Detection and evolution analysis of code clones for efficient management of large-scale software systems

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2015

Eunjong CHOI

Abstract

In recent decades, large-scale software systems have become mainstream. Such software systems have complicated the maintenance process by increasing efforts such as inspection and understanding of the existing source code. Therefore, to maintain these systems, a great deal of work and time are necessary. To alleviate this problem, this research focus on a well-known factor hindering the software maintenance task, a code clone (i.e., a code fragment that has other code fragments identical or similar to it in the source code). It is widely believed that code clones complicate software maintenance. For example, when changes to code clones in a clone set (i.e., a set of code clones that are identical or similar to each other) are inconsistent, the developer needs to identify inconsistently changed code clones and apply consistent changes to them.

Thus far, many tools and techniques have been proposed for supporting the detection and management of code clones. However, most are insufficient for supporting code clone related tasks during the software maintenance process for large-scale software systems. To resolve this problem, this study attempts to solve two important problems that code clones face. That is, “Which type of normalization dose make code clones to detected with high speed from large-scale software systems? ” and “Which supports are necessary for more widely used tools that support clone refactoring? ”.

To solve the first problem, this research proposes six approaches for detecting code clones with preprocessing input source files using different degrees of normalizations (e.g., the removal of white spaces, tokenization, and the regularization of identifiers). More precisely, each type of normalization is applied to the input source files, and equivalence class partitioning of the files is then conducted during the preprocessing. Code clones are then detected from a set of files that are representatives of each equivalence class using a token-based code clone detection tool called CCFinder. The proposed approaches can be categorized into two types, an approach with non-normalization and approaches with normalization. The former type is the detection of only identical files without normalization, whereas the latter category is the detection of identical files with different degrees of normalization

such as the removal of all lines containing macros in C program. From a case study, it was observed that the detection times of the proposed approaches are at least two-times faster than an approach that uses only CCFinder. It was also found that the approach with non-normalization is the fastest of the proposed approaches for many cases.

To resolve the second problem, this research presents an investigation of clone refactoring (i.e., merging code clones into a new method) carried out during the development of open source software systems for promoting the development of refactoring tools that can be more widely utilized. In this investigation, it was identified that a "Replace Method with Method Object" is the most frequently used refactoring pattern for clone refactoring. Moreover, this research discovered that merged code clone token sequences and the differences in the token sequence lengths vary for each refactoring pattern.

List of Publications

Major Publications

[1-1] Eunjong Choi, Norihiro Yoshida, Yoshiki Higo, Katsuro Inoue: *Proposing and Evaluating Clone Detection Approaches with Preprocessing Input Source Files*. IEICE Transactions on Information and Systems, Vol.E98-D, No.2, February 2015 (to appear).

[1-2] Eunjong Choi, Kenji Fujiwara, Norihiro Yoshida, Shinpei Hayashi: *A Survey of Refactoring Detection Techniques Based on Change History Analysis*. Computer Software, Vol.32, No.1, February 2015 (in Japanese) (to appear).

[1-3] Eunjong Choi, Norihiro Yoshida, Katsuro Inoue: *An Investigation into the Characteristics of Merged Code Clones during Software Evolution*. IEICE Transactions on Information and Systems, Vol.E97-D, No.5, pp.1244-1253, May 2014.

[1-4] Eunjong Choi, Norihiro Yoshida, Katsuro 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), pp.1-7, Rapperswil, Switzerland, June 2012.

Related Publications

[2-1] Manamu Sano, Eunjong Choi, Norihiro Yoshida, Yuki Yamanaka, Katsuro Inoue: *Supporting Clone Analysis with Tag Cloud Visualization*. in Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices (InnoSWDev 2014), pp.94-99, Hong Kong, China, November 2014.

[2-2] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue: *A High Speed Function Clone Detection Based on Information Retrieval Techniques*.

IPSJ Journal, Vol.55, No.10, pp.2245-2255, October 2014 (in Japanese).

[2-3] Akira Goto, Norihiro Yoshida, Masakazu Ioka, Eunjong Choi, Katsuro Inoue: *Method Differentiator Using Slice-based Cohesion Metrics*. in Proceedings of the 12th Annual International Conference Companion on Aspect-oriented Software Development (AOSD 2013 Companion), pp.11-14, Fukuoka, Japan, March 2013.

[2-4] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, Tateki Sano: *Applying Clone Change Notification System into an Industrial Development Process*. in Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC 2013), pp.199-206, San Francisco, California, USA, March 2013.

[2-5] Akira Goto, Norihiro Yoshida, Masakazu Ioka, Eunjong Choi, Katsuro Inoue: *How to Extract Differences from Similar Programs? A Cohesion Metric Approach*. in Proceedings of the 7th International Workshop on Software Clones (IWSC 2013), pp.23-29, San Francisco, California, USA, March 2013.

[2-6] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, Tateki Sano: *A Development of Clone Change Management System and Its Application to Actual Project*. IPSJ Journal, Vol.54, No.2, pp.883-893, February 2013 (in Japanese).

[2-7] Yuki Yamanaka, Eunjong Choi, Norihiro Yoshida, Katsuro Inoue, Tateki Sano: *Industrial Application of Clone Change Management System*. in Proceedings of the 6th International Workshop on Software Clones (IWSC 2012), pp.67-71, Zurich, Switzerland, June 2012.

[2-8] Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, Tateki Sano: *Extracting Code Clones for Refactoring Using Combinations of Clone Metrics*. in Proceedings of the 5th International Workshop on Software Clones (IWSC 2011), pp.7-13, Waikiki, Hawaii, USA, May 2011.

Acknowledgement

First of all, I would like to express my most sincere gratitude to my respected supervisor Katsuro Inoue. Without his warm supports and valuable comments regarding my research, this study would not have been possible. I feel extremely happy and blessed for the opportunity to have been supervised by him and have been inspired by his enthusiasm and integral view toward research.

I would especially like to express my gratitude to Professors Toshimitsu Masuzawa, and Shinji Kusumoto for their valuable comments and helpful suggestions regarding this thesis. I would also like to acknowledge the guidance of Professors Kenichi Hagihara and Yasushi Yagi.

In addition, I would like to express my gratitude to the professors and staff members of Inoue laboratory. I appreciate Associate Professor Makoto Matsushita and Assistant Professor Takashi Ishio for their kind consideration. I also appreciate Specially Appointed Professor Shusuke Haruna and Specially Appointed Associate Professor Yukio Mohri for their kind support. Moreover, I would like to express my thanks to Specially Appointed Assistant Professors Kula Raula Gaikovina and Ali Ouni for their helpful comments and suggestions regarding this research. In particular, I owe many thanks to Ms. Mizuho Karube for her continual support and understanding.

I am grateful to have had the chance to conduct the research with such great people. I really appreciate Assistant Professor Yoshiki Higo at Osaka University for his insightful encouragements and comments regarding my research. I would also like to thank Assistant Professor Hayashi Shinpei at Tokyo Institute of Technology for his valuable comments and gentle advices. In addition, I am grateful to Mr. Kenji Fujiwara at Nara Institute of Science and Technology for his friendly and practical support. I would also like to acknowledge my appreciation to Mr. Tateki Sano at NEC Corporation for his support. I would additionally like to thank Dr. Coen De Roover and Mr. Reinout Stevens at Vrije Universiteit Brussel, Belgium for their warm supports and advice. I also want to express my thanks to Mr. Akira Goto at NS Solutions Corporation for his diligent work and kind support and to Mr. Yuki Yamanaka at Hitachi, Ltd for his hard work and heartful kindness.

I am particularly appreciative of Assistant Professor Zhenchang Xing at Nanyang Technological University, and Dr. Yinxing Xue and Associate Professor Stan Jarzabek at the National University of Singapore for their warm welcome and kindness during my stay in Singapore as an international exchange student. In particular, I am very grateful to have had the chance to conduct research with Professor Zhenchang Xing.

I also would like to thank Professor Daniel German at the University of Victoria, Canada for his valuable comments regarding my research. I also am very grateful to Associate Professor Toshihiro Kamiya at Future University Hakodate for his great work on code clones, particularly the development of CCFinder. I would like to thank Assistant Professor Marouane Kessentini at the University of Michigan, USA for his helpful advice and comments. I would also like to express my appreciation to Assistant Professor Hideki Hata at Nara Institute of Science and Technology for his warm advice. Thanks also to the anonymous reviewers of my papers for their valuable suggestions and comments. Moreover, I would like to particularly thank the great researchers, in the software engineering community. Although I cannot name all of them here owing to a lack of space, their significan work has consistently thrilled me.

I am grateful to my ex-colleagues Mr. Masahiro Uchida and Mr. Sahichi Yamanaka in JN System Partners Co., Ltd for their kindness and support. I would also like to express my thanks to the members of the Osaka-Yodogawa Rotary Club, particularly Ms. Toshiko Nakatsu and the Nishimura International Scholarship Foundation for their economic support and warm kindness. In addition, I would like to express my thanks to my friends in Japan and Korea, particularly Jaejong Li, Naewook Jung, Minji Kim, Semi Kim, Sunyoung Park, Mijung Park, Minyong Jeong, Aya Suzuki, Yuko Muko, Junya Nakamura, and students of IT Spiral, 2012 class. Thanks are also due to my many friends in the Department of Computer Science, especially Mr. Yonghwan Kim in the Masuzawa laboratory and the students in the Inoue and Kusumoto laboratories.

I would also like to express my gratefulness to my loving family members and relatives for their continued support and for consistently being in my corner. In particularly, I want to thank my parents, Jaein Choi and Keumsoon Park, for their warm support and encouragement and for always being on my side. Finally, I would like to send a warm and well-deserved thank to my husband, Associate Professor Norihiro Yoshida at Nagoya University, for his years of mentoring and guidance. None of these studies described in this thesis would have been impossible without his great support and continued patience. Thank you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a short introduction to this thesis. First, Section 1.1 introduces the motivation. Section 1.2 then illustrates the contributions. Finally, Section 1.3 describes an outline of this thesis.

1.1 Motivation

In recent decades, many companies have released new models in rushed intervals to achieve superiority over their rivals [16]. To do so, they have frequently reused robust parts of existing source code for new developments. As a consequence, a significant amount of code clones (i.e., code fragments that have other code fragments identical or similar to them in the source code) exist within software systems and between different releases of the models/versions of software systems.

These code clones complicate the maintenance process by increasing efforts, such as inspections and an understanding of the existing source code. For example, if code clones belonging to the same clone set (i.e., a set of code clones that are identical or similar to each other) have been inconsistently-changed, generating defects in the software system, the developer will need to find the inconsistently-changed code clones and propagate the required changes to them. To alleviate this problem, code clones should be documented and properly maintained (e.g., merging code clones into a function/method).

There are a multitude of techniques and implemented tools for automatically detecting code clones and supporting their management [21, 44, 78]. Using a code clone management tool, a developer can easily apply consistent changes to code clones [21] or conduct clone refactoring (i.e., merging code clones into a single function/method) [44]. Moreover, when code clones have been inconsistently changed, the developer can readily identify inconsistently-changed code clones us-

ing a code clone detection tool [46, 62]. Such tools help developers effectively maintain their software systems.

However, most of these techniques and tools are insufficient for large-scale software systems. With respect to the detection of code clones, most of them take a long time when the size of the input source code is large. For example, a token-based code clone detection tool called CCFinder takes about 40 days on a single PC-based workstation to detect code clones from 400 million lines of code [63]. Currently, the tools for code clone management are commonly underused. In particular, although clone refactoring is a promising approach, the tools supporting clone refactoring are underused compared to refactoring tools (e.g., Eclipse's refactoring features) that were not intended to support clone refactoring. To facilitate these problems, this study aims to solve the following two Research Questions (RQs).

RQ1: Which type of normalization dose make code clones to detected with high speed from large-scale software systems?

RQ2: Which supports are necessary for more widely used tools that support clone refactoring?

In this thesis, a new approach for detecting code clones in large-scale software systems and an investigation into clone refactoring during the software evolution are presented to answer RQ1 and RQ2, respectively.

1.2 Contributions of This Thesis

The contributions of this thesis are as follows:

- First, six approaches for detecting code clones from different release models/versions are presented and implemented using preprocessing input source files with CCFinder [49], a token-based code clone detection tool. During the preprocessing of the proposed approaches, the input source file is normalized to the different degrees of the program elements. For example, one of the proposed approaches normalizes each input source file by removing all lines containing only comments, and comments and white spaces before and after comments. Different degrees of normalization lead to different granularities of the source code to be detect as code clones.

- Second, for a set of different released models/versions of various software systems, it was found that the proposed with a preprocessing of the input source files detect code clones faster than an approach that uses only

CCFinder. It was also found that any normalization takes significant amount of time during the preprocessing and post-processing and is unable to reduce the total detection time in many cases.

- Third, an approach using a code clone identification technique called undirected similarity (usim) and a refactoring detection tool called Ref-Finder to investigate instances of clone refactoring (i.e., the merging of a set of various code clone types into a single function or a method using refactoring patterns) are presented.

- Finally, it was discovered that the *Extract Method (EM)* and *Replace Method with Method Object (RMMO)* patterns are used the most when developers conduct clone refactoring. Moreover, it was found that large token differences exist between merged code clones when the *RMMO* and *EM* patterns are used on pairs of code clones.

1.3 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2: Related works
The thesis begins with a review of studies to help deeply understand the problemns involved with this thesis. In this chapter, studies on automatic code clone detection and analyzing code clones during the software evolution are first reviewed. Moreover, studies on the automatic detection of refactoring instances from the source code history and analyzing the refactoring instances during the software evolution will also be introduced.

Chapter 3: Proposing and Evaluating Clone Detection Approaches
This chapter describes six code clone detection approaches with a preprocessing of the input source files. These approaches extend CCFinder by adapting the preprocessing and post-processing to detect code clones effectively from different model/versions released. Moreover, a comparison of the proposed approaches and an "approach that uses only CCFinder" will be presented. The goal of this study is to investigate how the normalizations impact the code clone detection for quickly detecting code clones from large-scale software systems.

