments as the first step of use. Once links of two code fragments are specified, modifications applied on a code fragment will be automatically applied to the other code fragment. If users do not want to apply a modification simultaneously to the other fragment, they can flexibly avoid the automatic modification. Another technique to encourage simultaneous modifications on clones is the one proposed by Higo et al. [46]. Their technique detects cloned fragments that have clone relationships between the user-specified fragment and shows them to users. Unlike linked editing, Higo et al.'s technique requires only a single code fragment to be specified, because it automatically detects clones with a code clone detector.

Other techniques monitor developers' copy-and-paste operations to prevent clone related bugs. They automatically record links between the original and the

destination fragments, and support simultaneous modifications on them [50,57,58, 155]. Another technique achieves prevension of introducing clone related bugs by showing code clone information as comments [137].

There are some systems available to prevent clone related bugs. One of them is *JSync*, which is proposed by Nguyen et al. [119]. *JSync* automatically detects uninconsistent changes on clones and alerts its users. In addition to that, *JSync* has a function to suggest how to fix the uninconsistent changes.

Duala-Ekoko and Robillard developed another clone management system, which informs users of modifications applied on clones [27]. It tells the locations of cloned fragments that have clone relationships to the code fragment that users are modifying.

The system proposed by Yamanaka et al. reports how clones are modified across version histories [157]. Their method informs users of additions, deletions, or changes on clones, which prevents unawareness of them. They evaluated the effectiveness of their method on an experimental web system developed in NEC Corporation.

2.8.2 Detecting Clone Related Bugs

Many techniques have been proposed to detect clone related bugs. It is said that most of clone related bugs are introduced by unintended inconsistencies on clones. Hence, most of these techniques find and use inconsistencies of cloned fragments in code clones to detect bugs. Inconsistencies used to detect clone related bugs have a wide variety, including identifier names [53, 98], and control structures of owners of clones [62]. In addition, there is a technique for detecting clone related bugs that uses differences of results of multiple clone detectors [45].

Li and Ernst developed *CBCD*, which suggests code fragments having a high possibility to have a clone related bug [97]. *CBCD* is used when its users find a bug. *CBCD* takes the code fragment having the found bug as its input, and detects code fragments that are suspected to have similar bugs.

There exists some other ways that are similar to *CBCD*, including using information about identifiers [160, 161] or targetting concurrent processing [59].

Lucia et al. stated that bug detection tools for clones tend to report many false positives, and proposed a technique that eliminates such false positives from the results of the tools [104]. Their method refines clone anomaly reports to achieve the objective. Using thier method after clone related bug detection enables to find clone related bugs more efficiently. In similar to Lucia et al,'s technique, Hayase et al. proposed a filtering for *CP-Miner* to eliminate false positives [41].

In addition, there are some techniques that combine fault-prone module prediction with code clones [6, 68].

Some configuration management systems have been proposed for code clones in recent years [39, 121]. It is difficult to detect clone related bugs only with the code of the latest revision of the target software system if many modifications were added after copy-and-paste operation. This is because the origin and the destination of the cloning are no longer detected as clones because of gaps between them. Clone-aware configuration management systems will resolve such issues because they are able to provide historical information of the clones. In addition, they have another advantage than clone related bug detection. As Hata et al. have stated in the literature [39], using clone-aware configuration management systems allows us to gain rich data about clones including process related information or human attributes. That is, by using clone-aware configuration management systems, we can apply techniques of repository mining, which is one of the most hot topics in the research area of software engineering, to histories of code clones.

Chapter 3

An Empirical Study on Influences for Clones on Software Evolution

3.1 Background

It *was* generally believed that code clones negatively affect software evolution. However, some researchers doubted the accepted notion, and conducted empirical studies to reveal whether it is really true.

In order to answer the question whether code clones are harmful or not, they compare characteristics of cloned code and non-cloned code instead of directly investigating maintenance costs that they require. This is because measuring the actual maintenance costs is quite difficult. The experimenal results are split: some of them supported the accepted notion and claimed that code clones are harmful for software evolution [100, 101, 115], and others argued that code clones do not have seriously negative impacts [34, 35, 78, 88, 89, 103, 128]. There even exists opinions that code cloning is a good choice for design of the source code [74].

Such a variation of opinions on harmfulness of clones may imply that many of clones are not harmful, however, there still exists some instances having negative impacts on software maintenance. This fact indicates that managing all of clones carefully is not only effort-consuming but also uneffective. In other words, *we thus have to carefully select the clones to be managed to avoid unnecessary effort managing clones with no risk potential* [35]. To achieve such a challenging objective, it is necessary to know the characteristics that harmful clones have.

In this study, we conduct an empirical study that compares cloned code to non-cloned code from a different standpoint of previous research, and reports the experimental result on open source software. The features of the investigation in this study are as follows:

- every line of code is investigated whether it is cloned code or not. Such a fine-grained investigation enables an accurately judge on whether every modification conducted to cloned code or to non-cloned code;
- maintenance cost consists of not only source code modifications but also several phases prior to it. In order to estimate maintenance cost more appropriately, this study defines and uses a new indicator that is not based on modified lines of code but the number of modified places;
- we evaluate and compare modifications of cloned code and non-cloned code on multiple open source software systems with multiple clone detectors. That is because every clone detector detects different code clones from the same source code.

We also conducted a comparison experiment with two investigation methods previously proposed by other researchers. The purpose of this experiment is to reveal whether comparisons between cloned code and non-cloned code with different methods yield the same result or not. In addition, we carefully analyzed the results in the cases that the comparison results were different from each method to reveal the causes behind the differences.

3.2 Motivation

3.2.1 Motivating Example

As described in Chapter 2, much research effort has been performed on evaluating the influence of code clones. However, these investigation methods still have some points that they did not evaluate. This subsection explains these points with the example shown in Figure 3.1. In this example, there are two similar methods and some places are modified, with the modified places classified into four parts, modification A, B, C, and D.

Investigated Units

In some studies, large units (e.g. files or methods) are used as their investigation units. In those investigation methods, it is assumed that code clones have a negative impact if files or methods having a certain amount of code clones are modified, which can cause a problem. The problem is the incorrectness of modifications count. For example, if modifications are performed on a method which has a certain amount of code clones, all the modifications are assumed as performed on the cloned code even if they are actually performed on non-cloned code of the

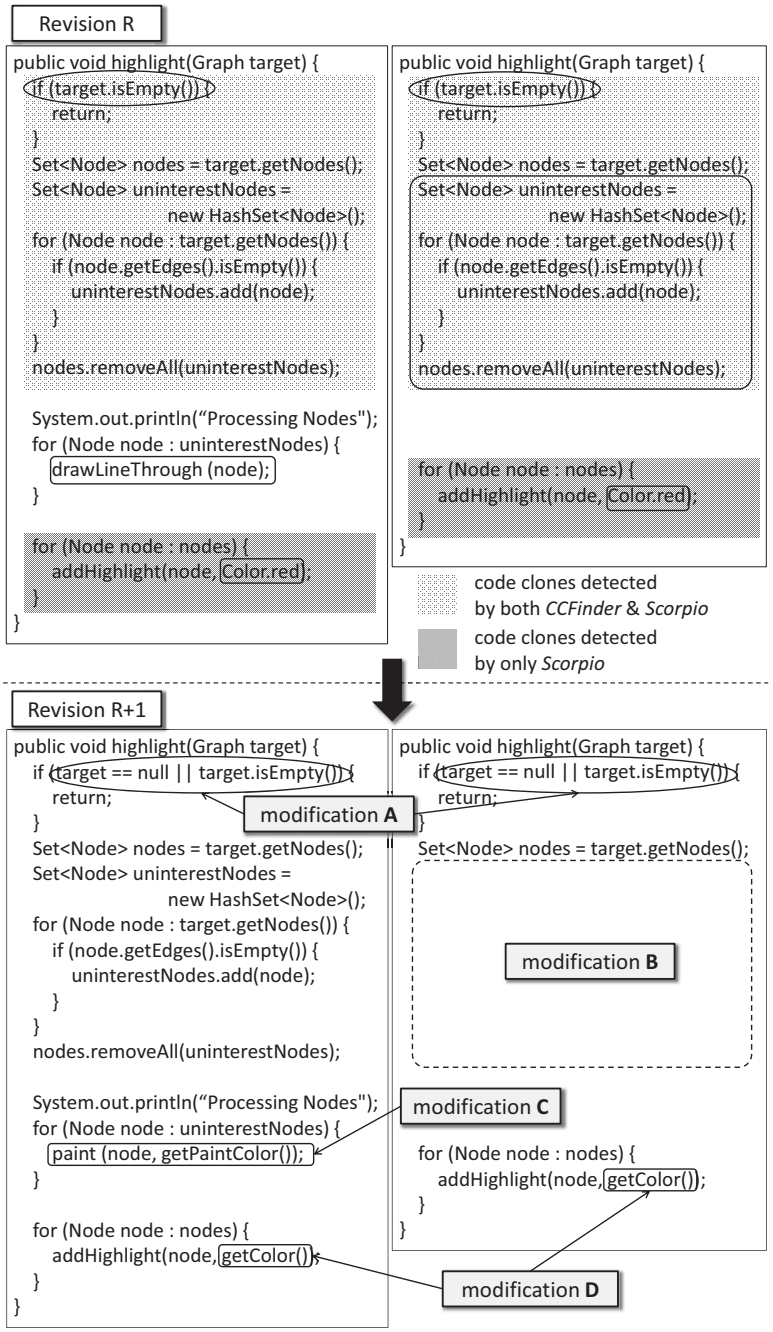


Figure 3.1: Motivating Example of Our Empirical Study

method. Modification C in Figure 3.1 is an instance that suffers from this problem. This modification is performed on non-cloned code, nevertheless method or file based investigations regard that this modification is performed on cloned code.

Line-based Barometers

Some other studies used line-based barometers to measure the influence of code clones on software evolution. Herein, the line-based barometer indicates a barometer calculated with the amount of added/changed/deleted lines of code. However, line-based barometer cannot distinguish the following two cases: the first case is that *consecutive 10 lines of code were modified for fixing a single bug*; the second case is that *1 line modification was performed on different 10 places of code for fixing 10 different bugs*. In real software maintenance, the latter should require much more costs than the former because software maintainers have to conduct several steps before actual modifications on source code such as identifying buggy modules, identifying buggy instructions and so on.

In Figure 3.2.2, Modification A is a single line modification, and performed on two places, meanwhile Modification B is a modification including seven consecutive lines on a single place. With line-based barometers, it is regarded that Modification B has the impact 3.5 times larger than Modification A. However, this is not true because software maintainers have to identify two places of code for Modification A meanwhile a single place need to be identified for Modification B.

A Single Clone Detector

Each of the previous studies used a single clone detector to detect code clones in target software systems. However, as discussed in Chapter 2, there is neither a generic nor strict definition of code clones. Each detector has its own unique definition of code clone, and it detects code clones based on the own definition. Consequently, different code clones are detected by different detection tools from the same source code. Therefore, the investigation result with one detector is different from the one from another detector. Figure 3.1 shows the differences of detected clones between a detector *CCFinder* and another detector *Scorpio*. Consequently, if we use *Scorpio*, Modification D is regarded as being affected with code clone, nevertheless it is regarded as not being affected with code clone if we use *CCFinder*. Therefore, the investigation with a single detector is not sufficient to get a generic result about the impact of code clones on software evolution.

3.2.2 Objective of this Study

This study stands on a different point from previous research. The features of this study are as follows:

Fine-grained Investigation Units: In this study, every line of code is investigated whether it is included in any code clones or not, which offers us with more accurate judgements on whether every modification is conducted on cloned code or non-cloned code.

Place-based Indicator: This study uses a new indicator based on the number of modified places, not the number of modified lines. The purpose of using such a place-based indicator is to evaluate the impact of the presence of cloned code with different standpoints from the previous research.

Multiple Detectors: This study uses four clone detectors to reduce biases of each detector.

3.3 Terms

This section describes clone detectors used in this study, and then explains some terms to which the remainder of this study refers.

3.3.1 Clone Detectors Used in This Research

CCFinder

CCFinder is one of the famous token-based clone detectors developed by Kamiya et al. [72]. The major features of *CCFinder* are as follows:

- *CCFinder* replaces user-defined identifiers such as variable names or function names with special tokens before the matching process. Consequently, *CCFinder* can identify code fragments that use different variables as code clones.

- Detection speed is very fast. *CCFinder* can detect code clones from millions lines of code within an hour.

- *CCFinder* can handle multiple popular programming languages including C/C++, Java, and COBOL.

CCFinderX

CCFinderX is a major version up from *CCFinder* [21]. *CCFinderX* is a token-based clone detector as well as *CCFinder*, but the detection algorithm was changed to *bucket sort* from *suffix tree*. *CCFinderX* can handle more programming languages than *CCFinder*. Moreover, it can effectively use resources of multi-core CPUs, which realizes faster detection.

Simian

Simian is a text-based clone detector [140]. As well as the *CCFinder* family, *Simian* can handle multiple programming languages. Its text-based technique realizes clone detection on small memory usage and short running time. Also, *Simian* allows fine-grained settings to its users. For example, users can configure to ignore *import* statements in the case of Java language.

Scorpio

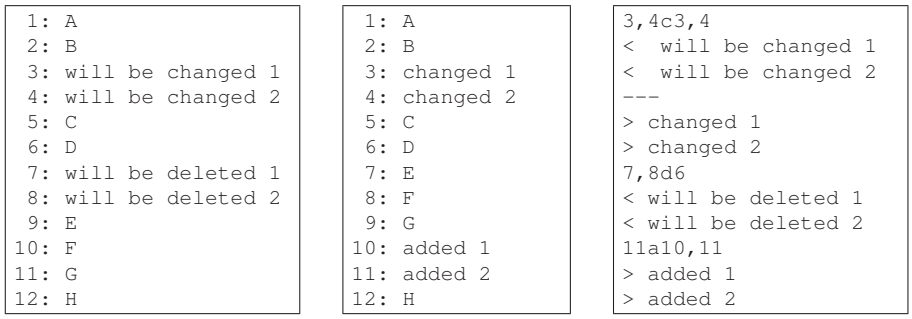
Scorpio is one of the graph-based clone detectors developed by Higo et al. [42]. Currently, *Scorpio* can handle software systems written in Java. The major features of *Scorpio* are as follows.

Detecting code clones with different user-defined variables: *Scorpio* replaces user-defined identifiers by special characters. Therefore, it can detect code clones having different user-defined variables.

Detecting Type-3 code clones and non-contiguous code clones: *Scorpio* can detect Type-3 code clones and non-contiguous code clones because it is a PDG-based clone detector.

Robustness for detecting contiguous code clones: One of the disadvantages of PDG-based clone detectors is that they cannot regard sequences of program elements as code clones if every element in the sequences has no dependence between other elements in the sequences. To improve this matter, *Scorpio* introduces execute dependence, which enables it to expand the range of program slicing, so that the ability to detect contiguous code clones is improved.

Using both of two graph search algorithms: There are two ways to search graphs, forward and backward slicing. *Scorpio* uses both of forward and backward slicing, which enlarges results of clone detection because there are similar subgraphs that cannot be detected by using only forward or backward slicing.



(a) before modification (b) after modification (c) *diff* output

Figure 3.2: A Simple Example of Comparing Two Source Files with *diff*

Heuristics for reducing costs of detection: To reduce detection costs, *Scorpio* makes a limitation on start points of slicing. Unnecessary slicing points are identified and removed by this heuristic.

3.3.2 Revision

In this study, it is necessary to analyze historical data of code. Therefore, this study targets software systems managed by version control systems. Version control systems store information about snapshots of software products, including source code and documents, and changes applied to them. Each snapshot is identified by a number, called "**revision**". We can get source code in arbitrary revision, and we can also get modified files, change logs, and the name of developers who made changes in arbitrary two consecutive revisions with version control systems.

Due to the limit of implementation, we restrict the target version control system to Subversion. However, it is possible to use other version control systems such as CVS.

3.3.3 Target Revision

This study is only interested in changes in source files. Therefore, we find out commits that have some modifications in source files. The remainder of this chapter uses a term **target revisions**, which indicates revisions that are related to any of such commits. That is, a revision R is regarded as a target revision if at least one source file is modified from R to $R + 1$.

3.3.4 Modification Place

This study uses the number of places of modified code, instead of lines of modified code. That is, even if multiple lines are modified, this study regards it as a single modification if the modified lines are consecutive. In order to identify the number of modifications, this study uses the traditional *diff* command of UNIX. Figure 3.2 shows an example of *diff* output. In this example, we can find three modification places. One is a change in the third and fourth lines, another is a deletion of the seventh and eighth lines, and the other is an addition of statements between the 11th line and the 12th line. As shown in Figure 3.2, it is very easy to identify modified places from *diff* outputs; all we have to do is just parsing the output of *diff* so that start lines and end lines of all the modifications are identified.

3.4 Proposed Method

This section describes our research questions and the investigation method.

3.4.1 Research Questions and Hypotheses

The purpose of this study is to reveal whether the presence of code clones really affects software evolution or not. This study is under an assumption that *if cloned code is more frequently modified than non-cloned code, the presence of cloned code has a negative impact on software evolution*. The basis of this assumption is that if much cloned code that is never modified during its lifetime is included in source code, the presence of cloned code never causes inconsistent changes or additional maintenance effort. Our research questions are as follows.

RQ1: Is cloned code more frequently modified than non-cloned code?

RQ2: Are the comparison results of stability between cloned code and non-cloned code different from multiple detection tools?

RQ3: Is cloned code modified uniformly throughout its lifetime?

RQ4: Are there any differences in the comparison results on types of modifications?

To answer these research questions, we define a new indicator, named **modification frequency** (in short, MF). We measure and compare MF of cloned code (in short, MF_d) and MF of non-cloned code (in short, MF_n) for investigation.

3.4.2 Modification Frequency

Definition

As described above, this study uses MF to estimate the influence of code clones. MF is an indicator based on the number of modified code, not lines of modified code.

The following formula (3.1) describes the definition of MF_d .

$$MF_d = \frac{\sum_{r \in R} MC_d(r)}{|R|} \tag{3.1}$$

where,

- R is a set of target revisions.
- $MC_d(r)$ is the number of modifications on cloned code between revision r and $r + 1$.

We also define MF_n in the formula (3.2).

$$MF_n = \frac{\sum_{r \in R} MC_n(r)}{|R|} \tag{3.2}$$

where,

- $MC_n(r)$ is the number of modifications on non-cloned code between revision r and $r + 1$.

These values mean the average number of modifications on cloned code or non-cloned code per revision. However, in these definitions, MF_d and MF_n are very affected by the amount of code clones included the source code. For example, if the amount of code clones is very small, it is quite natural that the number of modifications on cloned code is much smaller than non-cloned code. However, if a small amount of cloned code is included but it is quite frequently modified, we need additional maintenance effort to judge whether its correspondents need the same modifications or not. We cannot evaluate the influence of code clones in these situations in these definitions.

In order to eliminate the bias of the amount of code clones, we normalize the formulae (3.1) and (3.2) with the ratio of cloned code. Here, we assume that:

- $LOC_d(r)$ is the total lines of cloned code in revision r.

- $LOC_n(r)$ is the total lines of non-cloned code on r.
- $LOC(r)$ is the total lines of code on r, so that the following formula is satisfied:

$$LOC(r) = LOC_d(r) + LOC_n(r) \tag{3.3}$$

Under these assumptions, the normalized MF_d and MF_n are defined in the following formulae (3.4) and (3.5).

$$normalized MF_d = \frac{\sum_{r \in R} MC_d(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_d(r)} \tag{3.4}$$

$$normalized MF_n = \frac{\sum_{r \in R} MC_n(r)}{|R|} \times \frac{\sum_{r \in R} LOC(r)}{\sum_{r \in R} LOC_n(r)} \tag{3.5}$$

In the reminder of this chapter, the normalized MF_d and MF_n are refered as just MF_d and MF_n, respectively.

Measurement Steps

MF_d and MF_n are calculated with the following steps:

STEP1: Identify target revisions from the repositories of target software systems. Then, all the target revisions are checked out into our local storage.

STEP2: Normalize all the source files in every target revision.

STEP3: Detect code clones within every target revision, and then the detection result is analyzed in order to identify the file path and the lines of all the detected cloned code.

STEP4: Identify differences between two consecutive revisions. The start lines and the end lines of all the differences are stored.

STEP5: Count the number of modifications on cloned code and non-cloned code.

STEP6: Calculate MF_d and MF_n.

The remainder of this subsection explains each step of measurement in detail.

STEP1: Obtains target revisions

In order to measure MF_d and MF_n, it is necessary to obtain historical data of source code. As described above, this study uses a version control system, Subversion, to obtain the historical data.

This step identifies which files were modified, added, or deleted in each revision and finds out target revisions. After identifying all the target revisions from the historical data, they are checked out into the local storage.

STEP2: Normalizes source files

STEP2 normalizes every source file in all the target revisions with the following rules:

- deletes blank lines, code comments, and indents,
- deletes lines that consist of only a single open/close brace, and the open/close brace is added to the end of the previous line.

The presence of code comments influences the measurement of MF_d and MF_n. If a code comment locates within cloned code, it is regarded as a part of cloned code even if it is not a program instruction. Thus, the LOC of cloned code is counted greater than it really is. Also, there is no common rule how code comments should be treated if they are located in the border of cloned code and non-cloned code, which can cause a problem that a certain detector regards such a code comment as cloned code meanwhile another tool regards it as non-cloned code.

As mentioned above, the presence of code comments makes it more difficult to identify positions of cloned code accurately. Consequently, this step completely eliminates all the code comments from source code. As well as code comments, different clone detectors handle blank lines, indents, lines including only a single open or close brace in different ways, which also influences results of clone detection. For this reason, blank lines and indents are removed, and lines that consist of only a single open or close brace are removed, and the removed open or close brace is added to the end of the previous line.

STEP3: Detects code clones

This step detects code clones from every target revision with clone detectors, and stores the detection results into a database. Each detected cloned code is identified as a three-tuple (v, f, l), where: v is the revision number where a given cloned

code was detected; f is the absolute path to the source file where a given cloned code exists; l is a set of line numbers where cloned code exists. Note that storing only the start line and the end line of cloned code is not feasible because a part of cloned code is non-contiguous.

This step is very time consuming because this step need to run each clone detector on every target revisions. That is, if the history of the target software includes 1,000 target revisions, cloned code detection need to be performed 1,000 times. However, this step is fully-automated, and no manual work is required.

STEP4: Identifies differences between two consecutive revisions

STEP4 finds out modification places between two consecutive revisions. As described above, just parsing outputs of *diff* offers this information.

STEP5: Counts the number of modifications

In this step, we count the number of modifications on cloned code and non-cloned code with the results of the previous two steps. Here, we assume the variable for the number of modifications of cloned code is MC_d, and the variable for non-cloned code is MC_n. Firstly, MC_d and MC_n are initialized with 0, then they are increased as follows; if the range of specified modification is completely included in cloned code, MC_d is incremented; if it is completely included in non-cloned code, MC_n is incremented; if it is located across the border of cloned code and non-cloned code, both MC_d and MC_n are incremented. All the modifications are processed with the above algorithm.

STEP6: Calculates MF_d and MF_n

STEP6, the last step of measurement, calculates values of MF_d and MF_n with the definitions in the formulae (3.4) and (3.5).

3.5 Design of Experiment

The investigation in this study consists of the following two experiments.

Experiment 1: Compare MF_d and MF_n on 15 open source software systems.

Experiment 2: Compare the result of the proposed method with two conventional investigation methods on five open source software systems.

The remainder of this section describes the design of these experiments in detail.

3.5.1 Experiment 1

Outline

The purpose of this experiment is to answer our research questions. This experiment consists of the following two sub-experiments.

Experiment1-1: Compare MF_d and MF_n on software systems having various sizes with a scalable clone detector, *CCFinder*.

Experiment1-2: Compare MF_d and MF_n on small size software systems with four clone detectors described in section 3.3.1.

The following items are investigated in each sub-experiment.

Item A: Investigate whether cloned code is modified more frequently than non-cloned code, with MF_d and MF_n calculated on the entire periods.

Item B: Divide the entire period into 10 sub-periods and calculate MFs on every of the sub-periods.

Target Software Systems

Experiment 1 targets 15 open source software systems shown in Table 3.1. Five software systems are investigated in Experiment 1-1, and the other 10 software systems are investigated in Experiment 1-2. The criteria for the selection of these target software systems are as follows:

- the source code is managed with Subversion;
- the source code is written in C/C++ or Java.

Note that we took care not to bias the domains of the targets.

3.5.2 Experiment 2

Outline

Experiment 2 compares the results of the proposed method and two investigation methods that are previously proposed by other researchers on the same targets. The purpose of this experiment is to reveal whether comparisons of cloned code and non-cloned code with different methods always introduce the same result. Also, we evaluate the efficacy of the proposed method compared to the other methods.

Investigation Methods to be Compared

Here, we describe investigation methods used in Experiment 2. Experiment 2 uses two investigation methods. One is proposed by Krinke [88] (in short, Krinke's method) and the other is one proposed by Lozano et al. [101] (in short, Lozano's method). Table 3.2 shows an overview of these methods and the proposed method. The selection of the two investigation methods was performed based on the following criteria:

Table 3.1: Target Software Systems - Experiment 1

(a) Experiment 1-1

Name	Domain	Programming Language	# of Revisions	LOC (Latest Revision)
EclEmma	Testing	Java	788	15,328
FileZilla	FTP	C++	3,450	87,282
FreeCol	Game	Java	5,963	89,661
SQuirrel SQL Client	Database	Java	5,351	207,376
WinMerge	Text Processing	C++	7,082	130,283

(b) Experiment 1-2

Name	Domain	Programming Language	# of Revisions	LOC (Latest Revision)
ThreeCAM	3D Modeling	Java	14	3,854
DatabaseToUML	Database	Java	59	19,695
AdServerBeans	Web	Java	98	7,406
NatMonitor	Network(NAT)	Java	128	1,139
OpenYMSG	Messenger	Java	141	130,072
QMailAdmin	Mail	C	312	173,688
Tritonn	Database	C/C++	100	45,368
Newsstar	Network(NNTP)	C	165	192,716
Hamachi-GUI	GUI,Network(VPN)	C	190	65,790
GameScanner	Game	C/C++	420	1,214,570

Table 3.2: Overview of Investigation Methods

Method	Krinke [88]	Lozano et al. [101]	Proposed Method
Target Revisions	a revision per week	all	all
Investigation Unit	line	method	place
Measure	ratio of modified lines	*work*	MF

- The investigation is based on the comparison of some characteristics between cloned code and non-cloned code.
- The method has been published at the time when our research started (at 2010/9).

In the experiments of Krinke's and Lozano's papers, only a single clone detector, *Simian* or *CCFinder*, was selected, respectively. However, this experiment applies all the four clone detectors used in Experiment 1 for bringing more general results.

We have developed software tools for Krinke's and Lozano's methods based on their papers. The followings are brief explanations for Krinke's method and Lozano's method.

Krinke's Method

Krinke's method compares stability of cloned code and non-cloned code [88]. Stability is calculated based on ratios of modified lines in cloned code and in non-cloned code. This method uses not all the revisions but a revision per week.

First of all, a revision is extracted from every week history, and then cloned code is detected from every of the extracted revisions. Next, every consecutive two revisions are compared for obtaining where added lines, deleted lines, and changed lines are. With this information, the ratios of added lines, deleted lines, and changed lines on cloned code and non-cloned code are calculated and compared.

Lozano's Method

Lozano's method categorizes Java methods, then compares distributions of maintenance costs based on the categories [101].

First, Java methods are traced based on their owner class's full qualified name, start/end lines, and signatures. Methods are categorized as follows:

AC-Method: methods that always had cloned code during their lifetimes;

NC-Method: methods that never had cloned code during their lifetimes;

SC-Method: methods that sometimes had cloned code and sometimes did not.

Lozano's method defines the following terms, where m is a method, P is a period (a set of revisions), and r is a revision.

- $ChangedRevisions(m, P)$: a set of revisions that method m was modified in period P,
- $Methods(r)$: a set of methods that existed in revision r,
- $ChangedMethods(r)$: a set of methods that were modified in revision r,
- $CoChangedMethods(m, r)$: a set of methods that were modified simultaneously with method m in revision r. If method m is not modified in revision r, it becomes 0. Note that the following formula holds if method m was modified in revision r.

$$ChangedMethod(r) = m \cup CoChangedMethod(m, r) \tag{3.6}$$

Then, this method calculates the following formulae with the above definitions. Lozano's method regards *work* as an indicator for estimating the maintenance costs.

$$likelihood(m, P) = \frac{ChangedRevisions(m, P)}{\sum_{r \in P} |ChangedMethods(r)|} \tag{3.7}$$

$$impact(m, P) = \frac{\sum_{r \in P} \frac{|CoChangedMethods(m, r)|}{|Methods(r)|}}{|ChangedRevisions(m, P)|} \tag{3.8}$$

$$work(m, P) = likelihood(m, P) \times impact(m, P) \tag{3.9}$$

In this research, we compare *work* between AC-Method and NC-Method. In addition, we also compare SC-Methods' *work* on cloned period and non-cloned period.

Target Software Systems

Experiment 2 targets five open source software systems shown in Table 3.3. Two targets, OpenYMSG and EclEmma, are selected as well as Experiment 1. Note that the number of revisions and LOC of the latest revision of these two targets are different from Table 3.1. This is because they had been being in development between the time-lag in Experiment 1 and Experiment 2. Every source file is normalized with the rules described in Section 3.4.2 as well as Experiment 1. In addition, automatic generated code and testing code had been removed from all the revisions before the investigation methods were applied.

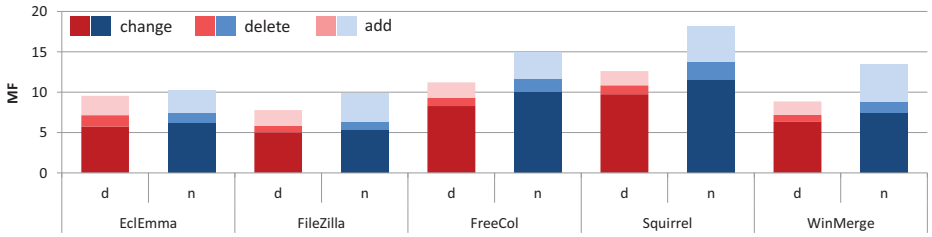


Figure 3.3: Result of Item A on Experiment 1-1

3.6 Experiment 1 - Result and Discussion

3.6.1 Overview

Table 3.4 shows the average ratio for each target of Experiment 1. Note that "ccf", "ccfx", "sim", and "sco" in the table are the abbreviated form of *CCFinder*, *CCFinderX*, *Simian*, and *Scorpio*, respectively.

Table 3.5 shows the overall result of Experiment 1. In this table, label "C" means $MF_d > MF_n$ in that case, and "N" means the opposite result. For example, the comparison result in ThreeCAM with *CCFinder* is $MF_d < MF_n$, which means cloned code was not modified more frequently than non-cloned code.

The following subsections describes the experimental results in detail.

3.6.2 Result of Experiment 1-1

Figure 3.3 shows all the results of Item A on Experiment 1-1. The labels 'd' and 'n' in X-axis means MF in cloned code and non-cloned code, respectively, and every bar consists of three parts, which means *change*, *delete*, and *add*. As shown in Figure 3.3, MF_d is lower than MF_n on all the target systems. Table 3.6 shows the average values of MF based on the modification types. The comparison

Table 3.3: Target Software Systems - Experiment 2

Name	Domain	Programming Language	# of Revisions	LOC (Latest Revision)
OpenYMSG	Messenger	Java	194	14,111
EclEmma	Testing	Java	1,220	31,409
MASU	Source Code Analysis	Java	1,620	79,360
TVBrowser	Multimedia	Java	6,829	264,796
Ant	Build	Java	5,412	198,864

results of MF_d and MF_n show that MF_d is less than MF_n in the cases of all the modification types. However, the degrees of differences between MF_d and MF_n are different for each modification type.

Figure 3.4 shows the result of Item B on Experiment 1-1. X-axis is the divided periods. Label '1' is the earliest period of the development, and label'10' is the most recent period. In the case of EclEmma, the number of periods that MF_d is greater than MF_n is the same as the number of periods that MF_n is greater than MF_d. In the case of FileZilla, FreeCol, and WinMerge, there is only one period that MF_d is greater than MF_n. In the case of Squirrel SQL Client, MF_n is greater than MF_n in all the periods. This result implies that if the number of revisions becomes large, cloned code tends to become more stable than non-cloned code. However, the shapes of MF transitions are different from every software system.

For WinMerge, we performed detail investigation on two periods. One is period '2', where MF_n is much greater than MF_d, and the other is period '10', where

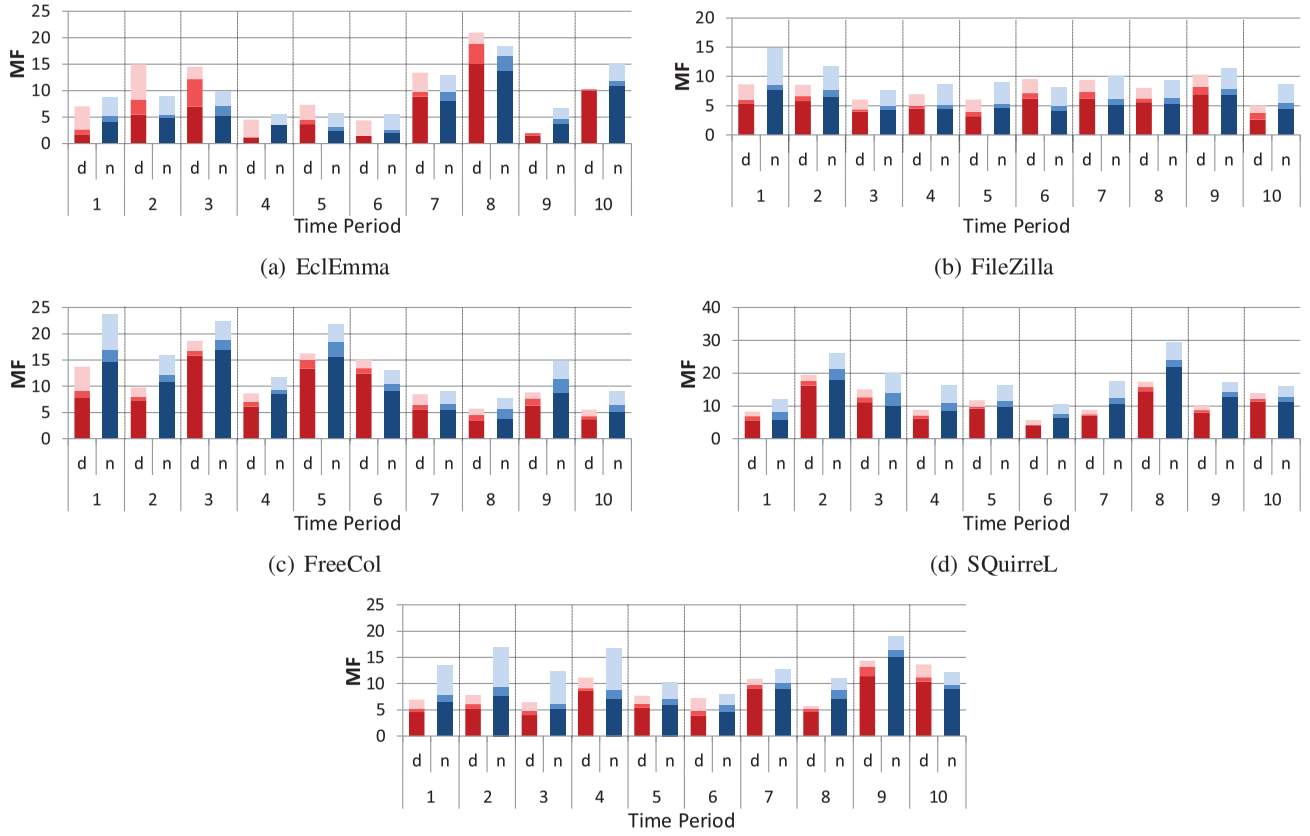
Table 3.4: Ratio of Code Clones - Experiment 1

(a) Experiment 1-1

Software Name	ccf	ccfx	sim	sco
EclEmma	13.1%	-	-	-
FileZilla	22.6%	-	-	-
FreeCol	23.1%	-	-	-
SQuirrel	29.0%	-	-	-
WinMerge	23.6%	-	-	-

(b) Experiment 1-2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	29.8%	10.5%	4.1%	26.2%
DatabaseToUML	21.4%	25.1%	7.6%	11.8%
AdServerBeans	22.7%	18.2%	20.3%	15.9%
NatMonitor	9.0%	7.7%	0.7%	6.6%
OpenYMSG	17.4%	9.9%	5.8%	9.9%
QMailAdmin	34.3%	19.6%	8.8%	-
Tritonn	13.8%	7.5%	5.5%	-
Newsstar	7.9%	4.8%	1.5%	-
Hamachi-GUI	36.5%	23.1%	18.5%	-
GameScanner	23.1%	13.1%	6.6%	-



(a) EclEmma

(b) FileZilla

(c) FreeCol

(d) SQuirreL

(e) WinMerge

Figure 3.4: Result of Item B on Experiment 1-1

is only the period that MF_d is greater than MF_n. In period '10', there are many modifications on test cases. The number of revisions that test cases are modified is 49, and the ratio of cloned code in test cases is 88.3%. Almost all modifications for test cases were performed on cloned code, so that MF_d is greater than MF_n. Omitting the modifications for test cases, MF_d and MF_n became inverted.

Table 3.5: Overall Results - Experiment 1

(a) Experiment 1-1

Software Name	ccf	ccfx	sim	sco
EclEmma	N	-	-	-
FileZilla	N	-	-	-
FreeCol	N	-	-	-
SQuirrel	N	-	-	-
WinMerge	N	-	-	-

(b) Experiment 1-2

Software Name	ccf	ccfx	sim	sco
ThreeCAM	N	C	N	N
DatabaseToUML	N	N	N	N
AdServerBeans	N	N	N	N
NatMonitor	C	C	N	C
OpenYMSG	C	C	C	N
QMailAdmin	C	C	C	-
Tritonn	N	C	N	-
Newsstar	N	N	N	-
Hamachi-GUI	N	N	N	-
GameScanner	C	C	N	-

Table 3.6: The Average Values of MF in Experiment 1-1

Modification Type	MF	
	cloned code	non-cloned code
change	7.0337	8.1039
delete	1.0216	1.4847
add	1.9539	3.7378
ALL	10.0092	13.3264

The summary of Experiment 1-1 is as follows: cloned code detected by *CCFinder* was modified less frequently than non-cloned code. Consequently, we conclude that code clones detected by *CCFinder* do not have a negative impact on software evolution even if the software is large and has a long lifetime.