Chapter 4: Investigating Merged Code Clones during Software Evolution
This chapter presents an investigation into merged code clones during the evolution of three Open Sorce Software (OSS) systems. This investigation

was conducted using a refactoring detection tool and a token similarity metric. The goal of this investigation was to uncover clues that can contribute to the development of more widely used tools for clone refactoring.

Chapter 5: Conclusion and FutureWork

The final chapter concludes the thesis by summarizing the main findings, and providing directions for future research.

Chapter 2

Related work

This chapter describes previous works related to code clones and refactoring to help with a deeper understanding of this thesis. Section 2.1 reviews existing studies on code clones, and Section 2.2 describes state-of-the-art studies on refactoring detection and analysis during the software evolution.

2.1 Code Clones

A code clone is a code fragment that has other code fragments identical or similar to it in the source code. A clone set is a set of code clones that are identical or similar to each other. Code clones can be categorized into the following four types based on the textual (Type-1, Type-2, and Type-3) and semantic (Type-4) similarities between the pairs of code clones [82]:

Type-1: Identical code fragments except for variations in whitespace, layout, and comments.

Type-2: Syntactically identical fragments except for variations in the identifiers, literals, types, whitespace, layout, and comments.

Type-3: Copied fragments with further modifications such as changed, added, or removed statements, in addition to variations in the identifiers, literals, types, whitespace, layout, and comments.

Type-4: Two or more code fragments that conduct the same computations but are implemented through different syntactic variants.

The dissimilarity and abstraction in the definition of code clones increase from Type-1 to Type-4.

Code clones are usually generated through the copying and pasting of existing code fragments with or without modifications. Such "copy and paste" tasks are the result of programmer's mental macros or the inexpressiveness of the programming languages used. Meanwhile, the avoidance of a new defects, the reuse of a template/design, the programmer's lack of knowledge of the domain/product, and the development resources also trigger such tasks [50, 81].

Details of previous code clone related studies are introduced in the following subsections. Existing studies on code clone detection and analysis are reviewed in Sections 2.1.1 and 2.1.2, respectively.

2.1.1 Code Clone Detection

Thus far, numerous code clone detection techniques and their implemented tools have been proposed, which can be roughly classified into the following categories:

- String-based techniques [24, 22]
- Token-based techniques [5, 4, 49, 62, 70, 63, 101]
- Tree-based techniques [10, 47, 58, 57]
- Program dependence graph-based techniques [30, 56, 59]
- Memory-based techniques [52]
- Hybrid techniques [7, 45, 80, 79]

Hereafter, techniques and tools from the abovementioned categories are introduced.

String-based techniques

The techniques in this category conduct string-by-string comparisons of the input source code and detect similar sequences of the strings as code clones. Ducasse et al. proposed a language-independent clone detection technique based on line similarities, and implemented a tool called Duploc [24]. After removing comments and white spaces from the input source code, Duploc detects Type-1 and Type-2 code clones using dynamic pattern matching. It also provides a scatter plot that visualizes the results of code clones for supporting analysis tasks.

Ducasse et al. also presented string-matching techniques with six different degrees of code normalization (i.e., replacing certain elements of a program

with generic placeholders with the aim of removing only nonessential information), and then measured their impact on the quality of the code clone detection [22]. First, noise (e.g., white spaces, tabulations, and comments) is eliminated from the input source code, and a different degree of code normalization (e.g., replacing identifiers, labels, and basic numeric types with a special token) is applied to the input source code. Next, code clones are detected using line-by-line string matching. In a case study, the authors compared different proposed techniques and confirmed that more complicated normalization results in a decrease in the recall and precision.

Token-based techniques

These techniques tokenize the input source code through normalization, and then compare the token sequences of lines to detect any code clones. Baker developed a token-based code clone detection tool called Dup [5, 4]. First, Dup tokenizes the input source code, and the tokens of the identifiers and constants are then replaced with placeholders. Next, Dup extracts the pairs of longest matches using a suffix tree algorithm [66]. It detects pairs of exactly the same code fragments or parameterized matched strings (i.e., a pair of code fragments in which each identifier or constant in one code fragment is consistently changed into another identifier and constant in another code fragment), with the exception of comments, blank lines, and white spaces, as code clones.

Kamiya et al. developed a tool called CCFinder that detects Type-1 and Type-2 code clones based on token similarities [49]. The steps used to detect code clones from the input source code are as follows: First, each line of the input source files is divided into tokens corresponding to a lexical rule of the programming language. The tokens of all files are concatenated into a single token sequence. In addition, the white spaces (including '\n' and '\t') and comments between tokens are removed from the token sequence. The token sequence is then transformed, (i.e., tokens are added, removed, or changed based on the transformation rules) and, each identifier related to the types, variables, and constants is replaced with a special token for detecting Type-2 code clones. Next, from all of the substrings in the transformed token sequence, equivalent code fragments are detected as code clones. A suffix-tree matching algorithm [38] is used to compute the matching, in which the clone location information is represented as a tree with sharing nodes for leading identical subsequences, and the clone detection is conducted by searching the leading nodes on the tree. Finally, each location of a code clone is converted into line numbers of the original input files. D-CCFinder extends CCFinder to detect code clones from large-scaled software systems [63].

To detect code cones at high speed, it partitions the code clone detection into smaller pieces for distribution in very large software systems.

CP-Miner, proposed by Li et al., detects code clones and bugs that are induced by code clones using a frequent subsequence mining (i.e., an association analysis technique used to discover frequent subsequences in a collection of sequences) technique [62]. CP-Miner tokenizes the input source code with replacement of identifier names such as variables, functions, and types with a special token. After converting a tokenized source code into hash values, CP-Miner detects code clones using an enhanced algorithm of closed sequential pattern mining [103] from sequences of the hash values.

Murakami et al. proposed a code clone detection technique using Smith-Waterman and the longest common subsequence (LCS) algorithm to detect gapped code clones [70]. This technique normalizes (i.e., user-defined identifiers are replaced with specific tokens) a tokenized source code and then calculates a hash value for every statement. After identifying similar hash sequences using the tailored Smith-Waterman algorithm [86], it identifies gapped code clones using the LCS algorithm.

Yamanaka et al. presented a lightweight technique aimed at detecting function clones, using information retrieval techniques [101]. A feature vector is generated for each function of the input source code using the Term Frequency Inverse Document Frequency (TF-IDF) and reserved keywords, and clustering of the generated vectors is then conducted by means of locality sensitive hashing (LSH) [31]. Finally, clones are detected based on the similarities between each pair of feature vectors.

Tree-based techniques

In these techniques, input source code is represented as a tree structure, and code clones are then detected by identifying isomorphic subtrees. CloneDR, developed by Baxter et al., detects code clones using abstract syntax tree [10]. CloneDR partitions subtrees of abstract syntax tree using hash values, and then detects code clones by comparing subtrees that have the same hash values.

Deckard, developed by Jiang et al., computes certain characteristic vectors from the abstract syntax trees [47]. These vectors are clustered using LSH, and subtrees with vectors in a cluster are then detected as code clones.

Koschke et al. proposed a technique that uses a suffix tree to identify code clones from an abstract syntax tree [58]. This approach parses the input source code and generates an abstract syntax tree. It then serializes the abstract syntax tree and the abstract syntax tree input into suffix tree. It detects

code clones of syntactic units by decomposing the identified code clones into complete syntactic units.

Koschke proposed a technique for detecting code clones between a subject system and a corpus (i.e., a set of software systems) aimed at identifying potential license violations [57]. This technique generates a suffix tree for a subject system and then compares every file in the corpus with the generated suffix tree using an MD5 hash function.

Program dependence graph-based techniques

These techniques transform input source code into a program dependence graph [26], a representation of a program that represents only the control and data dependency between statements and predicates, and then identifies isomorphic program dependence graph subgraphs to detect code clones. Komondoor and Horwitz presented a program dependence graph-based code clone detection technique and implemented a tool called PDG-DUP [56], which partitions all program dependence graph nodes into equivalence classes and then finds isomorphic program dependence graph subgraphs using (backward) program slicing. It then reports the isomorphic program dependence graph subgraphs as code clones.

Krinke proposed a code clone detection technique based on fine-grained program dependence graphs [59]. The technique detects similar syntactic structures (Type-1, Type-2, and Type-3 code clones) as well as similar semantics (Type-4 code clones) by comparing the subgraphs of the program dependence graphs.

Gabel et al. presented a technique for detecting code clones including semantic code clones (i.e., semantically similar code fragments) [30]. This technique maps carefully selected program dependence graph subgraphs to their related structured syntax, and then detects clones using Deckard's vector generation and LSH-based clustering.

Memory-based techniques

Kim et al. proposed an approach for detecting code clones based on abstract memory, and implemented a tool called MeCC [52]. MeCC computes the abstract memory states from input programs using a static analysis, and then detects code clones by comparing the abstract memory states. It is able to detect Type-4 clones such as pairs of statement-reordered code clones.

Hybrid-based techniques

A hybrid approach (e.g., hybrid code representation/techniques) is also used

in such techniques to detect code clones. Roy and Cordy developed a text-based hybrid clone detection tool called NiCad [80, 79]. NiCad identifies and normalizes potential clones using flexible pretty-printing. It then proceeds to normalize the input source code and compare text lines of potential code clones using the LCS algorithm.

Basit and Jarzabek developed a tool called Clone Miner for detecting structural clones (i.e., larger granularity code clones such as similar files or directories) [7]. Clone Miner detects structural clones using frequent itemset mining [37] from the repeated combinations of simple clones (i.e., fragments of duplicated code) detected by Repeated Tokens Finder(RTF) [8], a token-based code clone detection tool.

Finally, Hummel et al. proposed an incremental index-based code clone detection approach [45]. An index allows the lookup of all clones for a single file, and can be updated efficiently, when files are added, removed, or modified. This approach computes MD5 hash values from tokenized source code and then generates statement indices from the MD5 hash values. It detects code clones by retrieving the index from the databases.

2.1.2 Analysis of Clone Evolution

Many studies have been conducted on identifying the possible impact of code clones during the software evolution. Mandal et al. investigated the clone evolution history to identify candidates for clone refactoring [65, 69]. They analyzed the clone evolution from 13 subject systems using association rules [65]. Their analysis was conducted based on the idea that if code clones belonging to the same clone set are changed together, preserving their similarity during the software evolution, they can be important candidates for clone refactoring. Thus, the authors detected code clones from 13 subject systems using NiCad, and then mined for code clones following the similarity-preserving change patterns using association rules. They found that, on average, 7.04% of the code clones follow similarity-preserving change patterns, and that more than half of them are method-level clones. However, they also found code clones that have changed with non-cloned code or code clones belonging to other clone sets, which should not be removed through clone refactoring but should be tracked to consistently update them in the future. Thus, to find candidates for tracking, they also mined code clones from six subject systems, and identified candidates for tracking at an overall rate of 10.27% [69].

Thus far, several studies have investigated clone removal during the software evolution [11, 32, 105, 106]. Göde investigated clone removal aimed at gaining insight to improving clone detection and refactoring tools. In this study, Type-1 and

Type-2 code clones were detected from four subject systems, and then manually investigated to determine whether such code clones were deliberately removed. This investigation revealed that method extraction is the most frequently used refactoring pattern to eliminate code clones, and that most of the removed code clones are contained within the same or closely related files. Bazrafshan and Koschke expanded on Göde's study by analyzing not only deliberately removed but also accidently removed code clones from seven subject systems. Their analysis found that code clones are accidently removed more frequently than they are deliberately removed, and that Type-2 code clones are more frequently removed than Type-1 code clones. Zibran et al. investigated the changes and removal patterns of Type-1, Type-2, and Type-3 code clones from a total of 228 releases of six OSS systems to characterize the patterns of clone change and removal during the software evolution. Their study was conducted based on code clones detected by NiCad and the clone evolution model constructed by gCad [85], a code clone genealogy extractor. Their study found that a few early releases of software systems experience more significant clone removal than later releases. It was also found that the most of the code clones underwent changes only once, before they were removed. Furthermore, they investigated 329 releases of nine software systems and additionally revealed that inconsistent changes were found to have dominated over consistent changes of code clones [106].

Wei and Godfrey analyzed clone refactoring from Linux kernel to understand the ratio of intentional clone refactoring [95]. Their study found that only a small fraction of code clones are intentionally refactored.

Several studies have analyzed OSS systems to understand how code clones evolve [55, 84]. Kim et al. initially presented a model of a clone genealogy (i.e., history of how each code clone in a clone set has changed with respect to other clones in the same set) by mapping code clones across multiple consecutive revisions from two Java OSS systems. Their study confirmed that code clones are either very volatile (i.e., disappear shortly after they are created), or hard to remove [55]. Saha et al. investigated the evolution of clones at the release level from 17 subject systems and found that many clone sets are alive and long-lived, either without any changes or with changes only in the identifier renaming [84].

Two investigations have aimed at identifying how developers create and maintain clones [6, 39]. Balint et al. analyzed three OSS systems to identify how developers copy code [6]. Their study related detected code clones with the developer information by using cvs annotate and found that an inconsistent rate of changes to code clones correlated with the number of developers. In addition, Harder analyzed the relationship between code clones and the number of developers involved in the creation and maintenance of the clones for five OSS systems [39]. He found that several differences, such as the rationale for cloning and the changes applied

to the clones, exist between single- and multiple-author clones.

Several studies have also analyzed the stability of cloned and non-cloned code during the software evolution [33]. Krinke analyzed the code clones and changes in five OSS systems and confirmed that cloned code is more stable than non-cloned code (i.e., a non-cloned code is more often added and deleted than a cloned code) [60]. Göde and Harder extended Krinke's study using a more detailed measurement and additionally found that Type-1 code clones are less stable than Type-2 and Type-3 code clones [33]. Mondal et al. also extended Krinke's study by analyzing code clones in 12 subject systems written in three different program languages, namely Java, C, and C# [68]. This study found that Type-1 and Type-2 clones are unstable, and that code clones in Java and C systems are not as stable as those in C# systems.