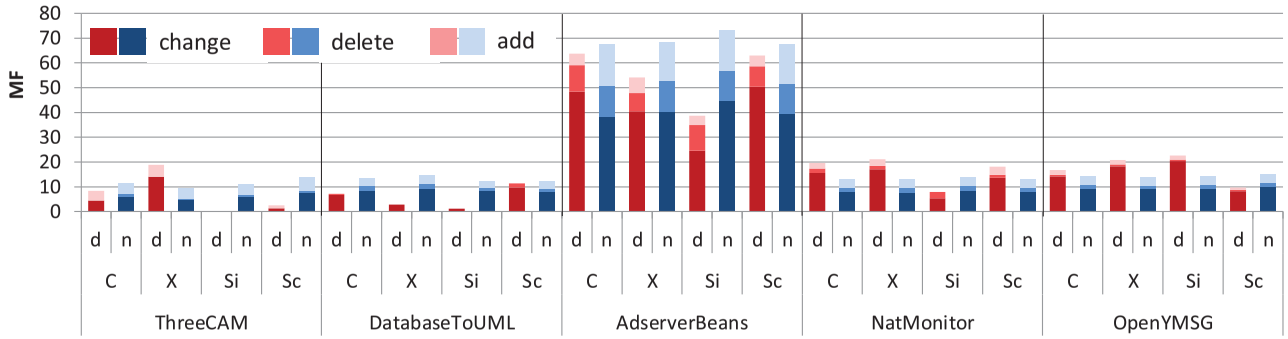
3.6.3 Result of Experiment 1-2

Figure 3.5 shows all the results of Item A on Experiment 1-2. In Figure 3.5, each clone detector is abbreviated as follows: $CCFinder \rightarrow C$; $CCFinderX \rightarrow X$; $Simian \rightarrow S_i$; $Scorpio \rightarrow S_c$. There are the results of three detectors except *Scorpio* on C/C++ systems, because it does not handle C/C++. As shown in the figure, MF_d is less than MF_n in the 22 comparison results out of 35. In the four target systems out of 10, cloned code is modified less frequently than non-cloned code in the cases of all the detectors. In the case of the other target system, MF_d is greater than MF_n in the cases of all the detectors. In the remaining systems, the comparison results are different from the detectors. In addition, we compared MF_d and MF_n based on programming languages and detectors. The comparison results are shown in Table 3.7. The result shows that MF_d is less than MF_n on all the programming languages, and MF_d is less than MF_n on the three detectors, *CCFinder*, *CCFinderX*, and *Simian*, whereas the opposite result is shown in the case of *CCFinderX*. We also compared MF_d and MF_n based on modification types. The result is shown in Table 3.8. As shown in Table 3.8, MF_d is less than MF_n in the cases of change and addition, whereas the opposite result is shown in the case of deletion.

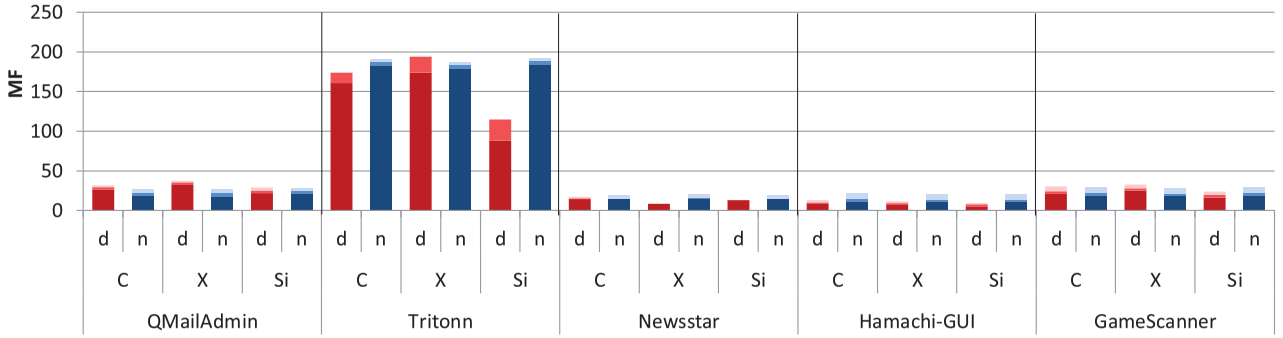
We investigated whether there is a statistically-significant difference between MF_d and MF_n by t-test. The result is as follows: there is no difference between them where the level of significance is 5 %. Also, there is no significant difference in the comparison based on programming languages and detectors. It is generally said that the presence of code clones makes it more difficult to maintain software. However, these experimental results do not show such a tendency.

Figure 3.6 shows the result of Item B on Experiment 1-2. Figure 3.6 contains graphs for the results of three software systems: Figure 3.6(a) shows the results of AdServerBeans, Figure 3.6(b) shows those of OpenYMSG, and Figure 3.6(c) shows those of NatMonitor, respectively.

In Figure 3.6(a), period '4' shows that MF_n is greater than MF_d on all the detectors meanwhile period '7' shows exactly the opposite result. In period '5', there are also hardly differences between cloned code and non-cloned code. We investigated the source code of period '4' to reveal the reason why such grate differences between MF_d and MF_n occured. In this period, many source files were created by copy-and-paste operations, and a large amount of code clones was detected by

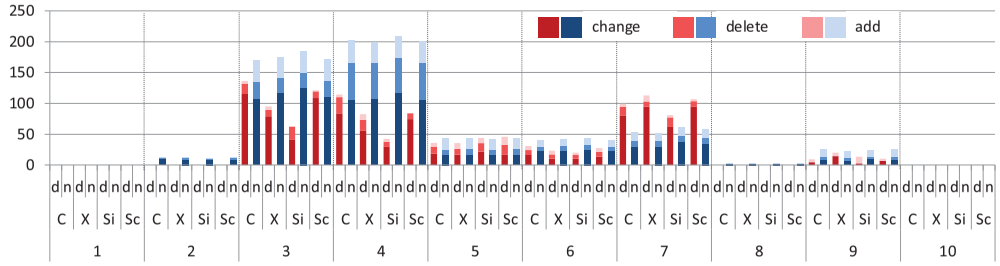


(a) Java Software

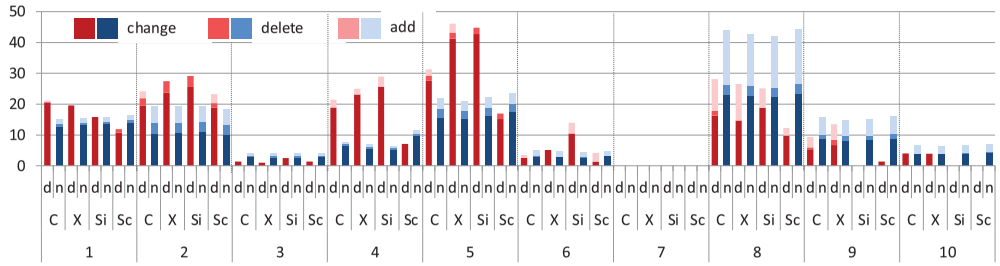


(b) C/C++ Software

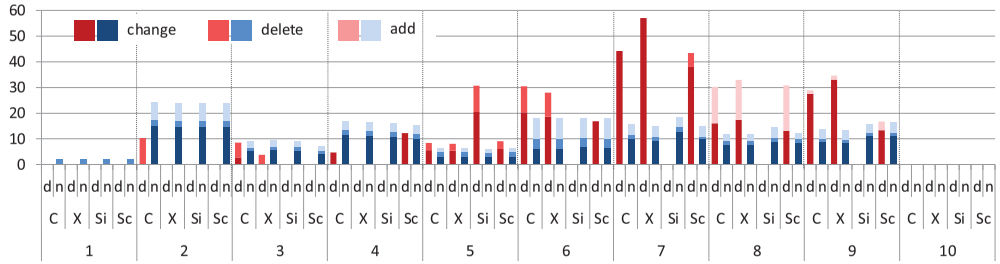
Figure 3.5: Result of Item A on Experiment 1-2



(a) AdServerBeans



(b) OpenYMSG



(c) NatMonitor

Figure 3.6: Result of Item B on Experiment1-2

each clone detector. The code generated by the copy-and-paste operations was very stable whereas the other source files were modified as usual. This is the reason why MF_n is much greater than MF_d in period '4'.

Figure 3.6(b) shows a tendency of modification frequencies: cloned code tends to be modified more frequently than non-cloned code in the anterior half of the period, whereas the opposite occurred in the posterior half. We found that there was a large number of cloned code that was repeatedly modified in the anterior half. On the other hand there was rarely such cloned code in the posterior half, which results in the occurrence of such a tendency.

Table 3.7: Comparing MFs based on Programming Language and Detector

(a) Comparison on Programming Language

Programming Language	MF	
	cloned code	non-cloned code
Java	20.4370	24.1739
C/C++	49.4868	57.2246
ALL	32.8869	38.3384

(b) Comparison on Detection Tool

Detection Tool	MF	
	cloned code	non-cloned code
CCFinder	38.2790	40.7211
CCFinderX	40.3541	40.0774
Simian	26.0084	42.1643
Scorpio	20.9254	24.1628
ALL	32.8869	38.3384

Table 3.8: The Average Values of MF in Experiment 1-2

Modification Type	MF	
	cloned code	non-cloned code
change	26.8065	29.2549
delete	3.8706	3.5228
add	2.2098	5.5608
ALL	32.8869	38.3384

Figure 3.6(c) shows an opposite tendency of modification frequencies of Figure 3.6(c): cloned code was modified more frequently in the posterior half of the period. In the anterior half, the amount of code duplications was very small, and modifications were rarely performed on it. In the posterior half, amount of clones became large, and modifications were performed on it repeatedly. In the case of *Simian* detection, no cloned code was detected except period '5'. This is because *Simian* detects only the exact-match clones whereas the other tools detect exact-match and renamed clones in the default setting.

The summary of Experiment 1-2 is as follows: we found some instances that cloned code was modified more frequently than non-cloned code in a short period on each clone detector; however, in the entire period, cloned code was modified less frequently than non-cloned code on every target software with all the detectors. Consequently, we conclude that the presence of code clones does not have a seriously negative impact on software maintenance.

3.6.4 Answers to Research Questions

This subsection offers the answers to every research question.

RQ1: *Is cloned code more frequently modified than non-cloned code?*

The answer is **No**. In Experiment 1-1, we found that MF_d is lower than MF_n in all the target systems. Also, we found a similar result in Experiment 1-2: 22 comparison results out of 35 show that MF_d is lower than MF_n, also MF_d is lower than MF_n in average. This result indicates that code clones tend to be more stable than non-cloned code. Hence, the presence of code clones does not seriously affect software evolution, which is different from the common belief.

RQ2: *Are the comparison results of stability between cloned code and non-cloned code different from multiple detection tools?*

The answer is **Yes**. In Experiment 1-2, the comparison results with *CCFinderX* are different from the results with other three detectors. Moreover, MF_n is much greater than MF_d in the case of *Simian*. At present, we cannot have found the causes of the differences of the comparison results. One of the causes may be the ratio of code clones. The ratio of code clones is quite different in each clone detector on the same software systems. However, we cannot see any relation between the ratio of code clones and values of MF.

RQ3: *Is cloned code modified uniformly throughout its lifetime?*

The answer is **No**. The results of Item B of Experiment 1-1 and Experiment 1-2 showed that there are some instances that cloned code was modified more frequently than non-cloned code in a short period even though MF_d is less than MF_n in the whole period. However, these MF's tendencies varied according to target software systems, so that we cannot find any common tendency. We also cannot find any characteristic that causes such variability.

RQ4: *Are there any differences in the comparison results on types of modifications?*

The answer is **Yes**. The experimental results in Experiment 1-1 revealed that MF_d is less than MF_n on all the modification types. However, there is a small difference between MF_d and MF_n in the case of deletion, whereas there is a large difference in the case of addition. The results in Experiment 1-2 showed similar results in the cases of change and addition: MF_d is less than MF_n. Specifically, MF_n is more than twice as large as MF_d in the case of addition. However, MF_d is greater than MF_n in the case of deletion. These results show that deletion tends to be affected by code clones, meanwhile addition tends not to be affected by them.

3.6.5 Discussion

The results of Experiment 1 showed that cloned code tends to be more stable than non-cloned code, which indicates that the presence of code clones does not have a negative impact on software evolution. To reveal the reasons why code clones are stable, we investigated how the software evolved in some of the periods that MF_d is less than MF_n. The manual investigation found that the following activities should be a part of factors that cloned code is modified less frequently than non-cloned code.

Reusing stable code: When developers want to implement new functionalities, reusing stable code will be a good way to reduce the number of introduced bugs. If most of cloned code is created by reusing stable code, MF_d becomes less than MF_n.

Using generated code: Automatically-generated code is rarely modified manually. The generated code also tends to be cloned code. Consequently, MF_d will become less than MF_n if the amount of generated code is high.

On the other hand, there are some cases that cloned code was more frequently modified than non-cloned code in a short period. The period '7' on AdServer-

Beans (Experiment 1-2, Item B) is one of these instances. We analyzed the source code of this period to detect why MF_d was greater than MF_n in this period even though the opposite results were shown in the other periods. Our manual inspection showed that there are some instances that the same modifications were applied to multiple places of code, which is a situation that the precense of code clones has a bad effect on software evolution.

Figure 3.7 shows an instance of unstable code clones. This code clone has five code fragments as its members, and the figure shows one of the code fragments included in the code clone. First, lines labeled with '%' (shown in Figure 3.7 (b)) were modified to replace the getter methods into direct accesses into fields. In the next, a line labeled with '$\#'$ is removed (shown in Figure 3.7 (c)). These two modifications were concentrically conducted in period '7'. Reusing unstable code like this example can cause additional costs for software evolution. Moreover, a code fragment was not simultaneously changed with its correspondents at the second modification. If this inconsistency was introduced unintentionally, it might cause a bug. If so, this is a typical situation that the presence of code clones affects software evolution.

3.7 Experiment 2 - Result and Discussion

3.7.1 Overview

Table 3.9 shows the average ratios of code clones included in each target, and Table 3.10 shows the comparison results of all the targets. In Table 3.10, "C" means that cloned code requires more costs than non-cloned code, and "N" means its opposite. The discriminant criteria of "C" and "N" are different in each investigation method.

In the proposed method, if MF_d is lower than MF_n, the column is labeled with "C", and the column is labeled with "N" in its opposite case.

Table 3.9: Ratio of Code Clones - Experiment 2

Software Name	ccf	ccfx	sim	sco
OpenYMSG	12.4%	6.2%	2.7%	5.5%
EclEmma	6.9%	4.8%	2.0%	3.7%
MASU	25.6%	26.5%	11.3%	15.4%
TVBrowser	13.6%	10.9%	5.4%	19.0%
Ant	13.9%	12.1%	6.2%	15.6%

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() *
             (dataGridDisplayCriteria.getPage() - 1);
  if (offsetTmp > 0) --offsetTmp;
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
  String sortColumn =
         dataGridDisplayCriteria.getSortColumn();
  Order orderTmp =
         dataGridDisplayCriteria.getOrder()
             .equals(AdServerBeansConstants.ASC) ?
                     Order.asc(sortColumn) :
                         Order.desc(sortColumn);
```

(a) Before Modification

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() *
             (dataGridDisplayCriteria.getPage() - 1);
  if (offsetTmp > 0) --offsetTmp;
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
  String sortColumn =
%         dataGridDisplayCriteria.sortColumn;
  Order orderTmp =
%         dataGridDisplayCriteria.order
             .equals(AdServerBeansConstants.ASC) ?
                     Order.asc(sortColumn) :
                         Order.desc(sortColumn);
```

(b) After 1st Modification

```
  int offsetTmp = dataGridDisplayCriteria
         .getItemsPerPage() *
             (dataGridDisplayCriteria.getPage() - 1);
#
  if (offsetTmp < 0) offsetTmp = 0;
  final int offset = offsetTmp;
  String sortColumn =
         dataGridDisplayCriteria.sortColumn;
  Order orderTmp =
         dataGridDisplayCriteria.order
             .equals(AdServerBeansConstants.ASC) ?
                     Order.asc(sortColumn) :
                         Order.desc(sortColumn);
```

(c) After 2nd Modification

Figure 3.7: An Example of Unstable Cloned Code

In Krinke's method, if the ratio of *changed* and *deleted* lines on cloned code is greater than *changed* and *deleted* lines on non-cloned code, the column is labeled with "C", and in its opposite case the column is labeled with "N". Note that herein we do not consider *added* lines because the amount of *add* is the lines of code added in the next revision, not in the current target revision.

In Lozano's method, if *work* in AC-Method is statistically greater than one in NC-Method, the column is labeled with "C". On the other hand, if *work* in NC-Method is statistically greater than one in AC-Method, the column is labeled with "N". Here we use Mann-Whitney's U-test under setting 5% as the level of significance. If there is no statistically-significant difference in AC- and NC-Method, we compare *work* in cloned period and non-cloned period in SC-Method with Wlcoxon's singed-rank test. We also set 5% as the level of significance. If there is no statistically-significant difference, the column is labeled with "-".

As this table shows, different methods and different clone detectors brought almost the same result in the case of EclEmma and MASU. On the other hand, in the case of other targets, we get different results with different methods or different detectors. Specifically, in the cases of TVBrowser and Ant, the proposed method brought the opposite results to those of Lozano's and Krinke's method.

Table 3.10: Overall Results - Experiment 2

Software Name	Method	Tools			
		ccf	ccfx	sim	sco
OpenYMSG	Proposed	N	C	C	N
	Krinke	N	C	C	N
	Lozano	-	-	N	-
EclEmma	Proposed	N	N	N	N
	Krinke	N	N	N	C
	Lozano	N	N	-	-
MASU	Proposed	C	N	C	C
	Krinke	C	C	C	C
	Lozano	C	C	C	C
TVBrowser	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C
Ant	Proposed	N	N	N	N
	Krinke	C	C	C	C
	Lozano	C	C	C	C

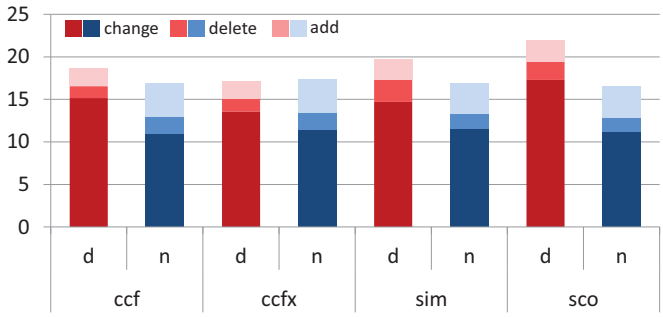


Figure 3.8: Result of the Proposed Method on MASU

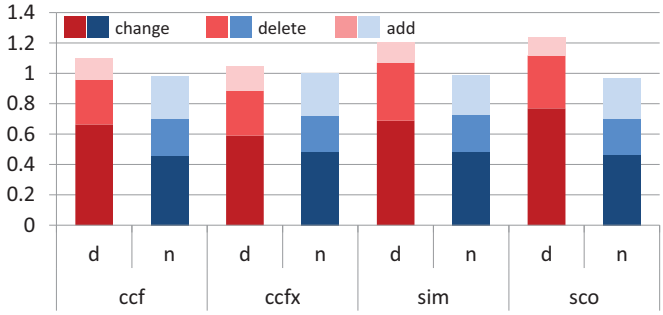
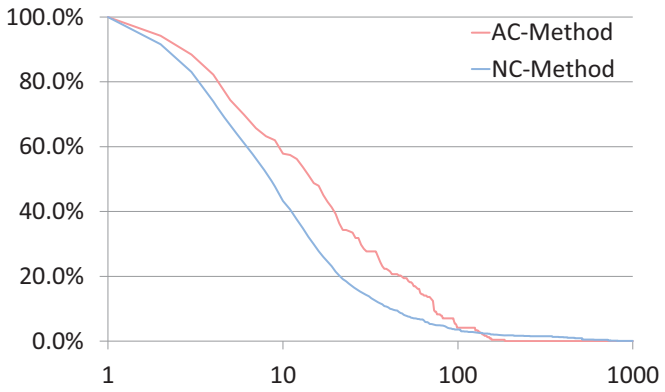


Figure 3.9: Result of Krinke's Method on MASU

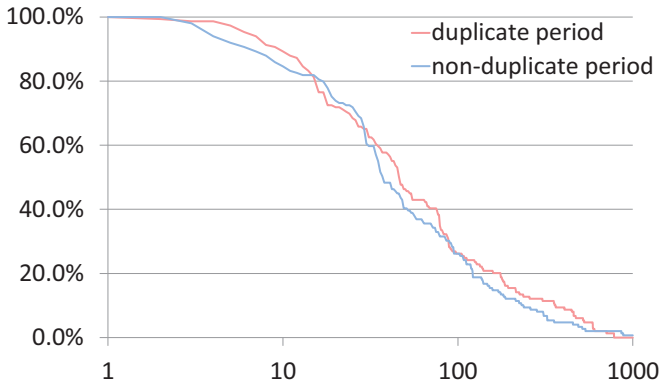
3.7.2 Result of MASU

Herein, we show comparison figures of MASU. Figure 3.8 shows the results of the proposed method. In this case, all the clone detectors except *CCFinderX* brought the same result that cloned code is more frequently modified than non-cloned code. Figure 3.9 shows the results of Krinke's method on MASU. As this figure shows, the comparison of all the detectors brought the same result that cloned code is less stable than non-cloned code. Figure 3.10 shows the results of Lozano's method on MASU with *Simian*. Figure 3.10(a) compares AC-Method and NC-Method. X-axis indicates maintenance costs (*work*) and Y-axis indicates cumulated frequency of methods. For readability, we adopt logarithmic axis on X-axis. In this case, AC-Method requires more maintenance costs than NC-Method. Figure 3.10(b) compares cloned periods and non-cloned periods of SC-Method. In this case, the maintenance cost in cloned period is greater than in non-cloned period.

In the case of MASU, Krinke's method and Lozano's method regard cloned code as requiring more costs than non-cloned code in the cases of all the detectors.



(a) AC-Method v. NC-Method



(b) SC-Method

Figure 3.10: Result of Lozano's Method on MASU with Simian

The proposed method indicates that cloned code is more frequently modified than non-cloned code with *CCFinder*, *Simian*, and *Scorpio*. In addition, there is little differences between MF_d and MF_n in the result of the proposed method with *CCFinderX*, which is the only case that cloned code is more stable than non-cloned code. Considering all the results, we can say that cloned code has a negative impact on MASU. This result is reliable because all the investigation methods show such tendencies.

3.7.3 Result of OpenYMSG

Figures 3.11, 3.12, and 3.13 show the results of the proposed method, Krinke's method, and Lozano's method on another experimental target, OpenYMSG. In

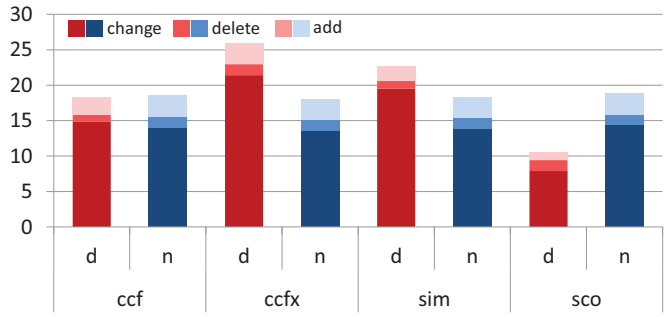


Figure 3.11: Result of the Proposed Method on OpenYMSG

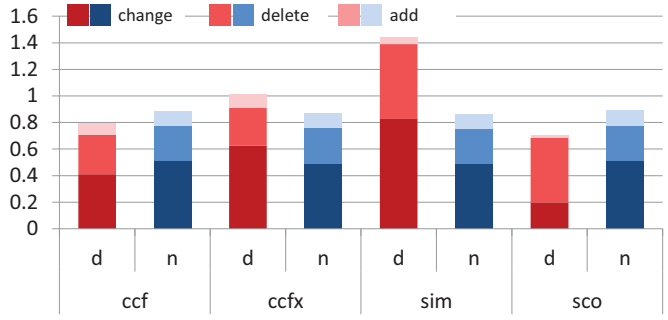


Figure 3.12: Result of Krinke's Method on OpenYMSG

the cases of the proposed method and Krinke's method, cloned code is regarded as having a negative impact with *CCFinderX* and *Simian*, whereas the opposing results are shown with *CCFinder* and *Scorpio*. In Lozano's method with *Simian*, cloned code is regarded as not having a negative impact. Note that we omit the comparison figure on SC-Method because there are only three methods that are categorized into SC-Method in OpenYMSG with *Simian*.

As these figures show, the comparison results are different for detectors or investigation methods. Therefore, we cannot judge whether the presence of cloned code has a negative impact or not on OpenYMSG.

3.7.4 Discussion

In the case of OpenYMSG, TVBrowser, and Ant, different investigation methods and different clone detectors brought opposing results. Figure 3.14 shows an actual modification in found in Ant. Two methods were modified in this modification. The hatching parts are detected as cloned code and frames in them mean pairs

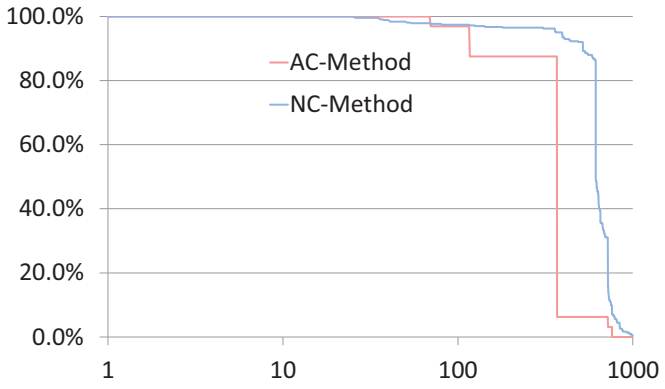


Figure 3.13: Result of Lozano's Method on OpenYMSG with Simian

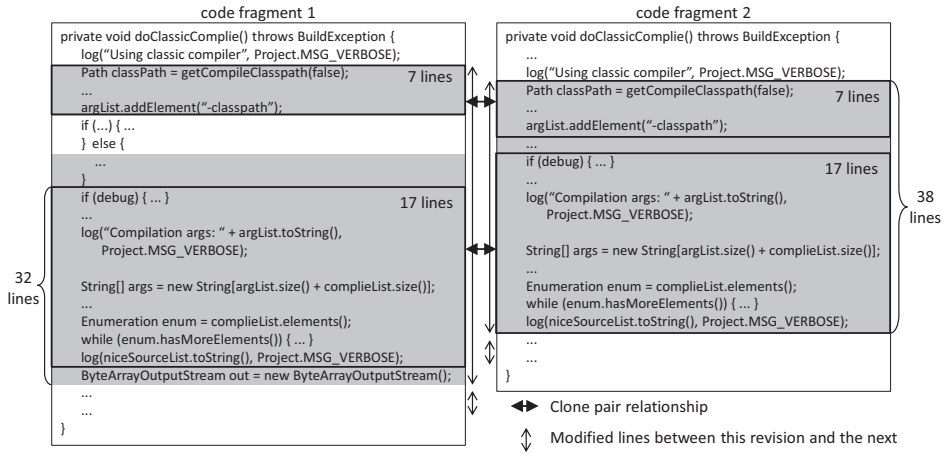


Figure 3.14: An Example of Modification by Refactoring

of cloned code between two methods. Vertical arrows show modified lines by this modification (77 lines of code were modified).

This modification is ain instance of Extract Method refactoring, which extracts the duplicated instructions from the two methods and merges them as a new method. In the proposed method, it is regarded that there are two modification places in cloned code and four places in non-cloned code, so that MF_d and MF_n became 51.13 and 18.13, respectively. In Krinke's method, $DC + CC$ and $DN + CN$ became 0.089 and 0.005, where DC, CC, DN, and CN indicate the ratio of deleted lines on cloned code, changed lines on cloned code, deleted lines on non-cloned code, and deleted lines on non-cloned code, respectively.

In this case, both the proposed method and Krinke's method regarded that cloned code required more maintenance costs than non-cloned code. However, there is a great difference in Krinke's method than the proposed method. This is caused by the difference of the barometers used in each method. In Krinke's method, the barometer depends on the amount of modified lines, whereas the barometer depends on the amount of modified places in the proposed method. This example is one of refactorings on code clones. In Krinke's method, if removed cloned code is large, cloned code is regarded as having more influence. However, in the cases of cloned code removal, we have to spend much effort if the number of cloned fragments is high. Therefore, we can say that the proposed method can accurately measure the influence of cloned code in this case.

This is an instance that is advantageous for the proposed method. However, we cannot investigate all the experimental data because the amount of the data is too vast to conduct manual checking for all the modifications. Hence, there is a possibility that the proposed method cannot accurately evaluate the influence of cloned code in some situations.

In Experiment 2, we found that the different investigation methods or different detectors draw different results on the same target systems. In Experiment 1, we found that cloned code is less frequently modified than non-cloned code. However, the result of Experiment 2 indicates that we cannot generalize the result of Experiment 1. We have to conduct more experiments and analyze the results of them in detail to gain more general results.

3.8 Threats to Validity

This section describes threats to validity of this study.

Cost required for every modification

In this study, we assume that cost required for every modification is equal to one another. However, the cost must be different between every modification in the actual software evolution. Consequently, the comparison based on MF values may not appropriately represent the costs required for modifying cloned code and non-cloned code.

In addition, when we modify cloned code, we have to consider keeping the consistency between the modified cloned code and its correspondents. If the modification lacks consistency by error, we have to re-modify them for repairing the consistency. The effort for keeping consistency is not necessary for modifying non-cloned code. Consequently, the average costs required for cloned code may

be different from the one required for non-cloned code. In order to compare them more appropriately, we have to consider the costs for keeping consistency.

Identifying the number of modifications

This study regards modifying consecutive multiple lines are regarded as a single modification. However, it is possible that such an automatically processing identifies an incorrect number of modifications. If multiple lines that were not contiguous are modified for fixing a single bug, the proposed method presumes that multiple modifications were performed. On the other hand, if multiple consecutive lines were modified for fixing two or more bugs by chance, the proposed method presumes that only a single modification was performed. Consequently, it is necessary to manually identify modifications if we have to use the exactly correct number of modifications.

Besides, we investigated how many the identified modifications occurred across the boundary of cloned code and non-cloned code. If this number is high, then the analysis suspects because such modifications increase both the counts at the same time. The investigation result is that, in the highest case, the ratio of such modifications is 4.8%. That means that almost all modifications occurred within either cloned code or non-cloned code.

Category of modifications

This study does not consider meanings of modifications, and so we counted all the modifications regardless of their meanings. As a result, the number of modifications might be incorrectly increased by unimportant modifications such as format transformations. A part of unimportant modifications remained even if we had used the normalized source code described in section 3.4.2. Consequently, manual categorization for the modifications is required for using the exactly correct number of modifications.

In addition, the code normalization used in this study removed all the comments in the source files. If considerable cost was expended to make or change code comments on the development of the target systems, we incorrectly missed the cost.

Property of target software

In this study, we used only open source software systems, so that different results may be shown with industrial software systems. It is generally said that industrial software includes more cloned code than open source software. Consequently, cloned code may not be managed well in industrial software, which may

increase MF_d. Also, properties of industrial software are quite different from ones of open source software. In order to investigate the impact of cloned code on industrial software, we have to compare MF on industrial software systems.

Division of development period

In this study, we divided the development periods of target software systems in an automatic way based on the number of revisions. However, a different division may yield different results. For example, if we divide the periods based on the border of versions, we may be able to grasp the properties of every version. Or, more fine grained division, that is, the period of every version is divided into multiple sub-periods, which will let us know how cloned code and non-cloned code are modified from a start of a version to the end of the version.

Settings of detection tools

In this study, we used default settings for all clone detectors. All the four clone detectors used in this study have flexible settings, including the minimum length of detected code clones or the way of code normalization before clone detection. It is natural that the same clone detector reports different clones even in the same software if it runs under different settings. Therefore, different results will be shown if we adopt each clone detector in different settings.

3.9 Summary

In this study, we conducted an empirical study on the impact of the presence of cloned code on software evolution. We assumed that if cloned code is modified more frequently than non-cloned code, the presence of cloned code affects software evolution, and compared the stability of cloned code and non-cloned code. To evaluate from a different standpoint from previous studies, we used a new indicator, modification frequency, which is calculated with the number of modified places of code. In addition, we used four clone detectors to reduce the bias of detectors. We conducted an experiment on 15 open source software systems, and the result showed that cloned code was less frequently modified than non-cloned code. We also found some cases that cloned code was intensively modified in a short period though cloned code was stable than non-cloned code in the whole development period.

Moreover, we compared the proposed method to other two investigation methods to evaluate the efficacy of the proposed method. We conducted another experiment on five open source software systems, and in the cases of two targets, we

got the opposing results to other two methods. We carefully investigated the result in detail, and found some instances that the proposed method could evaluate more accurately than other methods.

Our experimental result showed that cloned code tends to be stable than non-cloned code though there exists some clones that negativelly affect software evolution. However, more studies are required to generalize this result, because we found that different investigation methods may bring different results. In summary, our experimental results indicated that it is necessary to select and address harmful clones to be managed from all the detected clones to achive effective clone management.

Chapter 4

Enhancing CRD-based Clone Tracking based on Similarity of CRD

4.1 Motivation

As described in Chapter 3, our experimental results revealed that not all but a part of clones has negative impacts on software evolution. However, there are still some open issues. One of the issues is *how many clones have negative impacts on software evolution*. To tackle these open issues, it is necessary to analyze clones in more detail ways. That is, extracting and analyzing clone genealogies are required.

A clone genealogy indicates how a clone has been evolved across the version history of the software having the clone [77, 78]. Therefore, using clone genealogies enables to analyze how many times a *particular* clone has been changed. This means that we can know how many clones have negative impacts on software evolution, which is a fundamental qustion not addressed by our previous study.

There are some research reports that analyze clone genealogies [5, 8, 35, 78, 87, 143]. However, these studies focus on investigating how clones have evolved, not tracking clones. That means, tracking techniques that they used were not feasible if the clones were drastically modified or the clones were moved into other files between the revisions.

Currently, there are several techniques for tracking clones. Some techniques detect clones from every revision, and then link them between every of two consecutive revisions [8, 33, 78]. These techniques link clones based on textual similarity of clones between two revisions, thus they will miss some links of clones if modifications applied on the clones are not small, or the clones were moved. Some

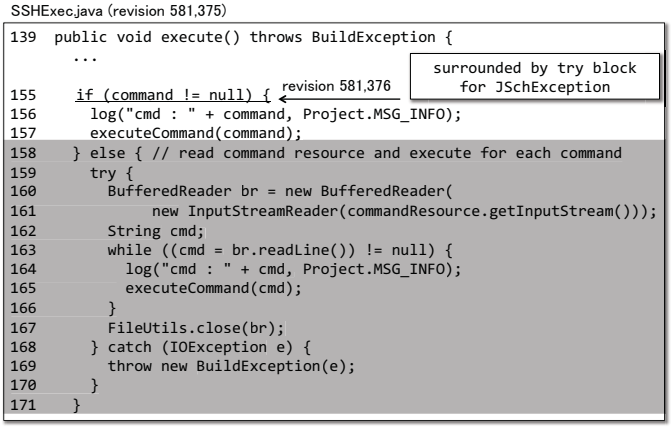


Figure 4.1: Actual modification that existing techniques cannot track clones

other techniques detect clones from the initial revision, then tracking the clones using change histories stored in historical code repositories [5, 87, 143]. These techniques are robust for modifications on clones. That means, they can extract some links of clones that techniques based on textual similarities of clones miss. However, they have a weak point that they cannot track clones that appeared during the target period because they are interested in only clones that exist in the initial revision.

Among these techniques that track clones across version histories, Clone Region Descriptor (in short **CRD**) based tracking is one of the best techniques. CRD was proposed by Duala-Ekoko and Robillard [27], and it is an abstract form to represent locational information of a clone. Clone tracking using CRDs is robust for large modifications on clones compared to the other clone tracking techniques. The reason is that the linking technique of clones with CRD is based on locations of clones, not their textual similarities. The location-based linking enables to link clones even if they were drastically modified between two revisions.

Even though CRD is a well-designed clone tracking technique, it still has a room for improvement. This means that the CRD based tracking is not so robust for changes that affect locations of clones. Figure 4.1 shows an actual example of code modifications on which the existing CRD-based technique cannot work well, which is found in Ant. In the figure, the hatched part is detected as a clone. In revision 581,376, a `try`-block for catching `JSchException` was added outside of the clone. This modification also added a new parameter to the signature. In the CRD-based technique, blocks are tracked based on the nesting structures. Consequently, the technique cannot regard that the `else`-block after the modification corresponds to the `else`-block inside the new `try`-block. As a result, the clone is regarded as

disappeared.

Tracking clones is a fundamental technique in the field of software evolution because it should be a basis of a variety of research relating to software evolution. Consequently, accurate and scalable clone tracking techniques are necessary for better results of research.

This study proposes another clone tracking technique based on CRDs. The proposed technique borrows the basic idea of the original CRD-based tracking, with the idea enhanced. The key idea of the enhancement is to use similarities of CRDs, not CRDs themselves. Using similarities of CRDs enables clone tracking to be robust compared to the original CRD, with detecting few false positives. In addition, we have implemented a clone tracking system based on the proposed method. The system includes a function to detect clones, which has a high scalability because it is incremental and block-based. This study evaluates the correctness of clone tracking of the proposed method, and also shows an application result of the proposed method for revealing why clones are gone in software evolution, which has not been investigated yet.

4.2 Tracking clones

This section describes the proposed clone tracking technique. The primary requirements for tracking clones are *accuracy* and *speed*. In order to achieve accurate and rapid clone tracking, the proposed technique uses incremental hash-based clone detection. The processing of the proposed technique consists of two phases, **hash generation** and **clone linking**, each of which is described briefly as follows.

HASH GENERATION

Hash values are calculated from every block in source files in every revision. If two or more blocks have the same hash value, they are regarded as clones. All the hash values are stored into a database with their locational information such as file paths, start lines, and end lines. In addition, a CRD [27] is measured from every block, and it is also stored into the database.

CLONE LINKING

Cloned blocks in every revision are linked to blocks in its next revision based on similarities of their CRDs. Then, their hash values are checked: if two blocks having the same hash value (h_r) in revision r have also the same hash value (h_{r+1}) in revision $r+1$, they are regarded as maintaining a clone relationship during r and $r+1$ even if h_{r+1} is different from h_r.

The remainder of this section firstly introduces the definition CRD with a simple example in Subsection 4.2.1. Secondly, it describes the processings of the phase

```
<CRD> ::= <file> <class> <CM> [<method>]
<method> ::= <signature> <block>*
<block> ::= <btype> <anchor> <CM>
<btype> ::= 'for' | 'while' | 'do' | 'if' | 'else' | 'switch' |
            'try' | 'catch' | 'finally' | 'synchronized'
```

(a) Definition

```
public class DeleteManager {
   ...
   public void delete (int n) {
      ...
      for(int i = n ; i < delete.size() ; i++ ) {
         ...
         if (delete.get(i) instanceof ElementNode) {
            // some code
         }
         ...
```

A

(b) Code situation

```
packagename.DeleteManager.java,DeleteManager,5
delete(int),5
for,delete.size(),4
if,delete.get(i) instanceof ElementNode,2
```

(c) Example

Figure 4.2: Clone Region Descriptor

"*hash generation*" in Subsection 4.2.2. Finally, it explains the processings of the phase "*clone linking*" in Subsection 4.2.3.