Finally, other studies have investigated the relationship between code clones and defect proneness [14, 77]. Nicolas et al. studied code clones at the release level to aim at investigating the effect of inconsistent changes on software quality. The authors analyzed code clones in three subject systems and found that only a small fraction of code clones induce software defects at the release level [14]. In addition, Rahman et al. analyzed four OSS systems using code clones to verify whether code clones are the source of a really bad smell [77]. Their study revealed that clones are much less defect-prone than non-cloned code.

2.2 Refactoring

Refactoring was defined by Fowler as a disciplined technique for restructuring an existing body of source code, altering its internal structure without changing its external behavior [28]. Refactoring leads to a reduction in the number of bugs and improved software quality and readability of the source code. The term "refactoring" was originally introduced by Opdyke in his Ph.D. thesis [72]. Fowler presented 72 refactoring patterns, including the *Extract Method* and *Rename Method*, along with the motivation and specific steps for conducting each refactoring pattern [28]. Hereafter, related works on refactoring detection and investigations into refactoring instances during the software evolution are summarized. In particular, these works will help readers understand the study, introduced in Section 4, in which merged code clones were analyzed during the software evolution based on detected refactoring instances.

The definition of refactoring detection used in this paper is as follows: Suppose that $(v_a, v_b)(0 \leq a < b \leq n)$ are a pair of versions extracted from a set of successive versions $(v_0, \ldots, v_{n-1}, v_n)$, and $C = \{c_0, \ldots, c_{m-1}, c_m\}$ is a set of changes occurring between versions v_a and v_b. If a subset (not empty) of set C subsumes

refactoring operations defined in the refactoring catalog suggested by Fowler, the subset is detected as a refactoring performed between versions v_a and v_b. In general, refactoring detection tools take a pair of versions (v_1, v_2) as input, and then outputs a list of refactoring performed between versions v_1 and v_2.

State-of-the-art studies for the automatic detection of refactoring instances from histories of source code changes can be categorized as follows [19]:

- Rule-based techniques [1, 2, 20, 35, 29, 54, 75, 92, 96, 97, 98, 100]
- Code clone-based techniques [15, 96]
- Metrics-based techniques [18, 64]
- Dynamic analysis-based techniques [89]
- Graph matching-based techniques [51, 90]
- Search-based techniques [13, 41, 42, 51, 64, 73, 93]

In this paper, only rule-based detection techniques are discussed, a complete discussion can be found in [19].

The rule used in these techniques is the criterion for determining whether refactoring was performed based on changes (e.g., additions, deletions, and movements) of the program elements (e.g., classes, methods, and parameters) and the similarity of elements between two versions. For example, to detect refactoring instances of the *Extract Method* pattern, a technique proposed by Prete et al. [54, 75] extracts program elements as facts and then computes the similarities of the facts between two versions. Finally, if the computed results match a predefined rules that states the "source code of a new method is extracted from a changed method in the old version, the new method calls the old method, and source code of the new and old methods are similar to each other" then the target source code is detected as a refactoring instance of the *Extract Method* pattern. Figure 2.1 summarizes the process of this type of detection.

The techniques proposed by Antoniol et al. and Advani et al. detect refactoring instances based on Fowler's definition of refactoring patterns [1, 2]. Antoniol et al. presented a technique for detecting refactoring instances at the class level (e.g., *Class Extraction* and *Class Split*) based on the predefined conditions to investigate the evolution of classes in Java software systems. Their technique extracts identifiers from each class, and then weights the extracted identifiers based on the TF-IDF (Term Frequency-Inverse Document Frequency) [3]. Next, it converts each class into a vector based on the weight of the class, and, finally, determines the refactoring instances according to the conditions based on changes (e.g., a newly

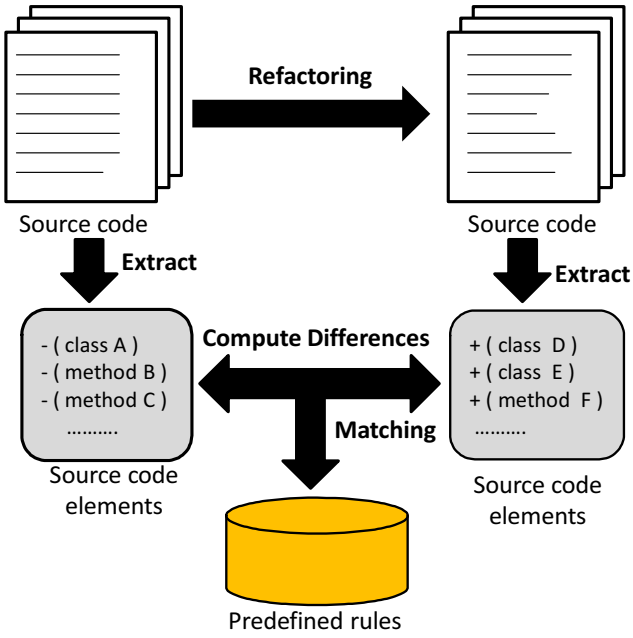


Figure 2.1: Overview of a rule-based refactoring detection technique

added class) in each class and the cosine of the angle between two vectors representing the classes. Furthermore, this approach was applied to 40 releases of *dnsjava* and identified the *Class Replacement*, *Merge and Split*, and *Factor Out* refactoring. Advani et al. developed a tool for detecting refactoring instances according to the predefined criteria aimed at investigating whether certain refactoring patterns are related [1]. This tool reports refactoring instances when predefined criteria for 15 refactoring patterns are matched with changes in the class entities (e.g., methods and fields). As a result of applying the tool to seven OSS systems, this study found that the *Rename Method*, *Rename Field*, *Move Method*, and *Move Field* patterns are frequently related with other refactoring patterns.

Görg and Weißgerber implemented a tool called REFVIS for detecting the refactoring instances based on changes (e.g., add, remove and unchanged) in the signatures of the classes and methods between two versions [35]. REFVIS also provides a feature that visualizes the detection results at the classes and methods levels. Weißgerber and Diehl presented a technique for detecting refactoring instances based on added, changed, or removed classes; interfaces; methods; and fields between two versions [96]. It then ranks the refactoring instances based on similarities in the source code between the two versions using CCFinder. This

technique is able to detect similar source codes as refactoring instances, whereas REFVIS only reports two exact source code as refactoring instances.

Xing and Stroulia's UMLDiff detects refactoring instances between two versions [97, 98, 100]. UMLDiff extracts logical facts such as the types, names, and modifiers of the packages and classes from two input program versions. It then computes their similarities based on their changes, additions, movements, and deletions. Finally, if the computed similarities are matched with queries representing refactoring patterns, it detects the target source code as a refactoring instance.

Prete et al. developed an Eclipse plug-in called Ref-Finder that detects refactoring instances of 63 refactoring patterns between two program versions based on the predefined rules [54, 75]. Ref-Finder extracts the code elements (e.g., packages, classes, and interfaces), structural dependencies (e.g., containment and overriding relationships), and the contents of the code elements (e.g., if-then-else control structures) as facts from two input program versions. It then computes the differences in facts between the two program versions. Finally, it determines the refactoring instances based on the predefined rules of the refactoring patterns [74]. Ref-Finder detects both atomic refactoring and complex refactoring using other atomic refactoring as a pre-requisite. Furthermore, it can detect more instances of refactoring patterns using code information such as a conditional branch and exception handling.

A technique proposed by Fujiwara et al. detects refactoring instances from multiple revisions [29]. This technique speeds up the refactoring detection by extracting code elements from each revision and matching them using Historage [40].

Rule-based techniques are also used to detect refactoring instances from the changes in histories of the components. In general, it is difficult to detect refactoring instances from changes in histories of the components because of backward compatibility (i.e., obsolete source codes coexist with their newer counterparts until they are no longer supported). To address this problem, Dig et al. and Taneja et al. presented techniques for detecting refactoring instances between two versions of components based on predefined rules [20][92]. Dig et al. developed an Eclipse plug-in called RefactoringCrawler [20], which identifies similar pairs of entities (e.g., methods and classes) in two versions of components using Shingles [17] to find refactoring candidates, and then analyzes references among the source code entities in each of the two versions of the components to detect real refactoring instances. Taneja et al. developed a tool called Refac Lib, which extracts similar entities from the source code from two versions of an Application Programming Interface (API) and then reports the refactoring instances based on a syntactic analysis, the similarities and size of the entities, and information regarding obsolete entities.

Next, several studies that have analyzed refactoring instances during the soft-

ware evolution for several different purposes are described. Note that only studies using automation techniques for detecting refactoring instances are introduced.

Several studies have aimed at finding the relationship between defects and refactoring [9, 36, 53]. Görg and Weißgerber investigated *jEdit* and *Tomcat* to find refactoring instances containing bugs [36]. After preprocessing the extracted data from CVS repositories, they identified refactoring instances of ADD/Remove Parameter and Rename Method patterns based on changes in the methods and classes between two versions [35]. They then checked whether these refactoring patterns were consistently applied to all related methods to identify incomplete refactoring instances. As a result, they found five candidate buggy methods.

Kim et al. investigated the development histories of three OSS systems, aiming at determining the relationships between API-level refactorings (i.e., *Rename* and *Move Package*/*Class*/*Method*, and *Method Signature Change*) and bug fixes [53]. They analyzed revisions containing API-level refactoring, revisions containing bug fixes, and bug-introduction changes in *Eclipse JDT*, *jEdit*, and *Columba*. They concluded that the number of bug fixes increases after refactoring, whereas the time taken to fix the bugs decreases after refactoring. Meanwhile, Bavota et al. reported that refactoring induces a small number of defects [9]. They analyzed the refactoring instances detected by Ref-Finder from three Java OSS systems to investigate to what extent refactoring induces bugs. As a result, they found that only 15% of the refactoring instances induced bug fixes and that several refactoring patterns including the *Pull Up Method* and *Extract Subclass* induce more bug fixes than others.

Some studies have investigated the actual practice of refactoring to better understand the process [87, 99]. Xing and Stroulia analyzed three pairs of released versions of Eclipse JDT using the UMLDiff algorithm to investigate the actual refactoring practice and suggested a direction for improving refactoring support tools [99]. In their study, they found that 70% of the structural changes were the result of refactoring, and that existing integrated development environments lack support for complex refactoring such as refactoring of the containment-hierarchy. Soares et al. analyzed the frequency of refactoring, program structures affected by refactoring, and the scope of refactoring from almost 41,000 software versions of five OSS systems [87]. From the identified refactoring based on behavior preservation between pairs of versions, they found that 27% of the changes during the software evolution were from refactoring. They also identified that most refactorings is low-level refactorings (i.e., only changing blocks of code within methods) or local refactorings (i.e., performed within a package).

Rachatasumrit and Kim investigated 14 pairs of versions of three Java OSS systems (*Apache JMeter*, *XML Security Library*, and *Apache ant*), aiming at determining the relationship between refactoring and regression testing [76]. This in-

vestigation was conducted based on refactoring instances detected by Ref-Finder and affected tests (i.e., a set of regression tests in the old version that are relevant to atomic changes) identified by FAULTTRACER [104], an automated change impact analysis tool. The results of this investigation revealed that 22% of the refactored methods and fields are covered by existing regression tests and that 38% of the affected tests involve refactoring. Furthermore, it was also found that 50% of the failed affected tests involve refactoring.

Finally, Tsantalis et al. studied Git repositories of three OSS systems, namely *JUnit*, *HTTPCore*, and *HTTPClient* [94]. They analyzed the refactoring histories, using refactoring detection rules suggested by Biegel et al. [15]. As a result, they observed that most types of the refactoring applied are conducted by specific developers, who usually have a key role in the management of the project. Moreover, they found a wide variety of reasons motivating the application of refactoring; for instance, the decomposition of methods was the most dominant motivation for applying *Extract Method* refactoring to deal with code smell.

Chapter 3

Proposing and Evaluating Clone Detection Approaches

3.1 Motivation

Electronics companies are currently releasing new models of their products in regular and rushed intervals [16, 25, 83]. To release a new model within a short timeframe, a number of companies simply reuse existing files with or without modifications. This saves time and cost, and avoids the high risk entailed in creating new code logic. However, it generates many identical or similar files between different versions and models, making software systems difficult to maintain.

It is important to detect code clones from different released versions and models. For example, when a defect is contained in a code clone in one version/model, all of its cloned code fragments in the other versions/models should be inspected for the same bug. This takes a significant amount of time and effort, particularly in large-scale software systems. To date, researchers have proposed code clone detection approaches using various granularities such as lines, tokens and abstract syntax trees and have evaluated them to find the most effective approach [12, 82].

Different degrees of normalizations (i.e., the transformation of program elements) for detecting code clones have also been proposed. Each type of normalization makes subtly different but similar source code to be detected as code clones. For instance, a code clone detection tool called Dup normalizes the input source files by tokenizing each file into a single token sequence [4]. This normalization leads to the detection of source code with different white spaces, layout, and comments as code clones. A token-based code clone detection tool called CCFinder normalizes the input source files by replacing identifiers related to the types, variables, and constants using a special token, and then concatenating all tokens in the

same file into a single token sequence [49]. This normalization leads to the detection of source code with different identifiers, white spaces, layout, and comments as code clones. Different degrees of normalization cause different granularities of source code to be detected as code clones, but only little is known about how such normalization impacts the code clone detection [23].

To investigate how normalization impacts the code clone detection, this study proposes and evaluates six approaches for detecting code clones with preprocessing using different degrees of normalization. More precisely, each type of normalization is applied to the input source files, and equivalence class partitioning is then conducted on the files based on the MD5 hash function during the preprocessing. The goal of this preprocessing is to avoid an irrelevant code clone detection caused by identical files. Identical files increase the computational complexity of the code clone detection because code clones are repeatedly detected within them. The proposed approaches can be categorized into two types, an approach with non-normalization and approaches with normalization. The former category is the detection of code clones based on identical files without normalization, whereas the latter category is the detection of clones based on different degrees of normalization, such as removing macros from the input source files. After preprocessing, code clones are detected only in a **corpus** (i.e., a set of files each of which is a representative of each equivalence class) by CCFinder. As a case study, the proposed approaches and an approach that uses only CCFinder are applied to different versions of three OSS systems. The contributions of this study can be summarized as follows:

- Both of the proposed approaches with a preprocessing of the input source files are faster than the approach using only CCFinder.