4.2.1 Clone Region Descriptor

CRD (, which is an abbreviation of **Clone Region Descriptor**) is an abstract description for location of a clone region in a software system. A CRD provides an approximate location of a clone. CRD is independent of specification of based on lines of source files. Figure 4.2 shows the definition and an example of CRD[1]. Figure 4.2(a) shows the definition of CRD in the extended Backus-Naur form. Figure 4.2(c) shows an example of CRD, which represents the block labeled with "**A**" in Figure 4.2(b). In the technique proposed by Duala-Ekoko and Robillard [27], if a block in revision r has the same CRD as a block in revision $r+1$, they are regarded

[1]These figures shows the same example used in the literature [27].

as the same block. On the other hand, the proposed technique uses the similarities of CRDs for clone tracking to improve the clone tracking based on CRDs.

4.2.2 Hash generation

The purpose of this step is to analyze source code in every revision in the target software system and to store the results into a database. The processing of this phase takes a repository of a target system as its input. From the given repository, it extracts every block in all the source files of all the target revisions, calculates hash values and CRDs for all the extracted blocks, and store the result of the calculation into a database. Thus, the output of this processing is a database storing all the blocks with hash values and CRDs. There are five steps in this processing as follows.

STEP1 (Syntax Analysis) This step performs lexical and syntax analyses for all the source files. Every source file is transformed into a sequence of tokens through lexical analysis. After the lexical analysis, this step detects all the blocks from the sequences of tokens through syntax analysis. Note that the syntax analysis is necessary because performing only lexical analysis is not sufficient for identifying blocks from source files.

STEP2 (Splitting) Every subsequence of tokens corresponding to a block is extracted from the token sequences of the source files in this step. This step puts a special token into the extracted position of token sequences that represents the information of the extracted block. The information described in the special token includes the following information:

- type of the block (e.g., *if*, *while*, *for*),
- the conditional predicate (if the block is conditional one), and
- parameters (if the block is method or constructor).

STEP3 (Normalization) This step normalizes token sequences to detect clones even if they have some trivial differeces. In this normalization, variable names and literals are replaced with special tokens. Note that invoked method names and type names are not normalized. It is easy to distinguish types of tokens because syntax analysis performed in STEP1 provides type information of tokens. The type information enables to normalize only variable names and literals.

STEP4 (Building block texts) This step builds a character sequence for every of token sequences that represent blocks. A hash value is calculated from every

of the character sequences with a hash function. A CRD is also calculated from every of them in this step.

STEP5 (Persisting hash values) Hash values of all the blocks are stored into a database. At the same time, file paths, start lines, end lines, revision numbers, and CRDs of every block are stored into the same database.

Figure 4.3 shows how source files are handled in every step. A remarkable point of the processing "*hash generation*" is included in STEP2, which is a set of operations that every of the sub-blocks in a block is extracted, then a small marker is put into every of the extracted positions. This operation enables to identify blocks including the same instructions as clones even if their sub-blocks are different. If a block is totally duplicated to another block, their hash values and all the hash values of their sub-blocks become the same.

4.2.3 Clone Linking

The input of the processing of this phase is an output of the processing "*hash generation*", which is a database that contains hash values and other attributes of all the blocks in all the target revisions. This phase links blocks in every revision to blocks in the next revision.

Hash values of blocks are used for identifying clone relationships in every revision. The same hash value means that blocks having the hash value are texutually identical except for differences in their sub-blocks, variable names, and literals. In other words, blocks having the same hash value are regarded as clones. The remainder of this paper uses a term **Equivalent Block Group** (in short, **EBG**), which represents a group of blocks having the same hash value.

In this processing, the proposed technique identifies links of blocks between revisions r and $r+1$ based on the similarities of their CRDs. If block b_r in revision r and block b_{r+1} in revision $r+1$ satisfy all the following conditions, b_{r+1} is regarded as the corresponding block of block b_r.

CONDITION1 The type of b_{r+1} is the same as the one of b_r.

CONDITION2 If b_r and b_{r+1} are conditional blocks, their conditions are the same except variable names, method names and literals in them.

CONDITION3 If b_r and b_{r+1} are methods or constructors, their names are the same or their parameters are the same.

CONDITION4 In revision $r+1$, CRD of b_{r+1} has the highest similarity with one of b_r.

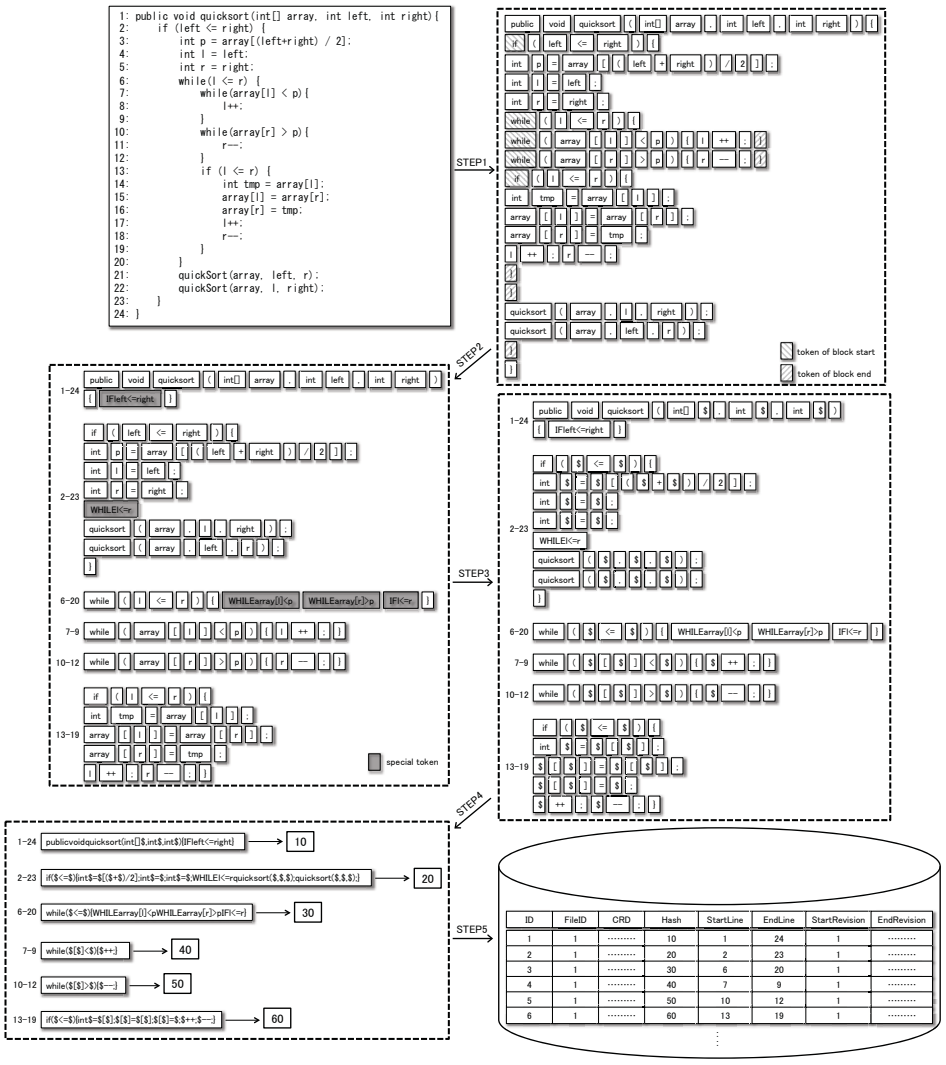


Figure 4.3: Intuitive example how hash values are measured from source code

CONDITION5 In revision r, CRD of b_r has the highest similarity with one of b_{r+1}.

The CRD similarity between blocks b_r and b_{r+1} are measured by Levenshtein Distance (in short, LD) of string representations of two CRDs calculated from b_r and b_{r+1}. If the two blocks have the identical CRDs, the LD between them becomes 0, which is the minimum value of LD. If the two blocks have completely different CRDs, the LD becomes maximum value, which is the higher value of the lengths of two string representations of two CRDs. Considering the similarity of two CRDs enables to track clones even if they were moved.

Figure 4.4 shows examples of a revision history and a result of clone linking from it. In Figure 4.4(a), the following modifications are performed.

- Method $b2$ in file $B.java$ was changed ($r_1 \rightarrow r_2$)
- Method $a1$ in file $A.java$ was changed ($r_2 \rightarrow r_3$)
- $b2$ was moved to file $C.java$ ($r_2 \rightarrow r_3$)
- Method $c1$ in $C.java$ was deleted ($r_2 \rightarrow r_3$)
- $b1$ in $B.java$ was changed ($r_3 \rightarrow r_4$)

In Figure 4.4(b), circles, arrows, and rectangles have the following meanings.

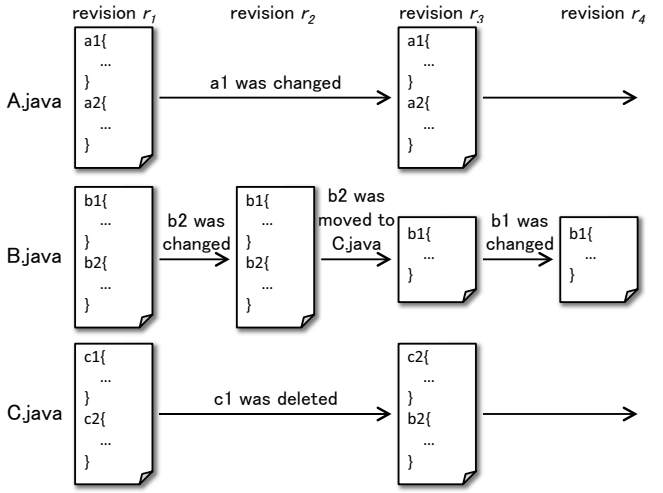
Circles: Every circle means a block. Numbers in blocks are their hash values.

Arrows: Every arrow means a link of two blocks between consecutive two revisions.

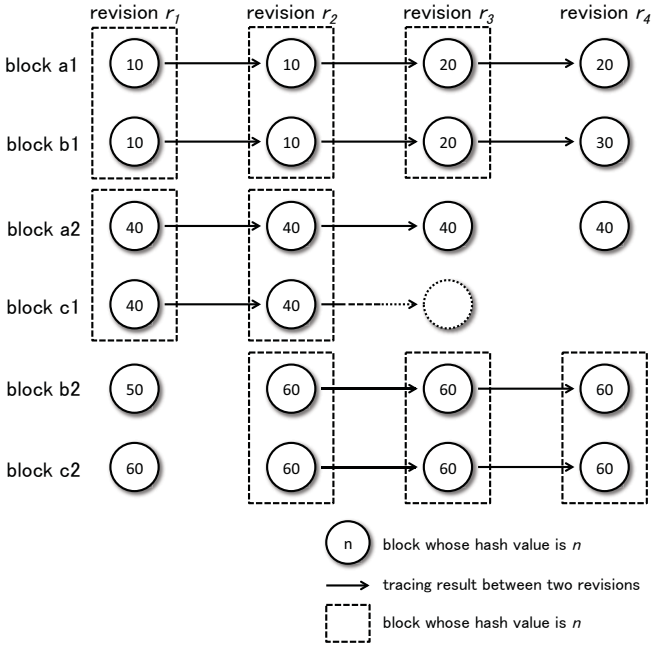
Rectanbles: Every rectangle means an EBG. All the blocks in a rectangle have the same hash value.

For example, blocks $a1$ and $b1$ are modified between revisions r_2 and r_3, so that each of them has different hash values between the revisions. Even after the modification, they have the same hash value, which means that they continue to be duplicated in revision r_3. However, between revisions r_3 and r_4, only block $b1$ is modified, and the two blocks have different hash values to each other in revision r_4. Hence, the EBG consisting of $a1$ and $b1$ is regarded as disappeared in revision r_4.

Blocks $a2$ and $c1$ have the same hash value in revisions r_1 and r_2, which means that they consist of a clone in the revisions. However, in revision r_3, block c_1



(a) Revisions



(b) Tracking result

Figure 4.4: Example of clone tracking

itself disappeared. In this case, the EBG consisting of $a2$ and $c1$ is regarded as disappeared in revision r_3.

Blocks $b2$ and $c2$ have the different hash values in revision r_1. Block $b2$ is modified between revisions r_1 and r_2, so that they have the same hash value in revision r_2. Even block $b2$ was moved to C.java in revision r_3, it can be tracked. This is because the proposed method tracks cloned blocks based on similarities of their CRDs.

4.3 Implementation

We have developed a software tool, **CTEC** based on the proposed technique. Currently, *CTEC* handles only Java language. However, it is not difficult to expand *CTEC* to be able to handle other programming languages because it performs only lexical and syntax analyses as language dependent processings. The language dependent processing has been implemented with *Java Development Tools* [60].

CTEC uses *SQLite* as the database module because of its ease to use. Any other database systems can take the place of *SQLite*. If we use other SQL database systems such as *PostgreSQL* or *Oracle* database, the analysis speed will become more rapid. However, we believe that the speed with *SQLite* is sufficient.

In *CTEC*, the two processings "*hash generation*" and "*clone linking*" are implemented as independent ones. "*Hash generation*" registers hash values of blocks in the target revisions into an SQL-based database. "*Clone linking*" takes the SQL-based database that the "*hash generation*" phase outputs, and identifies links of blocks in the next revision for every cloned block in every revision with comparing similarities of CRDs.

The remainder of this section describes each of the processings, respectively.

4.3.1 Hash generation

In order to achieve high scalability, *CTEC* adopts an incremental hash generation. The following is an explanation of "*processing for the first revision*" and "*processing for the 2nd or later revisions*". This explanation is under an assumption that the target revisions are $\{r_1, r_2, \cdots, r_n\}$. Figure 4.5 describes an example of initializing and updating a database.

In the processing for the first revision (r_1), all the blocks in all the source files in revision r_1 are stored into an SQL database. Note that column "*EndRevision*" of all the blocks are set as r_n, which is the last revision of the target.

In the processing for the 2nd or later revision ($r_k, 2 \leq k \leq n$), blocks in only source files modified, added, and deleted in r_k are stored into the database. If

After analyzing revision r_1

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_4
1	A.java	A.a2	40	11	20	r_1	r_4
2	B.java	B.b1	10	1	10	r_1	r_4
3	B.java	B.b2	50	11	20	r_1	r_4
4	C.java	C.c1	40	1	10	r_1	r_4
5	C.java	C.c2	60	11	20	r_1	r_4

After analyzing revision r_2

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_4
1	A.java	A.a2	40	11	20	r_1	r_4
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	C.c1	40	1	10	r_1	r_4
5	C.java	C.c2	60	11	20	r_1	r_4
6	B.java	B.b1	10	1	10	r_2	r_4
7	B.java	B.b2	60	11	20	r_2	r_4

After analyzing revision r_3

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_2
1	A.java	A.a2	40	11	20	r_1	r_2
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	B.b1	40	1	10	r_1	r_2
5	C.java	B.b2	60	11	20	r_1	r_2
6	B.java	B.b1	10	1	10	r_2	r_2
7	B.java	B.b2	60	11	20	r_2	r_2
8	A.java	A.a1	20	1	10	r_3	r_4
9	A.java	A.a2	40	11	20	r_3	r_4
10	B.java	B.b1	20	1	10	r_3	r_4
11	C.java	C.c2	60	1	10	r_3	r_4
12	C.java	C.b2	60	11	20	r_3	r_4

After analyzing revision r_4

ID	FileID	CRD	Hash	StartLine	EndLine	StartRevisio	EndRevision
0	A.java	A.a1	10	1	10	r_1	r_2
1	A.java	A.a2	40	11	20	r_1	r_2
2	B.java	B.b1	10	1	10	r_1	r_1
3	B.java	B.b2	50	11	20	r_1	r_1
4	C.java	B.b1	40	1	10	r_1	r_2
5	C.java	B.b2	60	11	20	r_1	r_2
6	B.java	B.b1	10	1	10	r_2	r_2
7	B.java	B.b2	60	11	20	r_2	r_2
8	A.java	A.a1	20	1	10	r_3	r_4
9	A.java	A.a2	40	11	20	r_3	r_4
10	B.java	B.b1	20	1	10	r_3	r_3
11	C.java	C.c2	60	1	10	r_3	r_4
12	C.java	C.b2	60	11	20	r_3	r_4
13	B.java	B.b1	30	1	10	r_4	r_4

Figure 4.5: Example of Database Updating

the database already includes blocks in the files, their columns "*EndRevision*" are updated to r_{k-1}.

The following explanation uses the example shown in Figure 4.5 to describe how these processings work with. This example shows how database is updated for revisions shown in Figure 4.4. Figure 4.5 shows the database content after the processing for every revision. Gray cells mean that they have just been inserted or updated. Note that, in this example, column "*FileID*" contains file names for ease to explain. However, in the actual implementation, column "*FileID*" includes IDs for source files. *CTEC* has another database table for mapping file name and its ID.

In the processing for revision r_1, which is the first revision of the target, all the source files are analyzed and their blocks are stored into the database. Note that the values of column "*EndRevision*" of their blocks become r_4, which is the last revision of the target.

In the processing for revision r_2, file *B.java* is reanalyzed, and its blocks are stored into the database. The reason why only file *B.java* is reanalyzed is that only file *B.java* is modified between revisions r_1 and r_2. In this case, the database already has blocks in file *B.java*, and so their columns "*EndRevision*" are updated to r_1. This updating operation is performed for only blocks whose "*EndRevision*" are r_4.

In revision r_3, files *A.java*, *B.java*, and *C.java* are modified. Hence, the processing for those files in revision r_3 is performed as well as the processing for file *B.java* in revision r_2. Note that the rows whose "*ID*" values are 2 and 3 are no longer updated even though their owner file *B.java* is modified. This is because the values of "*EndRevision*" in these rows are not r_4.

In a similar way, the processing for revision r_4 analyzes file *B.java*, inserts newly detected blocks into the database, and updates the column "*EndRevision*" at the row whose "*ID*" is 10.

4.3.2 Clone Linking

In order to link cloned blocks in every revision to corresponding blocks in the next revision (r_k and r_{k+1}), it is necessary to obtain EBGs in revision r_k from the given database. The followings describe the steps to obtain EBGs in revision r_k.

STEP1 obtaining records (blocks) satisfying the following formula. This operation means obtaining all the blocks existing in r_k.

$$(StartRevision \leq r_k) \quad \bigwedge \quad (r_k \leq EndRevision) \tag{4.1}$$

STEP2 classifying the blocks obtained in the STEP1 based on their hash values. Two or more blocks having the same hash value form an EBG.

For each block in the EBGs identified in the STEP2, its corresponding block is found with the proposed technique described in Subsection 4.2.3.

4.4 Experiment

This section describes the result of an experiment on well-known systems with the proposed clone tracking technique. The purpose of this experiment is confirming that the proposed technique has a beneficial effect on tracking clones. In order to achieve the purpose, this experiment has investigated tracking results in the way to answer the following questions.

QUESTION1 Could the proposed technique track clones that the conventional technique could not track?

QUESTION2 Did clones that the proposed technique could not track really disappear?

Firstly, this section describes the experimental setup, then shows the performance of *CTEC*. Lastly, it provides answers for our research questions.

4.4.1 Setup

We selected Ant and ArgoUML as the targets of this experiment. Tables 4.1 and 4.2 show an overview of the target systems. They are managed by using *Subversion* (in short, *SVN*) The targets of the investigation are the source code under directory

Table 4.1: Overview of Target Software - Target Revisions -

Software	Start revision (date)	End revision (date)	# of target revisions
ArgoUML	15,880 (2008-10-04)	19,794 (2011-11-17)	2,222
Ant	268,587 (2001-02-05)	904,537 (2010-01-30)	5,143

Table 4.2: Overview of Target Software - LOC -

Software	LOC of start revision	LOC of end revision
ArgoUML	329,170	362,604
Ant	57,124	211,855

"*/ant/core/trunk/src/main*" and "*/trunk/src*", respectively. This experiment is interested only in the source code under the directories "*trunk*", which is the main line of the development in *SVN* repositories. This experiment also narrows down the experimental target to particular subdirectories of "*trunk*" to exclude test files. The target period of the investigation was carefully decided because we would like to investigate development histories of multiple versions.

In this experiment, we specified 30 tokens as the threshold of minimum clone length. If and only if a block includes 30 or more tokens, it can be stored into databases and can be a member of a code clone. On the other hand, blocks including less than 30 tokens will be ignored, and so they are not inserted into the database.

4.4.2 Performance

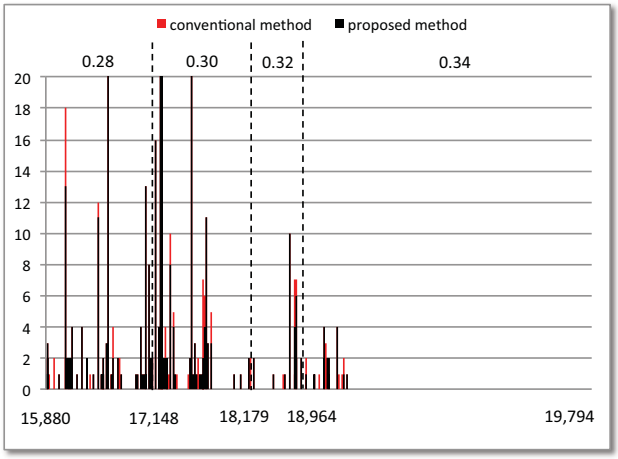
Table 4.3 shows the performance of two processings, "*hash generation*" and "*clone linking*", from the view point of the elapsed time. A processing "*hash generation*' consists of the analysis on all the target revisions of a target software system. Hence, the processing "*hash generation*" was performed only once on each of the experimental targets. On the other hand, a processing "*clone linking*" means analyzing a pair of two consecutive revisions. Therefore, "*clone linking*" needs to be performed multiple times, the number of pairs of two consecutive revisions, on each of the experimental targets. The table shows total, maximum, minimum, and average time of "*clone linking*". We can see that maximum time is 46 seconds, which means that *CTEC* can perform a processing "*clone linking*" interactively on demand from users. In the case of batch executions, *CTEC* takes a few hours to analyze several thousand of revisions. Considering the elapsed time shown in the table, it can be said that *CTEC* works very well in the point of scalability.

4.4.3 Answer to QUESTION1

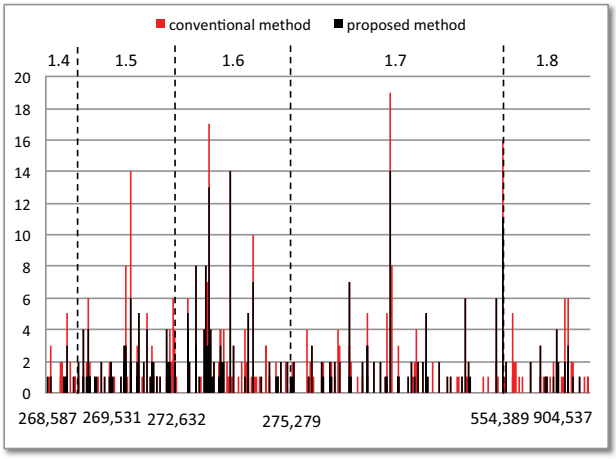
We tracked clones by using the proposed technique and a conventional technique proposed by Duala-Ekoko and Robillard [27]. The conventional technique has been implemented by us based on the description of the paper of Duala-Ekoko

Table 4.3: Timing information on experiment (execution with eight threads)

Software	Hash generation	Clone linking			
		total	max.	min.	ave.
ArgoUML	132 mins.	43 mins.	46.0 secs.	0.043 secs.	21.3 secs.
Ant	100 mins.	50 mins.	31.2 secs.	2.8 secs.	17.1 secs.



(a) ArgoUML



(b) Ant

Figure 4.6: Number of blocks that were not tracked by the proposed or conventional methods

and Robillard. In the whole of the target period, the number of untrackable clones of the proposed technique and the conventional technique is 345 and 581 in Ant, and 537 and 739 in ArgoUML, respectively. There exists no clones that the conventional technique tracked but the proposed technique could not. That is, 236 and 202 clones were tracked only by the proposed technique, respectively.

Figure 4.6 shows the number of clones that did not tracked by the proposed or conventional techniques in every revision. Clones not tracked by the conventional

technique are colored red, and ones not tracked by the proposed technique are colored black. Every black bar is drawn in front of red ones. If a red bar is taller than its corresponding black bar, there are clones that were tracked only by the proposed technique. The length of difference between red and black bars means the number of such clones. Again, there exists no clones that the conventional technique tracked but the proposed technique could not. Therefore, no black bar is taller than its corresponding red bar.

We investigated clones tracked only by the proposed technique to reveal whether tracking by the proposed technique had been correct or not. This was a manual investigation, so that we narrowed down period "1.8" in Ant and "0.32" in ArgoUML. In this investigation, we checked what kinds of modifications were performed in both the cases that tracking was correct and not correct. The following is a list of kinds of modifications. Prefix "**T**" means that tracking was correct even if the modifications were performed and "**F**" means that tracking was incorrect because of the modifications.

T1 Clones (and their surrounding code) were extracted as new methods.

T2 New blocks were added as surrounding code of clones such as null checking.

T3 Conditional predicates on conditional blocks including clones were changed.

T4 Methods including clones were moved to other classes.

T5 New catch clauses were added on try blocks including clones. In the CRD definition of `try`-block, it includes exception types of catch clauses attached to the `try`-block [27]. Consequently, if a new catch clause is added to a try block, CRD of the try block changes.

T6 Methods were inlined to other methods.

F1 Cloned blocks were deleted. As a result, other blocks in the next revision were incorrectly linked to the deleted blocks.

F2 Cloned block became smaller than the threshold (30 tokens). Smaller blocks than the threshold are not registered to the database. They are treated as the same as deleted blocks. As a result, incorrect linking happened.

Figure 4.7 shows actual clones classified into **T1**. In this case, cloned `if`-blocks and their subsequent code were extracted as new methods. Such modifications change CRDs of cloned `if`-blocks, hence the conventional technique misses the links of clones. Therefore, this clone is regarded as disappeared if we use the

```
459    */
460   public synchronized void createStreams() {
461     if (out != null && out.length > 0) {
462       String logHead = new StringBuffer("Output ").append(
463           ((append) ? "appended" : "redirected")).append(
464           " to ").toString();
465       outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
466     }
467     if (outputProperty != null) {

...       ...

479     }
480
481     if (error != null && error.length > 0) {
482       String logHead = new StringBuffer("Error ").append(
483           ((append) ? "appended" : "redirected")).append(
484           " to ").toString();
485       errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
486     } else if (!(logError || outputStream == null)) {

...       ...

510     }
511     if (alwaysLog || outputStream == null) {

...       ...
```

(a) Before modification (revision 567,592)

```
459    */
460   public synchronized void createStreams() {
461     outStreams();
462     errorStreams();
463     if (alwaysLog || outputStream == null) {

...       ...

573   }
574
575   /** outStreams */
576   private void outStreams() {
577     if (out != null && out.length > 0) {
578       String logHead = new StringBuffer("Output ").append(
579           ((append) ? "appended" : "redirected")).append(
580           " to ").toString();
581       outputStream = foldFiles(out, logHead, Project.MSG_VERBOSE);
582     }
583     if (outputProperty != null) {

...     ...

595     }
596   }
597
598   private void errorStreams() {
599     if (error != null && error.length > 0) {
600       String logHead = new StringBuffer("Error ").append(
601           ((append) ? "appended" : "redirected")).append(
602           " to ").toString();
603       errorStream = foldFiles(error, logHead, Project.MSG_VERBOSE);
604     } else if (!(logError || outputStream == null)) {

...     ...

628     }
629   }
```

(b) After modification (revision 567,593)

Figure 4.7: An EBG that only the proposed method tracked

conventional technique. On the other hand, the proposed technique could continue to track these blocks.

Table 4.4 shows the number of clones fallen into each category. The number of correct tracking is 40, and the number of incorrect is only four. That is, the accuracy of tracking for clones that were tracked only the proposed technique becomes about 91%.

4.4.4 Answer to QUESTION2

We investigated whether clones not tracked by the proposed technique had really disappeared. We conducted a manual investigation on period "1.8" in Ant and "0.32" in ArgoUML as well as the investigation for QUESTION1. As a result, we revealed the following modifications were factors that clones were not tracked by the proposed technique. Note that prefixes "**T**" and "**F**" mean correct and incorrect tracking respectively, as well as the investigation for QUESTION1.

T1 Cloned blocks existed after modifications. However, their sizes became smaller than the threshold (30 tokens), so that they were regard as disappeared by the proposed technique.

T2 Cloned blocks (and their surrounding code) were deleted from the source code.

T3 Cloned blocks evolved to different code by large modifications.

F1 Types appearing in conditions of cloned blocks were changed. In the proposed method, variable names, method names, and literals are normalized but types

Table 4.4: Modification types that the proposed technique could track

category			Ant	ArgoUML
T	tracking was appropriate		37	3
	T1	extracting as new methods	18	0
	T2	becoming deeper nested	9	0
	T3	changing block's conditions	5	2
	T4	moving methods	2	1
	T5	adding new catch clauses	2	0
	T6	in-lined to other methods	1	0
F	tracking was NOT appropriate		4	0
	F1	deleting blocks	3	0
	F2	shrinking blocks	1	0

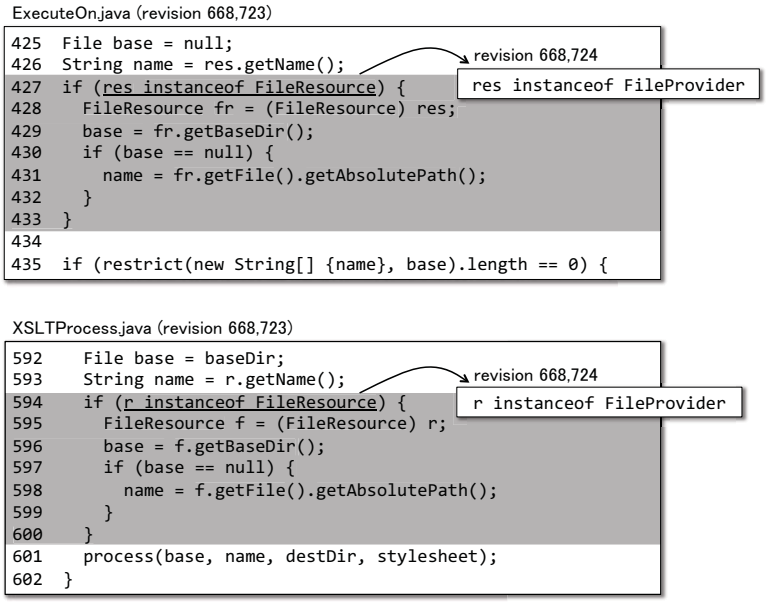


Figure 4.8: Cloned blocks not tracked by the proposed method because types in their conditions were changed

are not. Consequently, changes of types makes CONDITION2 unsatisfied, which makes a failuer of tracking.

F2 Conditions of cloned blocks were changed. This kind of change makes CONDITION2 unsatisfied.

F3 New catch clauses were added to cloned try blocks. Such modifications make CONDITION2 unsatisfied.

Figure 4.8 shows an actual instance of untracked clones found in Ant because of a type in its condition (**F1**). In revision 688,724, `FileResource` was changed to `FileProvider` in the condition of the cloned block. If the proposed technique was designed to normalize types in conditions of clone blocks, this clones would tracked correctly. However, the more normalized conditions are, the more incorrectly blocks will be tracked.

Figure 4.9 shows another example of clones not tracked by the proposed technique (**F2**), which is also found in Ant. The change performed in this example is larger than the one in Figure 4.8. In order to track clones even if this kind of large modifications were performed on conditions of cloned block, CONDITION2

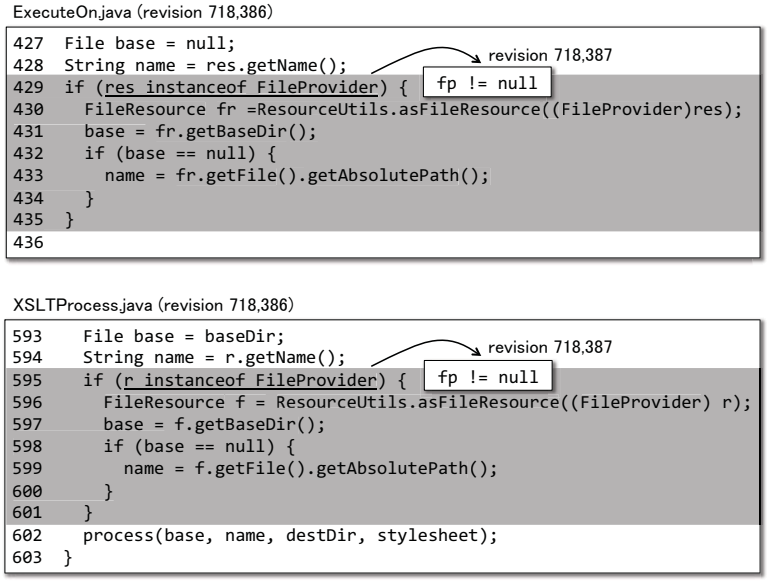


Figure 4.9: Cloned blocks not tracked by the proposed method because their conditions were changed

must become much weaker or even be removed. However, such changes on CONDITION2 will yield much more incorrect tracking. Consequently, tracking clones correctly even if their conditions are largely changed is not realistic on CRD-based clone tracking approaches.

Table 4.5 shows the number of clones not tracked by the proposed technique because of such modifications. We manually investigated 61 untracked clones, and 56 out of them were correct. That is, accuracy of untracking of the proposed technique is about 92%.

4.5 Revealing why clones are gone

As an application of the proposed technique, we investigated why clones are gone in software evolution[2]. In the past, several studies investigated occurrences and evolutions of clones [74, 78, 93, 121]: however there is no research focusing on investigation of reasons *why clones are gone*. Several empirical investigations on clones found that a part of clones was gone in software evolution [78, 121].

[2]In this application, we investigated why clone relationships among blocks had disappeared. On the other hand, the experiment described in Section 4.4 focuses on tracking each cloned block

However, in those investigations, clone removal is a pattern of clone evolution. They did not investigate why clones were gone.

This investigation was also conducted on Ant and ArgoUML. Figure 4.10 shows the number of EBGs whose elements disappeared in every revisions. As shown in this figure, clone are gone in every time of software evolution.

In order to reveal the reasons why clones are gone, we investigated disappeared EBGs manually. We narrowed the target periods of this manual inspection because investigating all the disappeared EBGs is not realistic. Herein, we investigated all the disappeared EBGs in periods "0.32" of ArgoUML (see Figure 4.10(a)) and "1.8" of Ant (see Figure 4.10(b)) as well as the manual inspections conducted in the experiment described in section 4.4. The number of investigated EBGs are 43 and 37, respectively. Table 4.6 summarizes the investigation result.

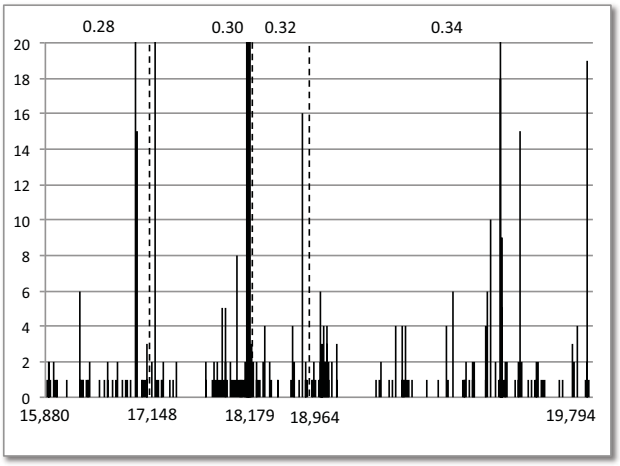
The number of disappeared EBGs due to *refactoring* is 10 and 7, respectively. However, some of those refactorings were not intended for removing duplicate code. We found that the other intentions were shortening long methods or simplifying complicated methods. Most of *refactoring* that caused disappearances of

Table 4.5: Moditications preventing the proposed method from tracking clones

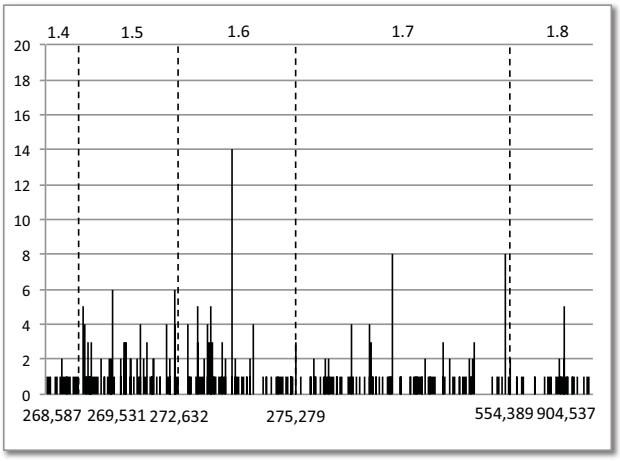
	category		Ant	ArgoUML
T	not tracking was appropriate		23	33
	T1	shrinking blocks	7	9
	T2	deleting blocks	8	24
	T3	changing blocks	8	0
F	not tracking was NOT appropriate		5	0
	F1	changing Types in block's conditions	2	0
	F2	changing block's conditions	2	0
	F3	adding catch clauses to try blocks	1	0

Table 4.6: Reasons why clones were gone

Reason	ArgoUML	Ant
Refactoring	10 (9)	7 (3)
Different evolution	6	11
Unintended inconsistency	15	10
Unneeded code deletion	8	5
Shrinking	4	0
CRD limitation	0	3
Total	43	36



(a) ArgoUML



(b) Ant

Figure 4.10: Number of EBGs whose elements disappeared

clones were Extract Method pattern. As a result of refactorings, CONDITION2 became unsatisfied, so that EBGs could not be tracked.

Different evolution means that, different modifications (e.g., functionality enhancements or expansions) were applied to one or more blocks in an EBG, so that they evolved differently. We classified 6 and 11 EBGs into this category.

Unintended inconsistency means that clones were gone unintentionally. For example, incomplete simultaneous modifications for bug fix or error checking were classified into this category. Disappearances of clones by these reasons will cause

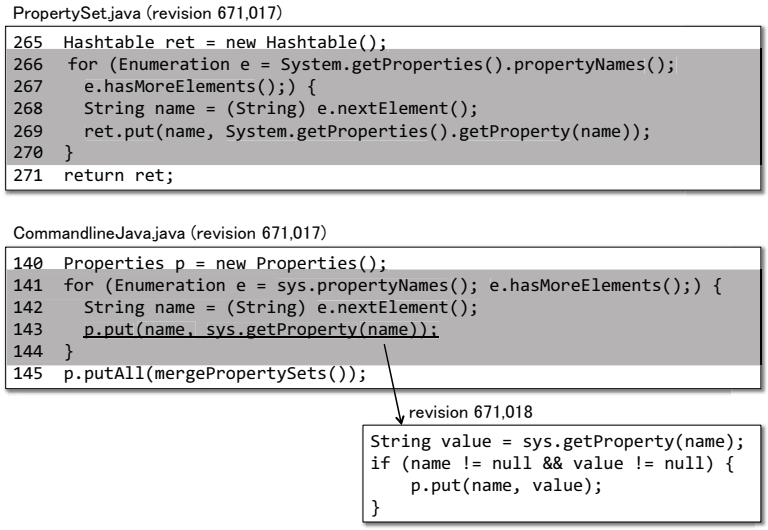


Figure 4.11: Code where an unintended inconsistency occurred

maintenance problems because they have high risks to introduce or to remain bugs. In this investigation, 15 and 10 EBGs were classified into this category. Figure 4.11 shows actual code of this category. This EBG consists of two blocks, which are different source files. Only one of them was modified (null checking code was added) in revision 671,018. However, those two blocks are logically the same. The null checking must be added to the other code simultaneously.

Unneeded code deletion means EBGs were gone by deleting unneeded code. We investigated commit logs for deciding whether the commits were for deleting unneeded code or not. We classified 8 and 5 EBGs into this category.

Shrinking means the size of blocks in EBGs becomes smaller than the threshold of minimum clone size to be detected. All the blocks consisting of an EBG continue to be duplicated: however their size became smaller than the threshold by consistency modifications. In consequent, they are not detected as clones after the modifications. In this investigation, four EBGs in ArgoUML were classified into this category.

CRD limitation means EBGs are judged as disappeared because tracking was performed incorrectly. In this investigation, three EBGs in Ant were classified into this category.

4.6 Threats To Validity

EBGs categorization

In this experiment, we conducted manual investigations on open source systems. However, the investigation result may not be correct entirely because the authors are not developers of the target systems. In order to eliminate incorrectness as much as possible, two of the authors performed the investigation together. Totally, we spent approximately 10 hours for the categorizations.

Target systems

In this experiment, we targeted only two system written in Java. Currently, it is difficult to generalize the investigation result because (1) only one programming language was investigated and (2) the number of investigated systems is only two. We selected ArgoUML and Ant as our targets because they are popular and successful systems. However, if we selected other systems that are no more than moderately successful, the investigation results may be different from this experiment. In addition, a further investigation on industrial software systems is required to generalize the experimental results on industrial software systems. This is because industrial software systems should have different characteristics compared to open source software systems.

Clone detection

This study adopted a block-based clone detection technique. The block-based approach enables *CTEC* to find clones rapidly. However, the block-based detector should miss clones that detectors based on other approaches can find. Our first experiment, which validates the accuracy of clone tracking with the proposed technique, never be affected by this factor because it did not use information of code clones: it only uses links of blocks. However, the experiment on investigating why clones are gone will be affected by this factor. That is, using different clone detectors should introduce different results. However, it is difficult to use other fine-grained clone detectors such as text-based or token-based techniques in the experiment because the experiment forces the detectors to run a few thousand times.

4.7 Summary

This study proposed a technique for tracking clones in software evolution. The proposed technique is an enhanced version of the CRD-based clone tracking. The

proposed technique includes incremental hash-based clone detection for realizing rapid clone tracking. We conducted experiments on open source systems and confirmed the followings.

- The proposed technique tracked many clones that were not tracked by a conventional technique. The correctness of tracking such clones was 91%.
- In the experiment, many clones were not tracked by even the proposed technique. The untracking correctness for such clones was 92%.

Moreover, we investigated why clones are gone with the proposed technique. We revealed that refactoring, different evolution, and unintended inconsistencies are major factors of clone disappearing. Interestingly, some of the refactorings were not intended for removing duplicate code but shortening long methods or simplifying complicated methods.

Chapter 5

Analyzing Clone Genealogies with the Enhanced Clone Tracking

5.1 Motivation

As discussed in Chapter 4, the research area of code clones still has some open issues even though many researchers have analyzed code clones to reveal characteristics of them. Concretely, the following questions have not been revealed even though they play fundamental roles on effecient management of clones.

- How many clones have negative impacts on software evolution?
- Do long-lived clones tend to be modified more frequently than short-lived ones?
- Are there any characteristics when clones are modified in their lifetime?

To answer the first question is quite important to decide how many costs we should pay to manage code clones. If there are many negative clones, software systems should require many costs to manage code clones. On the other hand, if there are few negative clones, paying too much attention on clones might be not cost-effective.

The answer to the second question should be useful to consider negative clones. If it is the case that long-lived clones tend to be modified more frequently than short-lived ones, it will be efficient to focus on long-lived clones. However, it will not be enough to detect long-lived clones if the above theory does not hold. This is

because clones will not require much attention if they are stable regardless of the length of their lifetimes.

Answering the third question will guide when we should start management of clones. If clones tend to be modified in the earlier periods of their lifetimes, managing clones in the earlier periods will be necessary. In addition, if clones tend to become more stable in the latter period, it is not cost-effective to manage clones after a certain period of time elapsed from their creation.

To reveal the above open questions, we conduct an empirical study on six open source software systems. The experiment analyzes clone genealogies detected with the enhanced CRD-based clone tracking, which is described in Chapter 4, to track code clones across version histories of code. Using the enhanced technique enables to overcome shortcomings of existing clone tracking techniques.

The empirical study that this chapter presents mainly has the following two contributions.

Revisiting Common Findings on Clone Evolution

Many researchers have analyzed evolution of clones, which offers many interesting findings. However, existing studies used their own techniques to track clones. As discussed above, these clone tracking techniques have some issues, and so it is not obvious that their findings still hold with more intelligent ways to track clones. Therefore, this study revisits the common findings on clone evolution with the enhanced CRD-based tracking. We select two findings as our revisiting target, both of the two have great influences on the research area of clones.

Answering Open Issues

This study analyzes clone genealogies detected with the enhanced technique to answer the above questions. The experimental results offer answers to the questions, and it also shows how important selecting clones to be managed carefully is.

5.2 Research Questions

The study that this chapter presents has two objectives as follows.

- Revisiting common findings on characteristics of clones with a clone tracking technique that has a high change-tolerance.
- Revealing some open issues about lifetime and the number of modifications of clones.

We investigate the following five research questions to achieve the above objectives.

RQ1: Does the finding 'most of clones are short-lived' hold as well as Kim et al. reported in the literature [78]?

RQ2: Does the finding 'there is a few clones that are modified multiple times' hold as well as Göde and Koschke reported in the literature [35]?

RQ3: Are there many long-lived clones that are modified multiple times?

RQ4: Is there a positive correlation between the length of clones' lifetime and the number of modifications applied to them?

RQ5: Do clones tend to be modified more frequently in the former half of their lifetimes than the latter one?

5.3 Detecing Clone Genealogies

This section describes definitions of code clone and clone genealogy in this study at first. Explanations of some terms relating to clone genealogies and an example of clone genealogies follow the definitions.

5.3.1 Detection of Code Clones

This study selects a block-based clone detector among a variety of clone detectors for its clone detection. The reason of this is that this study needs high scalability for its clone detection because it imposes its clone detectors on running every revision of the target software systems. In addition to that, block-based clones are compatible with our CRD-based clone tracking because CRDs are calculated on blocks.

More concretely, this study uses the clone detection technique described in section 4.2 in Chapter 4. This detector finds clones by comparing hash values created from string representations of blocks after normalized.

5.3.2 Definition of Clone Genealogy

This subsection offers the definition of clone genealogy in this study. The following explanations assume that every revision has a unique number (revision number) for its identification. In addition, they also assume that the revision numbers of contiguous revisions are also contiguous.

At the beginning, we define clone sets in formal.

Definition 5.3.1 (Clone Set)**.** Let B_r be a set of blocks in revision r, and $hash(b)$ be the hash value of the given block b. If a set of blocks c satisfies both of the following two forumlae (5.1) and (5.2), c is defined as a clone set.

$$\forall b_\alpha, b_\beta \in c[hash(b_\alpha) = hash(b_\beta)] \tag{5.1}$$

$$\forall b_\alpha \in c, \forall b_\beta \in B_r[(hash(b_\alpha) = hash(b_\beta)) \rightarrow b_\beta \in c] \tag{5.2}$$

The remainder of this section represents a set of all the clone sets in revision r as C_r.

The next defines correspondence relationships of blocks and clone sets between two contiguous revisions

Definition 5.3.2 (Correpondence Relationship of Blocks)**.** Let b_r and b_{r+1} be blocks in revision r and revision $r+1$, respectively. This study regards that b_r and b_{r+1} are under a correspondence relationship of blocks if they are linked with the enhanced CRD-based clone tracking. The remainder of this section uses the symbol $\leftrightarrow$ to represent the correspondence relationship of blocks. For instance, $b_r \leftrightarrow b_{r+1}$ means that b_r and b_{r+1} are under a correspondence relationship of blocks.

Definition 5.3.3 (Correspondence Relationship of Clone Sets)**.** Consider two clone sets c_r and c_{r+1}, and assume that c_r is in revision r and c_{r+1} is in revision $r+1$. This study regards that c_r and c_{r+1} are under a correspondence relationship of clone sets if the following formula (5.3) holds.

$$\exists b_r \in c_r,\ \exists b_{r+1} \in c_{r+1}[b_r \leftrightarrow b_{r+1}] \tag{5.3}$$

The remainder of this section uses the symbol $\Leftrightarrow$ to represent the correspondence relationship of clone sets, as well as those of blocks.

Using these definitions, a clone genealogy is defined as follows.

Definition 5.3.4 (Clone Genealogy)**.** Let C_{ALL} be a set consisting of all the clone sets that are detected in the revisions under the target period. This study calls a set of clone sets g as a clone genealogy if it satisfies both of the following two formulae (5.4) and (5.5).

$$\forall c_i \in g, \forall c_j \in C_{ALL}[(c_i \Leftrightarrow c_j) \rightarrow (c_j \in g)] \tag{5.4}$$

$$\forall c_i \in g\, \exists c_j \in g[c_i \Leftrightarrow c_j] \tag{5.5}$$

5.3.3 Definitions of Terms Related to Clone Genealogy

Lifetime of Clone Genealogy

For each clone genealogy g, its start revision, end revision, and lifetime are defined as follows. All the following definitions refer a set consisting of all the target revisions as T.

Definition 5.3.5 (Start Revision)**.** The revision $\mathit{first} \in T$ is defined as the start revision of g if it satisfies the following formula (5.6).

$$\exists c_{\mathit{first}} \in C_{\mathit{first}}[c_{\mathit{first}} \in g] \wedge \forall l \in T[l < \mathit{first} \rightarrow \forall c_l \in C_l(c_l \notin g)] \tag{5.6}$$

Definition 5.3.6 (End Revision)**.** The revision $last \in T$ is defined as the end revision of g if it satisfies the following formula (5.7).

$$\exists c_{last} \in C_{last}[c_{last} \in g] \wedge \forall l \in T[last < l \rightarrow \forall c_l \in C_l(c_l \notin g)] \tag{5.7}$$

Definition 5.3.7 (Lifetime)**.** Let first and $last$ be the start revision and the end revision of the given clone genealogy g. Then, the lifetime of g refered as R is defined in the following formula (5.8).

$$R = \{r \in T | \mathit{first} \leq r \leq last\} \tag{5.8}$$

Modifications Applied on Clone Genealogy

This study defines modifications applied on a clone genealogy g as follows.

Definition 5.3.8 (Modifications)**.** Let g be a clone genealogy, and r be a revision that are included in the lifetime of g. This study says "g was modified at the revision r" if at least one of the following two formulae (5.9) and (5.10) hold.

$$\exists c_r \in C_r[c_r \in g \wedge \exists b_r \in c_r, \exists b_{r+1} \in B_{r+1} \\ [b_r \leftrightarrow b_{r+1} \wedge hash(b_r) \neq hash(b_{r+1})]] \tag{5.9}$$

$$\exists c_r \in C_r[c_r \in g \wedge \exists b_r \in c_r, \forall b_{r+1} \in B_{r+1}[(b_r \not\leftrightarrow b_{r+1}) \\ \vee (b_r \leftrightarrow b_{r+1} \wedge \forall c_{r+1} \in C_{r+1}[b_{r+1} \notin c_{r+1}])]] \tag{5.10}$$

The formula (5.9) refers the case that the hash value of a block in revision r was changed bacause of modifications on the block. On the other hand, the formula (5.10) refers another case of modifications that a block in revision r disappeared in the next revision $r + 1$, or the block is no longer a member of any of clone sets in the revision $r + 1$.

In addition, genealogies should disappear if they do not exist in the latest revision of target revisions. In this case, it is natural to regard these genealogies as modified in their end revisions and these modifications made the genealogies disappear. In formal, let T be a set consisting of all the target revisions and $last_g$ be the end revision of a clone genealogy g, then g is regarded as modified in the revision $last_g$ if the following formula (5.11) holds. The remainder of this chapter calls modifications that satisfy the formula (5.11) "modification for disappearance".

$$\exists r \in T[last_g < r] \tag{5.11}$$

Furthermore, the number of modifications applied to a clone genealogy g as defined as follows.

Definition 5.3.9 (The Number of Modifications)**.** This study regards the number of revisions that the given clone genealogy g as the number of modifications applied to the clone genealogy.

5.3.4 Example of Clone Genealogy

Figure 5.1 shows an example of clone genealogies. This example has a clone genealogy consisting of six clone sets, A, B, C, D, E, and F.

In revision r, there exists a clone set A consisting of two blocks. The two blocks included in clone set A were modified in revision r, which results changes of hash values of the two blocks. Hence, this genealogy is regarded as modified in revision r because of the formula (5.9).

Clone set B that exists in revision $r + 1$ recieved a new member in the next revision $r + 2$. The genealogy, however, is not regarded as modified in revision $r + 1$ because this study is not interested in addition of cloned blocks.

The genealogy was branched by the modification from revision $r+2$ to revision $r + 3$. The formula (5.9) holds in the case that the genealogy was branched, hence the genealogy is regarded as modified in revision $r + 2$.

Both of the two blocks of clone set D that exists in revision $r+3$ have no correspondents in the next revision $r+4$. In other words, the two blocks disappeared by modifications between revisions $r + 3$ and $r + 4$. Hence, the genealogy is regarded as modified in revision $r + 3$ because of the formula (5.10) even though the other clone set E was not modified between the two revisions.

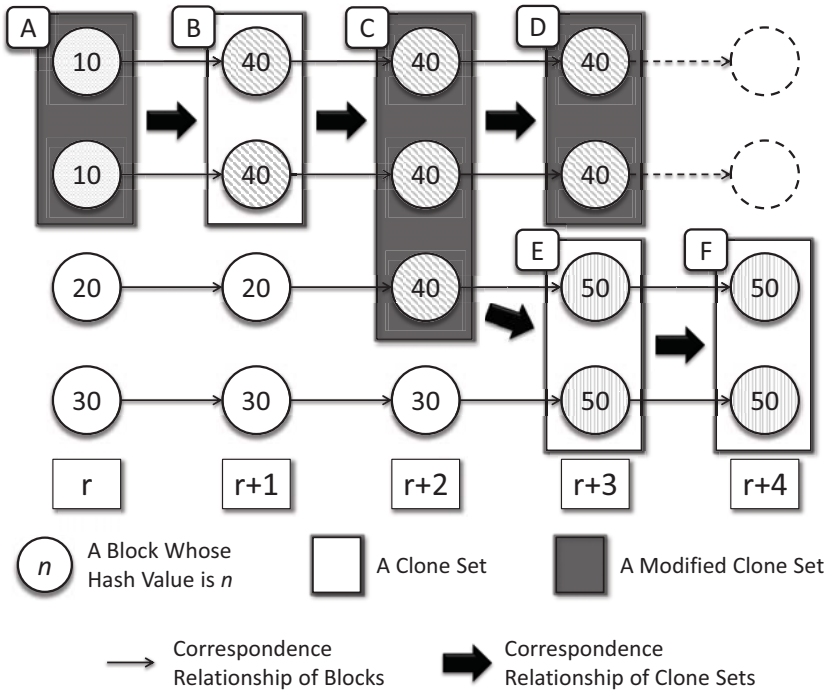


Figure 5.1: An Example of Clone Genealogies

In summary, it is regarded that the clone genealogy was modified three times in revisions r, $r+2$, and $r+3$. Note that there is no way to know the start revision and the end revision of the genealogy only from this figure because revisions before r and after $r+4$ are omitted in the figure.

5.4 Experimental Setup

The experiment uses the enhanced *CTEC* to be able to detect clone genealogies. The experimental targets are five open source software systems and *CTEC* itself.

Table 5.1 shows an overview of target software systems. Table 5.1(a) shows directories of the experimental target. Note that we narrowed the target directories on almost all of the target systems to eliminate uninterested code duplications, including test code or branches. Table 5.1(b) describes target revisions for every experimental targets. Herein, the number of revisions (the right most column) considers revisions that at least one source file in the target directories was modified. Table 5.1(c) shows LOCs and the number of detected genealogies of each target

Table 5.1: Target Software Systems

(a) Target Directories

Name	Target Directory
Ant	/ant/core/trunk/src/main/
ArgoUML	/trunk/src/
CTEC	/
Carol	/trunk/carol/src/
DNSJava	/trunk/org/xbill/DNS/
JabRef	/trunk/jabref/src/java/net/

(b) Target Revisions

Name	First Revision (Date)	Latest Revision (Date)	# of Revisions
Ant	268,623 (2001/2/9)	909,962 (2010/2/14)	5,154
ArgoUML	1 (1998/1/27)	19,893 (2012/7/10)	3,918
CTEC	1 (2012/6/19)	419 (2013/5/3)	152
Carol	9 (2002/8/6)	1,335 (2007/10/22)	250
DNSJava	2 (1998/9/6)	1,670 (2012/10/26)	1,285
JabRef	6 (2003/10/17)	3,718 (2011/11/11)	1,489

(c) LOCs and Numbers of Genealogies

Name	LOC (in the Latest Revision)	# of Genealogies
Ant	211,958	668
ArgoUML	362,783	2,710
CTEC	22,872	63
Carol	17,251	233
DNSJava	22,512	386
JabRef	113,277	682

system.

Every of our experimental targets is written in Java, and managed with *Subversion*. The reason why we chose these software systems as follows.

- Ant and ArgoUML have long histories of development, and they are widely used.
- CTEC has been developed by our own hands, and so we can deeply analyze it.
- Carol and DNSJava were used in the previous research conducted by Kim et al. [78].
- JabRef was used in the previous research conducted by Göde and Koschke [35].

5.5 Experimental Results

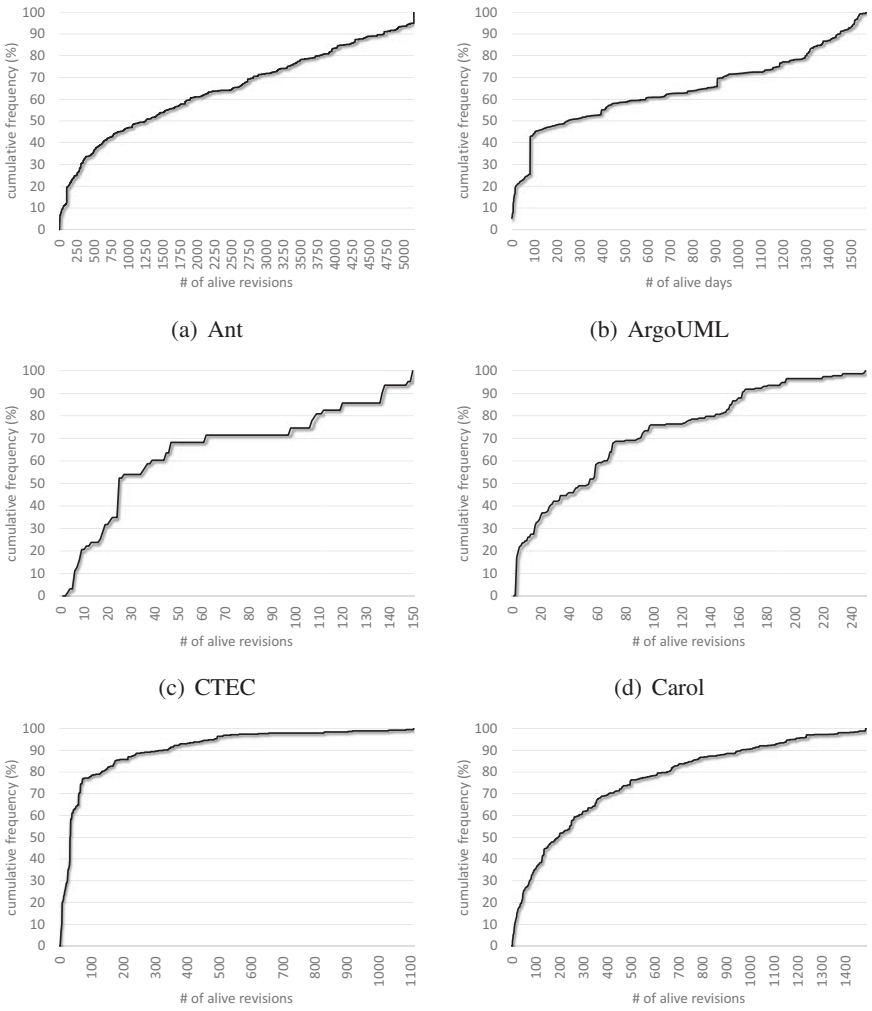
This section describes our experimental results for each of our research questions, and then offers the answers to them.

RQ1: *Does the finding 'most of clones are short-lived' hold as well as Kim et al. reported in the literature [78]?*

Figures 5.2 and 5.3 show the length of lifetimes of clone genealogies in each of the experimental targets. The x-axes of the graphs indicate the length of clone genealogies, which are the numbers of revisions in Figure 5.2 and the numbers of days in Figure 5.3, respectively. The y-axes indicate the cumulative frequencies of clone genealogies in percentage whose lifetimes are less than the number of revisions or days specified in the x-axes. For instance, we can see it from Figure 5.3 that approximately 40% of all the genealogies of Ant survived for less than 400 days.

All of the graphs have a common characteristic that they drastically grow in the left of the graphs. This means that most of genealogies have short lifetimes. This tendency is the most remarkable in DNSJava. That is, approximately 80% of genealogies in DNSJava are alive in less than 100 revisions and 250 days, which are less than 10% of its development period.

However, it is unclear from these graphs whether clone genealogies are short-lived or not. This is because these graphs take genealogies that are alive in the latest revisions of each software system into count. There is no way to know how long clone genealogies *will* survive if they are alive in the latest revisions.

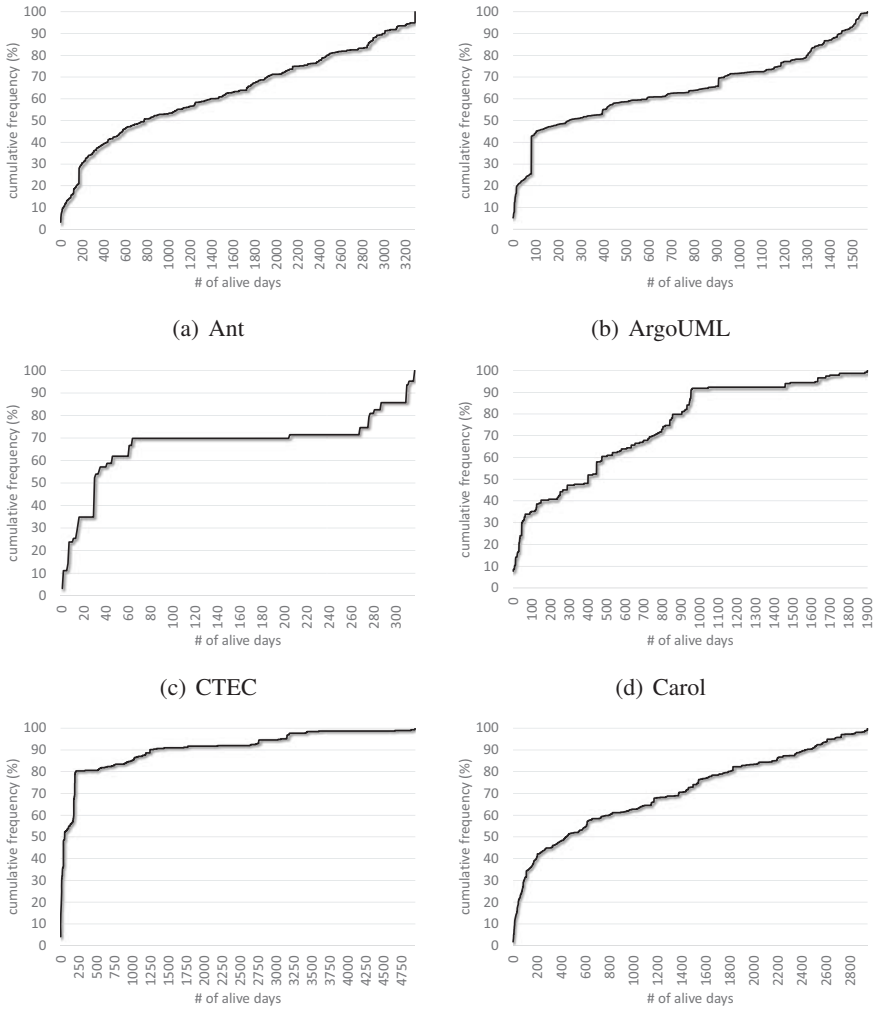


(a) Ant (b) ArgoUML

(c) CTEC (d) Carol

(e) DNSJava (f) JabRef

Figure 5.2: The Length of Lifetime (Revisions)



(a) Ant (b) ArgoUML

(c) CTEC (d) Carol

(e) DNSJava (f) JabRef

Figure 5.3: The Length of Lifetime (Days)

To investigate how long genealogies survived, this experiment uses some additional terms, all of which are used in the research conducted by Kim et al. [78]. The followings are the definitions of the terms.

Definition 5.5.1 (Dead Genealogy)**.** A genealogy g is regarded as a dead genealogy if the end revision of g is not the latest revision of the software.

Definition 5.5.2 (k-volatile Genealogy)**.** Let k indicate the number of revisions. For given k, it is defined that a dead genealogy g_{dead} is a k-volatile genealogy if the following formula (5.12) holds for g_{dead}.

$$0 \leq |R_{g_{dead}}| \leq k \tag{5.12}$$

,where $R_{g_{dead}}$ indicates the lifetime of g_{dead}.

Definition 5.5.3 ($CDF(k)$)**.** Let $f_{dead}(k)$ be the number of k-volatile genealogies. $CDF(k)$ is a cumulative distribution function of $f(k)$, whose definition is given as follows.

$$CDF(k) = \frac{\sum_{i=0}^{k} f_{dead}(k)}{\sum_{i=0}^{n} f_{dead}(k)} \tag{5.13}$$

, where n means the maximum value of k in the software.

Definition 5.5.4 ($R(k)$)**.** Let $f(k)$ be the number of all the genealogies whose lifetimes are less than k revisions, and $f_{dead}(k)$ be the number of k-volatile genealogies. $R(k)$ is defined by the following formula (5.14), which indicates the ratio of k-volatile genealogies among all the genealogies in the system.

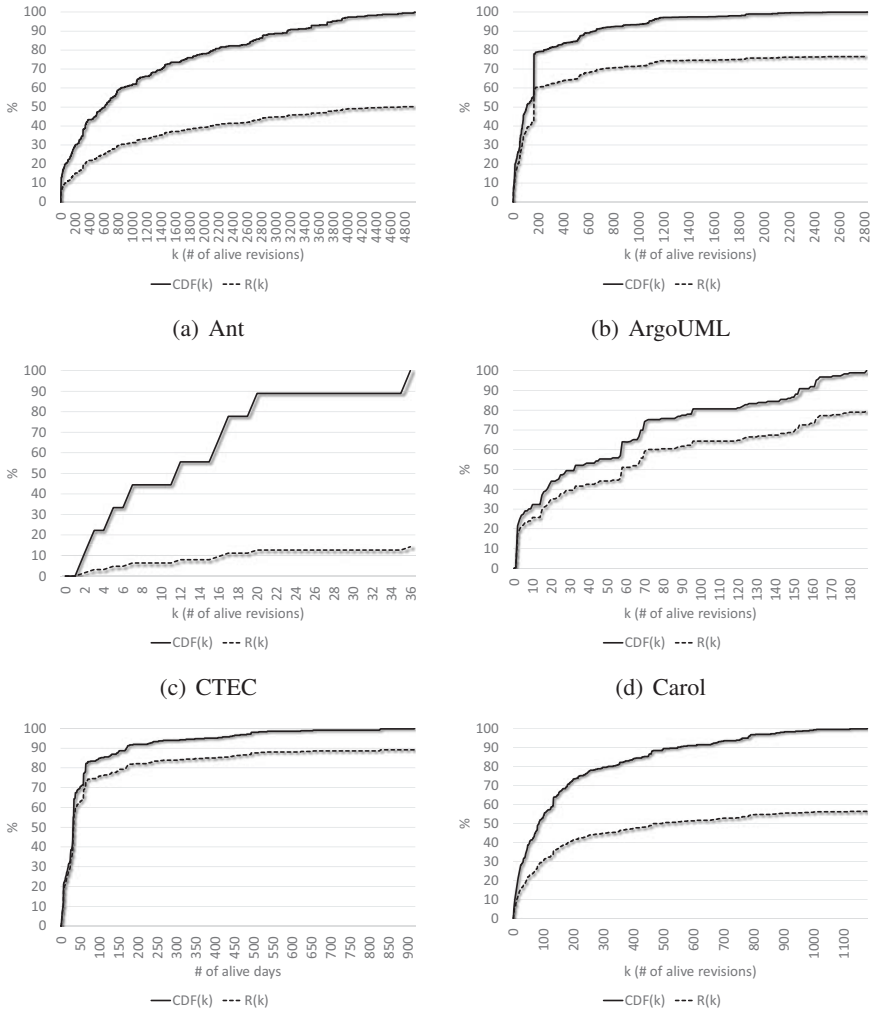
$$R(k) = \frac{\sum_{i=0}^{k} f_{dead}(k)}{\sum_{i=0}^{m} f(k)} \tag{5.14}$$

, where m indicates the maximum value of the length of the lifetimes among all the genealogies in the system.

The values of $CDF(k)$ and $R(k)$ tell us how many genealogies are short-lived. We calculate these values with k is being changed and plot them as graphs.

Figure 5.4 shows the graphs for every of the target systems. For all the graphs, the x-axes show the values of k, and the y-axes indicate the values of $CDF(k)$ and $R(k)$ in percentage. The values of $CDF(k)$ and $R(k)$ are shown with different types of lines, the solid ones show $CDF(k)$ and the broken ones show $R(k)$, respectively.

As we can see from the graphs, all the solid lines drastically grow in the left of the graphs, which means that most of dead genealogies are short-lived. The



(a) Ant (b) ArgoUML

(c) CTEC (d) Carol

(e) DNSJava (f) JabRef

Figure 5.4: *CDF*(k) and *R*(k)

broken lines also drastically grow in the left side with an exception of CTEC. ArgoUML and DNSJava have remarkable tendencies, which means that most of genealogies in the two systems are short-lived. Compared to the two systems, the degree of growth is not so high in the case of Carol even though it has a high ratio of dead genealogies in the maximum value of k. However, approximately 70% of genealogies died within 125 revisions, which is the half of target revisions of Carol. In the cases of Ant and JabRef, the maximum values of $R(k)$ are not high compared to ArgoUML, Carol, and DNSJava. This means that there exists a number of genealogies that are alive in the latest revisions of the systems. However, approximately 40% of genealogies in Ant and 50% in JabRef died within the half of the target revisions. In summary, we can say that most of genealogies are short-lived.

CTEC, however, has a different characteristic compared to the other five systems. That is, it has a low ratio of dead genealogies among all the genealogies. This means that genealogies in CTEC is not short-lived. A possible reason for the different characteristic of CTEC is that it has been developed by us, researchers of code clones. This might result in the less amount of dead genealogies.

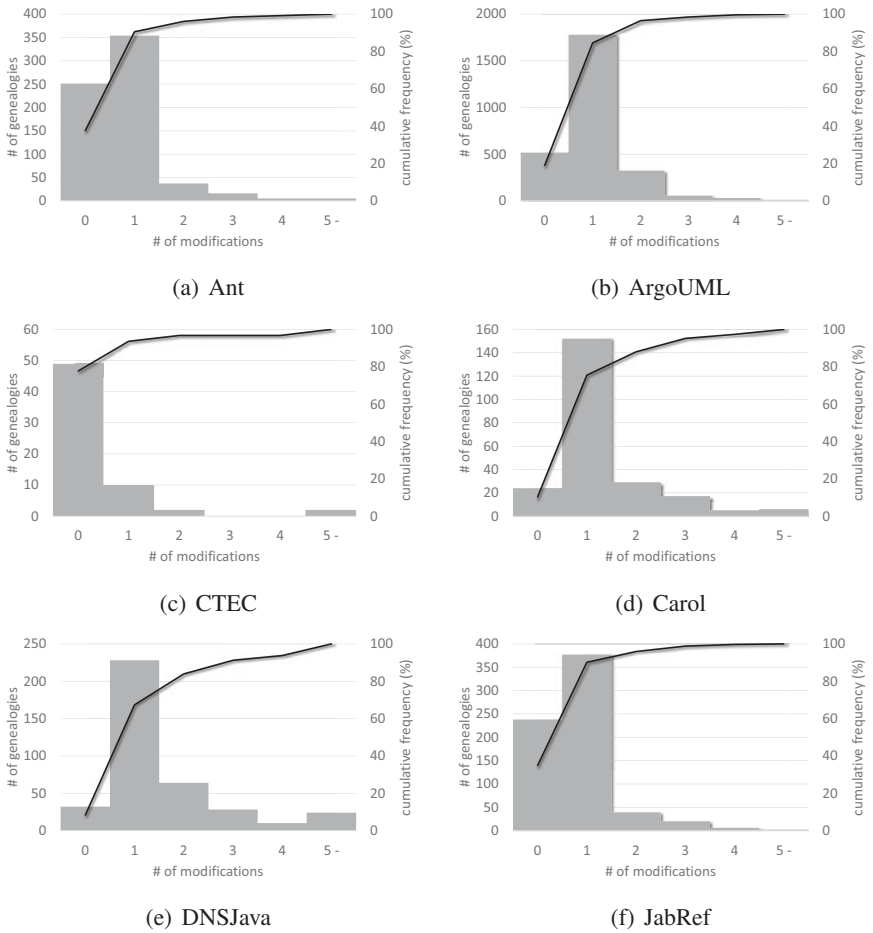
In summary of our experimental results for RQ1, it is revealed that most of clone genealogies have short lifetimes and are short-lived. This finding supports the findings reported by Kim et al. in the literature [78]. Therefore, our experimental results give **Yes** as the answer to RQ1.

RQ2: *Does the finding 'there is a few clones that are modified multiple times' hold as well as Göde and Koschke reported in the literature [35]?*

Figure 5.5 shows the number of modifications on clone genealogies in each target system. The x-axes indicate the number of modifications, and "5 -" indicates that the number of modifications are greater than or equal to five. The gray bars indicate the number of genealogies that is modified the times specified in the x-axes, and the values are shown in the y-axes in the left. The lines show cumulative frequencies with the y-axes in the right.

For all the experimental targets, over 70% of genealogies are modified at most once. This is the most remarkable in the case of CTEC, over 90% of whose genealogies are modified at most once. In addition, "0" shows the highest value in the case of CTEC, whereas "1" shows the highest values in the case of the other five systems.

In summary, these graphs tell us that most of genealogies are never modified or modified only once in their lifetimes. In other words, our experimental results support the finding reported by Göde and Koschke in the literature [35]. Therefore, we can answer **Yes** for RQ2.



(a) Ant (b) ArgoUML

(c) CTEC (d) Carol

(e) DNSJava (f) JabRef

Figure 5.5: The Number of Modifications

RQ3: *Are there many long-lived clones that are modified multiple times?*

To answer RQ3, we investigate how many genealogies are long-lived and modified multiple times. Herein, a genealogy is regarded as "long-lived" if it survive more than the half of target revisions of the software system. This criterion was also used in the research conducted by Kim et al. [78].

Table 5.2 shows the numbers and ratios of long-lived genealogies which are modified multiple times. The ratio of Carol is the highest among all the experimental targets, and Ant also has a high ratio compared to other four systems. In other words, Carol and Ant have more genealogies that developers or maintainers

should look out for than other systems.

In total, the ratio of genealogies that are long-lived and modified multiple times is approximately 3.0% among all the genealogies detected from the six software systems. Hence, the answer to RQ3 is **No**.

RQ4: *Is there a positive correlation between the length of clones' lifetime and the number of modifications applied on them?*

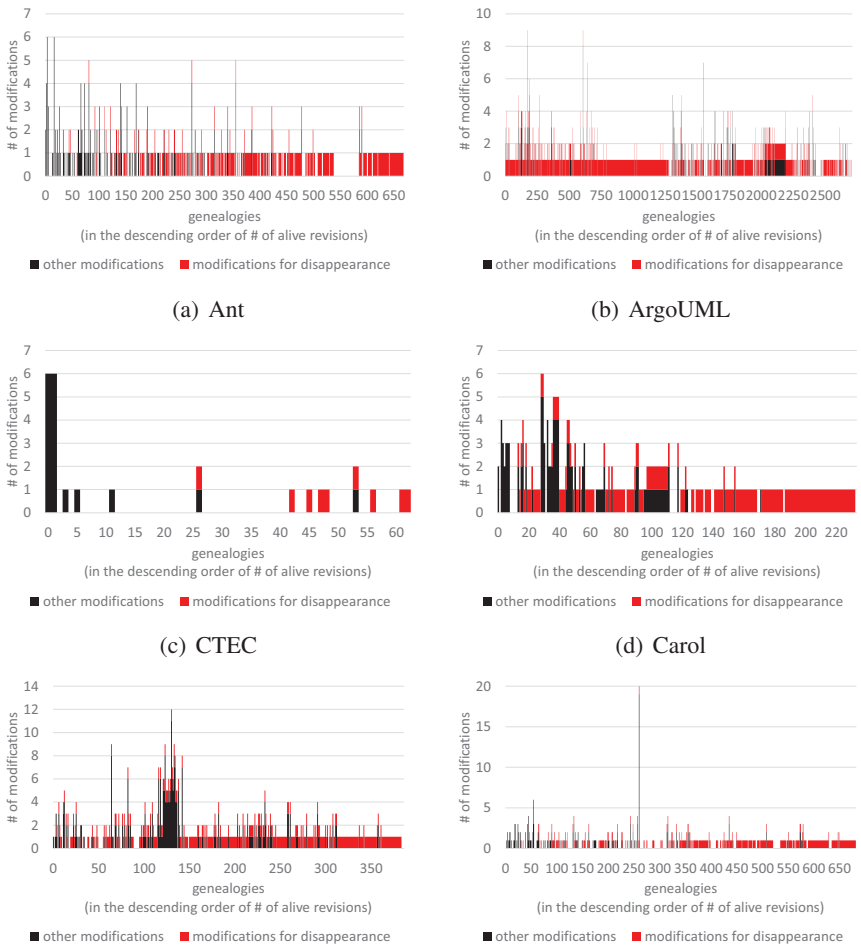
Figure 5.6 shows the numbers of modifications on each of clone genealogies. The x-axes indicate genealogies in the descending order of the length of their lifetimes, and the y-axes show the numbers of modifications. Each bar means the number of modifications on each genealogy. Note that the red and black bars indicate modifications for disappearance and the other modifications respectively, with the red bars putted on the black bars.