- Any normalization also takes a great deal of time during the preprocessing and post-processing, and is unable to reduce the total detection time in many cases.

- We have proposed and implemented code clone detection approaches using the preprocessing of the input source files.

The remainder of this study is organized into the following sections. Section 3.2 details the proposed code clone detection approaches. Section 3.3 describes a case study on different versions of three OSS systems, and Section 3.4 details the results. Next, Section 3.5 discusses the results of the case study, Section 3.6 then describes threats to the validity of the proposed approaches, and finally, Section 3.7 concludes this chapter.

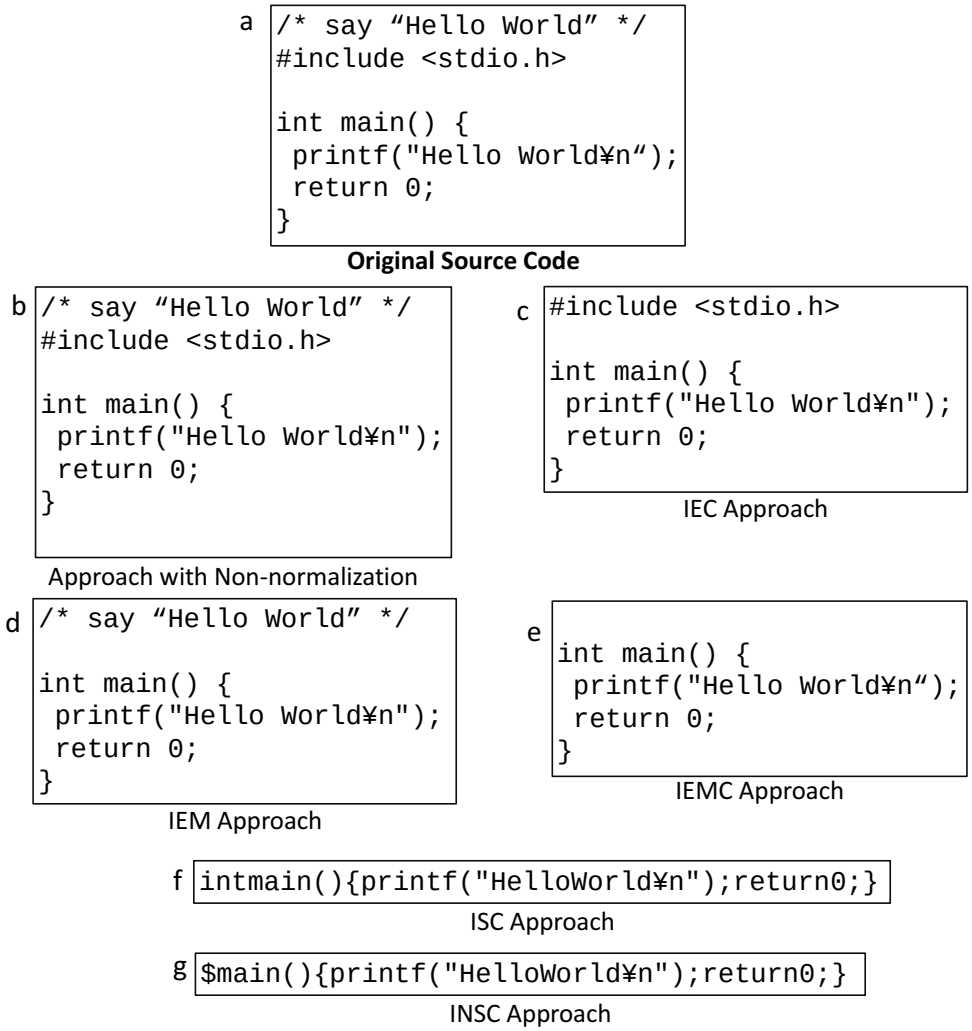


Figure 3.1: An example of the result of each normalization

3.2 Proposed Approaches

This study proposes and evaluates approaches for detecting code clones with a different type of preprocessing. The proposed approaches can be categorized into two categories: an approach with non-normalization (see Subsection 3.2.1) and approaches with normalization (see Subsection 3.2.2). The former is the detection

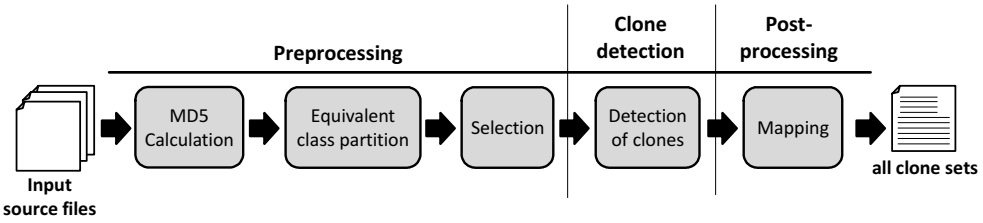


Figure 3.2: Overview of an approach with non-normalization

of code clones based on identical files without normalization, whereas the latter is the detection of code clones based on identical files with different degrees of normalization. Both approache types share the following pipeline phases:

i. **Preprocessing:** This is used to conduct equivalence class (i.e., a set of files that are identical to each other based on the hash values) partitioning and generate a corpus based on MD5 hash values of the input source files. For this study, the MD5 hash function was adopted because its probability of an accidental collision is extremely small.

ii. **Clone detection:** This is used to detect code clones in a corpus using CCFinder. In this phase, code clones are detected only on a corpus because identical files are detected as equivalence class in the preprocessing phase. As a result, time complexity from the repeatedly detection of code clones within the identical files can be reduced. To detect code clones, this study uses CCFinder because of its high accuracy in code clone detection.

iii. **Post-processing:** This is used to generate all clone sets by mapping the output of CCFinder, the equivalence classes and other information if necessary. The all cone sets exclude clone sets existing only within each equivalence class because they are already detected as equivalence classes.

The detailes of the approach with non-normalization and approaches with normalization are explained in Subsection 3.2.1 and 3.2.2 respectively.

3.2.1 Approach with Non-normalization

This type of approach identifies identical files without normalization. For example, Figures 3.1(a) and 3.1(b) show identical non-normalized files used in this type of approach. Code clones are then detected based on these identical files. Figure 3.2

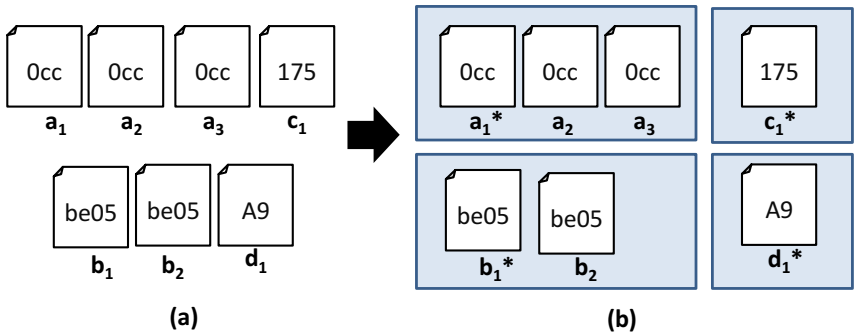


Figure 3.3: An example of equivalence class partition and selection

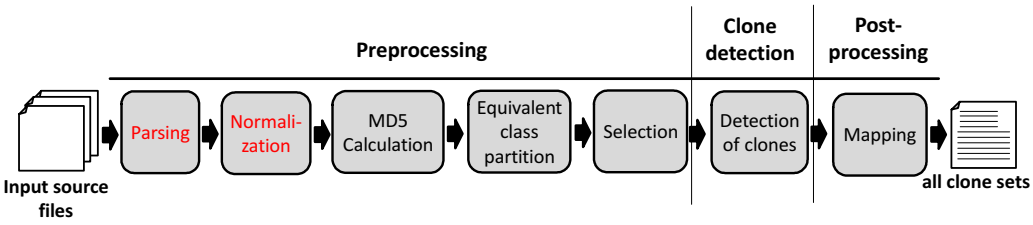


Figure 3.4: An overview of the approaches with normalization

illustrates an overview of the three phases of this approach type. The details of each phase are as follows:

a. **Preprocessing:** For each input source file, the MD5 hash value of the file is calculated. Equivalence class partitioning is then conducted based on the hash values. Namely, all files that have the same MD5 hash values are partitioned into the same equivalence class. Figure 3.3 shows an example of a partition. In this figure, characters written on the files represent the hash values, and a blue rectangle represents each equivalence class. Taking a closer look at the figure, files a_1, a_2, and a_3, which have the same hash value 'Occ', are partitioned into the same equivalence class. Files b_1 and b_2, which have the same hash value 'be05', are also partitioned into an equivalence class. In addition, files c_1 and d_1 are partitioned into their own singleton equivalence class. After the partitioning, a file is selected from each equivalence class as a representative and then added to the corpus. Figure 3.3 shows an example of such a selection. In this figure, an asterisk indicates files contained in the

corpus. That is, files a_1, b_1, c_1, and d_1 are selected and then contained in the corpus.

b. **Clone Detection:** Code clones are detected in a corpus using CCFinder.

c. **Post-processing:** It is easy to assume that if a code clone exists in one file within an equivalence class, code clones also exist in the same place in other files of the same equivalence class. Thus, during this phase, all clone sets are generated based on this assumption. That is, if a code clone is detected in a representative of an equivalence class during the clone detection phase, then a code fragment in the same place in the other files of the same equivalence class are also added to the same clone set as code clones.

3.2.2 Approaches with Normalization

This category contains approaches with different degrees of normalization, as follows:

- Identical Except for Comments (IEC) approach
- Identical Except for Macros (IEM) approach
- Identical Except for Macros and Comments (IEMC) approach
- Identical Source Code (ISC) approach
- Identical Normalized Source Code (INSC) approach

These approaches require additional processes compared to the approach with non-normalization described in Section 3.2.1. That is, they parse the input source files into tokens and then save the token information during the preprocessing phase. Figure 3.4 provides an overview of three common phases of the above approaches. In the figure, additional processes are shown in red. The details of each phase are as follows:

1. **Preprocessing:** The input source files are parsed into tokens and the following information is then extracted from each file:
    - Token list: a list of tuples (the token number, the start column number, the end column number and the line number) of each token, with the following attributes.
        - Token number: the number assigned to each token.
        - Start column number: the column number where the token starts.

– End column number: the column number where the token ends.
– Line number: the line number where the token exists.

One of the following normalizations is then applied to each input source file. Note that targets for normalizations are selected from the same program elements as CCFinder.

- For the IEC approach: All lines containing only comments, and comments or white spaces before and after the comments are removed from each input source file. This normalization transforms Figure 3.1(a) into Figure 3.1(c).
- For the IEM approach: All lines containing only macros are removed from each input source file. This normalization transforms Figure 3.1(a) into Figure 3.1(d).
- For the IEMC approach: All lines containing only macros or comments, and comments or white spaces before and after the comments are removed from each input source file. This normalization transforms Figure 3.1(a) into Figure 3.1(e).
- For the ISC approach: Tokens in the same file are concatenated into a single token sequence. This normalization transforms Figure 3.1(a) into Figure 3.1(f).
- For the INSC approach: Tokens of identifiers, literals, and types are replaced by a special token, and tokens in the same file are then concatenated into a single token sequence. This normalization transforms Figure 3.1(a) into Figure 3.1(g). In this figure, an identifier 'int' is replaced by $.

The rests of this phase are is same as the preprocessing used in the approach with non-normalization, as explained in Section 3.2.1-a. All files that have the same hash value are partitioned into the same equivalence class. After the partition, a file is selected from each equivalence class as a representative and then added to the corpus.

2. **Clone Detection:** Code clones are detected in the corpus using CCFinder.

3. **Post-processing:** As a result of normalization during the preprocessing phase, code clones may exist in different places between the files within the same equivalence class. Therefore, if a code clone is detected in a representative of an equivalence class, mapping is conducted as follows:

(a) The start token number (i.e., the first token number of the code clone) and the end token number (i.e., the last token number of the code clone) in the representative are identified.

(b) The start column number and the line number of the corresponding start token number in other files in the same equivalence class are identified based on the token list saved during the preprocessing.

(c) The end column number and the line number of the corresponding end token number in other files in the same equivalence class are also identified based on the token list.

(d) The code fragments ranging from the identified start column number in the line number to the identified end column number in the line number in other files of the same equivalence class are added into the same clone set as code clones

3.3 Case Study

In the case study, the proposed approaches and the approach that uses only CCFinder are applied to different versions of three OSS systems. Note that the proposed approaches detect code clones by excluding identical files within the same equivalence class, and therefore for the case study, different versions of the same software system were selected as subject systems because they contain many identical files. In particular, the case study was designed to address the following two RQs:

- RQ1. Can the proposed approaches detect code clones faster than an approach that uses only CCFinder?
- RQ2. Which approach is the fastest among the proposed approaches?

In the case study, 30 tokens (the default setting) were used as the minimum length of the token sequence of a code clone for CCFinder. During the case study, each approach was executed three times to obtain reliable results. This case study

Table 3.1: Statistics of subject systems

Name	#Versions	#Files	Line of code	#Tokens
Apache Ant	29	18,708	4,862,102	8,404,790
Linux kernel	12	7,839	5,690,967	12,537,555
Samsung Galaxy	2	29,573	19,920,387	43,924,235

was conducted on a 64-bit Windows 7 Professional workstation equipped with two processors, (i.e., 2.27GHz and 2.26GHz CPUs) and 24 GB of main memory.

As the subject systems, three OSS systems of different sizes and application domains were selected: *Apache Ant*[1], *Linux kernel*[2], and *Samsung Galaxy* [3]. An overview of these systems is shown in Table 3.1.

Apache Ant is a Java library and command-line tool for building systems written in Java. From this system, Java files under a directory called *main* were selected from 29 consecutive released versions (released versions 1.1 through 1.9.4). *Linux kernel* is a clone of the UNIX operating system written in C. From this system, C files having the file extensions .c, .cc, .cpp, and .cxx under a directory called *fs* were selected from 12 consecutive released versions (released versions 2.6.0 through 2.6.10). *Samsung Galaxy* is a Samsung mobile phone called Samsung Galaxy Y Pro, which is written in C. Two released versions of this model for different geographical areas, Latin America and China, were selected. From this system, C files having the file extensions .c, .cc, .cpp, and .cxx under a directory called *common* were selected from each version. Note that files that are lexically incomplete were excluded.