These graphs show that there exists a number of modifications for disappearance, but they do not show any obvious correlations between the length of clones' life time and the number of modifications. We calculated, therefore, Spearman's rank correlation coefficients to statistically judge whether there is any correlations or not.

Table 5.3 shows Spearman's rank correlation coefficients for all the experimental targets. For Ant, CTEC, and DNSJava, there exists no correlations between the length of clones' lifetimes and the number of modifications. On the other hand, there are correlations between them for the other three systems, ArgoUML, JabRef, and Carol. The correlations are positive in the case of Carol, and negative in the cases of ArgoUML and JabRef. Therefore, we cannot say that there exists positive or negative correlations between the length of clones' lifetimes and the number of modifications because there is no common tendencies of correlations. In addition, the ρ values are not so high in all the experimental targets.

Table 5.2: Long-Lived Genealogies which are Modified Multiple Times

Software	# of genealogies	ratio among all the genealogies
Ant	42	6.29%
ArgoUML	49	1.81%
CTEC	2	3.17%
Carol	29	12.45%
DNSJava	4	1.04%
JabRef	21	2.08%
Total	147	3.10%



(a) Ant

(b) ArgoUML

(c) CTEC

(d) Carol

(e) DNSJava

(f) JabRef

Figure 5.6: Modifications on Each Genealogy

These findings indicate that there is no obvious correlations between the length of clones' lifetimes and the number of modifications applied on them. Hence, the answer to RQ4 is **No**.

RQ5: *Do clones tend to be modified more frequently in the former half of their lifetimes than the latter one?*

Figure 5.7 shows the timing when each of clone genealogies was modified. The x-axes list clone genealogies in the descending order of the length of their lifetimes. The y-axes indicate normalized lifetime of each genealogy, with every lifetime of genealogies normalized from 0 to 100. Each dot means a modification that was applied at the timing specified with the y-axes. If a modifications was applied shortly after the modified clone genealogy was born, a dot will be plotted near 0 of y-axes. Note that these graphs do not consider any of modifications for disappearance because they are interested only in modifications applied during lifetimes of clones.

We cannot find any obvious tendencies of timings when clone genealogies were modified. That is, clone genealogies tend to be evenly modified among their lifetimes.

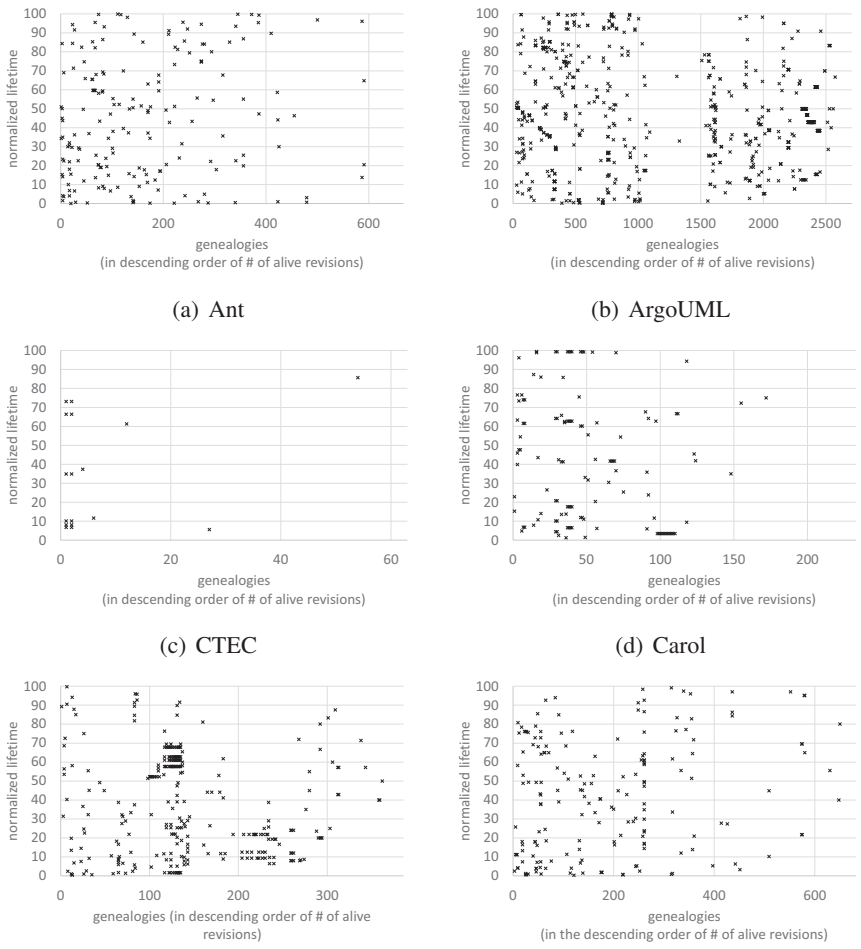
To reveal characteristics of timigs of modifications on clone genealogies, we calculated the number of modifications for every of quartered periods of lifetimes. Table 5.4 summarizes the results. As shown in the table, the first periods of all

Table 5.3: Spearman's Rank Correlation Coefficients

Software	ρ	p-value
Ant	-0.03350	0.3873
ArgoUML	-0.4529	under $2.2e^{-16}$
CTEC	-0.03290	0.798
Carol	0.3292	$2.724e^{-7}$
DNSJava	0.06206	0.2238
JabRef	-0.1901	$5.694e^{-07}$

Table 5.4: Timing of Modifications on Quartered Periods

Software	1st period 0 - 25	2nd period 25 - 50	3rd period 50 - 75	4th period 75 - 100	Total
Ant	65	37	41	39	182
ArgoUML	152	279	138	115	684
CTEC	8	3	5	1	17
Carol	49	23	28	21	121
DNSJava	124	50	92	21	287
JabRef	67	40	38	35	180
Total	465	432	342	232	1471



(a) Ant (b) ArgoUML

(c) CTEC (d) Carol

(e) DNSJava (f) JabRef

Figure 5.7: Timing of Modifications on Individual Clone Genealogies

the software systems except for ArgoUML have the highest numbers of modifications, and the second period of ArgoUML has the highest value. From the table, modifications seem to tend to be applied in the earlier periods of lifetimes of clone genealogies.

We further performed a statistical testing, which is the chi-square (χ^2) test, to reveal whether these tendencies have statistical meanings or not. Table 5.5 shows the results of chi-square tests. As the table shows, there exists a strong significant difference between the numbers of modifications on the former and latter halves

```
   …
 9: public class FinallyBlockInfo extends BlockInfo {
       …
15:     public FinallyBlockInfo(String core, List<String> types) {
16:        super(BlockType.FINALLY, core);
17:        this.types = new LinkedList<String>();
18:        this.types.addAll(types);
19:        StringBuilder builder = new StringBuilder();
20:        for (String type : types) {
21:           builder.append(type + " ");
22:        }
23:        this.concatenatedTypes = builder.toString();
24:        this.crdElement = new BlockCRD(bType);
25:     }
       …
40: }
```

Figure 5.8: An Instance of Long-Lived and Frequently Modified Clones

in the case of ArgoUML, and there exists a weak one in DNSJava. For the other systems, there are no significant differences between them. The numbers of modifications on the former half, however, are greater than those on the latter half for all the six experimental targets. A possible reason why there are no significant differences in four systems is that the numbers of modifications on the four systems are less than those of ArgoUML and DNSJava. Therefore, it has a high possibility that the differences between the numbers of modifications on the former half and the latter half will become larger if the four systems further evolve.

Hence, it can be said that these experimental results answer **Yes** for RQ5.

Table 5.5: The Results of Chi-Square Test

Software	The Former Half	The Latter Half	χ^2	p-value
Ant	102	80	1.1029	0.2963
ArgoUML	431	253	23.0332	$1.592e^{-6}$
CTEC	11	6	0.2706	0.603
Carol	72	49	1.8389	0.1751
DNSJava	174	113	6.1337	0.01326
JabRef	107	73	2.87	0.09024

5.6 Discussion

5.6.1 Long-Lived and Frequently Modified Code Clones

In this experiment, we stood a basis that we need to pay attention on code clones that are long-lived and modified multiple times. The experimental results revealed that approximately 3% of clones were instances of such negative clones. Figure 5.8 shows an instance of such clones. This instance was found in CloneTracker. It was introduced into the software in the third revision, and it is still alive in the latest revision. This clone was modified six times in revisions, 17, 19, 24, 121, 234, 244, respectively. Although this clone has been completely consistent across its lifetime, such a long-lived and frequently modified clone has a high risk to suffer inconsistencies of modifications. Dealing with such a code clone in the earlier stage of its lifetime should be effective to prevent such unintended inconsistencies.

5.6.2 Threats to Validity

Target Software Systems

This study analyzed only software systems written in Java, and so more investigations are necessary to genelarize our findings on software systms written in any other programming languages. In addition, none of the exprimental targets of this study is industrial software system. Therefore, it is possible to gain another finding if further analysis is performed on industrial software systems.

Detection of Code Clones

Clone detection in this study is a block-based detection. Such a fast detection of code clones is necessary for this study because it need to run a clone detector for all the target revisions, which is quite time-consuming. Therefore, further investigations with other clone detectors may report other different results because different clone detectors should find different clones in the same source code. In addition, the clone detection used in this study normalizes source code, including replacing sub blocks with special strings. Hence, changing the way of normalization will result another experimental findings.

Clone Tracking

This study used a new technique described in Chapter 4 to track clones across version histories. It has a high accuracy for the traditional clone tracking based on

CRD, but it is possible some linkes of clones were missed even with the enhancement technique. If a more intelligent technique is available, a further investigation with such a technique will offer different experimental results.

5.7 Summary

This chapter presented an empirical study on clone genealogies. This study used a clone tracking technique desribed in the previous chapter to detect clone genealogies. The purposes of this study are twohold. One is to revisit common findings on clone evolution with a new clone tracking technique having a high change-tolerance, and the other one is to answer some open questions that previous research did not reveal.

The experimenal results supported two major common findings on clone evolution. That is, they revealed that most of clones have short lifetimes, and most of clones are modified at most once through their lifetimes. Furthermore, the results revealed the following findings on clone evolution, all of which have not been addressed yet.

- Approximately 3% of all the code clones survived over the halves of experimental periods and were modified two or more times.
- There is no correlation between the length of lifetimes and the number of modificaions of clone genealogies.
- Clones tend to be modified more frequently in the former halves of their lifetimes than in the latter halves of them.

In summary, these findings indicate that managing all the clones equally is not a suitable way to cope with code clones. This is because approximately 97% of code clones did not require much attention of developers or maintainers because they disappeared in a short time period or they were not frequently modified. Hence, it is necessary to carefully select code clones that you pay any attention for achieving an effective code clone management. In addition, one of our findings, *clones tend to be modified more frequently in the former halves of their lifetimes than in the latter halves of them*, indicates that it should be effective to start dealing with harmful code clones as soon as possible. However, we cannot say that it is not a good way to pay attentions on clones that have survived a certain period. The reason of this is that clones were also modified in the latter halves of their lifetime even though the number of modifications on the period will be less than the former ones.

Chapter 6

Clone Removal with Form Template Method Refactoring

6.1 Background

One of the ways to prevent the influence of harmful code clones is removing them by refactorings. Refactorings can improve software maintainability without changing external behaviors of software. However, a fully manual refactoring is a difficult task for software maintainers. That is, it is quite difficult for maintainers to apply refactorings manually without introducing any human errors [118]. In other words, fully manual refactorings have high risks to introduce new bugs. Applying manual refactorings is not only complicated but also costly. This is because maintainers need to detect where they should refactor, consider how the candidates are refactored, and confirm that the refactorings do not change the behavior of the target software. Because of these factors, it is almost necessary to support maintainers with tools or techniques for applying refactorings. Tool supports enable maintainers to apply refactorings easily and safely. These needs encourage many researchers in recent years and they produce many tools and techniques [111].

It is quite natural that refactorings requiring complex procedures are more risky and more costly than ones requiring simple procedures. This implies that the former requires more supports than the latter. From the point of clone removal, clones having some gaps require more complex procedures to be removed than exact clones. Therefore, for the purpose of effective clone management, tool supports are required to remove clones having gaps. The majority of clone removal techniques, however, cannot handle gaps included in clones because they are based on "Extract Method" or "Pull-Up Method".

Using "Form Template Method" can overcome this issue. Form Template

Method refactorings pull the common statements between similar methods into a common base class, and leave gaps between target methods in the original classes. Form Template Method is a hybrid of Extract Method and Pull-Up Method. This implies that Form Template Method requires much effort and attention than Extract Method and Pull-Up Method.

Some researchers have proposed techniques to support refactorings with Form Template Method [55,66,107]. However, these techniques cannot support removing code clones if they include the following differences even if these differences have no impacts on the behavior of the program:

- Different order of code fragments and
- Different implementation styles (such as for- and while- loops).

Moreover, the existing techniques can handle only pairs of methods though Form Template Method refactoring can be applied to groups consisting of three or more similar methods.

This study proposes a new technique to support applying Form Template Method with program dependence graphs, which allows us to resolve the first issue. We also extend the proposed method to be able to handle groups of three or more similar methods.

6.2 Motivation

6.2.1 Issues of Previous Studies

As described in Chapter 2, there are some studies to support Form Template Method refactoring application. However, they still have some issues as follows.

- They cannot handle trivial differences that have no impact on the behavior of programs.
- They cannot handle groups of three or more similar methods in spite of that Form Template Method itself can be applied to them.

The following subsections describe these issues in detail.

Issue of Trivial Differences

In previous studies, all the differences between target methods are regarded as unique processing even if some of them do not affect the meaning of program. The following situations may be instances of the differences that do not affect the behavior of program.

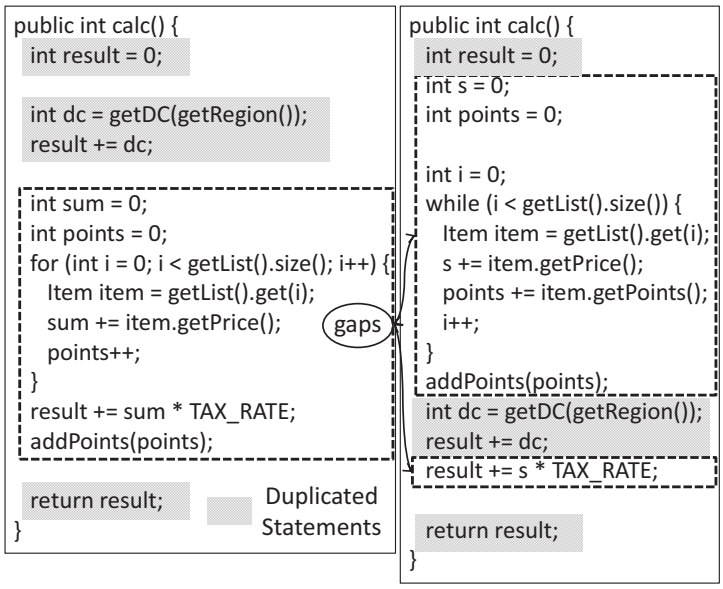
- The order of code statements is different in target methods. However, the behavior of the program is preserved even if they are reordered.
- Iterations are implemented with for statements in a method of target methods, meanwhile they are implemented with while statements in another method of target methods. However, the semantics of the iterations are exactly the same.

Figure 6.1 shows an example of our motivating example for this issue. In this example, there is a difference of the order of code statements, and there is also a difference of the implementation style of loop statements. However, these differences do not influence the meaning of the program. The only meaningful difference of these two methods is the ways of calculations of variable *points*. Nevertheless the methods described in previous studies regard these trivial differences as gaps between the two methods. Therefore, they can suggest only four lines as duplicate statements in the two methods (shown in Figure 6.1(a)). This study aims to improve this issue by using PDGs, and it will suggest 11 lines except the calculations of variable *points* as duplicate statements (shown in Figure 6.1(b)).

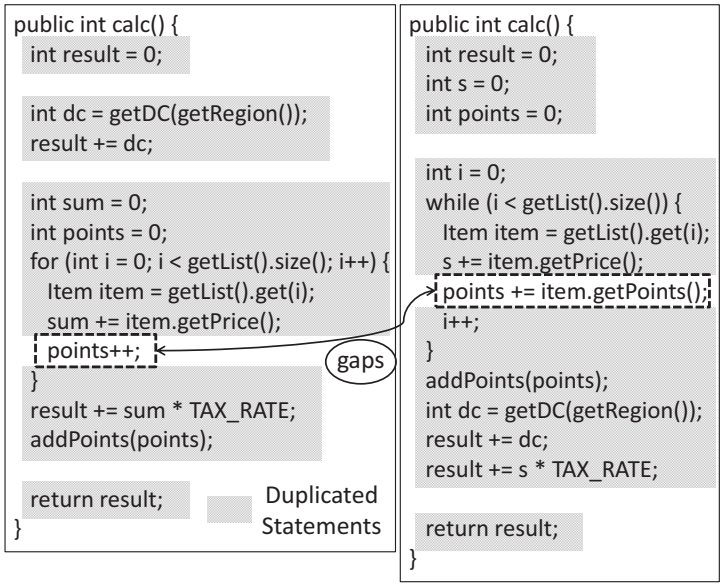
Issue of Groups of Three or More Methods

Form Template Method can be applied to a group of three or more similar methods. Nevertheless, the previous methods can handle only a pair of similar methods. Supporting Form Template Method application on only a pair of similar methods is not sufficient for clone removal. This is because code clones should remain after a refactoring with Form Template Method on a pair of methods if there are three or more similar methods.

In the example shown in Figure 6.2, there are four similar methods in four different classes, and these four classes have the same base class. If we apply Form Template Method refactoring on the pair of `method()` in `ClassA` and `method()` in `ClassB`, we get source code shown in Figure 6.2(b). As the figure shows, there are still code clones between `method()` in `ClassC` and `method()` in `ClassD` because we did not modify these two methods. Also, there are code clones between the template method and `method()` in `ClassC` and `ClassD`. Moreover, it is difficult to remove code clones from the source code of Figure 6.2(b) with Form Template Method refactoring. That is because a conflict of two template methods should occur if we apply Form Template Method on a pair of `method()` in `ClassC` and `method()` in `ClassD`. However, we can apply Form Template Method on all the four similar methods at a time. If we do so, we get the source code shown in Figure 6.2(c). As the figure shows, code clones are completely removed by the refactoring.

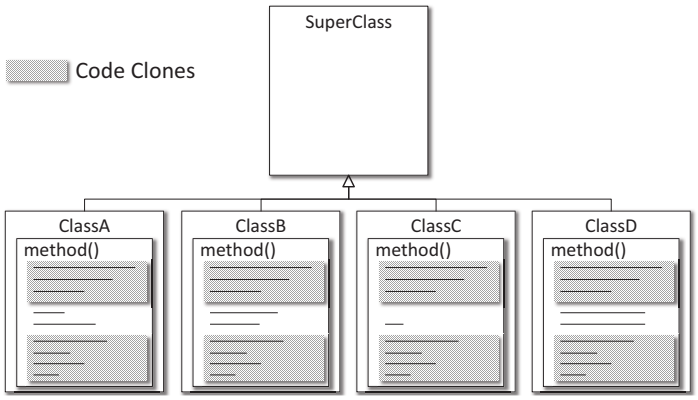


(a) The Method Proposed by Juillerat et al. [66]

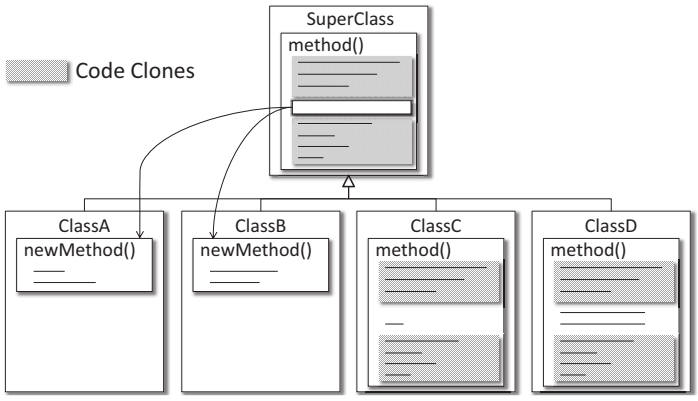


(b) The Proposed Method

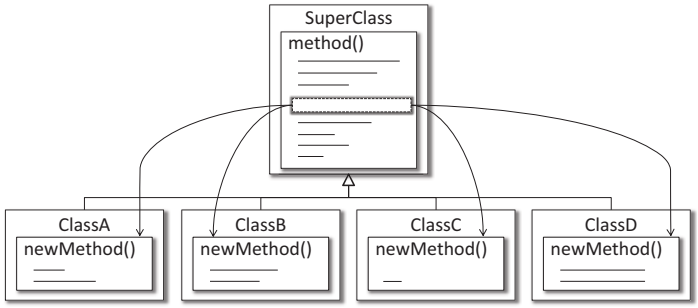
Figure 6.1: Motivating Example 1



(a) before refactoring



(b) after refactoring on two methods



(c) after refactoring on all the four methods

Figure 6.2: Motivating Example 2

Moreover, some researchers reported that the quality of software systems after some refactorings is affected by the order of the refactorings [94, 165]. In the case that refactorings on only a pair of methods are supported, the number of the candidates (pairs of methods) of Form Template Method refactorings is equal to the number of 2-combinations from a set of all the target methods. It is too difficult to detect the most appropriate order of refactorings from such a huge number of candidates. In the example of Figure 6.2, there are six pairs of methods that can be refactored with Form Template Method. However, it is difficult to decide which pair is most suitable to be refactored.

For these reasons, it is necessary to handle three or more methods at a time for effective clone removal with Form Template Method refactoring pattern. In this study, therefore, we propose a refactoring support technique on pairs of methods to be able to handle groups of three or more methods.

6.2.2 Objective of This Study

This study proposes a new refactoring support technique with Form Template Method refactoring pattern. We aim to resolve the first issue of previous studies (described in Section 6.2.1), and we aim to resolve the second issue described in Section 6.2.1 by expanding the proposed method on pairs of methods to be able to handle groups of three or more methods.

Moreover, we aim to assist users in detecting refactoring candidates with Form Template Method. Users need to specify a refactoring candidate (which means a pair of methods) for using the previous techniques for Form Template Method application assistance. The approach of previous studies is useful for actual modifications in source code associated with refactoring activities. However, it is not possible to reduce effort required for identifying opportunities on which users want to apply Form Template Method refactorings in this approach. Because software systems become larger and more complex, it is difficult to comprehend structures of software systems appropriately. Hence, it is difficult to identify suitable clone removal candidates. This is the reason why we aim to support the detection of refactoring candidates.

To reduce efforts for identifying refactoring candidates, the proposed method detects refactoring candidates automatically, and suggests all the candidates to its users. Consequently, the proposed method can suggest refactoring candidates of which users are not aware. In addition, the proposed method also suggests common processing and unique processing in each of refactoring candidate to reduce efforts required for modifying source code to apply Form Template Method refactoring pattern.

Note that the proposed method aims to suggest candidates that *can* be refac-

tored, not *should* be refactored. The reason is that there is not strict and generic standard to judge whether code clones should be removed. Also, there does not exist a strict and generic standard to judge whether Form Template Method should be used to remove code clones. Accordingly, the proposed method leaves such decisions to its users whether they need to apply refactorings on each candidate that the proposed method suggests.

6.3 Outline of the Proposed Method

6.3.1 Inputs and Outputs

The proposed method takes source code of target software systems as its input. Then, the proposed method detects all the candidates of Form Template Method refactoring, and it suggests them to users. For each of the refactoring candidates, the proposed method suggests program statements that can be merged into the base class as the common processes, and program statements that should be remained in each derived class as the unique processes. Additionally, for the unique processes, the proposed method suggests the following information.

- Sets of program statements that should be extracted as a single method.
- Relationships of new methods created by extracting the unique processes between the derived classes. This relationship means that the new methods under this relationship can be extracted as methods whose signatures are the same to each other.

Figure 6.3 shows the output information of the proposed method. In this example, there are two similar methods named `validate`, and the owner classes of these two methods have the same base class. The proposed method detects the common and unique processes between these methods. Herein, the hatched program statements are the common processes that should be merged into the base class. Program statements that are not included in the common processes are regarded as the unique processes in each derived classes. For the unique processes, the proposed method detects sets of program statements that can be extracted as a single method. In this case, we get three sets of program statements (labeled with ‘A’, ‘B’ and ‘C’ in the figure). The proposed method also detects relationships of new methods created by extracting the unique processes. In this example, the proposed method detects a relationship between ’A’ and ‘B’, which means that the new methods created by extracting ‘A’ and ‘B’ should have the same signature to each other. Here, there is no correspondence of ‘C’. In this case, we have to write an empty method that has the same signature of the method created from ‘C’ in the owner class of the left method.

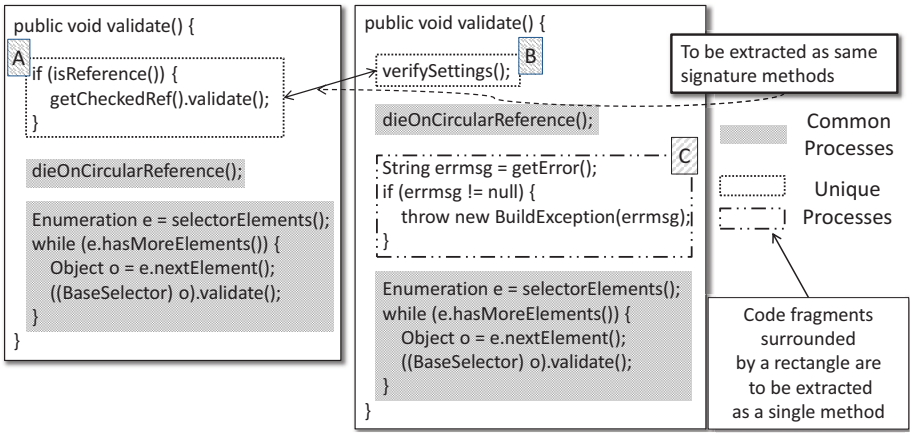


Figure 6.3: The Output of the Proposed Method

6.3.2 Specialization of PDGs

Definition of Traditional PDG

As metioned in 2.4.5, a PDG (Program Dependence Graph) is a directed graph that represents dependencies between the elements of the program [31, 153]. A node in a PDG indicates an element of a program (such as a statement and a conditional predicate), and an edge in a PDG indicates a dependence between two elements. PDG is created based on flows of data and controls. Therefore, we get the same PDGs from two programs if their flows of data and controls are same, though the programming styles are not equal.

There are the following two types of dependencies in PDG.

Data Dependence: There is a data dependence from element s to element t, if a value is assigned to variable x in s, and t references x without changing the value of x.

Control Dependence: There is a control dependence from element s to element t, if s is a conditional predicate and it directly determines whether t is executed or not.

Figure 6.4 shows an example of PDG. In this example, there are three data dependencies from the 2nd, 3rd, and 5th lines to the 4th line because variables y and z are referenced in the 4th line. On the other hand, there is a control dependence from the 4th line to the 5th line because the conditional predicate in the 4th line directly controls the execution of the 5th line. In addition, there is a node labeled

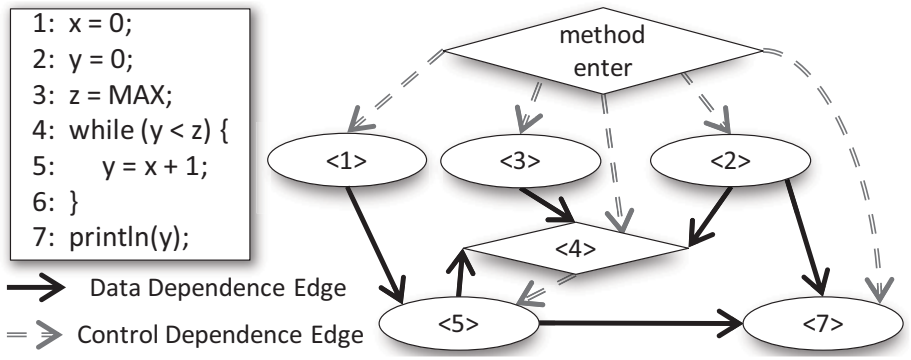


Figure 6.4: An Example of PDG

with "method enter" that means the enter node of the method. In general, PDG contains a method enter node, and there are control dependencies from the enter node to all nodes that are directly contained by the method. Note that we regard a node n as being directly contained by the method if s has no control dependencies from any other nodes in the PDG.

Specialization

PDGs used in this study is specialized for code clones detection and refactoring. The major differences of a traditional PDG and a specialized PDG are as follows:

- having execute dependences and
- tracing state changes of objects.

Execute Dependence

PDGs used in this study have an additional dependence called "execute dependence". The definition of execute dependence is as follows.

Execute Dependence: There is an execute dependence from element s to element t, if t can be executed in the next that s is executed.

Figure 6.5 shows an example of PDG with execute dependence edges. We can detect more code clones with PDGs having execute dependence than with traditional PDGs. This is because the range of program slicing is expanded by introducing this dependence.

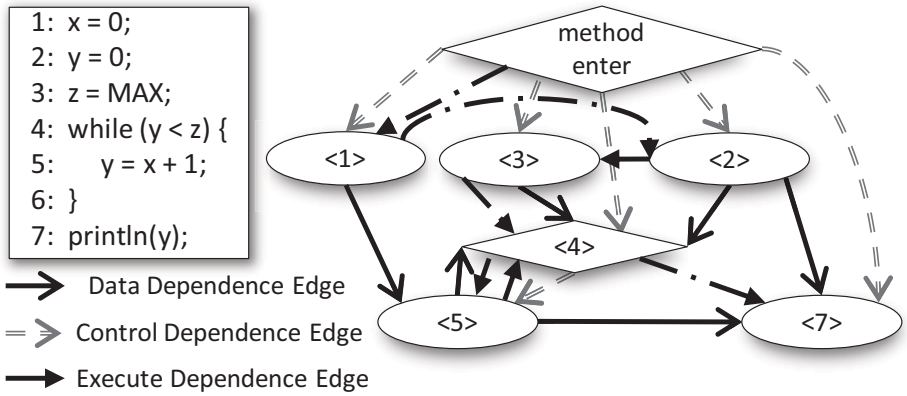


Figure 6.5: An Example of PDG with Execute Dependence Edges

Tracing State Changes of Objects

In this study, we create data dependence edges with considering state changes of objects caused by method calls. Concretely, we regard that there is a data dependence from a method call statement s to other statement t, if the state of any objects is changed in s and t references the objects without redefining them.

Figure 6.6 compares a traditional PDG and a specialized PDG created from the same source code. This figure omits control and execute dependences and the method enter node. In this example, the state of an object `builder` is changed in the 2nd, 3rd, and 4th lines by calling a method `append`. In the traditional PDG, all the elements that reference `builder` have data dependences from the 1st line. This is because the object `builder` does not re-defined or re-assigned until the end of the method. However, the specialized PDG used in this study considers state changes of objects. Therefore, we get the PDG shown in Figure 6.6(c) from the source code.

Note that it is regarded that states of objects are changed by a method call if the values of any fields in the objects are changed by the method [146].

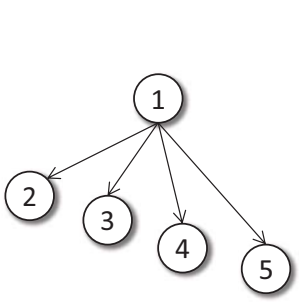
6.3.3 Processing Flow

The processing of the proposed method can be separated into one for method-pairs and one for method-groups. The processing for method-groups is implemented as an extended version of method-pairs one.

The processing flow of the proposed method on pairs of methods is shown below.

```
1: StringBuilder builder = new StringBuilder();
2: builder.append(“A”);
3: builder.append(“B”);
4: builder.append(“C”);
5: return builder.toString();
```

(a) Source Code



(b) Traditional

(c) Specialized

Figure 6.6: Data Dependence Considering State Changes of Objects

STEP-P1: Analyze target source code, and create PDGs.

STEP-P2: Detect code clones with PDGs.

STEP-P3: Identify pairs of methods on which Form Template Method can be applied.

STEP-P4: Detect common processes and unique processes for each of method pairs.

STEP-P5: Detect sets of statements included in unique processes that should be extracted as a single method.

STEP-P6: Detect pairwise relationships between new methods created by extracting unique processes.

STEP-P7: Show all the analysis results.

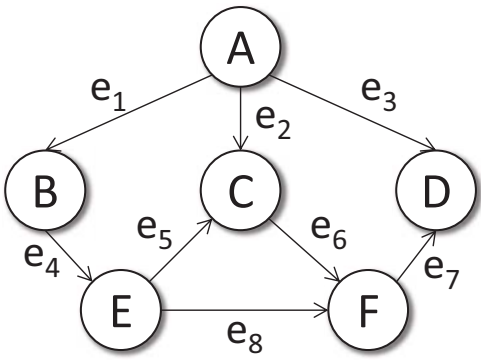


Figure 6.7: A Directed Graph

The processing for method-groups uses the results of the method-pairs version. Therefore, processing steps from STEP-S1 to STEP-S6 are exactly identical to the processing steps from STEP-P1 to STEP-P6. The processing flow of the proposed method on method groups after STEP-S6 is shown below.

STEP-S7: Detect groups of methods on which Form Template Method can be applied with the information about pairs of methods.

STEP-S8: Detect common processes and unique processes for each of method groups.

STEP-S9: Detect relationships between new methods created by extracting unique processes.

STEP-S10: Show all the analysis results.

We describe each step in detail in Sections 6.4 and 6.5.

6.3.4 Definitions

Here, we describe definitions of terms referenced in the following explanations.

A Directed Graph

A directed graph G is represented as $G = (f, V, E)$, where, V is a set of nodes, E is a set of edges, and f is a map from edges to ordered pairs of nodes ($f : E \rightarrow V \times V$). The remainder of this chapter writes the set of nodes in G as V_G, the set of edges in G as E_G, and the map between edges and ordered pairs of nodes in G as f_G, respectively.

Figure 6.7 shows an example of directed graphs. Given that the graph of the figure is G, V_G, E_G, and f_G become as follows.

$$
\begin{aligned}
&V_G = \{A, B, C, D, E, F\} \\
&E_G = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\} \\
&f_G(e_1) = (A, B),\ f_G(e_2) = (A, C),\ f_G(e_3) = (A, D),\ f_G(e_4) = (B, E) \\
&f_G(e_5) = (E, C),\ f_G(e_6) = (C, F),\ f_G(e_7) = (F, D),\ f_G(e_8) = (E, F)
\end{aligned}
$$

We define a tail of an edge $e \in E_G$ as $tail(e)$ and a head of e as $head(e)$. The definitions are as follows.

Definition 6.3.1 ($tail(e)$, $head(e)$)**.** We define $tail(e)$ as the first element of $f_G(e)$, and $head(e)$ as the last element of $f_G(e)$. In other words, $tail(e) := u$ and $head(e) := v$, where $f_G(e) = (u, v)$.

For example, for an edge e_1 in the graph of Figure 6.7, $tail(e_1) = A$ and $head(e_1) = B$.

In the next, we define sets of edges $BackwardEdges(v)$ and $ForwardEdges(v)$ for $v \in V_G$. $BackwardEdges(v)$ is a set of edges whose head is v (defined in the formula (6.1)), and $ForwardEdges(v)$ is a set of edges whose tail is v (defined in the formula(6.2)).

Definition 6.3.2 ($BackwardEdges(v)$, $ForwardEdges(v)$)**.**

$$
\begin{aligned}
BackwardEdges(v) &:= \{e \in E_G \mid head(e) = v\} && (6.1) \\
ForwardEdges(v) &:= \{e \in E_G \mid tail(e) = v\} && (6.2)
\end{aligned}
$$

For a node C in the graph of Figure 6.7, $BackwardEdges(C)$ and $ForwardEdges(C)$ become as follows.

$$
\begin{aligned}
BackwardEdges(C) &= \{e_2, e_5\} \\
ForwardEdges(C) &= \{e_6\}
\end{aligned}
$$

PDG

A PDG is one of the directed graphs. Given a PDG $G = (f, V, E)$, a node of G corresponds to an element of programs, and an edge of G corresponds to a dependence between two elements. In this study, an element of programs indicates a statement of programs. Note that we build a PDG in each of methods, therefore every method has a corresponding PDG.

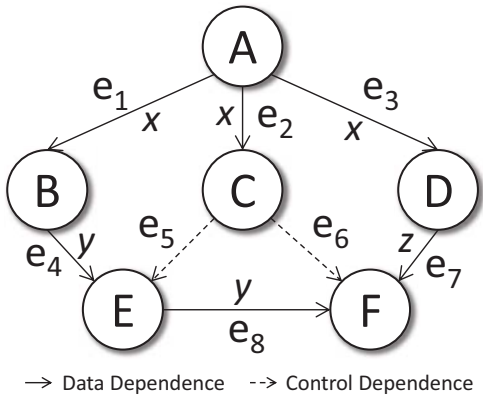


Figure 6.8: A PDG

As described above, there are three types of dependencies in PDGs used in this sutdy.

Definition 6.3.3 (Dependencies in PDG)**.** This study represents data dependencies as *data*, control dependencies as *control*, and execute dependencies as *execute*, respectively. It uses a term *type* to represent a map from edges to the types of dependences that the edges represent ($type : E \rightarrow EdgeType$), where $EdgeType = \{data, control, execute\}$. In addition, a data dependence edge has the information about the variable that the edge represents. We define $var(e_d)$ as the represented variable by a data dependence edge e_d.

In the PDG of Figure 6.8, $type$ and var become as follows.

$$type(e_1) = data,\ type(e_2) = data,\ type(e_3) = data, type(e_4) = data,$$
$$type(e_5) = control,\ type(e_6) = control,\ type(e_7) = data, type(e_8) = data,$$
$$var(e_1) = x,\ var(e_2) = x,\ var(e_3) = x,$$
$$var(e_4) = y,\ var(e_7) = z,\ var(e_8) = y$$

Clone Pairs

In the proposed method, code clones are detected with PDGs. PDG-based clone detectors regard isomorphic subgraphs of PDGs as code clones. Here, we define $ClonePairs(G_1, G_2)$ as a set of isomorphic subgraphs between PDGs G_1 and G_2.

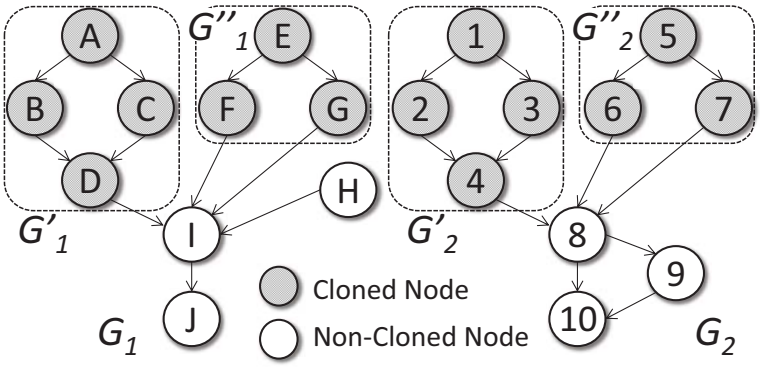


Figure 6.9: $ClonePairs(G_1, G_2)$

Definition 6.3.4 ($ClonePairs(G_1, G_2)$ and a clone pair)**.** $ClonePairs(G_1, G_2)$ is defined in the formula (6.3), and we call every element of $ClonePairs(G_1, G_2)$ a clone pair.

$$ClonePairs(G_1, G_2) := \{(G'_1, G'_2) \mid G'_1 \subset G_1 \wedge G'_2 \subset G_2 \wedge G'_1 \cong G'_2\} \quad (6.3)$$

where, G_1 and G_2 are PDGs given as input data, $G' \subset G$ indicates G' is a subgraph of G, and $G' \cong G''$ indicates G' and G'' are isomorphic subgraphs to each other.