3.4 Results

This section describes the results of the case study used to answer the above RQs.

3.4.1 Comparison with the Approach that uses Only CCFinder

To answer RQ1, this study compared the proposed approaches with the approach that uses only CCFinder with respect to the detection time. Tables 3.2, 3.3, and 3.4 list the detection times of the proposed approaches compared to the approach that uses only CCFinder for *Apache Ant*, *Linux kernel*, and *Samsung Galaxy*, respectively. Note that the "IEM Approach" and "IEMC Approach" are conducted based on the macros in C program, and thus these approaches were only applied to *Linux kernel* and *Samsung Galaxy*. In these tables, the column Total detection time represents the detection time needed to complete each approach. The columns Preprocessing, Clone detection, and Post-processing show the detection time of each phase. In these columns, the numbers in the parentheses represent their proportion of the total detection time. Note that these tables show the average detection time of the three executions.

[1] http://ant.apache.org/

[2] http://www.kernel.org/

[3] http://opensource.samsung.com/index.jsp

As these tables indicate, the proposed approaches reduce the code clone detection time compared with the "Approach that uses only CCFinder" for all subject systems. In particular, the detection times of the proposed approaches are at least two-times shorter than that of the "Approach that uses only CCFinder". Therefore, it can be concluded that the proposed approaches are able to detect code clones faster than the "Approach that uses only CCFinder".

> The proposed approaches are able to detect code clones faster than the "Approach that uses only CCFinder".

3.4.2 Comparison of Proposed Approaches

To answer RQ2, this study compared the detection times between the proposed approaches, and then examines the results, including the number of equivalence classes and code clones. In terms of the detection time, the approach with non-normalization is relatively faster than the other proposed approaches, as indicated in Tables 3.2, 3.3, and 3.4.

Table 3.5, 3.6, and 3.7 show the number of instances from *Apache Ant*, *Linux kernel*, and *Samsung Galaxy*, respectively. In these tables, the number of equivalence classes is shown in the column #Equivalence classes. The column #Files in equivalence classes represents the number of files that are contained in the non-singleton equivalence classes. Meanwhile, the column #Files in singleton equivalence classes represents the number of files contained in the singleton equivalence classes. The columns #Clone sets and #Code clones represent the number of detected clone sets and code clones, respectively. Note that from the "Approach with non-normalization" to "INSC Approach", the number of clone sets and code clones that exist within each equivalence class are excluded because they are already detected as equivalence classes. For the "Approach with non-normalization", these columns show the numbers of detected clone sets and code clones by CCFinder. Meanwhile, other approaches describe the numbers of clone sets and code clones except for clone sets and code clones within the identical files in the same equivalence class.

Similar tendencies can be seen in these tables. For *Apache Ant*, the least numbers of equivalence classes, files in singleton equivalence classes, and code clones are detected by the "INSC Approach", as shown in Table 3.5. Similarly, for *Linux kernel* and *Samsung Galaxy*, the least numbers of equivalence classes, files in singleton equivalence classes, and code clones are detected by the "ISC Approach" and "INSC Approach", as shown in Tables 3.6, and 3.7.

The "Approach with non-normalization" is the fastest for *Linux kernel* and *Samsung Galaxy*, whereas for *Apache Ant*, the "ISC Approach" and "INSC Approach" are faster than the other approaches.

3.5 Discussion

This section discusses the results of the case study described in Sections 3.4.1 and 3.4.2. As mentioned in Section 3.4.1, the proposed approaches detect code clones faster than the "Approach that uses only CCFinder". This was caused by the large decrease in the number of files in singleton equivalence classes. The numbers of files in singleton equivalence classes were decreased by 5.13-21.45%, 9.87-10.77%, and 0.19- 0.22% for *Apache Ant*, *Linux kernel*, and *Samsung Galaxy*, respectively.

Among the proposed approaches, the "Approach with non-normalization" is the fastest for *Linux kernel* and *Samsung Galaxy*, whereas for *Apache Ant*, the "ISC Approach" and "INSC Approach" are faster than the other proposed approaches. However, the time difference between the "Approach with non-normalization" is still very small (39 seconds at maximum). This is because, in the case of *Linux kernel* and *Samsung Galaxy*, the number of files in singleton equivalence classes is almost the same between the different approaches. This leads to the "Approach with non-normalization", which requires the least number of preprocess and post-process compared with the other approaches, being the fastest. For *Apache Ant*, the "ISC Approach" and "INSC Approach" output the least number of files in singleton equivalence classes, leading these approaches to detect code clones faster than the other proposed approaches.

Therefore, it is expected that if the files contain many unique files (i.e., singleton equivalence classes), the "ISC Approach" and "INSC Approach" will be the fastest; however, for the other cases, the "Approach with non-normalization" is the fastest among all of the approaches.

3.6 Threats to Validity

The following threats to the validity of this study were identified. The proposed approaches rely on the quality of the underlying clone detection tool and the hash function to detect code clones. This threat is countered by a careful selection of the clone detection tool and hash function, i.e., CCFinder, which is a widely-used clone detection tool with high accuracy for detecting code clones, and the MD5

hash function, which is very unlikely to cause collisions.

As a case study, three different sized OSS systems were chosen from diverse domains to achieve a generality of the results. However, the results of this case study may differ for other software systems. To alleviate this limitation, the proposed approaches will be applied to additional software systems in the future.

3.7 Summary

In this chapter, code clone detection approaches with a preprocessing of the input source files using different degrees of normalization were proposed to investigate how normalization impacts the code clone detection. The proposed approaches conduct equivalence class partitioning of the input source files based on the MD5 hash values during the preprocessing. After the preprocessing, code clones are only detected from a set of files each of which is selected from each equivalence class. To detect code clones, this study used CCFinder, which is a token-based code clone detection tool. The proposed approaches can be categorized into two types, an approach with non-normalization and approaches with normalization. The former is the detection of code clones based on identical files without normalization, wherease the latter is the detection of code clones based on identical files with different degrees of normalization, such as the removal of all lines only containing macros.

In this case study, the proposed approaches, as well as an approach that uses only CCFinder, were applied to different versions of three OSS systems and evaluated with respect to the code clone detection time. It was determined that the proposed approaches detect code clones faster than an approach using only CCFinder. It was also discovered the approach with non-normalization is the fastest among the proposed approaches for many of applied cases.

Table 3.2: Detection time in seconds (Apache Ant)

Approach Names	Total detection	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	716	-	-	-
Approach with non-normalization	253	3 (1.19%)	248 (97.89%)	2 (0.79%)
IEC Approach	232	17 (7.34%)	103 (44.60%)	111 (48.06%)
ISC Approach	214	13 (6.07%)	100 (46.73%)	101 (47.20%)
INSC Approach	214	14 (6.54%)	100 (46.57%)	100 (46.88%)

Table 3.3: Detection time in seconds (Linux Kernel)

Approach Names	Total detection	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	1,058	-	-	-
Approach with non-normalization	175	2 (1.14%)	172 (98.29%)	1 (0.57%)
IEC Approach	336	23 (6.74%)	172 (51.14%)	142 (42.12%)
IEM Approach	344	26 (7.56%)	176 (51.11%)	142 (41.34%)
IEMC Approach	333	22 (6.71%)	172 (51.70%)	138 (41.58%)
ISC Approach	328	18 (5.49%)	168 (51.37%)	141 (43.13%)
NSC Approach	335	21 (6.26%)	172 (51.39%)	142 (42.35%)

Table 3.4: Detection time in seconds (Samsung Galaxy)

Approach Names	Total detection	Preprocessing	Clone detection	Post-processing
Approach that uses only CCFinder	19,622	-	-	-
Approach with non-normalization	4,326	7 (0.16%)	4,307 (99.56%)	12 (0.28%)
IEC Approach	8,803	204 (2.31%)	4,686 (53.23%)	3,913 (44.46%)
IEM Approach	9,240	271 (2.93%)	4,601 (49.79%)	4,368 (47.28%)
IEMC Approach	8,711	227 (2.60%)	4,530 (52.01%)	3,954 (45.39%)
ISC Approach	8,513	242 (2.84%)	4,398 (51.67%)	3,873 (45.49%)
INSC Approach	8,894	234 (2.63%)	4,552 (51.18%)	4,108 (46.19%)

Table 3.5: Results of Apache Ant

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singleton equivalence classes	#Clone sets	#Code clones
Approach that uses only CCFinder	-	-	-	15,626	246,245
Approach with non-normalization	4,174	14,696 (78.55%)	4,012 (21.45%)	14,778	243,211
IEC Approach	3,119	17,652 (94.36%)	1,056 (5.64%)	13,127	234,006
ISC Approach	2,993	17,739 (94.82%)	969 (5.18%)	13,003	233,011
INSC Approach	2,973	17,749 (94.87%)	959 (5.13%)	12,976	232,812

Table 3.6: Results of Linux Kernel

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singleton equivalence classes	#Clone sets	#Code clones
Approach that uses only CCFinder	-	-	-	23,031	306,592
Approach with non-normalization	1,516	6,995 (89.23%)	844 (10.77%)	20,356	293,076
IEC Approach	1,513	7,002 (89.32%)	837 (10.68%)	20,346	293,013
IEM Approach	1,517	7,046 (89.88%)	793 (10.12%)	20,248	292,198
IEMC Approach	1,512	7,056 (90.01%)	783 (9.99%)	20,228	292,026
ISC Approach	1,494	7,065 (90.13%)	774 (9.87%)	20,196	291,766
INSC Approach	1,494	7,065 (90.13%)	774 (9.87%)	20,196	291,766

Table 3.7: Results of Samsung Galaxy

Approach names	#Equivalence classes	#Files in equivalence classes	#Files in singleton equivalence classes	#Clone sets	#Code clones
Approach that uses only CCFinder	-	-	-	274,186	2,529,843
Approach with non-normalization	14,737	29,508 (99.78%)	65 (0.22%)	113,929	2,208,830
IEC Approach	14,735	29,518 (99.81%)	55 (0.19%)	113,876	2,208,713
IEM Approach	14,640	29,516 (99.81%)	57 (0.19%)	113,853	2,208,619
IEMC Approach	14,611	29,518 (99.81%)	55 (0.19%)	113,897	2,208,701
ISC Approach	14,576	29,518 (99.81%)	55 (0.19%)	113,797	2,208,363
INSC Approach	14,576	29,518 (99.81%)	55 (0.19%)	113,797	2,208,363

Chapter 4

Investigating Merged Code Clones during Software Evolution

4.1 Motivation

In recent decades, many tools have been developed to detect code clones [49, 48, 62]. Lately, the code clone research community has gradually shifted its focus from detection to management [34, 102]. Clone refactoring is one of the most vital features of code clones management. It merges a set of code clones into a single function or method. Several tools for clone refactoring have been developed, for example, the Eclipse plug-in, which supports automatic clone refactoring based on the modified Eclipse refactoring engine [91], and a tool that provides metrics indicating how code clones can be merged [43]. However, such tools are not commonly used compared to refactoring tools (e.g., Eclipse's refactoring features) not intended for supporting clone refactoring.

Murphy-Hill et al. investigated instances of refactoring in the development of OSS systems [71]. Their study provided valuable insight that could be applied to develop more widely used refactoring tools. However, such insights has proven to be insufficient for developing tools for clone refactoring, in particular because the merging of code clones is considerably more complicated than other patterns of refactoring (e.g., simple code extraction and method renaming/moving) [28].

In this study, instances of clone refactoring in OSS systems were investigated to uncover clues that could contribute to the development of more widely used tools for clone refactoring. The study began by detecting instances of refactoring from consecutive program versions of OSS systems using a refactoring detection tool called Ref-Finder [75]. From the detected instances of refactoring, instances of seven refactoring patterns (e.g., *Extract Method* and *Replace Method with Method*

Object) suggested by Fowler [28], which can be used to merge sets of code clones into the same method, were further selected. Next, to mitigate the false-positive problem, the outputs of the Ref-Finder were manually analyzed. The similarity of the token sequences in the identified instances of refactoring was then measured to identify instances of clone refactoring. Finally, the statistics of the instances of clone refactoring from 63 releases of three OSS systems were analyzed. The contributions of this study can be summarized as follows:

- **Presenting an approach to investigate how clone refactoring was conducted** To investigate instances of clone refactoring in three OSS systems, this study presents an approach using a code clone identification technique called undirected similarity (usim) and a refactoring detection tool called Ref-Finder.
- **Discovering the most widely used refactoring patterns in clone refactoring and the characteristics of merged clones.** This study discovered that the *Extract Method (EM)* and *Replace Method with Method Object (RMMO)* patterns are the most widely used when developers conduct clone refactoring. Moreover, it was found that large token differences existed between merged code clones in cases where the *RMMO* and *EM* patterns were applied.
- **Suggestions for clone refactoring tools.** This study provides several suggestions for developing tools to support code refactoring based on the results of this investigation.

The remainder of this chapter is organized into the following sections. Section 4.2 describes the research questions of this study. Next, Section 4.3 details the steps for investigating the characteristics of merged code clones. Section 4.4 analyzes the results of this investigation for three OSS systems, Section 4.5 provides suggestions for tools to support clone refactoring based on the results, and Section 4.6 describes threats to the validity of this study. Finally, Section 4.7 summarizes this research.

4.2 Research Questions

The RQs in this study were devised to identify important clues regarding the development of more widely used tools for clone refactoring. The RQs are as follows:

Which refactoring patterns are the most frequently used in clone refactoring? (RQ1) Among the refactoring patterns that can be used for clone refactoring, tools for clone refactoring should preferentially support the most frequently used refactoring patterns. Therefore, RQ1 aims to identify the most commonly used

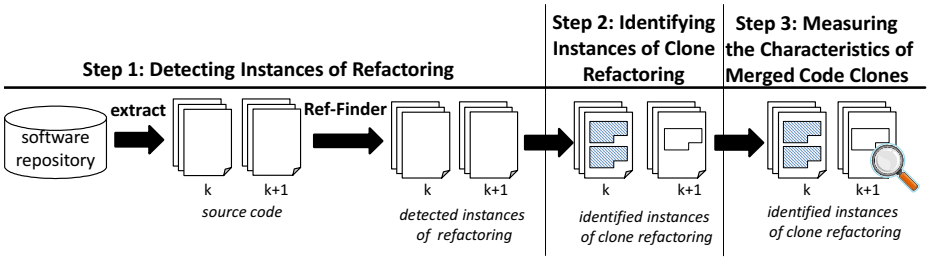


Figure 4.1: Overview of the investigation

refactoring patterns. By answering this RQ, it is believed that the information could help develop clone refactoring tools that support frequently used refactoring patterns.

How similar are the token sequences between pairs of merged code clones? (RQ2) . In addition, **how different are the lengths of token sequences between pairs of merged code clones? (RQ3)** Whether code clones are merged into the same method depends highly on their similarities. Code clones that are very similar to each other are more easily merged into the same method. However, even if they share few similarities, developers are often able to merge them with a certain amount of effort based on the refactoring patterns. For instance, after identical code fragments are merged into the same method, different parts are extracted as an each method using the *Form Template Method* pattern. These RQs aim to uncover how merged code clones differ with respect to the token content (RQ2), and token lengths (RQ3) based on their refactoring pattern. It is believed that the answers to these RQs could help with the development of clone refactoring tools to better detect candidates for pairs of code clones based on their similarities and differences.

How far apart are pairs of code clones located before clone refactoring? (RQ4) Tools for clone refactoring should be capable of suggesting candidates for clone refactoring. However, it is difficult to select the appropriate candidates because code clones are spread out over various locations (e.g., the same class, different packages). Therefore, RQ4 aims to determine how far apart code clones were located before they were refactored. It is believed that this would further improve a clone refactoring tool's ability to locate pairs of code clone candidates.

4.3 Steps of this Investigation

To the best of our knowledge, no techniques or tools for detecting instances of clone refactoring have been proposed. Therefore, a refactoring detection tool first detects instances of refactoring, and then identify instances of clone refactoring

from the results using a code clone identification technique was used. Figure 4.1 provides an overview of the investigation into the characteristics of merged code clones, which is composed of the following three steps:

Step 1. Detect instances of refactoring between two consecutive program versions.

Step 2. Identify instances of code refactoring from a set of instances of the detected refactoring .

Step 3. Measure the characteristics of the merged code fragments in an old version of the software and categorize the data based on the refactoring pattern.

The following sections describe the detailes of each step.

4.3.1 Step 1 : Detecting Instances of Refactoring

In this step, instances of refactoring in OSS systems were detected. To accomplish this, the source code of each system was extracted from its respective software repository. Next, Ref-Finder [75] was applied to two consecutive program versions (e.g., versions k and k+1) to detect instances of refactoring. Ref-Finder takes two consecutive program versions as input data and reports instances of refactoring. It can detect 65 of Fowler's refactoring patterns.

Seven refactoring patterns that could be used specifically for clone refactoring were then selected. They are *Extract Method(EM)*, *Extract Class (EC)*, *Parameterize Method (PM)*, *Pull Up Method (PUM)*, *Extract Superclass (ES)*, *Form Template Method (FTM)*, and *Replace Method with Method Object (RMMO)*.

Extract Method (EM) : Originally, *EM* can be applied to source code that is too complicated or long to understand its purpose. It can also be applied to remove source code that has the same expression in two methods of the same class. Figure 4.2 illustrates an example of clone refactoring using *EM*, which can be used to merge code clones with similar expressions in the same class based on Fowler's refactoring book [28]. In this figure, version k includes two duplicated statements (shown in bold) existing in two separate methods (printOwing and printAssets). After conducting clone refactoring based on the *EM* pattern (version k+1), these code clones are extracted from the two methods to create a new method (*printDetails*), and the old statements are replaced by caller statements to the new method.

Extract Class (EC) : Originally, *EC* can be applied when a class is too big or complicated to easily understand. It is also applied to similar methods or fields existing in the same class or unrelated classes.

Extract Superclass (ES) : *ES* can be applied when two or more classes have similar features but do not have a common parent class.

Form Template Method (FTM) : If a developer would like to merge two similar methods that conduct similar steps in the same order, yet the steps are different from subclasses into a superclass, *FTM* can be used. In this case, a developer can move the similar methods to the superclass and allow polymorphism to play its role in ensuring that the different steps conduct things differently. This kind of method is called a templated method .

Pull Up Method (PUM) : *PUM* can be applied to methods that have the same body in the subclasses.

Parameterize Method (PM) : If several methods do similar things but with different values contained in the method body, these separate methods are replaced with a single method that handles variations by different parameters, so-called *parameterize method*, which is an effective refactoring pattern. Such a change removes duplicated code and increases the flexibility because programmers can deal with other variations by adding other parameters.

Replace Method with Method Object (RMMO) : Originally, *RMMO* can be applied to a long method that uses local variables in such a way that the developer cannot apply the *EM*. The *RMMO* can also be used to merge code clones that use local variables by extracting code clones into a new method that is its own object, where all of the local variables become fields of that object. Figure 4.3 illustrates an example of clone refactoring using the *RMMO* pattern. Before clone refactoring (version k), two cloned methods (normalPrice and salePrice), shown in bold, use local variables. After clone refactoring (version k+1), these code clones are extracted to a new method of a new class (PriceCalculator), and all the local variables are moved into fields of the PriceCalculator class.

Finally, the outputs of Ref-Finder were manually validated because they contained many false positives [88]. To accurately validate the outputs, existing validated output data of Ref-Finder that was used in a previous study by Bavota et al. [9] was referenced. In the study, two master course students at the University of Salerno conducted the manual validation.

For the purposes of this investigation, the same released versions used for the previously validated data were selected, including 63 released versions of three

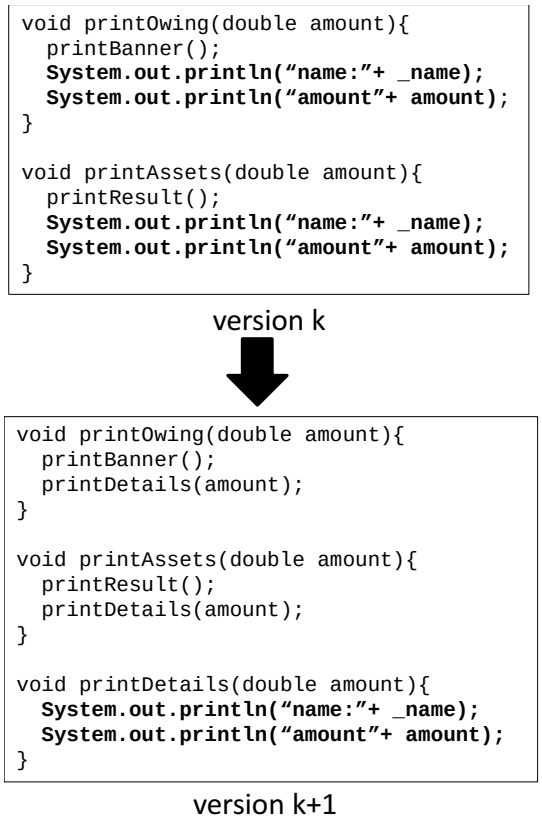


Figure 4.2: An example of clone refactoring using the *EM* pattern

Java OSS systems: *Apache Ant* [1], *ArgoUML* [2], and *Xerces-J* [3]. Table 4.1 provides statistical data on each of these software systems.

[1] http://ant.apache.org/
[2] http://argouml.tigris.org/
[3] http://xerces.apache.org/xerces-j/

Table 4.1: Statistics of subject systems

Software	Versions	#Versions	Period
Apache Ant	1.2-1.8.2	17	Jan 2000-Dec 2010
ArgoUML	0.12-0.34	13	Oct 2002-Dec 2011
Xerces-J	1.0.4-2.9.1	33	Nov 1999-Nov 2010

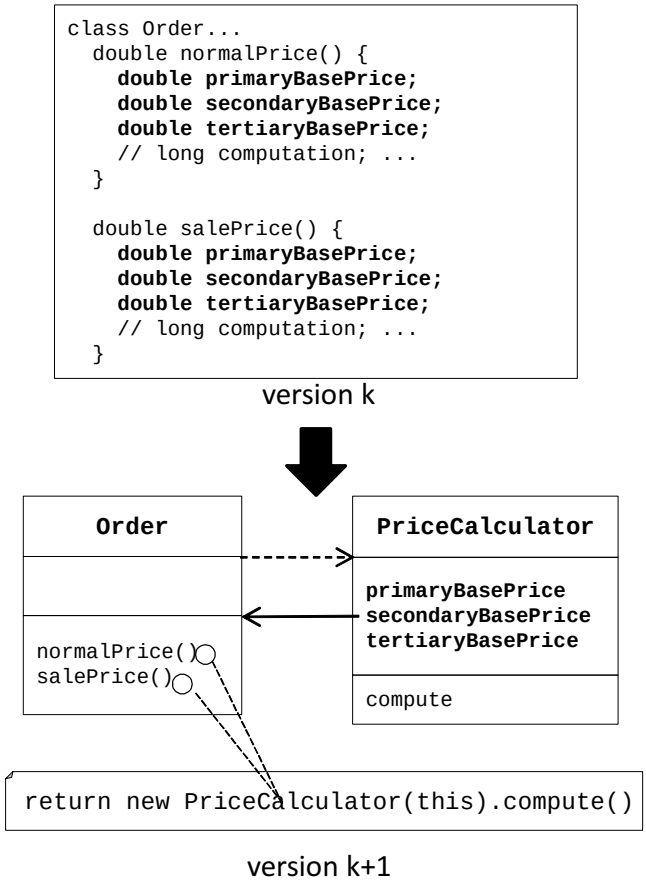


Figure 4.3: An example of clone refactoring using the *RMMO* pattern]

4.3.2 Step 2 : Identifying Instances of Clone Refactoring

In this step, *undirected similarity (usim)* [67] was used to identify instances of clone refactoring from the instances of refactoring detected in Step 1. To elaborate, each refactored pair was defined as an instance of clone refactoring, only if it satisfied the following three conditions:

Condition 1 : Each pair of code fragments was refactored into the same new method in the new software version. This means that each pair of code fragments in the old version was merged into the same new method in the new version.

Condition 2 : The computed *usim* value of each pair of code fragments in the old version was more than 65%. Many token-based clone detection tools

have been proposed [49, 62] because such tools detect code clones with high accuracy [12]. However, existing tools fail to identify many code clones when they contain large dissimilarities, which are often found in Type-3 and Type-4 code clones. For example, CCFinder can only detect Type-1 and Type-2 code clones [49]. Consequently, existing token-based clone detection tools are incapable of accurately detecting certain instances of clone refactoring because developers sometimes conduct Type-3 and Type-4 clone refactoring.

To identify all code clone types, this study used the *usim* to identify code clones. Originally used to identify code clones to find candidates for clone refactoring in the software evolution, *usim* uses the *Levenshtein distance*, which measures the minimum amount of changes necessary to transform one sequence of items into a second sequence of items [61]. For instance, the *Levenshtein distance* between *survey* and *surgery* is 2, and that between *color* and *colour* is 1 [3].

The definition of *usim* is given in Equation (4.1) [67]. Each instance of refactoring in its original version is represented as a normalized sequence $sf_x = norm(fx)$, where the normalization function $norm$ removes comments, line breaks, and insignificant white spaces. The resulting distance $\Delta f_{x,y} = LD(sf_x, sf_y)$ then describes the number of tokens that must be changed to turn the code fragment f_x into f_y. The *Levenshtein distance* can be normalized to a relative value using the length of the token sequence $l_x = len(sf_x)$:

$$\textit{usim}\,(f_x, f_y) = \frac{\max\,(l_x, l_y) - \Delta f_{x,y}}{\max\,(l_x, l_y)} \times 100\,(\%) \tag{4.1}$$

To confirm this condition, the old version of each instance of refactoring is first concatenated into a single token sequence. During this process, comments and white spaces are removed from the token sequences. Then, the concatenated token sequences are normalized by replacing variables and identifiers with a special token. Finally, the *usim* values of each pair of token sequences that were merged into the same new method in a later version of the software are computed. Pairs are defined as code clones if the *usim* values between their token sequences are greater than 65%. This threshold was used based on Mende's study [67] because the authors discovered that the best compromise between recall and precision can be obtained at a *usim* value of 65%.

Condition 3 : The token length of each refactored pair was greater than 10 in the old version. In Mende's study [67], the best compromise between recall and precision was obtained at usim value of 65% with a minimum token length parameter of ten tokens. Therefore, instances of refactoring where the code fragment in the old version had a token length of fewer than ten tokens were also excluded.

4.3.3 Step 3 : Measuring the Characteristics of Merged Code Clones

After identifying instances of clone refactoring, the next step was to measure their characteristics in order to answer the RQs introduced in Section 4.2.

To answer **RQ1**, the number of sets of merged code clones between refactoring patterns was analyzed. In this analysis, pairs of code clones were categorized based on whether they were merged into the same newly-created method using the refactoring patterns.

To address **RQ2**, the token similarities between pairs of merged code clones were measured using *usim*, which was also used to identify code clones in Section 4.3.2. Among a set of merged code clones, the *usim* values can sometimes differ from each other because pairs of code clones are categorized into the same set based on the merged method in the new version. To analyze the distribution of the *usim* values accurately, U_{mi} (a set containing the minimum usim values of merged code clones), U_{av} (a set containing the average usim values of merged code clones), and U_{mx} (a set containing the maximum usim values of merged code clones) were measured between the refactoring patterns.