In the example of Figure 6.9, there are two isomorphic subgraphs between G_1 and G_2. Therefore, $ClonePairs(G_1, G_2)$ become as follows.

$$ClonePairs(G_1, G_2) = \{(G'_1, G'_2), (G''_1, G''_2)\}$$

where, $V_{G'_1} = \{A, B, C, D\}$, $V_{G'_2} = \{1, 2, 3, 4\}$, $V_{G''_1} = \{E, F, G\}$, and $V_{G''_2} = \{5, 6, 7\}$.

We also define duplicate relationships on nodes of PDGs as follows.

Definition 6.3.5 (Duplication of nodes)**.** The two nodes $v_1 \in V_{G_1}$ and $v_2 \in V_{G_2}$ are duplicated to each other if and only if they satisfy the formula (6.4). This chapter uses a symbol $\sim$ to represent a node duplication. For instance, $v_1 \sim v_2$ means that v_1 and v_2 are duplicated to each other.

$$\exists (G'_1, G'_2) \in ClonePairs(G_1, G_2)[v_1 \in V_{G'_1} \wedge v_2 \in V_{G'_2} \wedge \varphi(v_1) = v_2] \quad (6.4)$$

where, G_1 and G_2 are PDGs, and φ indicates the isomorphism between G'_1 and G'_2 ($G'_1 \cong G'_2$).

In the example of Figure 6.9, the binary relation $\sim$ becomes as follows.

$$\sim = \{(A,1),(B,2),(C,3),(D,4),(E,5),(F,6),(G,7)\}$$

6.4 Supporting for Method Pairs

6.4.1 STEP-P1: Create PDGs

The proposed method internally uses an existing PDG-based clone detector, *Scorpio* [139], to detect code clones. In addition, *Scorpio* internally uses a source code analysis tool, *MASU* [108], to create PDGs. The first step of the proposed method is covered with *MASU*.

In PDGs created by *MASU*, a node corresponds to a statement of program. Additionally, PDGs created by *MASU* have another dependence, "execution dependence", in addition of traditional two dependences, data and control dependences. Execution dependences indicate execution-next links.

Note that PDGs used in the proposed method need not to be always created by *MASU*. Any tools or techniques that are able to create PDGs can take place of *MASU*.

6.4.2 STEP-P2: Detect Code Clones

As described above, the proposed method uses *Scorpio* to detect code clones. Therefore, the second step of the proposed method is fully covered with *Scorpio*. Here, we describe the clone detection algorithm used in *Scorpio* briefly.

First, *Scorpio* calculates hash values for every node of PDGs. The hash values are calculated with information about the structure of the statement that every node represents. *Scorpio* replace variables' names or literals by their types, which enables to detect code clones with different variables' names or literals. Next, *Scorpio* classifies every node with its hash value. Nodes having the same hash value are classified as an equivalence class. Then, every pair (r_1, r_2) of nodes are selected from every equivalence class, and two isomorphic subgraphs that include r_1 and r_2 are identified. Both forward and backward slices are used to identify isomorphic subgraphs.

Algorithms of each slicing are shown in Algorithm 6.1 and Algorithm 6.2, respectively. Suppose G_1 and G_2 are the target PDGs. The algorithm to detect isomorphic subgraphs between G_1 and G_2 with the forward slice is shown in Algorithm 6.1. Note that R_1 and R_2 must be initialized as empty sets to run this algorithm. In *Scorpio*, hash values are used to compare two nodes. Therefore,

Algorithm 6.1 *ForwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2)

Require: $G_1, G_2, r_1, r_2, R_1, R_2, r_1 = r_2$
Ensure: $R_1 \cong R_2$
1: $R_1 \overset{+}{\leftarrow} r_1$
2: $R_2 \overset{+}{\leftarrow} r_2$
3: **for all** $e_1 \in ForwardEdges(r_1)$ **do**
4: **for all** $e_2 \in ForwardEdges(r_2)$ **do**
5: $r'_1 \leftarrow head(e_1)$
6: $r'_2 \leftarrow head(e_2)$
7: **if** $r'_1 \neq r'_2$ **then**
8: continue
9: **end if**
10: **if** $r'_1 \in R_1$ or $r'_2 \in R_2$ **then**
11: continue
12: **end if**
13: **if** $r'_1 \in R_2$ or $r'_2 \in R_1$ **then**
14: continue
15: **end if**
16: $ForwardSlice(G_1, G_2, r'_1, r'_2, R_1, R_2)$
17: **end for**
18: **end for**

Algorithm 6.2 *BackwardSlice*(G_1, G_2, r_1, r_2, R_1, R_2)

Require: $G_1, G_2, r_1, r_2, R_1, R_2, r_1 = r_2$
Ensure: $R_1 \cong R_2$
1: $R_1 \overset{+}{\leftarrow} r_1$
2: $R_2 \overset{+}{\leftarrow} r_2$
3: **for all** $e_1 \in BackwardEdges(r_1)$ **do**
4: **for all** $e_2 \in BackwardEdges(r_2)$ **do**
5: $r'_1 \leftarrow tail(e_1)$
6: $r'_2 \leftarrow tail(e_2)$
7: **if** $r'_1 \neq r'_2$ **then**
8: continue
9: **end if**
10: **if** $r'_1 \in R_1$ or $r'_2 \in R_2$ **then**
11: continue
12: **end if**
13: **if** $r'_1 \in R_2$ or $r'_2 \in R_1$ **then**
14: continue
15: **end if**
16: $BackwardSlice(G_1, G_2, r'_1, r'_2, R_1, R_2)$
17: **end for**
18: **end for**

$r_1 = r_2$ indicates that the hash value of r_1 is equal to that of r_2. Also, the algorithm with the backward slice is shown in Algorithm 6.2. Both of the forward and backward slices are used to detect code clones in *Scorpio*.

Isomorphic subgraphs detected in this step is regarded as a clone pair. We set a minimal size of each isomorphic subgraph to six nodes to be detected as code clones. In the next step, *Scorpio* removes uninteresting clone pairs. The algorithm is that if a clone pair (s_1, s_2) is subsumed by another clone pair (s'_1, s'_2), it is removed from the set of clone pairs. Finally, clone sets are generated from clone pairs sharing the same isomorphic subgraphs.

Note that it is not necessary to detect code clones with this way to use the proposed method. The proposed method only needs $ClonePairs(G_{m_1}, G_{m_2})$ for any pair of methods (m_1, m_2) contained in the target program, where G_{m_i} indicates a PDG of method m_i. The proposed method does not care how they are identified.

6.4.3 STEP-P3: Identify Method Pairs

This step detects pairs of methods on which Form Template Method can be applied with the information about code clones detected by *Scorpio*. The proposed method regards a pair of methods as a refactoring candidate if it satisfies following requirements.

Requirement A: The two methods in the method pair are defined in different classes.

Requirement B: The owner classes of the two methods have the same base class.

Requirement C: There is at least one clone pair between the method pair.

The followings discuss these requirements in detail.

Requirement A

Form Template Method cannot be applied on methods defined in the same class because it uses the inheritance relationships and the polymorphism. Thus, the method pair that the proposed method targets has to be defined in different classes.

Requirement B

Form Template Method targets similar methods whose owner classes have the same base class. It is possible that we apply Form Template Method on methods whose owner classes do not have the same base class. The way is that we

insert a new class into class hierarchy and make the owner classes inheriting the new class. However, refactorings with this way may decay the quality of program design because two non-related classes are forced to be jointed in the class hierarchy. For this reason, the target method pairs are limited to having the same base class.

Requirement C

If there is no duplicate statement, Form Template Method cannot be applied on such a method pairs because no statement is pulled up into the base class. Therefore, we make a requirement that there is at least one clone pair between the two methods of every target method pair.

Suppose that G_{m_1} and G_{m_2} are PDGs of methods m_1 and m_2. If there is no clone pair between a method pair (m_1, m_2), $ClonePairs(G_{m_1}, G_{m_2})$ is empty. Therefore, we can check whether there is at least one clone by checking whether $ClonePairs(G_{m_1}, G_{m_2})$ is empty or not. In other words, the method pair (m_1, m_2) must satisfy the formula (6.5).

$$ClonePairs(G_{m_1}, G_{m_2}) \neq \emptyset \tag{6.5}$$

6.4.4 STEP-P4: Detect Common and Unique Processes

In this step, the proposed method detects common and unique processes in each method pair. Suppose that a method pair of m_1 and m_2 is the given method pair, and $G_{m_{1(2)}}$ is the PDG of method $m_{1(2)}$.

The proposed method regards statements as common processes if and only if they are included in code clones existing between the two methods of the given method pair. We define $CommonNodes(G_{m_{1(2)}})$ as a set of nodes in $G_{m_{1(2)}}$ whose representing statements form common processes. The formula (6.6) represents the definition, where $G_{m_{1(2)}}$ indicates the PDG of method $m_{1(2)}$.

$$CommonNodes(G_{m_{1(2)}}) := \{v \in V_{G_{m_{1(2)}}} \mid \exists w \in V_{G_{m_{2(1)}}} [v \sim w]\} \tag{6.6}$$

However, a node in $G_{m_{1(2)}}$ can be duplicated between two or more nodes in $G_{m_{2(1)}}$. In other words, the formula (6.7) can be satisfied in some cases, considering the two clone pairs $(G'_{m_1}, G'_{m_2}), (G''_{m_1}, G''_{m_2}) \in ClonePairs(G_{m_1}, G_{m_2})$.

$$\exists v \in V_{G'_{m_{1(2)}}} [v \in V_{G''_{m_{1(2)}}}] \tag{6.7}$$

Algorithm 6.3 Removing Redundant Clone Pairs

Require: $ClonePairs(G_{m_1}, G_{m_2})$
Ensure: $ClonePairs(G_{m_1}, G_{m_2})$ after repaired

```
 1: CopyOfClonePairs = ∅
 2: CopyOfClonePairs +← ClonePairs(G_{m1}, G_{m2})
 3: for all (G'_{m1}, G'_{m2}) ∈ CopyOfClonePairs do
 4:    for all (G''_{m1}, G''_{m2}) ∈ CopyOfClonePairs do
 5:       if (∃v1 ∈ G'_{m1}[v1 ∈ G''_{m1}])&(G'_{m1} ≠ G''_{m1}) then
 6:          if |G'_{m1}| < |G''_{m1}| then
 7:             ClonePairs(G_{m1}, G_{m2}) -← (G'_{m1}, G'_{m2})
 8:          else
 9:             ClonePairs(G_{m1}, G_{m2}) -← (G''_{m1}, G''_{m2})
10:          end if
11:       end if
12:       if (∃v2 ∈ G'_{m2}[v2 ∈ G''_{m2}])&(G'_{m2} ≠ G''_{m2}) then
13:          if |G'_{m2}| < |G''_{m2}| then
14:             ClonePairs(G_{m1}, G_{m2}) -← (G'_{m1}, G'_{m2})
15:          else
16:             ClonePairs(G_{m1}, G_{m2}) -← (G''_{m1}, G''_{m2})
17:          end if
18:       end if
19:    end for
20: end for
```

In this case, we cannot merge all the nodes that are duplicate to other nodes in the other method. We remove some clone pairs from $ClonePairs(G_{m_1}, G_{m_2})$ to resolve this problem. Algorithm 6.3 shows the algorithm for removing clone pairs. Note that $|R|$ means the number of elements in a set R and $R \xleftarrow{-} r$ means the process to remove an element r from R.

This algorithm ensures that there is at most one duplicate node in the other method for all nodes in method m_1 and m_2. Nodes should be pulled up into the base class if they are contained in $CommonNodes(G_{m_{1(2)}})$ after this processing.

Figure 6.10 shows an instance of method pairs that contain redundant clone pairs. There are two clone pairs; one is labeled with 'α', and another one is labeled with 'β'. The clone pair α consists of $(\{a, b, c, d, e\}, \{A, B, C, D, E\})$, and the clone pair β consists of $(\{a, b, d, e\}, \{F, G, H, I\})$. In this case, the algorithm selects α as the remaining clone pair, and removes β from $ClonePairs(G_{m_1}, G_{m_2})$ because the number of elements of α is larger than those of β. As a result, the common statements that the proposed method detects in this method pair (m_1, m_2) become as follows.

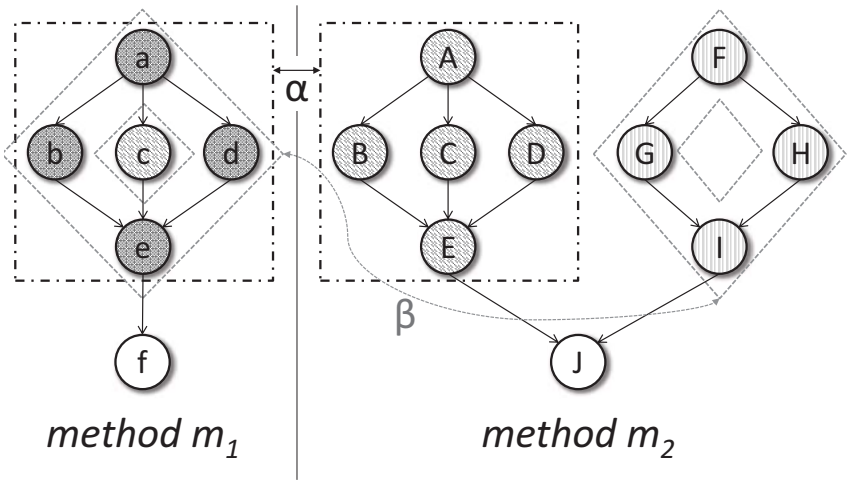


Figure 6.10: An example of Method Pairs Including Redundant Clone Pairs

$$\begin{aligned} CommonNodes(G_{m_1}) &= \{a, b, c, d, e\} \\ CommonNodes(G_{m_2}) &= \{A, B, C, D, E\} \end{aligned}$$

On the other hand, the proposed method regards that program statements form unique processes in a given method pair if they are not included in the common processes. We define $DiffNodes(G_{m_{1(2)}})$ as a set of nodes in $G_{m_{1(2)}}$ that need to remain in the derived class that has method $m_{1(2)}$. Formula (6.8) shows the definition of $DiffNodes(G_{m_{1(2)}})$.

$$DiffNodes(G_{m_{1(2)}}) := \{v \in V_{G_{m_{1(2)}}} \mid v \notin CommonNodes(G_{m_{1(2)}})\} \quad (6.8)$$

In the method pair (m_1, m_2) shown in Figure 6.10, $DiffNodes(G_{m_{1(2)}})$ becomes as follows.

$$\begin{aligned} DiffNodes(G_{m_1}) &= \{f\} \\ DiffNodes(G_{m_2}) &= \{F, G, H, I, J\} \end{aligned}$$

6.4.5 STEP-P5: Detect Sets of Statements Extracted as a Single Method

In this step, the proposed method detects sets of statements that can be extracted as a single method in the unique processes. For applying Form Template Method

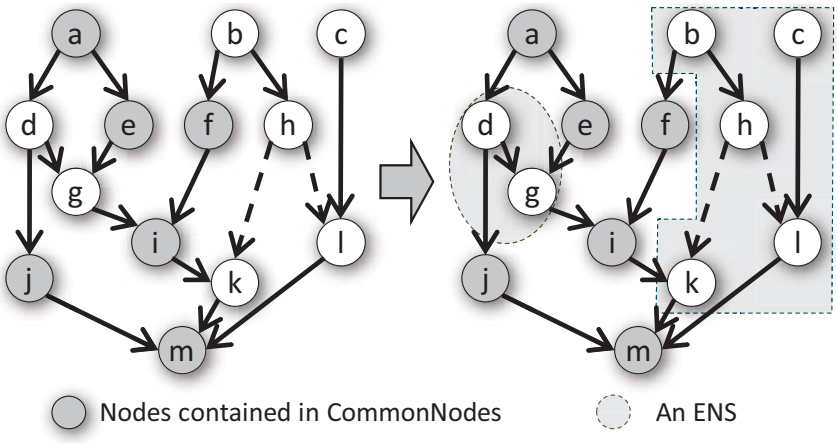


Figure 6.11: An example of the Detection of ENSs

refactorings, it is necessary that nodes remaining in derived classes are extracted as new methods. Therefore, we have to detect sets of program statements included in $DiffNodes(G_{m_{1(2)}})$, each of which can be extracted as a single method. The reminder of this chapter represents a set of nodes that should be extracted as a single method as an **Extract Node Set** (in short, **ENS**).

Definition of the Extract Node Set

The proposed method regards nodes included $DiffNodes(G_{m_{1(2)}})$ as an ENS if there is at least one path that does not include nodes in $CommonNodes(G_{m_{1(2)}})$ for any pairs of the nodes in it ignoring directions of each edges. In other words, we regard a set of nodes $S_{m_{1(2)}} \subset V_{G_{m_{1(2)}}}$ as an ENS if there is at least one path that satisfies the formula (6.9) for any two nodes $v_1, v_n (v_1 \neq v_n)$ in $S_{m_{1(2)}}$.

$$\forall i \in \{1 \ldots n\}[v_i \in DiffNodes(G_{m_{1(2)}})] \tag{6.9}$$

In the example shown in Figure 6.11 we can find two ENSs; one consists of $\{d, g\}$ and the other consists of $\{b, c, h, k, l\}$. As shown in this example, each of methods in refactoring candidates can contain multiple ENSs. This study uses a term $DiffNodeSets(G_{m_{1(2)}})$ to represent a family of ENSs in method $m_{1(2)}$. Suppose that m_1 indicates the name of the method shown in the figure, and G_{m_1} indicates its PDG, then, in the example of Figure 6.11, $DiffNodeSets(G_{m_1})$ becomes as follows.

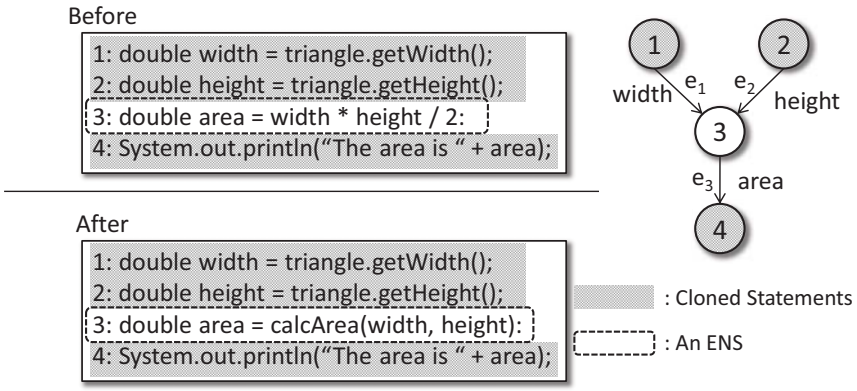


Figure 6.12: An Example of Inputs and Outputs of ENSs

$$\mathit{DiffNodeSets}(G_{m_1}) = \{\{d, g\}, \{b, c, h, k, l\}\}$$

Note that any of nodes in $\mathit{DiffNodes}(G_{m_{1(2)}})$ must be included in at least and at most one ENS in $\mathit{DiffNodeSets}(G_{m_{1(2)}})$ (forumla (6.10)).

$$\forall v \in \mathit{DiffNodes}(G_{m_{1(2)}})\, \exists S \in \mathit{DiffNodeSets}(G_{m_{1(2)}})[v \in S] \tag{6.10}$$

Parameters of ENSs

Parameters of the method created by extracting an ENS S can be defined as variables represented by data dependence edges whose heads are included in S and whose tails are not included in S. Assume that G indicates a PDG, and S indicates an ENS of G. Under these assumptions, we define a set of data dependence edges whose tails are not included in S and whose heads are included in S as $\mathit{InputDataEdges}(G, S)$. Formula (6.11) shows the definition of $\mathit{InputDataEdges}(G, S)$.

$$\begin{aligned}&\mathit{InputDataEdges}(G, S) := \\ &\qquad \{e \in E_G \mid (\mathit{tail}(e) \notin S) \wedge (\mathit{head}(e) \notin S) \wedge (\mathit{type}(e) = \mathit{data})\}\end{aligned} \tag{6.11}$$

Here, we define a set of variables to represent parameters of the method created by extracting S consist as $\mathit{InputVariables}(S)$ in forumla (6.12).

$$InputVariables(G, S) := \{p \mid \exists e \in InputDataEdges(G, S)[var(e) = p]\} \quad (6.12)$$

In the example of Figure 6.12, there is an ENS consisting of the 3rd line. In this case, $InputDataEdges(G, S)$ and $InputVariables(G, S)$ become as follows

$$\begin{aligned} InputDataEdges(G, S) &= \{e_1, e_2\} \\ InputVariables(G, S) &= \{width, height\} \end{aligned}$$

Thus, a method created by extracting the ENS needs two parameters, one is $width$, and the other is $height$.

Output of ENSs

Suppose that G indicates a PDG of a method, and S indicates an ENS of G. The output values of the method created by extracting S are defined as variables that are represented by data dependence edges whose heads are not included in S and whose tails are included in S.

First, we define $OutputDataEdges(G, S)$ as a set of data dependence edges whose tails are included in S and whose heads are not included in S. The definition is shown in formula (6.13).

$$OutputDataEdges(G, S) := \{e \in E_G \mid tail(e) \in S \wedge head(e) \notin S \wedge type(e) = data\} \quad (6.13)$$

Herein, we can define a set of output variables of S with this definition. We define it as $OutputVariables(G, S)$ in the formula (6.14).

$$OutputVariables(G, S) := \{p \mid \exists e \in OutputDataEdges(G, S)[p = var(e)]\} \quad (6.14)$$

In the example of Figure 6.12, $OutputDataEdges(G, S)$ and $OutputVariables(G, S)$ become as follows.

$$\begin{aligned} OutputDataEdges(G, S) &= \{e_3\} \\ OutputVariables(G, S) &= \{area\} \end{aligned}$$

Therefore, a method created from the ENS needs to return a value of `double`.

Conditions for Call

The conditions to call methods created by extracting ENSs are represented by control dependence edges. For example, if there are control dependences from a conditional predicate of `if` statement to all the nodes included in an ENS S, a method created from S should be called in the case that the conditional predicate is satisfied.

First, we define $InputControlEdges(G, S)$ as a set of control dependence edges whose tails are not included in S and whose heads are included in S in the formula (6.15), where G is a PDG of a method and S is an ENS of G.

$$InputControlEdges(G, S) := \{e \in E_G \mid tail(e) \notin S \wedge head(e) \in S \wedge type(e) = control\} \quad (6.15)$$

In the next, we define nodes that have control dependences to nodes included in S as $InputControlNodes(G, S)$. The definition is shown in formula (6.16).

$$InputControlNodes(G, S) := \{v \in V_G \mid \exists e_c \in InputControlEdges(G, S)[v = tail(e_c)]\} \quad (6.16)$$

As described above, a PDG has a method enter node, and there are control dependencies from the node to all the nodes that are directly contained by the method. In addition, nodes contained in conditional blocks have control dependence from the conditional predicates of the blocks. In this case, there is no control dependence from the method enter node to nodes contained in conditional blocks because these nodes are not directly contained by the method. Therefore, all the nodes except the method enter node have at least and at most one control dependence from other nodes.

Requirements for ENSs to be Extracted as a Single Method

In some cases, we cannot extract each of ENSs as a single method. Concretely, we cannot extract an ENS S as a single method if it satisfies the following conditions.

- There are multiple return values in the method created from S.
- S includes a part of nodes in a block statement, and it also includes some nodes out of the block statement.

Multiple Return Values

It is necessary that an ENS S has at most one return value to be extracted as a method. Therefore, if there are two or more return values of S, it cannot be extracted as is.

To resolve this problem, we divide S into multiple ENSs satisfying the condition. Here, we describe the algorithm.

First, we define a set of nodes in S that locate at boundary of data dependences between S and out of S as $BoundaryNodes(G, S)$. Formula (6.17) is its definition.

$$BoundaryNodes(G, S) := \{v \in V_G \mid \exists e \in OutputDataEdges(G, S)[tail(e) = v]\} \quad (6.17)$$

Then, we divide S with the Algorithms 6.4, 6.5, and 6.6. Note that $R \overset{+}{\leftarrow} r$ means adding an element r into a set R. Besides, $detect(v, S)$ and $parse(S, R)$ indicates the following processing, respectively.

$detect(v, S)$**:** Return an ENS S' satisfying the condition that $BoundaryNodes(G, S') = \{v\}$ by dividing the original ENS S.

$parse(S, R)$**:** Return a node set R' created by adding some nodes into the specified node set R. The added nodes must be reached from a node in R by tracing an edge in the reverse direction. Moreover, the addition of nodes must preserve the condition that $|BoundaryNodes(G, R)| = 1$.

Here, we describe the behavior of the algorithm with the example shown in Figure 6.13.

In the beginning, $BoundaryNodes(G, S) = \{n, o\}$. Here we consider the case that $detect(n, S)$ is called in the 3rd line in Algorithm 6.4.

In $detect(n, S)$, a node set R is initialized with n, then R is expanded by trace edges in the reverse direction. Obviously, $|BoundaryNodes(G, R)| = 1$.

In the next, $parse(S, R)$ is called. At first, we reach a node k and we need to judge whether it can be added into R or not. In this case, k has no dependences to nodes except n, therefore we judge that it can be included in R. In the next, we reach another node f. The node f has two dependences whose tail is f; one is to k, and the other is to another node i. Herein, the node i are not included in R. Consequently, if we add f into R, $BoundaryNodes(G, R)$ becomes $\{f, n\}$. Thus we judge that we cannot add f into R. $parse(S, R)$ stops here because there is no nodes that can be a candidate of expansion, and it returns $R' = \{k, n\}$.

Algorithm 6.4 Division of an ENS

Require: G, S
Ensure: $SeparetedNodeSets$
1: **while** $S \neq \emptyset$ **do**
2: **for all** $v \in BoundaryNodes(G, S)$ **do**
3: $SeparetedNodeSets \xleftarrow{+} detect(v, S)$
4: **end for**
5: **for all** $T \in SeparetedNodeSets$ **do**
6: **for all** $v' \in T$ **do**
7: $S \xleftarrow{-} v'$
8: **end for**
9: **end for**
10: **end while**

Algorithm 6.5 *detect*(v, S)

Require: v, S
Ensure: R
1: $R \leftarrow \{v\}$
2: **while** $|R| \neq |parse(S, R)|$ **do**
3: **for all** $v' \in parse(S, R)$ **do**
4: $R \xleftarrow{+} v'$
5: **end for**
6: **end while**

Algorithm 6.6 *parse*(S, R)

Require: S, R
Ensure: R'
1: $R' = R$
2: **for all** $v \in R$ **do**
3: **for all** $e \in BackwardEdges(v)$ **do**
4: **if** $tail(e) \in S \wedge tail(e) \notin R$ **then**
5: **if** $\forall e_d \in ForwardDataEdges(tail(e))[head(e_d) \in R]$ **then**
6: $R' \xleftarrow{+} tail(e)$
7: **end if**
8: **end if**
9: **end for**
10: **end for**

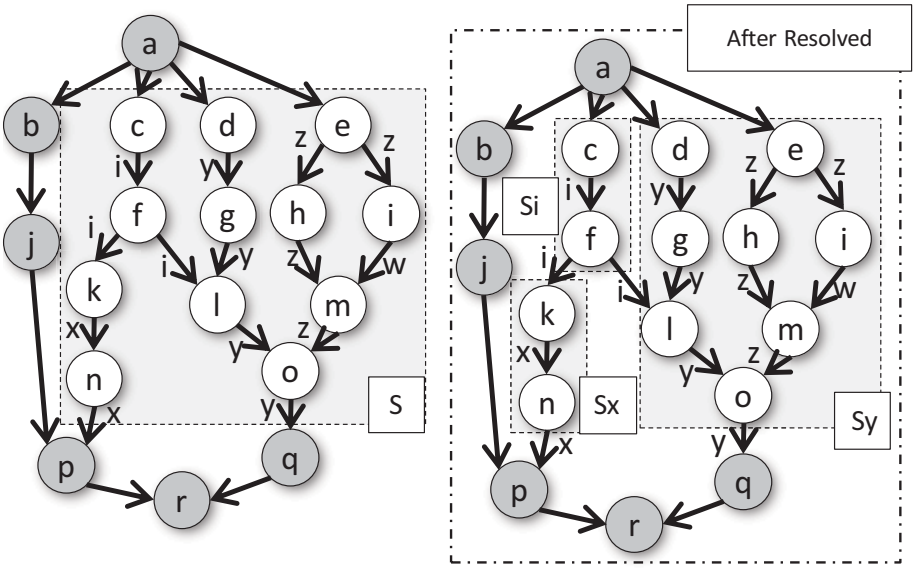


Figure 6.13: Behavior of Algorithm 6.4

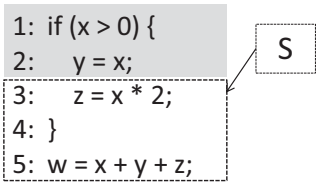


Figure 6.14: An Instance of Segmentalization of Block Statements

Then the algorithm backs to the 3rd line in Algorithm 6.4. Here, $detect(o, R)$ is called, and it returns $R' = \{d, e, g, h, i, l, m, o\}$.

Then the algorithm goes to the 5th line in Algorithm 6.4. Here, nodes included in any element in $SeparetedNodeSets$ are removed from the original ENS S. In this case, $SeparetedNodeSets = \{\{k, n\}, \{d, e, g, h, i, l, m, o\}\}$, therefore S becomes $S = \{f, c\}$.

The algorithm repeats this process until $S \neq \emptyset$. Finally, we get 3 ENSs S_x, S_y, and S_i from the original ENS S, and all of them have to return only a single value x, y, and i, respectively.

Segmentalization of Block Statements

Suppose that $Nodes(b)$ indicates a set of nodes that are included in the given block statement b. If an ENS S satisfies all the following formulae (6.18) and (6.19), we cannot extract it as a single method.

$$\exists v \in Nodes(b)(v \in S) \wedge \exists u \in Nodes(b)(u \notin S) \tag{6.18}$$

$$\exists v \in S(v \in Nodes(b)) \wedge \exists u \in S(u \notin Nodes(b)) \tag{6.19}$$

Figure 6.14 shows an instance of ENSs that satisfy these formulae. As this figure shows, we cannot extract S as is. This is because one node in S is included in `if` statement, and the other node is not included in the statement. To resolve this problem, we restrict nodes in each of ENSs to be in the same block statement. By this restriction, the 3rd line and the 5th line in Figure 6.14 can be included in the same ENS. Therefore, we get two ENSs in this example, and each of them can be extracted as a single method.

6.4.6 STEP-P6: Detect Pairwise Relationships

In this step, we detect pairwise relationships of ENSs in a given method pair. In other words, assuming that $\rightleftharpoons$ indicates the pairwise relationships and $S_{m_{1(2)}}$ is an ENS of method $m_{1(2)}$, for each of $S_{m_{1(2)}} \in DiffNodeSets(G_{m_{1(2)}})$ we detect whether $S_{m_{2(1)}} \in DiffNodeSets(G_{m_{2(1)}})$ satisfies $S_{m_1} \rightleftharpoons S_{m_2}$ exists or not. Note that $S_{m_1} \rightleftharpoons S_{m_2}$ indicates that S_{m_1} and S_{m_2} can be extracted as methods whose signatures are the same as each other. If an ENS S has no correspondent in the other method, we have to make an empty method whose signature is the same as S in the derived class that does not have S.

We regard a pair of ENSs S_{m_1} and S_{m_2} as $S_{m_1} \rightleftharpoons S_{m_2}$ if they satisfy the following two requirements.

Requirement P6-1: The types of return values of S_{m_1} and S_{m_2} are the same as each other.

Requirement P6-2: The conditions to call the new methods created by extracting S_{m_1} and S_{m_2} are the same as each other.

The following subsections describe these requirements in detail. Herein, $EM_{S_{m_{1(2)}}}$ means the method created by extracting the ENS $S_{m_{1(2)}}$.

Requirement P6-1: Requirement the Type of the Return Value

To make $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ have the same signature, it is necessary that the types of return values of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are same to each other.

As described in 6.4.5, the return values of $EM_{S_{m_{1(2)}}}$ are defined as $OutputVariables(G_{m_{1(2)}}, S_{m_{1(2)}})$ (formula (6.14)). In addition, the number of elements in $OutputVariables(G_{m_{1(2)}}, S_{m_{1(2)}})$ is at most one because of the processing described in 6.4.5.

We define that $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ have the same type of the return value if they satisfy formula (6.20).

$$\begin{aligned}(|OutputVariables(G_{m_1}, S_{m_1})| = |OutputVariables(G_{m_2}, S_{m_2})|) \\ \wedge (\forall p \in OutputVariables(G_{m_1}, S_{m_1}) \exists q \in OutputVariables(G_{m_2}, S_{m_2}) \\ [varType(p) = varType(q)])\end{aligned} \quad (6.20)$$

Note that we do not consider parameters of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ to detect the pairwise relationships. This is because we can make them having the same signature by adding non-used parameters in the case that the parameters of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are different. For example, suppose that $EM_{S_{m_1}}$ needs one parameter whose type is integer and $EM_{S_{m_2}}$ needs one parameter whose type is string. In this case, we can match the signatures of $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ by adding a string parameter in $EM_{S_{m_1}}$ and an integer parameter $EM_{S_{m_2}}$.

Requirement P6-2: Requirement about Conditions for Call

To extract $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ as same signature methods, it is necessary that $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are called under the same conditions.

Figure 6.15 shows an example of wrong correspondence of ENSs. This is caused by not considering the conditions for call of each ENSs. In this example, there are two ENSs $A1$ and $A2$ in `methodA`, and there are also two ENSs $B1$ and $B2$ in `methodB`. All of the ENSs are in `if` statements, which means that methods created by extracting these ENSs are called if the conditional predicates of the corresponding `if` statements are satisfied. However, the pairwise relationships shown in the figure do not consider the conditions, therefore the behavior of `methodB` is changed after the refactoring.

As described in 6.4.5, the conditions to call $EM_{S_{m_{1(2)}}}$ are represented by control dependence edges, and all the nodes always have one control dependence from other nodes. In addition, all the nodes in an ENS are contained by a single block statement or contained by their owner method directly by the process described in

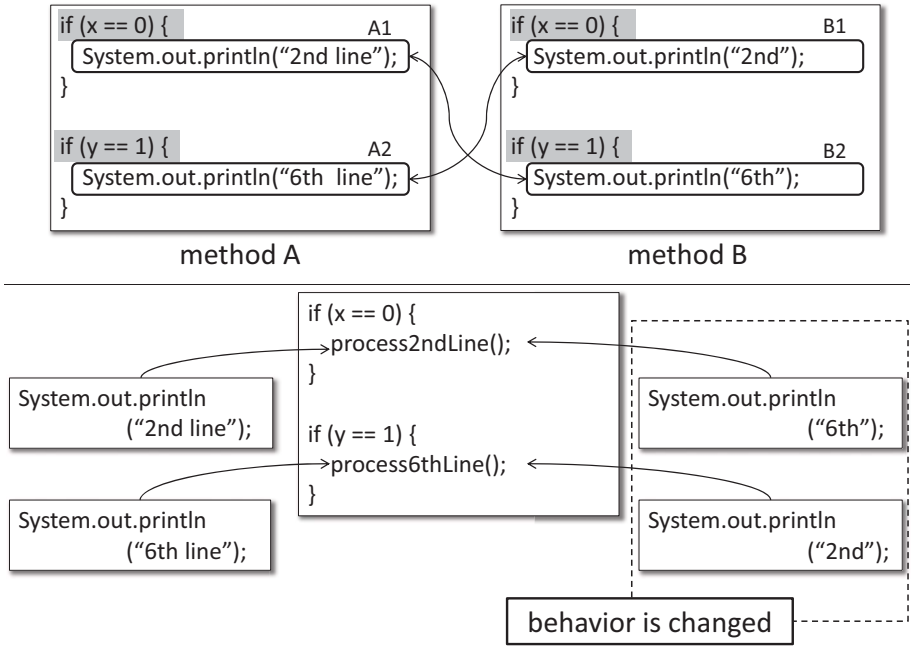


Figure 6.15: An Example of Wrong Pairwise Relationships Caused by not Considering Conditions for Call

6.4.5. Consequently, all the control dependence edges to S have the same tail node. In other words, the formula (6.21) is always satisfied for every ENS S.

$$|InputControlNodes(G, S)| = 1 \tag{6.21}$$

Here, we define ICN_S as the unique element in $InputControlNodes(G, S)$. We regard a pair of ENSs (S_{m_1}, S_{m_2}) as having same conditions for call if and only if they satisfy the formula (6.22).

$$\begin{aligned}(ICN_{S_{m_1}} \sim ICN_{S_{m_2}}) \vee \\ (\exists S'_1 \in DiffNodeSets(G_{m_1}) \exists S'_2 \in DiffNodeSets(G_{m_2}) \\ [S'_1 \rightleftharpoons S'_2 \wedge ICN_{S_{m_1}} \in S'_1 \wedge ICN_{S_{m_2}} \in S'_2])\end{aligned} \tag{6.22}$$

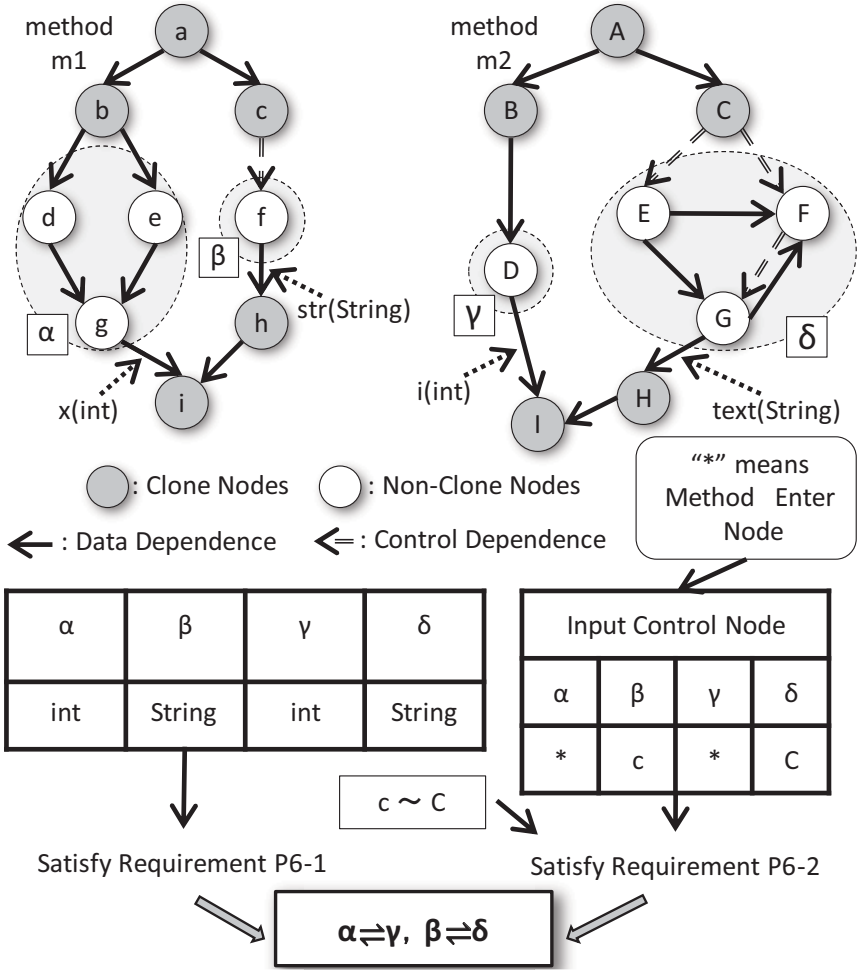


Figure 6.16: An Example of Pairwise Relationships

An Example of Pairwise Relationships Detection

Figure 6.16 shows an example of pairwise relationships detection. Note that this figure ommits method enter nodes and control dependence.

In this example, there are two ENSs α and β in method $m1$, and there are also two ENSs γ and δ in method $m2$. Return values of EM_α and EM_γ are integer values, and return values of EM_β and EM_δ are string values. Consequently, two pairs of ENSs (α, γ) and (β, δ) satisfy Requirement P6-1.

Then, the proposed method checks the correspondence of call conditions. In this example, ICN_α and ICN_γ are the method enter nodes, which means that a

pair of ENSs (α, γ) satisfies Requirement P6-2. In the case of (β, δ), ICN_β is c, and ICN_δ is C. Consequently, the pair of ENS (β, δ) satisfies Requirement P6-2 because $c \sim C$.

As a result, we get two pairs of ENSs (α, γ) and (β, δ) in this example.

6.5 Supporting for Method Groups

In this section, we describe the steps of the proposed technique for method groups. As described in 6.3.3, we use method pair information calculated in STEP-P1 to STEP-P6, therefore the steps from STEP-S1 to STEP-S6 are identical to from STEP-P1 to STEP-P6. Therefore, we describe the steps after STEP-S6 in the following subsections.

6.5.1 STEP-S7: Identify Method Groups

This step detects groups of methods on which Form Template Method can be applied. In the reminder of this chapter, suppose that $m_1 \doteq m_2$ indicates that a pair of methods m_1 and m_2 is a refactoring candidate detected in the process described in 6.4.3.

Obviously, the binary relation $\doteq$ is a symmetric relation ($m_1 \doteq m_2 \Rightarrow m_2 \doteq m_1$). However, it is not a transitive relation. Assume that there are three methods m_1, m_2 and m_3, and $m_1 \doteq m_2$, $m_2 \doteq m_3$. In this case, there is at least one clone pair between m_1 and m_2, and between m_2 and m_3 because of the definitions of $\doteq$. However, there is no clone pair between m_1 and m_3 if all the clone pairs between m_1 and m_2 are not overlapped by any of clone pairs between m_2 and m_3. If there is no clone pair between m_1 and m_3, $m_1 \not\doteq m_3$ because of its definitions.

However, the proposed method temporarily regards a group of methods as a candidate method group if it satisfies the formula (6.23).

$$\forall m \in MS\ ,\ \exists m' \in MS\ (m \doteq m') \tag{6.23}$$

Under this definition, if $m_1 \doteq m_2$ and $m_2 \doteq m_3$ are satisfied, a group of methods m_1, m_2, and m_3 is regarded as a candidate method group regardless of whether $m_1 \doteq m_3$ is satisfied or not. If there is no clone pairs between m_1 and m_3, the proposed method omits the method group from candidate method groups in the next step.

6.5.2 STEP-S8: Detect Common and Unique Processes

In this step, the proposed method detects common processes and unique processes in every method group. Suppose that MS indicates a method group and G_{m_i}

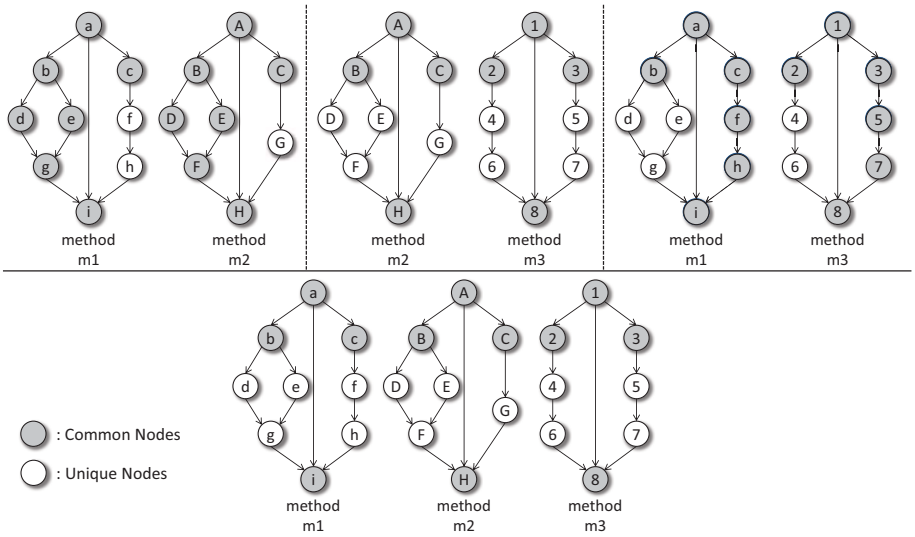


Figure 6.17: An Example of Method Group

means the PDG of method m_i.

Statements must be duplicated between all the methods in the method group to be pulled up into a base class as a template method. We use the representation $CommonNodes_{group}(G_{m_i})$ for a group of nodes in $V_{G_{m_i}}$ that are pulled up into a base class. The definition is shown in formula (6.24).

$$CommonNodes_{group}(G_{m_i}) := \{v_i \in V_{G_{m_i}} \mid \forall m_j \in MS\ ,\ \exists v_j \in V_{G_{m_j}}[v_i \sim v_j]\} \quad (6.24)$$

We define $DiffNodes_{group}(G_{m_i})$ as a group of nodes that need to remain in the derived class that has method m_i. The definition is shown in the formula (6.25).

$$DiffNodes_{group}(G_{m_i}) := \{v_i \in V_{G_{m_i}} \mid v_i \notin CommonNodes_{group}(G_{m_i})\} \quad (6.25)$$

Figure 6.17 shows an example of method group. In this example, there are three methods ($m1$, $m2$, and $m3$) and all the pairs of them are detected as candidate method pairs, in other words $m1 \doteq m2$, $m2 \doteq m3$, and $m1 \doteq m3$. In this example, $CommonNodes_{group}(G_{m_1})$ and $DiffNodes_{group}(G_{m_1})$ become as follows.

$$
\begin{aligned}
CommonNodes_{group}(G_{m_1}) &= \{a, b, c, i\} \\
DiffNodes_{group}(G_{m_1}) &= \{d, e, f, g, h\}
\end{aligned}
$$

In some cases, some nodes included in $CommonNodes(G_{m_i})$ are omitted to make $CommonNodes_{group}(G_{m_1})$. Consequently, the amount of common processes on method groups might be quite smaller than that on method pairs. As a result, the number of elements in $CommonNodes(G_{m_i})$ might be less than the threshold of minimum code clone size that is specified by users. Therefore, the proposed method omits method groups if the number of their common nodes is less than the minimum clone size. Consequently, in the case that $m_1 \doteq m_2$, $m_2 \doteq m_3$, and $m_1 \not\doteq m_3$, the proposed method omit the method group that consists of m_1, m_2, and m_3.

In the next, the proposed method detects ENSs for every method in MS. There is no difference in the definitions of ENSs between method pairs and method groups because the detection of ENSs is closed in each method.

6.5.3 STEP-S9: Detect Relationships on ENSs

In this step, the proposed method detects correspondences of ENSs between methods in MS.

Likewise on method pairs, the correspondence relationship means that ENSs in a correspondence relationship can be extracted as methods whose signatures are the same. As described in 6.4.6, the proposed method regards a pair of ENSs S_{m_1} and S_{m_2} as $S_{m_1} \rightleftharpoons S_{m_2}$ if they satisfy the requirements about their return values and their call conditions. We can detect this relationship by expanding that on method pairs.

Suppose that S_{m_1}, S_{m_2}, and S_{m_3} are ENSs in methods m_1, m_2, and m_3, respectively. In addition, assume that $S_{m_1} \rightleftharpoons S_{m_2}$ and $S_{m_2} \rightleftharpoons S_{m_3}$. Moreover, assume that EM_S means a method created by extracting an ENS S. Under these assumptions, the types of return values $EM_{S_{m_1}}$ and $EM_{S_{m_2}}$ are the same. Moreover, those of $EM_{S_{m_2}}$ and $EM_{S_{m_3}}$ are also the same. Therefore, the return values of $EM_{S_{m_1}}$ and $EM_{S_{m_3}}$ are the same. Similarly, the call conditions for $EM_{S_{m_1}}$, $EM_{S_{m_2}}$, and $EM_{S_{m_3}}$ are same to each other. Consequently, the binary relationship $\rightleftharpoons$ is a transitive relation ($S_{m_1} \rightleftharpoons S_{m_2} \wedge S_{m_2} \rightleftharpoons S_{m_3} \Rightarrow S_{m_1} \rightleftharpoons S_{m_3}$).

Obviously, the binary relation $\rightleftharpoons$ is a symmetric relation ($S_{m_1} \rightleftharpoons S_{m_2} \Rightarrow S_{m_2} \rightleftharpoons S_{m_1}$). Moreover, it is also a reflexive relation ($S_{m_1} \rightleftharpoons S_{m_1}$). Consequently, the binary relation $\rightleftharpoons$ is an equivalence relation.

Therefore, we can detect correspondence relationships between three or more ENSs by detecting equivalent classes.

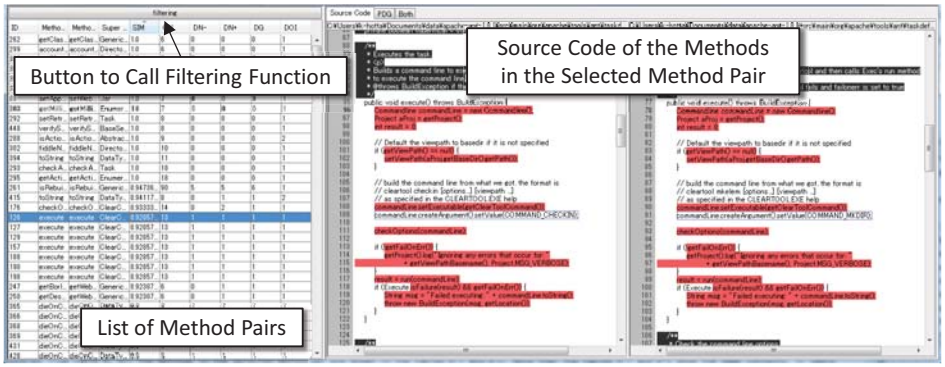


Figure 6.18: A Whole Snapshot of *CRat* (for Method Pairs)

6.6 Implementation

6.6.1 Overview

We have implemented the proposed method as a tool named *CRat* (*Clone Removal Assistant Tool*) in Java. *CRat* can handle software systems written in Java, because *Scorpio*, the clone detection tool used in *CRat*, can handle only Java. However, the proposed method can be applied to other programming languages if PDGs can be built.

The LOC of *CRat* is 17,290 with comments and white lines. It becomes 11,125 without comments and white lines. Moreover, *CRat* consists of 136 source files. In addition, it uses the external libraries except for *Scorpio* and *MASU*.

JUNG: *JUNG* (*Java Universal Network/Graph Framework*) is a framework that provides software libraries for the modeling, analysis, and visualization of data that can be represented as a graph or network [67]. *CRat* uses it to visualize PDGs. *JUNG* is an open source project in SourceForge likewise *MASU*.

CRat has two modes. One is for method pairs, and the other is for method groups. We describe each of them in detail in the following subsections. Note that *CRat* does not modify the source code by itself. Therefore, users need to perform source code modification by their own effort.

6.6.2 Functionalities for Method Pairs

Figure 6.18 shows a snapshot of *CRat* for method pairs. The table shows all the candidate method pairs that *CRat* detected. When users select a method pair

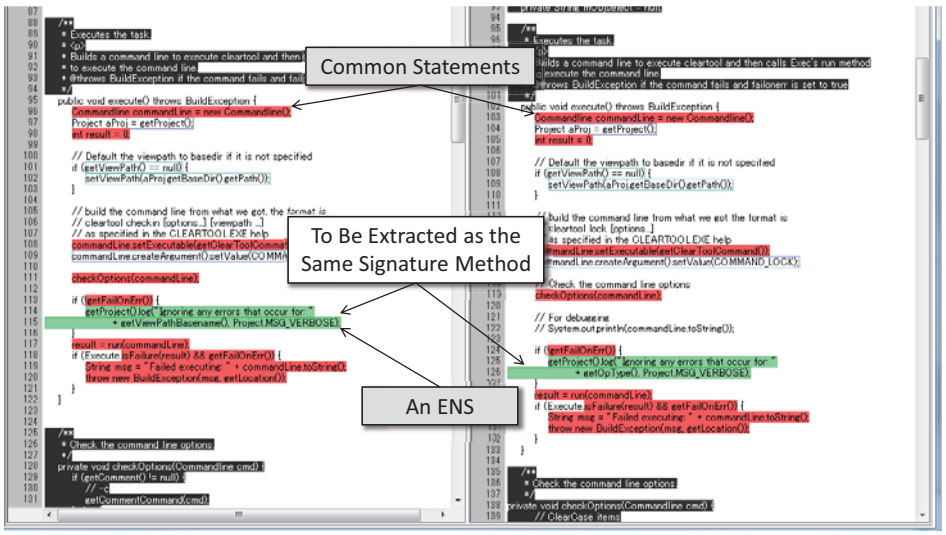


Figure 6.19: A Snapshot of Source Code View

from the table, the source code of methods included in the pair is shown in the right panel.

Figure 6.19 shows a snapshot of source code view. In the source code view, common statements are highlighted with red. Statements highlighted by red mean that they should be pulled up into the base class as a template method. On the other hand, the other statements are unique processes in each method. Statements surrounded by the same color rectangles make an ENS. In addition, if users click statements that are not highlighted by red, an ENS that includes the statements is highlighted. Moreover, if users click an ENS in one method, *CRat* also highlights the corresponding ENS in the other method. ENSs highlighted by the same color are under the correspondence relationship ($S_{m_1} \rightleftharpoons S_{m_2}$), which indicates that the methods created by extracting them have the same signature. Additionally, *CRat* shows the signature of the method created from an ENS if users put cursor on the ENS.

CRat also has a PDG view. Figure 6.20 shows a snapshot of PDG view. Each circle indicates a node of PDG, and each line indicates an edge of PDG. Nodes colored by red are nodes whose owner statements are included in common statements. The blue lines indicate data dependence edges, and the black broken lines indicate control dependence edges. The character string on each data dependence edge indicates the name of the variable that the edge represents. Note that *CRat* omits the method enter nodes and execute dependence edges in PDG view. To visualize

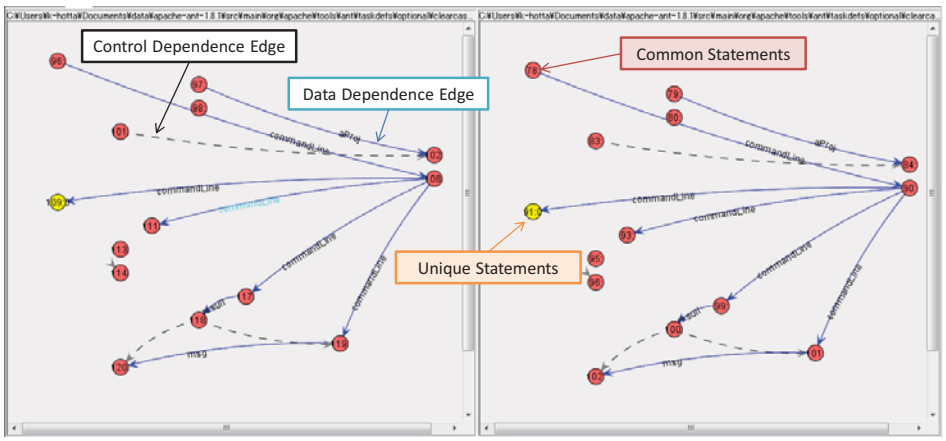


Figure 6.20: A Snapshot of PDG View

PDGs, *CRat* uses APIs provided by *JUNG*.

CRat can show both the source code view and PDG view at a time. Figure 6.21 shows the apposing view of source code view and PDG view. The functionalities of the source code view and the PDG view in the apposing view are exactly same to the original ones.

In addition, *CRat* has a filtering function of method pairs with some metrics. All the metrics are calculated for each method pair. The metrics are as follows under the assumption that m_1 and m_2 are methods in a method pair, and G_m is the PDG of the method m.

SIM: The similarity between two methods of each method pair (defined in the formula (6.26)).

$$SIM := \frac{|CommonNodes(G_{m_1})| + |CommonNodes(G_{m_2})|}{|V_{G_{m_1}}| + |V_{G_{m_2}}|} \tag{6.26}$$

CN: The number of nodes whose owner statements are included in common statements (defined in the formula (6.27)).

$$CN := |CommonNodes(G_{m_{1(2)}})| \tag{6.27}$$

DN+, DN-: The number of nodes whose owner statements are not included in common statements. Note that the values of this metric are different between

Figure 6.21: A Snapshot of Apposing View of Source Code View and PDG View

each method. Therefore, we define DN+ as the larger one (formula (6.28)), and DN- as the smaller one (formula (6.29)), respectively.

$$DN+ := max(|DiffNodes(G_{m_1})|, |DiffNodes(G_{m_2})|) \quad (6.28)$$
$$DN- := min(|DiffNodes(G_{m_1})|, |DiffNodes(G_{m_2})|) \quad (6.29)$$

LOC+, LOC-: The number of lines of each method. Obviously, the values of this metric are different between each method. Likewise DN+ and DN-, we define LOC+ as the larger one, and LOC- as the smaller one, respectively.

DG: The number of new methods that are created by extracting ENSs. DG is defined in the formula (6.30), where N is the number of ENSs that have their correspondents in the other method. Note that the 'correspondent' of an ENS S_1 is $\leftrightharpoons (S_1)$.

$$DG := |DiffNodeSets(G_{m_1})| + |DiffNodeSets(G_{m_2})| - N \quad (6.30)$$

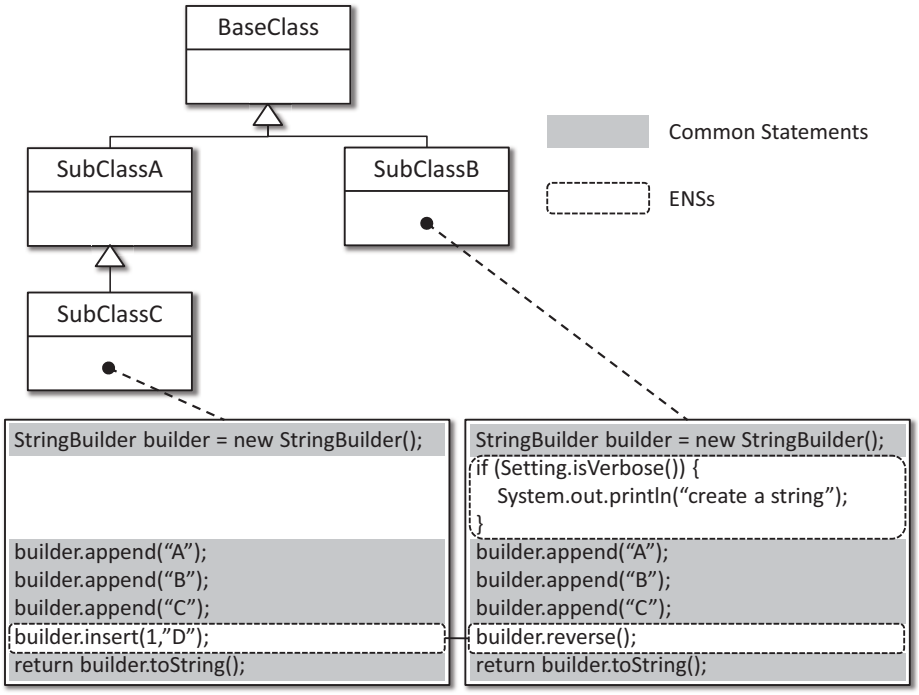


Figure 6.22: An Example of Candidate Method Pair

DOI: The depth of inheritance from the common base class to the owner classes of the two methods. If the value is different for each method, we choose the larger one as the value of DOI.

Table 6.1 shows the values of metrics of the method pair shown in Figure 6.22. Note that the values of inheritance depth from the common base class are different for each class that has the target method, therefore the larger value ‘2’ is chosen as the value of DOI in this example.

Users can make a short list of candidate method pairs with the filtering function. The filtering function returns a list of method pairs whose metrics values are in the range that users specified. To call the filtering function, users push the button on the top of the table listing the method pairs.

Table 6.1: The Values of Metrics in the Method Pair of Figure 6.22

SIM	CN	DN+	DN-	LOC+	LOC-	DG	DOI
0.769	5	3	1	9	6	2	2

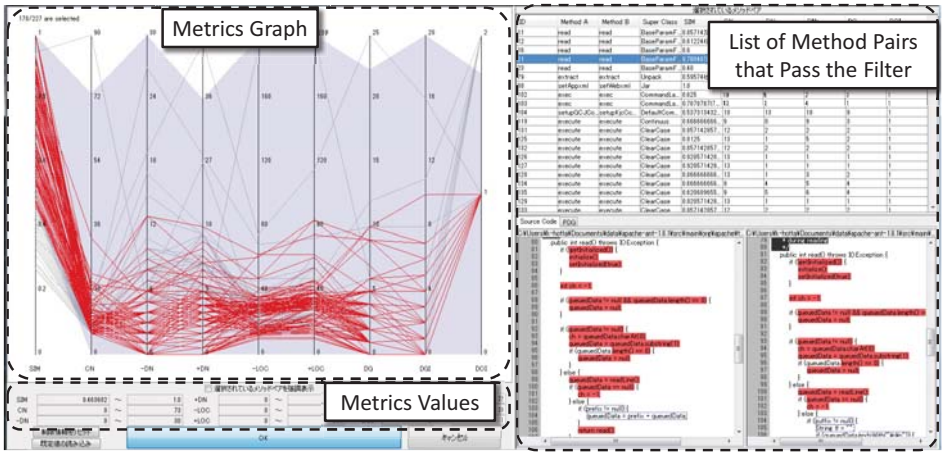


Figure 6.23: A Snapshot of Filtering View

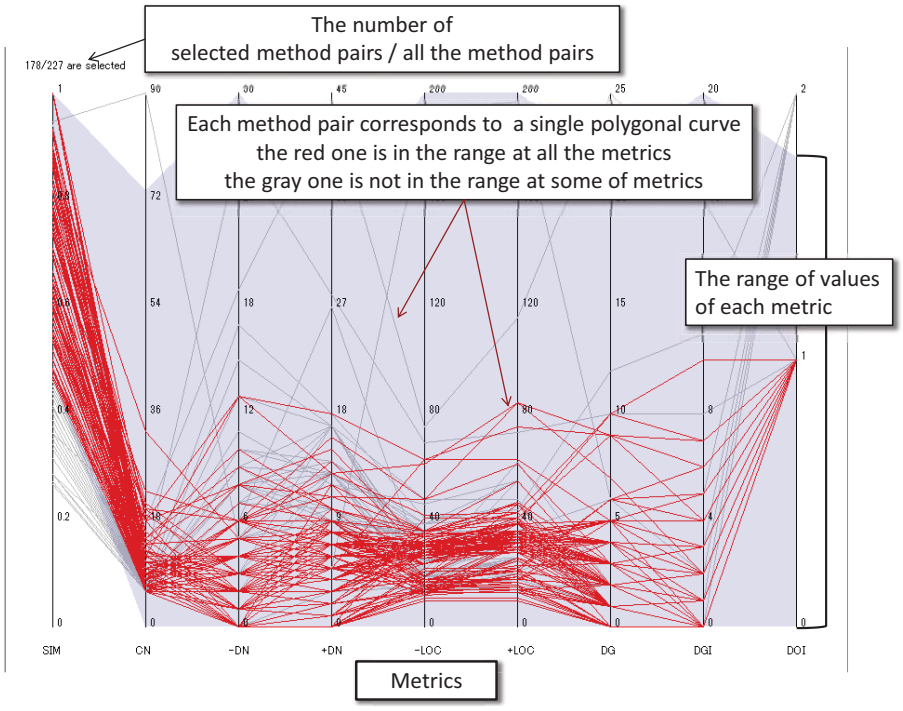


Figure 6.24: A Metrics Graph

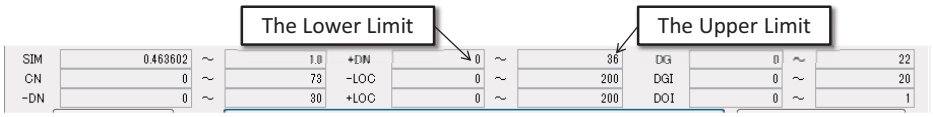


Figure 6.25: View of the Metrics Values

A filtering view is launched when users push the button. Figure 6.23 shows a snapshot of the filtering view. The filtering view consists of three parts: a metrics graph, a list of metrics values, and a list of method pairs that pass the filtering. Figure 6.24 shows a metrics graph. Users specify the thresholds of each metric by dragging the graph. The area whose background color is gray indicates the range of thresholds for every metric, and the area whose background color is white indicates the outside of the range. In the metric graph, each polygonal curve corresponds to a method pair. The polygonal curve becomes red if and only if all the metrics of the method pair represented by the polygonal curve are in the specified threshold. If any of the metrics is not in the threshold, the polygonal curve becomes gray. The specified lower limit and the specified upper limit of each metric are shown in the metrics values view (Figure 6.25). The list of selected method pairs is shown in the right of the filtering view. Users can view the source code and the PDGs of method pairs that are listed in the view. The functionalities of the source code view and the PDG view in the filtering view are exactly same to the original ones.

6.6.3 Functionalities for Method Groups

Figure 6.26 shows a snapshot of *CRat* for method groups. The left table shows all the candidate method groups that *CRat* detected. When users select a method group from table A, all the methods in the selected method group are shown in the tables B-1 and B-2. Note that the tables B-1 and B-2 show the same contents. If users choose one of the methods in table B-1, the source code of the selected method is shown in the source code view C-1. Similarly, if users choose one of the methods in table B-2, its source code is shown in the source code view C-2. The source code view and the PDG view are the same to those of described in 6.6.2.

6.7 Evaluation

In order to evaluate the proposed method, we conducted experiments on two open source software systems. Table 6.2 shows the target software systems, their scale, and the environment of the experiments. The following subsections describe each of the experiments.

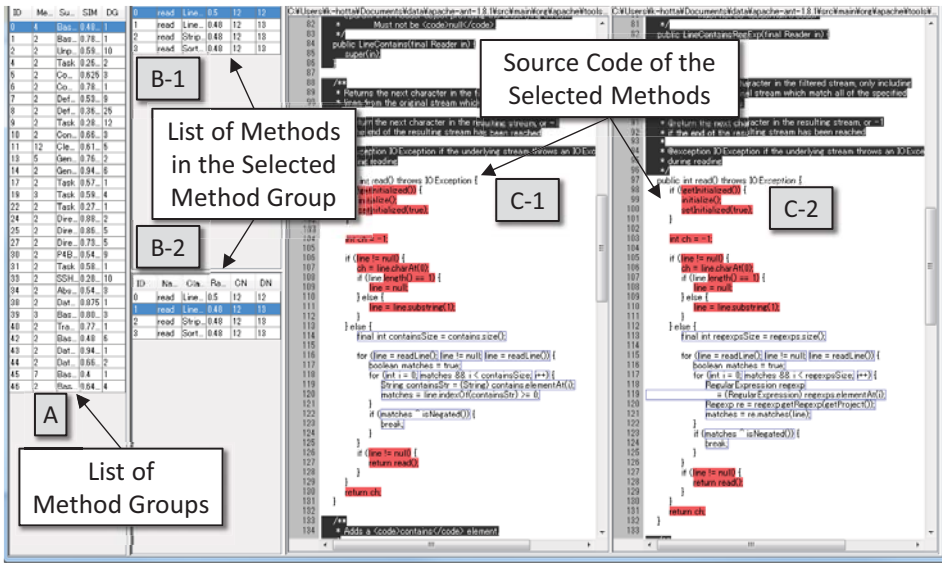


Figure 6.26: A Whole Snapshot of *CRat* (for Method Groups)

6.7.1 Evaluation of Supporting for Method Pairs

Table 6.3 shows the the number of detected candidates, and elapsed time to execute *CRat* on each target software system. The numbers of candidate method pairs are 226 and 45, so that it can be difficult for users to identify all the candidates manually. In addition, *CRat* can detect all the candidates in a few minutes although the target software systems have hundreds of source files.

Figure 6.27 shows a refactoring candidate in Ant detected by *CRat* and the result of the refactoring. In this example, there is a base class, `ClearCase`,

Table 6.2: Target Software Systems

Name	LOC	# of Files	Environment
Ant	212,401	829	CPU: Xeon 2.27GHz(8 core), RAM: 32GB
Synapse	58,418	383	

Table 6.3: The Number of Detected Candidates and Elapsed Time on Method Pairs

Name	# of Candidates	Elapsed Time [s]
Ant	226	178
Synapse	45	66

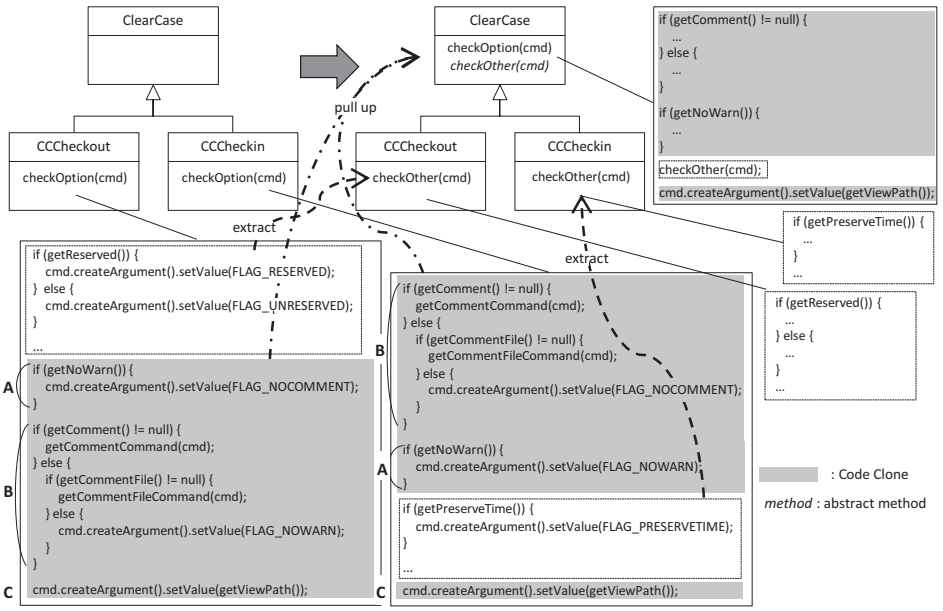


Figure 6.27: An Example of Application of Form Template Method with the Proposed Method

and there are two derived classes, `CCCheckout` and `CCCheckin`.There are also similar methods in the derived classes, `checkOption`. By applying Form Template Method to this target, duplicate statements are pulled up into in the method `checkOption` defined in the base class and new methods `checkOther` are created to implement the unique statements in each derived class. Note that there is a difference of the order of code fragments in code clones: in `CCCheckout` the code fragments labeled A, B, and C are executed in this order, however in `CCCheckin` the order of code fragments is B-A-C. Therefore, this example is an instance that the previous techniques cannot detect.

In addition, we applied Form Template Method refactoring to all the 45 candidates that the proposed method had suggested in sf Synapse in order to confirm the adequacy and the efficiency of the proposed method as a technique to support refactorings. In this experiment, we successfully refactored all the 45 candidates detected with *CRat* in sf Synapse, and confirmed that the behavior of the program is preserved by using test suites attached to the software system. Additionally, we measured the time needed to each of the refactorings. Figure 6.28 shows the boxplots of the time needed to apply refactorings. Because *CRat* suggests that all the candidates can be refactored at a time, we run *CRat* at once and apply refactorings

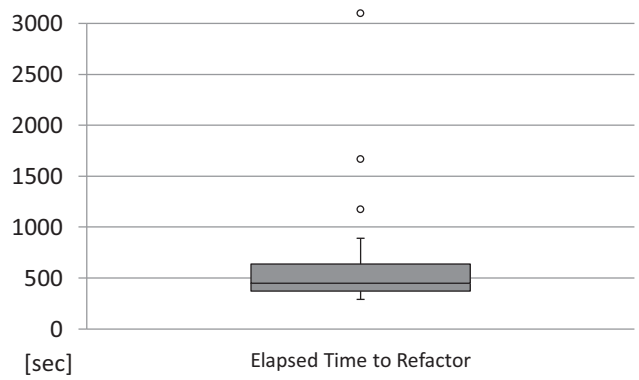


Figure 6.28: The Box-Plot of the Time to Apply Form Template Method on Synapse

using the output. The time to execute *CRat* to sf Synapse is 95 seconds as shown in Table 6.3. As a result, we could apply refactorings in few minutes in average nevertheless we are unfamiliar with the software.

6.7.2 Evaluation of Supporting for Method Groups

Table 6.4 shows the number of detected method groups, the number of them that have three or more methods, and elapsed time to detect them on each target software.

Likewise on method pairs, we applied Form Template Method refactoring to all the six method groups that the proposed method had suggested in sf Synapse. As a result, we successfully refactored all the candidates and confirmed that the behavior is preserved by using test suites.

Table 6.4: The Number of Detected Candidates and Elapsed Time on Method Groups

Name	# of Candidates	# of Candidates (3 or more methods)	Elapsed Time [s]
Ant	48	18	195
Synapse	6	2	68

6.7.3 Experiment with Subjects

Overview of the Experiment

We conducted an experiment with seven subjects. All the subjects belong to Graduate School of Information Science and Technology, Osaka University, or Department of Information and Computer Sciences in School of Engineering Science, Osaka University.

The objective of this experiment is to investigate the effectiveness of the proposed method as refactoring support method. In this experiment, subjects applied Form Template Method refactoring after a short introduction and practice, and we measured elapsed time that they needed to finish the refactoring. All the subjects refactored two method groups described in 6.7.3. Subjects applied refactorings to one candidate method group with *CRat*, and they applied refactorings to the other candidate with *CCFinder*.

Target Method Groups

As described above, subjects applied Form Template Method to two method groups. We call the two method groups **Candidate-A** and **Candidate-B**, respectively.

Figure 6.29 shows the source code and the outputs of *CRat* on each candidate. The features of the two method groups are shown in Table 6.5.

Prodcedure of the Experiment

The procedure of the experiment consists of five steps as follows.

1. We give a brief introduction to subjects.
2. Subjects apply Form Template Method to a simple example.
3. We divide subjects into four groups. Table 6.6 shows the groups and assignments of each subject.
4. Subjects apply refactorings to the assigned method group.

Table 6.5: The Features of Target Method Groups

Label	# of Methods	# of Common Nodes	# of ENSs
Candidate-A	3	19	3
Candidate-B	12	9	5

```
public PlanarImage executeDrawOperation() {
   BufferedImage bi = new BufferedImage(width, height,
                                        BufferedImage.TYPE_4BYTE_ABGR_PRE);
   Graphics2D graphics = (Graphics2D) bi.getGraphics();

   if (!stroke.equals("transparent") {
      BasicStroke bStroke = new BasicStroke(stroke_width);
      graphics.setColor(ColorMapper.getColorByName(stroke));
      graphics.setStroke(bStroke);
      graphics.draw(new Ellipse2D.Double(0, 0, width, height);
   {
   if (!fill.equals("transparent")) {
      graphics.setColor(ColorMapper.getColorByName(fill));
      grahpics.fill(new Ellipse2D.Double(0, 0, width, height));
   }
   for (int i = 0; i < instructions.size(); i++) {
      ImageOperation instr = ((ImageOperation) instructions.elementAt(i));
      if (instr instanceof DrawOperation) {
         PlanarImage img = ((DrawOperation) instr).executeDrawOperation();
         graphics.drawImage(img.getAsBufferedImage(), null, 0, 0);
      } else if (instr instanceof TransformOperation) {
         graphics = (Graphics2D) bi.getGraphics();
         PlanarImage image = ((TransformOperation) instr)
               .executeTransformOperation(PlanarImage.wrapRenderedImage(bi));
         bi = image.getAsBufferedImage();
      }
   }
   return PlanarImage.wrapRenderedImage(bi);
}
```

: Common Statements : ENSs

(a) Candidate-A

```
public void execute throws BuildException {
   Commandline commandline = new Commandline();
   Project aProj = getProjcet();
   int result = 0;

   if (getViewPath() == null) {
      setViewPath(aProj.getBaseDir().getPath());
   }

   commandLine.setExecutable(getClearToolCommand());
   commandLine.createArgument().setValues(COMMAND_MKBL);

   checkOption(commandLine);

   if (!getFailOnErr()) {
      getProject().log("Ignoring any errors that occur for: " +
                       getBaselineRootName(), Ptoject.MAG_VERBOSE);
    }

   result = run(commandline);
   if (Execute.isFailure(reault) && getFailOnErr()) {
      String msg = "Failed executing: " + commandLine.toString();
      throw new BuildException(msg.getLocation());
   }
}
```

: Common Statements : ENSs

(b) Candidate-B

Figure 6.29: Candidate Method Groups

5. Subjects apply refactorings to the other method group.

Introduction to subjects

At first, we gave an introduction to subjects about the background of this study and the experimental procedure. The introduction includes the following information.

- Code clones and their removal techniques.
- Form Template Method refactoring pattern.
- How to apply Form Template Method.
- How to use *CRat*.
- The procedure of the experiment.

Practice

Second, we have subjects practice applying Form Template Method with a simple method group. The target method group consists of two methods, and it contains two ENSs (note that we count a pair of ENSs (S_1, S_2) as 1 ENS if $S_1 \rightleftharpoons S_2$). The purposes of the practice are (1) to have subjects understand refactoring steps of Form Template Method, and (2) to have subjects be familiar with the tool.

Table 6.6: Groups of Subjects

Group ID	Subjects	First	Second
1	Subject 1 Subject 2	Candidate-A *CRat*	Candidate-B *CCFinder*
2	Subject 5	Candidate-B *CRat*	Candidate-A *CCFinder*
3	Subject 6 Subject 7	Candidate-A *CCFinder*	Candidate-B *CRat*
4	Subject 3 Subject 4	Candidate-B *CCFinder*	Candidate-A *CRat*

Grouping

In the next, we divide subjects into four groups. Table 6.6 shows the groups of subjects. As this table shows, the differences between each group are as follows:

- Which candidate do they refactor first?
- Which candidate do they use *CRat*?