Suppose that $S_1, S_2, \cdots, S_i, \cdots, S_n$ (where $1 \leq i \leq n$) represent sets of merged code clones refactored using the same refactoring pattern, u_{mi_i} represents the minimum *usim* value of S_i, u_{av_i} represents the average *usim* value of S_i, and u_{mx_i} represents the maximum *usim* value of S_i, then $U_{mi} = \{u_{mi_1}, u_{mi_2}, \cdots, u_{mi_n}\}$, $U_{av} = \{u_{av_1}, u_{av_2}, \cdots, u_{av_n}\}$, and $U_{mx} = \{u_{mx_1}, u_{mx_2}, \cdots, u_{mx_n}\}$.

To answer **RQ3**, how the length of the token sequences differs between pairs of merged code clones was investigated. First, the length of the differences in token sequences between a pair of merged code clones c_1 and c_2 as $LD = |lt_1 - lt_2|$ was defined, where lt_1 represents the length of the token sequences of c_1, and lt_2 represents the length of the token sequences of c_2. Secondly, because the LD values among a set of merged code clones occasionally vary, L_{mi} (a set containing the minimum LD values of merged code clones), L_{av} (a set containing the average LD values of merged code clones), and L_{mx} (a set containing the maximum LD values of merged code clones) were measured between refactoring patterns.

Suppose that $S_1, S_2, \cdots, S_i, \cdots, S_n$ (where $1 \leq i \leq n$) represent sets of merged code clones refactored by the same refactoring pattern, l_{mi_i} represents the minimum LD value of S_i, l_{av_i} represents the average LD value of S_i, and l_{mx_i} represents the maximum LD value of S_i, then $L_{mi} = \{l_{mi_1}, l_{mi_2}, \cdots, l_{mi_n}\}$, $L_{av} = \{l_{av_1}, l_{av_2}, \cdots, l_{av_n}\}$, and $L_{mx} = \{l_{mx_1}, l_{mx_2}, \cdots, l_{mx_n}\}$.

In response to **RQ4**, the term *class distance* was defined as an indicator of the location between pairs of merged code clones in the old version of the code. Locations can be categorized into one of the following categories: Same Class, Same Package, and Different Packages.

Only the *class distances* for code clones refactored by the *RMMO* pattern were investigated because the other six refactoring patterns already contained constraints regarding the location of pairs of the code clones. For example, the *PUM* pattern can only be used for pairs of code clones in subclasses that have a common superclass. To answer RQ4, the *class distances* between pairs of merged code clones refactored by the *RMMO* pattern were analyzed.

4.4 Results of the Investigation

This section details the investigation results and provides answers to the RQs based on these results[4].

Which refactoring patterns are the most frequently used in clone refactoring? (RQ1)

To answer **RQ1**, Table 4.2 shows the number of sets of merged code clones, as well as the numbers of pairs of merged code clones (in the parentheses) organized by each refactoring pattern.

The table reveals that a total of 35 sets of merged code clones were identified from the three software systems investigated. Surprisingly, only four types of clone refactoring (the *EM*, *ES*, *FTM*, and *RMMO* patterns) were found, while there were no detected instances of the *EC*, *PM*, and *PUM* patterns.

Figures 4.4 and 4.5 show examples of found pairs of merged code clones in the subject systems. Note that the layouts of these examples were changed to save space. Figure 4.4 shows an example of clone refactoring using the *EM* pattern in *Apache Ant* between releases 1.6.2 and 1.6.3. In this figure, releases 1.6.2 includes pairs of code clones (shown in bold) existing in two separate methods (setIncludes and setExcludes) in a class called org.apache.tools.ant.DirectoryScanner. In release 1.6.3, these code clones are extracted from the two methods to create a new method called normalizePattern, which is shown in red, and the old statements are replaced by caller statements in the new method. Figure 4.5 shows an example of clone refactoring using the *RMMO* pattern in *Xerces-J* between releases 1.0.4 and 1.2.0. In release 1.0.4, the two cloned methods (getContentSpecHandle and getContentSpecType), shown in bold, use local variables. In release 1.0.6, these code clones are extracted as a new method called getElementDecl, shown in red, of a new class (org.apache.xerces.validators.common.Grammar), and all local variables are also moved into fields of the Grammar class.

Taking a closer look at the table, 11 sets, and 12 pairs of merged code clones across eight releases from three software systems were identified as having been

[4]Our analyzed data is available at `http://sel.ist.osaka-u.ac.jp/~ejchoi/refactoredclones`

refactored by the *EM* pattern. In most cases, it was found that the *EM* pattern was only used to merge one pair of code clones, meaning that it was frequently used to merge small sets of code clones.

In examining the instances of the *ES* and *FTM* patterns, it was determined that they were only found in one release of *ArgoUML* for each pattern. The *ES* and *FTM* patterns were used to merge 15 and 6 pairs of code clones, respectively. To summarize, the *ES* and *FTM* patterns were rarely used for clone refactoring, but were used to merge a large number of code clones.

In contrast, 22 sets, and 455 pairs of merged code clones were refactored using the *RMMO* pattern across ten releases in two software systems, *ArgoUML*, and *Xerces-J*. In particular, the *RMMO* pattern was most commonly used in release 0.26 of *ArgoUML*. In it, 34 pairs of coded methods called initWizard, which were distributed across 11 classes, were merged into a single getToDoItem method in the org.argouml.cognitive.critics.Wizard class. Furthermore, 142 pairs of coded methods called doIt, getChoices, and getSelected located in 17 classes were also merged into a single getTarget method in a class called org.argouml.uml.ui.AbstractActionAddModelElement2. An additional 245 pairs of coded methods in 23 classes called stillValid were merged into a method called isActive in the org.argouml.cognitive.Critic class. It was observed that the *RMMO* pattern was used to merge sets of code clones of various sizes.

The *RMMO* pattern was the most frequently used refactoring pattern observed, followed by the *EM* pattern. Conversely, the *ES* and *FTM* patterns were used the least.

How similar are the token sequences between pairs of merged code clones? (RQ2)

The results of **RQ2** can be seen in the box-plots of Figure 4.6. It was observed that U_{av} had the same distribution as U_{mi}, and U_{mx} for the *EM*, *ES*, and *FTM* patterns. This was caused by the fact that sets of merged code clones were mainly comprised of a pair of code clones (particularly with the *EM* pattern), or because only one set of merged code clones was identified (shown with the *ES* and *FTM* patterns). Figure 3(a) shows the distribution of U_{av} for the *EM*, *ES*, *FTM*, and *RMMO* patterns. The distributions of U_{mi}, U_{av}, and U_{mx} are different for the

Table 4.2: The number of sets of merged code clones, and the number of pairs of merged code clones in parentheses from overall subject systems

Refactoring pattern	EM	ES	FTM	RMMO
# Instances	11 (12)	1 (15)	1 (6)	22 (455)

RMMO pattern, as shown in Figure 3(b). In these figures, the vertical axis represents the *usim* value, which starts from 65%, because this is the minimum value used herein to define a pair of merged code clones (see Section 4.3.2).

Figure 3(a) shows that the token similarities of pairs of merged code clones refactored by the *EM* and *RMMO* patterns were relatively low compared to those of the *ES* and *FTM* patterns. This was believed to be caused by the fact that the *ES* and *FTM* patterns merge pairs of code clones from subclasses into the same superclass. On the other hand, the *EM* and *RMMO* patterns merge pairs of code clones into the same new method within the same class or different classes.

It was discovered that the *EM* and *RMMO* patterns were used to merge pairs of code clones of various token similarities. However, these two patterns were mainly used to merge pairs of relatively dissimilar code clones. The *EM* pattern (median of 73%) was used to merge pairs of code clones that were less similar than the pairs refactored by the *RMMO* pattern (median of 88%). This implies that the *EM* pattern was used to merge pairs of code clones with fewer similarities. Largely dissimilar pairs of code clones refactored by the *EM* pattern were consistently observed across all three software systems. Compared to the *EM* pattern, pairs of code clones merged using the *RMMO* pattern shared more similarities with one another. (the median of U_{mi}, U_{av}, and U_{mx} was approximately 88%). Similar results were also obtained in *ArgoUML* and *Xerces-J*.

> The *EM* and *RMMO* patterns were mainly used to merge pairs of code clones of various token similarities. Conversely, the *ES* and *FTM* patterns were used mainly to merge highly similar pairs of code clones.

How different are the lengths of token sequences between pairs of merged code clones? (RQ3)

The results of **RQ3** were also analyzed via box-plots. It was observed that the distribution of L_{av} was the same as in L_{mi}, and L_{mx} for the *EM*, *ES*, and *FTM* patterns. As mentioned above, this was largely due to the fact that sets of merged code clones primarily comprised a pair of code clones (*EM* pattern), or because only one set of merged code clones was identified (*ES* and *FTM* patterns). Figure 4(a) shows the distribution of L_{av} for the *EM*, *ES*, *FTM*, and *RMMO* patterns. Further, the distributions of L_{mi}, L_{av}, and L_{mx} for the *RMMO* pattern, which can be seen in Figure 4(b), also differed. In the figures, the vertical axis represents the differences in the token lengths between pairs of merged code clones.

It was discovered that differences in the token lengths between pairs of merged code clones varied more for the *EM* and *RMMO* patterns than for the *ES* and *FTM* patterns. Even though the differences in token lengths between the merged code clones refactored by the *EM* pattern varied, this pattern was mainly used to merge

pairs with relatively small differences in token lengths (median of 15). Only small differences in token lengths between pairs of merged code clones were found in *Apache Ant* and *ArgoUML*. Conversely, the differences in token lengths varied relatively widely in *Xerces-J*. The *RMMO* pattern was also mainly used to merge pairs of code clones with similar differences in token lengths (median of 16). Similar results were obtained from *ArgoUML* and *Xerces-J*.

> The *RMMO* and *EM* patterns were used to merge pairs of code clones with tokens of varying lengths. In contrast, there was no difference in length in the token sequences of pairs of code clones refactored by the *ES* and *FTM* patterns.

How far apart are pairs of code clones located before clone refactoring? (RQ4)

In response to **RQ4**, Table 4.3 shows the three different class distance categories identified in instances of the *RMMO* pattern, along with the number of pairs of code clones (in the *# of pairs of code clones* column) and the percentage value (in the *Percentage* column) of that category. Table 4.3 shows that pairs of code clones within the same Java package were the most prevalent, followed by pairs of code clones in different packages and in the same class.

> Pairs of code clones in the same Java package were the most prevalent, followed by pairs in different packages and in the same class.

4.5 Suggestions for Clone Refactoring Tools

This section details the suggestions for developing clone refactoring tools based on the answers to the RQs above. These suggestions are as follows:

- It is vital for tools to support the *RMMO* and *EM* patterns, as evidenced by the answer to RQ1.

Table 4.3: The number of pairs of code clones and percentage share categorized by class distance

Class distance	# of pairs of code clones	Percentage (%)
Same Class	13	3
Same Package	324	71
Different Packages	118	28

- To support the *RMMO* pattern, the tools should suggest the following code clones as candidates for clone refactoring.
  - Pairs of code clones of various token similarities, as shown in the response to RQ2,
  - Pairs of code clones with various differences in token size, as shown in the response to RQ3,
  - Pairs of code clones that are distributed in the same Java package, as shown in the response to RQ4.
- To support the *EM* pattern, the tools should suggest pairs of code clones with various token similarities as candidates for clone refactoring, as shown in the response to RQ2. Moreover, code clone candidates with different token lengths should also be suggested on the basis of the results of RQ3.

These findings provide evidence of how (RQ1) and which code clones (RQ2, RQ3, and RQ4) were refactored. These findings can be utilized when tools are used to suggest candidates for clone refactoring for the *RMMO* or *EM* patterns. Figure 4.8 shows an overview of a tool suggested for supporting the *EM* pattern based the findings described herein. As shown in this figure, when a developer extracts a code clone as a new method, this tool detects the developer's action in conducting clone refactoring in the background, and then suggests candidates for clone refactoring to the developer. For this suggestion, the tool suggests candidates according to the results of RQ2 and RQ3, code clones with various tokens and/or different token lengths. On the other hand, to support the *RMMO* pattern, when a developer extracts a code clone as a new method in a newly created class, the tool detects the developer's action in conducting clone refactoring in the background, and then suggests candidates for clone refactoring. For this suggestion, the tool suggests code clones with various token similarities and/or different token lengths in the same Java package according to the results of RQ2, RQ3, and RQ4.

As a future challenge, a system for ranking code clones is needed for the efficient candidate suggestion. Further investigation into the OSS version archives should be conducted with a high priority to discover the characteristics of code clones. In addition, a study should be conducted on *how the suggested tool actively detects developer's action in conducting clone refactoring*. This type of study is necessary because the tool suggested in the previous paragraph needs to detect the developer's action in conducting clone refactoring. In the case of non-clone refactoring, an active detection of the refactoring was already realized by Foster et al [27]. Their tool, called WitchDoctor, observes the developer's programming activities conducted in the background process to detect the beginning of the refactoring process on the fly. Once the beginning of the refactoring process is detected,

WitchDoctor suggests code transformations to complete the process. By extending WitchDoctor, the next step is to realize the active detection of clone refactoring and thereby develop the tool suggested above. Moreover, after developing the suggested tool, validation on whether the suggested tool accurately detects clone refactoring should be conducted.

4.6 Threats to Validity

There are three limitations to the curren investigation.

The first limitation is that the investigation results might have been too dependent on the output of Ref-Finder and *usim* because this investigation was based significantly on the data from these two process. However, the outputs of both processes were validated in the [75] and [67], respectively. Moreover, the results of Ref-Finder were manually validated based on Bavota's study [9] to improve the accuracy of the investigation on clone refactoring. Therefore, the results of this investigation are deemed reliable.

As the second limitation, because ten tokens were used as the minimal token length parameter, and a 65% *usim* value was used to identify instances of clone refactoring, real instances of clone refactoring might have been missed. However, the results of this investigation are deemed reliable because these parameters were validated in Mende's study with the best compromise between the recall and precision [67]. Moreover, any small-scale instances that may have been missed are thought te be trivial to software maintenances.

As the final limitation, because the case study described herein was conducted on three particular OSS systems, an investigation into different systems could have led to different results. However, the investigation results are deemed generalizable and applicable to other OSS systems because they spanned 63 released versions from the three separate systems.