Apply Refactoring

Subjects apply refactoring to the assigned target method group. For example, subjects in Group 1 refactor candidate A with *CRat*. We measure the elapsed time required to finish the refactoring for every subject.

Result

Table 6.7 shows the elapsed time to finish Form Template Method for every subject. The numeric characters in this table 'hh:mm:ss' indicates that the subject need hh hours and mm minutes and ss seconds to finish their refactoring tasks. For example, Subject 1 had finished applying refactoring on Candidate-A in 22 minutes and 45 seconds. 'N/A' means that the subject cannot finish refactoring in the case.

Table 6.7: Elapsed Time to Finish Form Template Method Application

Subjects	Group ID	Candidate-A		Candidate-B	
Subject 1	1	with *CRat*	0:22:45	with *CCFinder*	0:50:30
Subject 2	1		0:52:00		1:17:00
Subject 3	4		0:19:22		N/A
Subject 4	4		0:09:58		0:27:45
Subject 5	2	with *CCFinder*	0:12:04	with *CRat*	0:25:30
Subject 6	3		0:22:55		0:50:28
Subject 7	3		0:35:20		1:04:15

Table 6.8: The Average Time

	Candidate-A	Candidate-B	Both
with *CRat*	0:28:14	0:46:44	0:34:54
with *CCFinder*	0:23:26	0:51:45	0:37:36
Both	0:25:50	0:49:15	0:36:09

As the table shows, the time required to finish refactoring tasks varies greatly among subjects. There is also great variability among Candidate-A and Candidate-B; all the subjects required much time on Candidate-B than Candidate-A. This is because the degree of difficulty of Candidate-B is higher than that of Candidate-A. Table 6.8 shows the average time to finish the refactorings. As the table shows, the elapsed time with *CRat* is higher than that with *CCFinder* in Candidate-A, meanwhile the opposite result is shown in Candidate-B. As a result, *CRat* cannot reduce time required for the refactorings in the easier candidate, but it can reduse time required for the refactorings in the more difficult candidate. Therefore, *CRat* is useful in a case that the target method group is complex and it has a number of the methods.

6.8 Discussion

6.8.1 PDG Creation

There are some other dependences except data, control, and execute dependence that should be considered in PDGs. In the proposed method, dependence of break and continue statements and dependence of exception are considered. However, the proposed method does not consider dependence caused by the following factors.

- Library call.
- Alias.
- Presence of innner classes.
- Reflection.

Of these factors, we can consider dependence caused by library calls by giving the source code of libraries as additional input of the proposed method. However, it is quite difficult to give the source code of all the libraries that are used in the target software systems as the input.

In the experiments of this study, we cannot find instances that suffer any problems by dependence that are not considered in the proposed method. However, there is a risk that the proposed method suggests refactoring candidate incorrectly by these dependence. Thus, it is necessary to consider these factors to make the proposed method robust.

6.8.2 Detection of Common Statements

As described in 6.4.4, the proposed method omits clone pairs except the most largest one in the case that there are duplications of clone pairs. The purpose of this is to suggest more nodes as common statements. However, in some cases, this selection may be not appropriate. We can avoid this problem by delegating the selection to users. However, the proposed method does not have this function at present.

6.8.3 Candidates that Need to be Tailored

As we described in Section 6.7.1, we applied Form Template Method refactoring to 45 method pairs detected in Synapse on method pairs. In some cases, we had to make some modifications that *CRat* did not indicate, or we had to make some tailoring to the output of *CRat* to apply the pattern. Table 6.9 shows the modifications or adjustments needed to apply refactorings, and the number of candidates that needed them. The definitions of the terms in the table are as follows: the term "modify ENS" means the cases in which we had to modify ENSs or their pairwise relationships between two methods that *CRat* suggests; the term "move methods into base class or change their visibility" means the cases in which some methods defined in derived classes are used in common processes and we had to move those methods into the base class and/or change their visibilities; and the term "replace field references to calls of getter methods" indicates the cases in which some fields are used in duplicate statements and they are not visible from the base class and we had to replace references of these fields to calls of getter methods of them.

Issues of Visibility

The proposed method does not consider the visibility of methods and fields in the source code. Therefore, code fragments that should be pulled up into the template method can call methods or reference fields that are not accessible from the base class. In such cases, we need additional modifications on the source code

Table 6.9: The Candidates that Need some Modifications for *CRat*'s Outputs

# of candidates that need no modifications	29
# of candidates that need some modifications	16
modify ENS	12
move methods into base class and/or change their visibilities	4
replace field references to calls of getter methods	2

to apply Form Template Method. We can apply the pattern to such candidates by changing the visibility of methods and fields. However, it is not desirable that code clone removal requires increasing the visibility of methods or fields, because such changes could cause vulnerability [106]. For fields, if fields have getters and setters, we can resolve this problem by using them.

Issues of ENSs and their Relationships

The proposed method automatically detects ENSs and correspondence relationships of ENSs. However, the automatically detected ENSs or relationships of ENSs may not fit with users' sensibilities. Although the automatically detected ENSs and their relationships do not always suitable, they can help users apply refactoring.

6.8.4 Detection of Method Groups

The proposed method forms method groups from all the methods that satisfy the requirements described in 6.4.3. However, it may be more suitable to form method groups from a subset of the methods. We can improve this issue by delegating the selection of methods that should be included in method groups to users. However, the proposed method currently does not have this functionality.

6.8.5 Threats to Validity of the Experiment with Subjects

In the experiment with subjects, we confirmed that the proposed method reduces time to refactor in a case that the target is complex and there is a number of methods in the target method group. However, we found the opposite result in a case that the target is not complex. There might be bias of subjects' abilities, so that the result might occur. We may get another result with different grouping of subjects.

6.9 Summary

Form Template Method enables maintainers to remove code clones with some gaps. Because of its difficulty, there exists some techniques to support Form Template Method applications. However, the existing techniques still remain some issues.

This chapter proposed a new technique to assist developers to apply Form Template Method refactorings to code clones. It detects refactoring candidates automatically and suggests them to its users. It uses program dependence graphs as

its data structure, which enables to assist developers removing code clones having trivial differences that have no impact on the meanings of the program. Moreover, it can handle a group of three or more methods, which increases the practicality of code clones removal.

We implemented the proposed method as a tool, and conducted an experiment to evaluate the proposed method. We applied Form Template Method to all the candidates that the tool suggests in an open source software, and confirmed that we can refactor the candidates with preserving the behavior of the program.

Chapter 7

Conclusion

7.1 Contributions

The objective of the work presented in this dissertation is to promote efficiencies of development and maintenance of software systems. This dissertation focused on code clones to achieve the objective, and it presented some studies on code clones. In summary, it contributes to the followings:

- A survey on research literatures that are related to code clone management.
- An empirical study on impacts of code clones on software evolution by comparing stabilities of cloned and non-cloned code.
- A new technique to track code clones across version histories of software systems based on an enhancement of an existing clone tracking technique, which is named CRD.
- An empirical investigation on clone genealogies to reveal characteristics of harmful clones.
- A refactoring support for code clones with Form Template Method refactoring pattern.

There are many research papers relating to code clones because code clones have recieved great attention and interests as a research topic on the field of software engineering. This dissertation, therefore, presented a survey at first to look down at the current state and open issues around code clones. The survey introduced discussions on code clones, specifically on the perspectives of causes of creation and harmfulness. After that, it introduced techniques, ideas, and findings

relating to code clones. The introduction of research achievements loosely classified them into five categories; detection, removal, prevention, analysis, and finding bugs.

Based on the survey results, we conducted four studies to promote efficient management of code clones.

An empirical study on impacts of code clones aimed to reveal whether code clones are generally harmful or not. The study stood on a basis that code clones are harmful *if code clones are frequently modified than non-cloned code*. The study defined a metric, named modification frequency, to calculate frequencies of modifications on cloned and non-cloned code, and then it calculated values of the metrics on 15 open source software systems with four clone detectors. As a result, it was revealed that cloned code tends to be stable compared to non-cloned code. This experimental finding indicates that clones do not have seriously negative impacts on software evolution. However, our detail analysis on the finding revealed that there existed some instances of code clones that were frequently modified. Therefore, we can say that not all but a part of clones have negative impacts on software evolution as a summary of the experimental results.

The dissertation presented the study result on **an enhancement of CRD-based clone tracking**. Tracking clones across version histories plays an important role on analyzing code clones. That is, it enables researchers to analyze evolution of code clones. Analyzing clone evolution has a wide variety of applications, including revealing characteristics of clones and finding inconsistencies of clones. Although some techniques have been proposed to track code clones, they have still some issues. The clone tracking technique proposed in this dissertation resolved issues of the previous technqiues by enhancing an existing technique, named CRD. CRD represents information of the location where a code fragment stands. The original CRD-based technique to track clones realizes clone tracking with exact matches of CRDs. On the other hand, the new clone tracking technique proposed in this dissertation uses not only exact matches of CRDs but also similarities of CRDs. The idea is quite simple, but it was confirmed that the simple idea improved accuracies of clone tracking through the experimental results on two open source software systems.

Analyzing clone genealogies with the new tracking technique followed the enhancement of CRD-based clone tracking. Our first empirical study revealed that a part of clones have negative impacts, but there still exists some questions on harmfulness of clones. To reveal such questions, we detected and analyzed clone genealogies that describe how individual clones evolved in version histories of software systems. The detection of clone genealogies was based on the clone tracking techniques proposed by this dissertation. First, this study revisited some common findings on clone evolution with the new clone tracking technique, and the experi-

mental results supported the common findings. In the next, this study investigated some open questions, including *how many clones have negative impacts on software evolution* and *there is any characteristics of timings when clones were modified across their lifetimes*. As a result, it revealed that approximately 3% of clones had negative impacts. Hence, this finding empirically supports our fingings that not all but a part of clones are harmful. Furthermore, it was revealed that clones tended to be modified more frequently in the former halves of their lifetimes than the latter halves. This fact indicates an importance for start managing clones in the earlier stage of their lifetimes.

Finally, we proposed **a refactoring support for code clones with** Form Template Method. One of the ways to cope with harmful code clones is removing them from source code. However, removing clones should be performed in a cautious manner because it is quite challenging to complete refactorings on clones without introducing any failures. This fact indicates a requirement for tool support. In addition, removing clones will become more difficult in the case they include some gaps. Form Template Method is a refactoring technique that can be used for removing clones including some gaps. However, applying the refactoring pattern is a more complicated task compared to other refactoring applications because of the complexity of the pattern. Hence, tool supports are strongly required to apply Form Template Method refactroing pattern. For these reasons, this dissertation proposed a refactoring support for Form Template Method to cope with harmful clones after they are detected. The proposed technique automatically detects all the refactoring candidates that can be refactored by Form Templete Method pattern. The purpose of automatic detection of refactoring candidates is to reduce costs to identify where to be refactored. Form Template Method is a complex pattern, and so detecting candidates of the refactoring pattern is a costly task. Therefore, supporting for identifications of refactoring candidates is necessary. Furthermore, the technique shows how to refactor each of detected candidates to be refactored. This function should reduce costs to apply refactorings. We have developed the proposed technique as a software tool named *CRat*, and confirmed the usefulness and effectiveness of the proposed method through two experiments.

7.2 Future Research Directions

This section discusses open issues and future research directions of the work presented in this dissertation.

The empirical study presented in Chapter 3 targets 15 software systems with four different clone detectors, and so we believe that the findings will have sufficient generalities. However, the study has a limitation that the experimental targets

of it are only open source software systems. Industrial software systems should have different characteristics from open source software systems, which makes it difficult to generalize our findings to industrial software systems. One of the future directions of this study is to reval characteristics of clones that are frequently modified. Revealing the characteristics of frequently modified clones should help us to detect or predict harmful clones.

Clone tracking technique proposed in Chapter 4 has higher accuracy than the original CRD based clone tracking. However, there exists some instances that even the improved technique cannot track. These failures of clone tracking will be resolved by relaxing the conditions for clones to be linked. On the other hand, relaxed conditions have a risk to introduce more false positives. Therefore, one of the important future work of this study is to seek more appropriate conditions for linking clones that improve accuracy of clone tracking without introducing any false positives.

The empirical study for clone genealogies presented in Chapter 5 has a similar issue of the study presented in Chapter 3. That is, the study targets only open source software systems. Hence, as well as the study in Chapter 3, it requires more experiments on industrial software systems to generalize the findings. In addition, it also has another limitation that it adopts only a block-based clone detector. It is required to use other clone detectors to generalize the findings although experiments with other detectors should need much time. Furthermore, the block-based clone detector used in this study can handle only Java. Therefore, more experiments are necessary on other software systems written in other programming langauges.

One of the important future work for the refactoring support proposed in Chapter 6 is automatic code transformation. Form Template Method refactoring is a complex refactoring pattern, so manual refactorings with this pattern have a high risk to introduce human errors. Automatic code transformation will contribute to safe refactoring with Form Template Method, and so it is necessary to realize this functionality to better refactoring support. Futhermore, combining this technique and other refactoring support techniques for other refactoring patterns will be useful to remove code clones. Form Template Method is nothing more than one of the choises for clone removal. Therefore, not Form Template Method but other refactoring patterns should be suited for some cases. Supporting multiple patterns at a time will be helpful to choose the best solution for individual cases.

The most fundamental and important challenge of our future research is to detect harmful clones. Our investigations on code clones are based on histories of them, but our reserach does not reach detection of harmful clones. Our findings suggested to manage a part of code clones, but they cannot answer the question that what clones should be managed. Idengifying harmful clones must be a basis of efficient clone management because developers or maintainers need to pay

attention only for such harmful clones. For these reasons, we need to develop a technique to detect harmful clones as a major future research direction.

One of the way to detect harmful clones is to learn characteristics of harmful clones existed in previous revisions of the software system with machine learning techniques. However, there is no generic definitions for the harmfulness of clones. To detect harmful clones, it is necessary to reveal what clones are harmful. This dissertation regarded clones as harmful if they are frequently modified, or they are long-lived. However, there will exist some other definitions of harmful clones, including *clones are harmful if they cause any bugs*. Hence, much more discussions are required for harmfulness of clones.

Bibliography

[1] IEEE Standard 12207. *Standard for Information Technology - Software Life Cycle Processes*, 1996.

[2] ISO/IEC 14764. *Software Engineering - Software Maintenance*, 1999.

[3] E. Adar and M. Kim. SoftGUESS: Visualization and Exploration of Code Clones in Context. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), May 2007.

[4] R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proceedings of the 11th International Conference on Data Engineering* (*ICDE 1995*), pages 3–14, Mar. 1995.

[5] L. Aversano, L. Cerulo, and M. Di Penta. How Clones are Maintained: An Empirical Study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering* (*CSMR 2007*), pages 81–90, Mar. 2007.

[6] S. Baba, N. Yoshida, S. Kusumoto, and K. Inoue. Application of Code Clone Information to Fault-Prone Module Prediction. *IEICE Transactions on Information and Systems*, J91-D(10):2559–2561, Oct. 2008. (in Japanese).

[7] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the 2nd Working Conference on Reverse Engineering* (*WCRE 1995*), pages 86–95, July 1995.

[8] T. Bakota, R. Ferenc, and T. Gyim'othy. Clone Smells in Software Evolution. In *Proceedings of the 23rd International Conference on Software Maintenance* (*ICSM 2007*), pages 24–33, Oct. 2007.

[9] M. Balazinska, E. Merlo, M. Dagenais, and B. Lague. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proceedings of*

the 7th Working Conference on Reverse Engineering (*WCRE 2000*), pages 98–107, Nov. 2000.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the 6th International Software Symposium on Software Metrics* (*METRICS 1999*), pages 292–303, Nov. 1999.

[11] H. Basit and S. Jarzabek. A Data Mining Approach for Detecting Higher-Level Clones in Software. *IEEE Transactions on Software Engineering*, 35(4):497–514.

[12] H. A. Basit and S. Jarzabek. Detecting Higher-level Similarity Patterns in Programs. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering* (*ESEC/FSE 2005*), pages 156–165, Sep. 2005.

[13] H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, and S. Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the 15th International Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2007*), pages 513–516, Nov. 2007.

[14] S. Bassil and R. K. Keller. Software Visualization Tools: Survey and Analysis. In *Proceedings of the 9th International Workshop on Program Comprehension* (*IWPC 2001*), pages 7–17, May 2001.

[15] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance* (*ICSM 1998*), pages 368–377, Mar. 1998.

[16] S. Bellon, R. Koschke, G. Antniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 31(10):804–818, Oct. 2007.

[17] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: A Road Map. In *Proceedings of the Conference on the Future of Software Engineering in the 22nd International Conference on Software Engineering* (*ICSE 2000*), pages 73–87, June 2000.

[18] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *Science of Computer Programming*, 77(6):760–776, June 2012.

[19] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[20] E. Burd and J. Bailey. Evaluating Clone Detection Tools for Use during Preventative Maintenance. In *Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation* (*SCAM 2002*), pages 36–43, Oct. 2002.

[21] CCFinderX. available at <http://www.ccfinder.net/ccfinderx-j.html>.

[22] E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored? : A case study of three OSS projects. In *Proceedings of the 5th Workshop on Refactoring Tools* (*WRT 2012*), pages 1–7, June 2012.

[23] CloneDR. available at <http://www.semdesigns.com/Products/Clone/>.

[24] R. Cottrell, J. J. Chang, R. J. Walker, and J. Denzinger. Determing Detailed Structural Correspondence for Generalization Tasks. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the 15th International Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2007*), pages 165–174, Nov. 2007.

[25] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice - Experiences from Windows. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation* (*ICST 2011*), pages 357–366, Mar. 2011.

[26] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proceedings of the 12th Annual Symposium on Computational Geometry* (*SCG 2004*)*, pages = 253-262, year = 2004, month = June*.

[27] E. Duala-Ekoko and M. P. Robillard. Clone Region Descriptors: Representing and Tracking Duplication in Source Code. *ACM Transactions on Software Engineering and Methodology*, 20(1):3:1–3:31, June 2010.

[28] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the 15th International Conference on Software Maintenance* (*ICSM 1999*), pages 109–118, Aug. 1999.

[29] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, Jan. 2001.

[30] R. Falke, P. Frenzel, and R. Koschke. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Software Engineering*, 13(6), Dec. 2008.

[31] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[32] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[33] N. Göde. Evolution of Type-1 Clones. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2009*), pages 77–86, Sep. 2009.

[34] N. Göde and J. Harder. Clone Stability. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering* (*CSMR 2011*), pages 65–74, Mar. 2011.

[35] N. Göde and R. Koschke. Frequency and Risks of Changes to Clones. In *Proceedings of the 33rd International Conference on Software Engineering* (*ICSE 2011*), pages 311–320, May 2011.

[36] N. Göde and R. Kosheke. Incremental Clone Detection. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering* (*CSMR 2009*), pages 219–228, Mar. 2009.

[37] A. Goto, N. Yoshida, M. Ioka, E. Choi, and K. Inoue. How to Extract Differences from Similar Programs? A Cohesion Metric Approach. In *Proceedings of the 7th International Workshop on Software Clones* (*IWSC 2013*), pages 23–29, May 2013.

[38] J. Harder and N. Göde. Cloned code: stable code. *Journal of Software : Evolution and Process*, Mar. 2012. doi: 10.1002/smr.1551.

[39] H. Hata, Y. Higo, and S. Kusumoto. Code Clone Version Control System for Mining Rich Clone Histories. *IPSJ Journal*, 54(2):894–902, Feb. 2013. (in Japanese).

[40] B. Hauptmann, V. Bauer, and M. Junker. Using Edge Bundle Views for Clone Visualization. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 86–87, June 2012.

[41] Y. Hayase, Y. Yong Lee, and K. Inoue. A Criterion for Filtering Code Clone Related Bugs. In *Proceedings of International Workshop on Defects in Large Software* (*DEFECTS 2008*).

[42] Y. Higo and S. Kusumoto. Code Clone Detection on Specialized PDGs with Heuristics. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering* (*CSMR 2011*), pages 75–84, Mar. 2011.

[43] Y. Higo, S. Kusumoto, and K. Inoue. A Survey of Code Clone Detection and Its Related Techniques. *IEICE Transactions on Information and Systems*, 91-D(6):1465–1481, June 2008. (in Japanese).

[44] Y. Higo, S. Kusumoto, and K. Inoue. Identifying Refactoring Opportunities for Removing Code Clones with A Metrics-based Approach. In K. Cai, editor, *Java in Academia and Research*, chapter 3, pages 57–82. Concept Press Ltd., 2011.

[45] Y. Higo, K. Sawa, and S. Kusumoto. Problematic Code Clones Identification using Multiple Detection Results. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference* (*APSEC 2009*), pages 365–372, Dec. 2009.

[46] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference* (*APSEC 2007*), pages 262–269, Dec. 2007.

[47] Y. Higo, Y. Ueda, M. Nishino, and S. Kusumoto. Incremental Code Clone Detection: A PDG-based Approach. In *Proceedings of the 18th Working Conference on Reverse Engineering* (*WCRE 2011*), pages 3–12, Oct. 2011.

[48] Y. Higo and N. Yoshida. An Introduction to Code Clone Refactoring. *Computer Software*, 28(4):42–56, 2011. (in Japanese).

[49] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, Sep. 2006.

[50] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming. In *Proceedings of the 17th International Conference on Program Comprehension* (*ICPC 2009*), pages 238–242, May 2009.

[51] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-Based Code Clone Detection: Incremental, Distributed, Scalable. In *Proceedings of the 26th International Conference on Software Maintenance* (*ICSM 2010*), pages 1–9, Sep. 2010.

[52] The Japan Patent Office in the Ministry of Economy, Trade, and Industry in Japan. `http://www.jpo.go.jp/shiryou/toushin/kenkyukai/jyouhou_iinkai.htm` (in Japanese).

[53] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of Finding Inconsistently-Changed Bugs in Code Clones of Mobile Software. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 94–95, June 2012.

[54] K. Inoue, T. Kamiya, and S. Kusumoto. Code-Clone Detection Methods. *Computer Software*, 18(5):529–536, 2001. (in Japanese).

[55] M. Ioka, N. Yoshida, T. Masai, Y. Higo, and K. Inoue. A Tool Support to Merge Similar Methods with a Cohesion Metric COB. In *Proceedings of the 3rd International Workshop on Empirical Software Engineering in Practice* (*IWESEP 2011*), pages 23–24, Nov. 2011.

[56] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Inter-Project Functional Clone Detection toward Building Libraries - An Empirical Study on 13,000 Projects. In *Proceedings of the 19th Working Conference on Reverse Engineering* (*WCRE 2012*), pages 387–391, Oct. 2012.

[57] 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 Workshop on Eclipse Technology eXchange* (*ETX 2007*), pages 16–20, Oct. 2007.

[58] F. Jacob, D. Hou, and P. Jablonski. Actively Comparing Clones Inside The Clone Editor. In *Proceedings of the 4th International Workshop on Software Clones* (*IWSC 2010*), pages 9–16, May 2010.

[59] K. Jalbert and J. S. Brandbury. Using Clone Detection to Identify Bugs in Concurrent Software. In *Proceedings of the 26th International Conference on Software Maintenance* (*ICSM 2010*), pages 1–5, Sep. 2010.

[60] Java Development Tools. available at <http://www.eclipse.org/jdt/>.

[61] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD : Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), May 2007.

[62] L. Jiang, Z. Su, and E. Chiu. Context-Based Detection of Clone-Related Bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the 15th International Symposium on the Foundations of Software Engineering* (*ESEC/FSE 2007*), pages 147–156, Nov. 2007.

[63] Z. M. Jiang and A. E. Hassan. A Framework for Studying Clones In Large Software Systems. In *Proceedings of the 7th Working Conference on Source Code Analysis and Manipulation* (*SCAM 2007*), pages 203–212, Oct. 2007.

[64] J.H. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the 10th International Conference on Software Maintenance* (*ICSM 1994*), pages 120–126, Sep. 1994.

[65] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 485–495, June 2009.

[66] N. Juillerat and B. Hirsbrunner. Toward an Implementation of the "Form Template Method" Refactoring. In *Proceedings of the 7th International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2007*), pages 81–90, Sep. 2007.

[67] JUNG. avilable at <http://jung.sourceforge.net/>.

[68] Y. Kamei, H. Sato, A. Monden, S. Kawaguchi, H. Uwano, M. Nagura, and K. Matsumoto. An Empirical Study of Fault Prediction with Code Clone Metrics. In *Proceedings of the Joint Conference of International Workshop on Software Measurement and International Conference on Software Process and Product Measurement* (*IWSM/Mensura 2011*), pages 55–61, Nov. 2011.

[69] T. Kamiya. Classifying Code Clones with Configuration. In *Proceedings of the 4th International Workshop on Software Clones* (*IWSC 2010*), pages 75–76, May 2010.

[70] T. Kamiya. Conte*t Clones or Re-thinking Clone on a Call Graph. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 74–75, June 2012.

[71] T. Kamiya, Y. Higo, and N. Yoshida. Evolving and Hot Topics on Code Clone Detection Techniques. *Journal of Computer Software*, 28(3):28–42, Aug. 2011. (in Japanese).

[72] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[73] C. Kapser and M. W. Godfrey. Aiding Comprehension of Cloning Through Categorization. In *Proceedings of the 7th International Workshop on Principles of Software Evolution* (*IWPSE 2004*), pages 85–94, Sep. 2004.

[74] C. J. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Enginieering*, 13(6):645–692, Dec. 2008.

[75] S. Kawaguchi, T. Yamashita, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A Tool for Automatic Code Clone Detection in the IDE. In *Proceedings of the 16th Working Conference on Reverse Engineering* (*WCRE 2009*), pages 313–314, Oct. 2009.

[76] M. Kim, L. Bergman, T. Lau, and D. Notokin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering* (*ISESE 2004*), pages 83–92, Aug. 2004.

[77] M. Kim and D. Notokin. Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones. In *Proceedings of the 2nd International Workshop on Mining Software Repositories* (*MSR 2005*), pages 1–5, May 2005.

[78] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering* (*ESEC/FSE 2005*), pages 187–196, Sep. 2005.

[79] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya. Detection of Type-1 and Type-2 Code Clones Using Textual Analysis and Metrics.

In *Proceedings of 2010 International Conference on Recent Trends in Information, Telecommunication and Computing* (*ITC 2010*), pages 241–243, Mar. 2010.

[80] R. Komondoor and S. Horwitz. Semantics-Preserving Procedure Extraction. In *Proceedings of the 27th Symposium on Principles of Programming Language* (*POPL 2000*), pages 155–169, Jan. 2000.

[81] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis* (*SAS 2001*), pages 40–56, July 2001.

[82] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 4th Working Conference on Reverse Engineering* (*WCRE 1997*), pages 44–54, Oct. 1997.

[83] R. Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar*, July 2006.

[84] R. Koschke. Frontiers on Software Clone Management. In *Proceedings of the Frontiers of Software Maintenance in the 24th International Conference on Software Maintenance* (*ICSM 2008*), pages 119–128, Oct. 2008.

[85] R. Koschke, R. Falke, and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the 13th Working Conference on Reverse Engineering* (*WCRE 2006*), pages 253–262, Oct. 2006.

[86] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings the 8th Working conference on Reverse Engineering* (*WCRE 2001*), pages 301–309, Oct. 2001.

[87] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proceedings of the 14th Working Conference on Reverse Engineering* (*WCRE 2007*), pages 170–178, Oct. 2007.

[88] J. Krinke. Is Cloned Code More Stable than Non-cloned Code? In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2008*), pages 57–66, Sep. 2008.

[89] J. Krinke. Is Cloned Code older than Non-Cloned Code? In *Proceedings of the 5th International Workshop on Software Clones* (*IWSC 2011*), pages 28–33, May 2011.

[90] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. P. Hudepohl. Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In *Proceedings of the 13th International Conference on Software Maintenance* (*ICSM 1997*), pages 314–321, Oct. 1997.

[91] F. Lanubile and T. Mallardo. Finding Function Clones in Web Applications. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering* (*CSMR 2003*), pages 379–386, Mar. 2003.

[92] M. Lanza and S. Ducasse. Polymetric Views - A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sep. 2003.

[93] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), May 2006.

[94] S. Lee, G. Bae, H. S. Chae, D. Bae, and Y. R. Kwon. Automated Scheduling for Clone-based Refactoring Using a Competent GA. *Software: Practice and Experience*, 41(5):521–550, Apr. 2010.

[95] S. Lee and I. Jeong. SDD: High Performance Code Clone Detection System for Large Scale Source Code. In *Proceedings of the Object Oriented Programming Systems Languages and Applications Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA 2005*), pages 140–141, Oct. 2005.

[96] H. Li and S. Thompson. Clone Detection and Removal for Erlang/OTP within a Refactoring Environment. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (*PEPM 2009*), pages 169–178, Jan. 2009.

[97] J. Li and M. D. Ernst. CBCD: Cloned Buggy Code Detector. In *Proceedings of the 34th International Conference on Software Engineering* (*ICSE 2012*), pages 310–320, June 2012.

[98] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transcations on Software Engineering*, 32(3):176–192, Mar. 2006.

[99] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Dis-

tributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering* (*ICSE 2007*), May 2007.

[100] A. Lozano and M. Wermelinger. Evaluating the Harmfulness of Cloning: A Change Based Experiment. In *Proceedings of the 4th International Workshop on Mining Software Repositories* (*MSR 2007*), May 2007.

[101] A. Lozano and M. Wermelinger. Assessing the Effect of Clones on Changeability. In *Proceedings of the 24th International Conference on Software Maintenance* (*ICSM 2008*), pages 227–236, Sep. 2008.

[102] A. Lozano and M. Wermelinger. Tracking clones' imprint. In *Proceedings of the 4th International Workshop on Software Clones* (*IWSC 2010*), pages 65–72, May 2010.

[103] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the Relation between Changeability Decay and the Characteristics of Clones and Methods. In *Proceedings of the 23rd International Conference on Automated Software Engineering* (*ASE 2008*), pages 100–109, Sep. 2008.

[104] Lucia, D. Lo, L. Jiang, and A. Budi. Active Refinement of Clone Anomaly Reports. In *Proceedings of the 34th International Conference on Software Engineering* (*ICSE 2012*), pages 397–407, June 2012.

[105] K. Maeda. Syntax Sensitive and Language Independent Detection of Code Clones. *World Academy of Science, Engineering and Technology*, (36):350–354.

[106] K. Maruyama and T. Omori. A Security-Aware Refactoring Tool for Java Programs. In *Proceedings of the 4th Workshop on Refactoring Tools* (*WRT 2011*), pages 22–28, May 2011.

[107] T. Masai, N. Yoshida, M. Matsushita, and K. Inoue. Supporting Difference Extraction for Merging Similar Methods. In *IEICE Technical Report*, pages 45–50, May 2010. (in Japanese).

[108] MASU. avilable at <http://sourceforge.net/projects/masu/>.

[109] 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 12th International Conference on Software Maintenance* (*ICSM 1996*), pages 244–253, Nov. 1996.

[110] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[111] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[112] E. Merlo and T. Lavoie. Computing Structural Types of Clone Syntactic Blocks. In *Proceedings of the 16th Working Conference on Reverse Engineering* (*WCRE 2009*), pages 274–278, Oct. 2009.

[113] H. Miyazaki, Y. Higo, and K. Inoue. Increasing Code Clone Analysis Efficiency Using Itemset Mining. In *Technical Report of IEICE* (*SIGSS*), volume 108, pages 31–36, July 2008. (in Japanese).

[114] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (*SAC 2012*), pages 1227–1234, Mar. 2012.

[115] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proceedings of the 8th IEEE Internaitional Software Metrics Symposium* (*METRICS 2002*), pages 87–94, June 2002.

[116] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Folding Repeated Instructions for Improving Token-Based Code Clone Detection. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2012*), pages 64–73, Sep. 2012.

[117] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped Code Clone Detection with Lightweight Source Code Analysis. In *Proceedings of the 21st International Conference on Program Comprehension* (*ICPC 2013*), pages 93–102, May 2013.

[118] E. Murphy-Hill and A. P. Black. Breaking the Barriers to Successful Refactoring. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 421–430, May 2008.

[119] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Clone Management for Evoluving Software. *IEEE Transactions on Software Engineering*, 38(5):1008–1026, Sep. 2012.

[120] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. ClemanX: Incremental Clone Detection Tool for Evoluving Software. In *Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 437–438, June 2009.

[121] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware Configuration Management. In *Proceedings of the 24th International Conference on Automated Software Engineering* (*ASE 2009*), pages 123–134, Nov. 2009.

[122] NiCad3 Clone Detector. available at <http://www.txl.ca/nicaddownload.html>.

[123] T. Omori, K. Maruyama, S. Hayashi, and A. Sawada. A Literature Review on Software Evolution Research. *Computer Software*, 29(3):3–28, Aug. 2012. (in Japanese).

[124] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois, 1992.

[125] J. Ossher, H. Sajnani, and C. Lopes. File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly. In *Proceedings of the 27th International Conference on Software Maintenance* (*ICSM 2011*), pages 283–292, Sep. 2011.

[126] K. J. Ottenstein. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *ACM SIGCSE Bulletin*, 8(4):30–41, Dec. 1976.

[127] J. Pearl. *Probablistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[128] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that Smell? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories* (*MSR 2010*), pages 72–81, May 2010.

[129] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, July 2013.

[130] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-Wide Code Duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering* (*WCRE 2004*), pages 100–109, Nov. 2004.

[131] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. *School of Computing Technical Report 2007-541, Queen's University*, 115, 2007.

[132] C. K. Roy and J. R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clons Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the 16th International Conference on Program Comprehension* (*ICPC 2008*), pages 172–181, June 2008.

[133] C. K. Roy and J. R. Cordy. A Mutation / Injection-based Automatic Framework for Ecaluating Code Clone Detection Tools. In *Proceedings of the International Conference on Software Testing, Verification, and Validation Workshops* (*ICSTW 2009*), pages 157–166, Apr. 2009.

[134] F. V. Rysselberghe and S. Demeyer. Evaluating Clone Detection Techniques from a Refactoring Perspective. In *Proceedings of the 19th International Conference on Automated Software Engineering* (*ASE 2004*), pages 336–339, Sep. 2004.

[135] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In *Proceedings of the 10th Working Conference on Source Code Analysis and Manipulation* (*SCAM 2010*), pages 87–96, Sep. 2010.

[136] R. K. Saha, C. K. Roy, and K. A. Schneider. Visualizing the Evolution of Code Clones. In *Proceedings of the 5th International Workshop on Software Clones* (*IWSC 2011*), pages 71–72, May 2011.

[137] T. Sasaki, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. A Code Clone Information Supplement Tool to Support Program Change. *IEICE Journal*, J87-D-I(9):868–870, Sep. 2004. (in Japanese).

[138] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding File Clones in FreeBSD Ports Collection. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories* (*MSR 2010*), pages 102–105, May 2010.

[139] Scorpio. available at <http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio>.

[140] Simian. available at <http://www.redhillconsulting.com.au/products/simian/>.

[141] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[142] R. Tairas and J. Gray. An Information Retrieval Process to Aid in the Analysis of Code Clones. *Empirical Software Engineering*, 14(1):33–56, Feb. 2009.

[143] 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):1–34, Feb. 2010.

[144] R. Tiarks, R.Koshcke, and R. Falke. An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2009*), pages 67–76, Sep. 2009.

[145] M. Toomim, A. Begel, and S. L. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of the 2004 Symposium on Visual Languages and Human Centric Computing* (*VL-HCC 2004*), pages 173–180, Sep. 2004.

[146] N. Tsantalis and A. Chatzigeorgiou. Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods. *Journal of Systems and Software*, 84(10):1757–1782, Oct. 2011.

[147] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *Proceedings of the 8th IEEE Internaitional Software Metrics Symposium* (*METRICS 2002*), pages 67–76, June 2002.

[148] R. D. Venkatasubramanyam, H. K. Singh, and K. Ravikanth. A Method for Proactive Moderation of Code Clones in IDEs. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 62–66, June 2012.

[149] H. V. Vliet. *Software Engineering: Principles and Practices*. John Wiley & Sons, 2008.

[150] W. Wang and M. W. Godfrey. We Have All of the Clones, Now What? Toward Integrating Clone Analysis into Software Quality Assessment. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 88–89, June 2012.

[151] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can I Clone This Piece of Code Here? In *Proceedings of the 27th International Conference on Automated Software Engineering* (*ASE 2012*), pages 170–179, Sep. 2012.

[152] M. Wattenberg. Arc Diagrams: Visualizing Structure in String. In *Proceedings of the IEEE Symposium on Information Visualization* (*InfoVis 2002*), pages 110–116, Oct. 2002.

[153] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Enginieering* (*ICSE 1981*), pages 439–449, Mar. 1981.

[154] R. Wettel and R. Marinescu. Archeology of Code Duplication: Recovering Duplication Chains From Small Duplication Fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (*SYNASC 2005*), Sep. 2005.

[155] M. D. Wit, A. Zaidman, and A. V. Deursen. Managing Code Clones Using Dynamic Change Tracking and Resolution. In *Proceedings of the 24th International Conference on Software Maintenance* (*ICSM 2009*), pages 169–178, Sep. 2009.

[156] Z. Xing, Y. Xue, and S. Jarzabek. CloneDifferentiator: Analyzing Clones by Differentiation. In *Proceedings of the 26th International Conference on Automated Software Engineering* (*ASE 2011*), pages 576–579, Nov. 2011.

[157] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Industrial Application of Clone Change Management System. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 67–71, June 2012.

[158] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Filtering Clones for Individual User Based on Machine Leraning Analysis. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 76–77, June 2012.

[159] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. A Method for Identifying Useful Code Clones with Machine Learning Techniques Based on Similarity Between Clones. *IPSJ Journal*, 54(2):807–819, Feb. 2013. (in Japanese).

[160] N. Yoshida, T. Hattori, Y. Hayase, and K. Inoue. Retrieving Similar Code Fragments Based on Synonymous Word Identification. *IPSJ Journal*, 50(5):1506–1509, May 2009. (in Japanese).

[161] N. Yoshida, T. Ishio, M. Katsushita, and K. Inoue. Retrieving Similar Code Fragments based on Identifier Similarity for Defect Detection. In *Proceedings of International Workshop on Defects in Large Software* (*DEFECTS 2008*), pages 41–42, July 2008.

[162] K. Yoshimura and R. Mibe. Visualizing Code Clone Outbreak: An Industrial Case Study. In *Proceedings of the 6th International Workshop on Software Clones* (*IWSC 2012*), pages 96–97, June 2012.

[163] G. Zhang, X. Peng, and Z. Xing W. Zhao. Cloning Practices: Why Developers Clone and What can be Changed. In *Proceedings of the 28th International Conference on Software Maintenance* (*ICSM 2012*), pages 285–294, Sep. 2012.

[164] Y. Zhang, H. A. Basit, S. Jarzabek, D. Anh, and M. Low. Query-based Filtering and Graphical View Generation for Clone Analysis. In *Proceedings of the 23rd International Conference on Software Maintenance* (*ICSM 2008*), pages 376–385, Oct. 2008.

[165] M. F. Zibran and C. K. Roy. A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proceedings of the 11th International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2011*), pages 105–114, Sep. 2011.