4.7 Summary

In this chapter, an investigation into instances of clone refactoring identified in three OSS systems was presented to provide insights into the development of clone refactoring tools that could be more widely used in the industry. In this investigation, instances of refactoring were detected from consecutive program versions of software systems using Ref-Finder. Next, instances of clone refactoring using an undirected similarity were identified. To improve the accuracy of this investigation, the instances of clone refactoring and the statistics of pairs of merged code clones to answer the RQs above were manually validated.

The investigation results show that it is vital for clone refactoring tools to support the *RMMO* and *EM* patterns. Such tools should also suggest pairs of code clones with varying tokens in the same Java package as candidates for the *RMMO* pattern. In addition, the suggested pairs of code clone candidates should have different in token lengths. To support the *EM* pattern, pairs of code clones with varying levels of similarities should be suggested as candidates for clone refactoring.

```
public final class XMLValidator .....
  private static final int CHUNK_SHIFT = 8; // 2^8 = 256
  private static final int CHUNK_SIZE = (1 << CHUNK_SHIFT);
  private static final int CHUNK_MASK = CHUNK_SIZE - 1;
   .....

  public int getContentSpecHandle(int elementIndex) {
    if (elementIndex < 0 || elementIndex >= fElementCount)
      return -1;
    int chunk = elementIndex >> CHUNK_SHIFT;
    int index = elementIndex & CHUNK_MASK;
    return fContentSpec[chunk][index];
  }
 .....
  public int getContentSpecType(int elementIndex) {
    if (elementIndex < 0 || elementIndex >= fElementCount)
      return -1;
    int chunk = elementIndex >> CHUNK_SHIFT;
    int index = elementIndex & CHUNK_MASK;
    return fContentSpecType[chunk][index];
  }
 .....
```

release 1.0.4

```
public final class XMLValidator .....
  public int getContentSpecType(int elementIndex) {
    int contentSpecType = -1;
      if ( elementIndex > -1) {
        if ( fGrammar.getElementDecl(elementIndex,fTempElementDecl) ) {
          contentSpecType = fTempElementDecl.type;
        }
      }
    return contentSpecType;
  }
  .....
  public int getContentSpecHandle(int elementIndex) {
    int contentSpecHandle = -1;
      if ( elementIndex > -1) {
        if ( fGrammar.getElementDecl(elementIndex,fTempElementDecl) ) {
          contentSpecHandle = fTempElementDecl.contentSpecIndex;
        }
      }
    return contentSpecHandle;
  }
  .....
```

```
public class Grammar .....
  private static final int CHUNK_SHIFT = 8; // 2^8 = 256
  private static final int CHUNK_SIZE = (1 << CHUNK_SHIFT);
  private static final int CHUNK_MASK = CHUNK_SIZE - 1;
  .....
  public boolean getElementDecl(int elementDeclIndex, XMLElementDecl elementDecl) {
    if (elementDeclIndex < 0 || elementDeclIndex >= fElementDeclCount) {
      return false;
    }

    int chunk = elementDeclIndex >> CHUNK_SHIFT;
    int index = elementDeclIndex &  CHUNK_MASK;
   .....
```

release 1.2.0

Figure 4.4: An example of clone refactoring using the *EM* pattern in *Apache Ant* between releases 1.6.2 and 1.6.3.

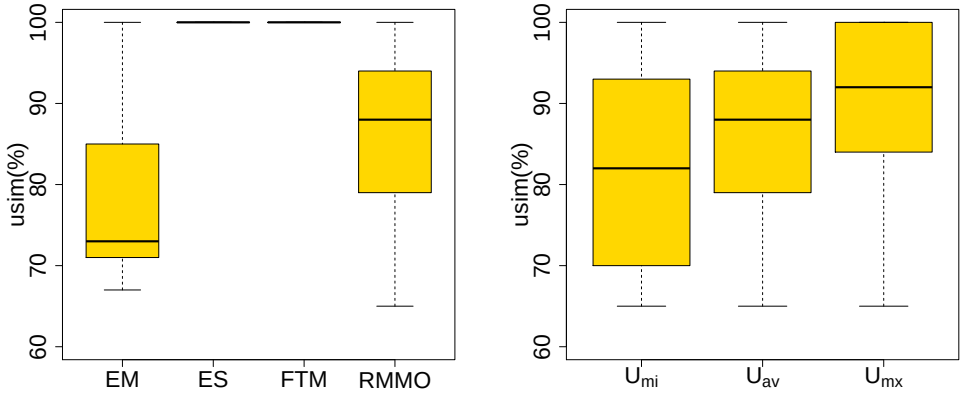
```
public class DirectoryScanner .....
  public void setIncludes(String[] includes) {
    if (includes == null) {
      this.includes = null;
    } else {
      this.includes = new String[includes.length];
      for (int i = 0; i < includes.length; i++) {
        String pattern;
        pattern = includes[i].replace('/', File.separatorChar).replace(
          '¥¥', File.separatorChar);
        if (pattern.endsWith(File.separator)) {
          pattern += "**";
        }
        this.includes[i] = pattern;
      }
    }
  }
 .....
  public void setExcludes(String[] excludes) {
    if (excludes == null) {
      this.excludes = null;
    } else {
      this.excludes = new String[excludes.length];
      for (int i = 0; i < excludes.length; i++) {
        String pattern;
        pattern = excludes[i].replace('/', File.separatorChar).replace(
          '¥¥', File.separatorChar);
        if (pattern.endsWith(File.separator)) {
          pattern += "**";
        }
      this.excludes[i] = pattern;
    }
  }
 .....
```

release 1.6.2

```
public class DirectoryScanner .....
  public synchronized void setIncludes(String[] includes) {
    if (includes == null) {
      this.includes = null;
    } else {
      this.includes = new String[includes.length];
      for (int i = 0; i < includes.length; i++) {
        this.includes[i] = normalizePattern(includes[i]);
      }
    }
  }
 .....
  public synchronized void setExcludes(String[] excludes) {
    if (excludes == null) {
      this.excludes = null;
    } else {
      this.excludes = new String[excludes.length];
      for (int i = 0; i < excludes.length; i++) {
        this.excludes[i] = normalizePattern(excludes[i]);
      }
    }
  }
 .....
  private static String normalizePattern(String p) {
    String pattern = p.replace('/', File.separatorChar)
      .replace('¥¥', File.separatorChar);
    if (pattern.endsWith(File.separator)) {
      pattern += "**";
    }
    return pattern;
  }
  .....
```

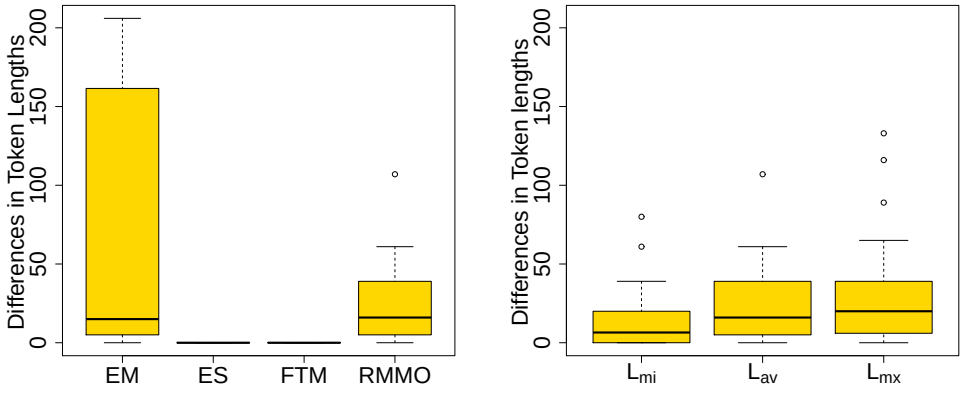
release 1.6.3

Figure 4.5: An example of clone refactoring using the *RMMO* pattern in *Xerces-J* between releases 1.0.4 and 1.2.0



(a) (b)

Figure 4.6: Box plots of U_{av} for the EM, ES, FTM, and RMMO patterns (a), and of U_{mi}, U_{av}, and U_{mx} for RMMO pattern (b)



(a) (b)

Figure 4.7: Bot plots of L_{av} for the EM, ES, FTM, and RMMO patterns (a), and L_{mi}, L_{av}, and L_{mx} for RMMO pattern (b)

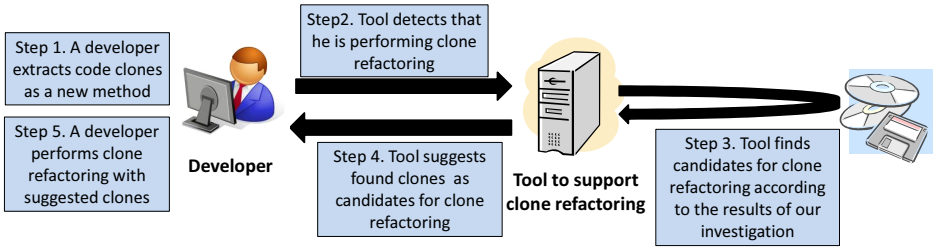


Figure 4.8: Overview of a tool that we suggest for supporting EM pattern based the findings

Chapter 5

Conclusion and Future Work

In this section, Section 5.1 first provides some concluding remarks regarding the studies in this thesis and Section 5.2 then discusses possible areas of future work.

5.1 Conclusion

In this paper, two types of approaches were proposed to answer two research questions (i.e. RQ1: which type of normalization dose make code clones to detected with high speed from large-scale software systems? and RQ2: which supports are necessary for more widely used tools that support clone refactoring?), as introduced in Section 1.1.

First, to answer RQ1, code clone detection approaches with a preprocessing of the input source files using different degrees of normalizations were proposed and presented. These approaches can be categorized into two types, an approach with non-normalization and approaches with normalization. The former is the detection of code clones based on identical files without normalization, whereas the latter category is the detection of code clones based on identical files with different degrees of normalization, such as the removal of all lines containing only macros. All of the proposed approaches are composed of the three following pipeline phases:

Preprosessing : During the preprocessing, equivalence class partitioning of the input source files is conducted based on the MD5 hash values.

Clone Detection : Code clones are only detected from a set of files that are selected from each equivalence class using CCFinder, which is a token-based code clone detection tool.

Post-processing : Here, all clone sets are generated by mapping the output of CCFinder, the equivalence classes, and other information if necessary.

In the case study, the proposed approaches and an approach that uses only CCFinder were applied to three OSS systems. As a result, it was found that the proposed approaches detect code clones faster compared to the approach that uses only CCFinder. It was also discovered that the approach with non-normalization is the fastest among the proposed approaches in many cases.

Second, to answer RQ2, instances of clone refactoring during the development process of three OSS systems were investigated. Instances of refactoring were detected from consecutive versions of the software systems using Ref-Finder, and instances of clone refactoring using an undirected similarity was identified. Furthermore, the instances of clone refactoring and the statistics of pairs of merged code clones were manually validated to improve the accuracy of the investigation. This investigation revealed that the *RMMO* pattern was the most frequently used refactoring pattern observed, followed by the *EM* pattern. Therefore, it would be vital for clone refactoring tools to support the *RMMO* and *EM* patterns. Such tools should also suggest pairs of code clones with varying tokens in the same Java package as candidates for the *RMMO* pattern. In addition, the suggested pairs of code clone candidates should have differences in token lengths. To support the *EM* pattern, pairs of code clones with varying levels of similarities should be suggested as candidates for clone refactoring.

5.2 Future Work

Based on the studies described in this thesis, there are several issues that require further investigation.

Higher speed : For the approach presented in Chapter 3, this study will be extended to aim at achieving high-speed execution time. For this, a distributed approach such as D-CCFinder is being considered. Moreover, the adaptation of other code clone detection tools such as NiCad [80, 79] and Deckard [47], as well as different hash functions such as CRC, may improve the speed.

Tool development : For the investigation presented in Chapter 4, a tool that supports clone refactoring in accordance with results of the investigation will be developed. This developed tool might be widely used for clone refactoring because it is developed based on developers' actual uses of clone refactoring.

Generality : For the both studies, additional OSS systems and industrial software systems belonging to different domains and develop using different programming languages will be investigated to achieve a generality for these studies.

Bibliography

[1] D. Advani, Y. Hassoun, and S. Counsell. Extracting refactoring trends from open-source software and a possible solution to the 'related refactoring' conundrum. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1713–1720, 2006.

[2] G. Antoniol, M. Di Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of the 7th International Workshop on the Principles of Software Evolution*, IWPSE '04, pages 31–40, 2004.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval: The Concepts and Technology behind Search (2nd Edition)*. Addison Wesley, 2011.

[4] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the 2th Working Conference on Reverse Engineering*, WCRE '95, pages 86–95, 1995.

[5] B. S. Baker. Finding clones with dup: Analysis of an experiment. *IEEE Transactions on Software Engineering*, 33(9):608–621, sep 2007.

[6] M. Balint, T. Girba, and R. Marinescu. How developers copy. In *Proceedings of the 14th IEEE International Conference on Program Comprehension, 2006*, ICPC '06, pages 56–68, 2006.

[7] 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, July 2009.

[8] 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 on European Software Engineering Conference and*

the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, ESEC-FSE companion '07, pages 513–516, 2007.

[9] G. Bavota, B. D. Carluccio, A. D. Lucia, M. D. Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the IEEE 12th International Working Conference on the source Code Analysis and Manipulation*, SCAM '12, pages 104–113, 2012.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, 1998.

[11] S. Bazrafshan and R. Koschke. An empirical study of clone removals. In *Proceedings of 2013 29th IEEE International Conference on Software Maintenance*, ICSM '13, pages 50–59, 2013.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.

[13] A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer. Search-based detection of high-level model changes. In *ICSM '12*, Proceedings of 2012 28th IEEE International Conference on Software Maintenance, pages 212–221, 2012.

[14] 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.

[15] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 53–62, 2011.

[16] J. Bosch and P. Bosch-Sijtsema. From integration to composition: On the impact of software product lines, global development and ecosystems. *Journal of Systems and Software*, 83(1):67–76, 2010.

[17] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*, SEQUENCES '97, pages 21–29, 1997.

[18] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[19] E. Choi, K. Fujiwara, N. Yoshida, and S. Hayashi. A survey of refactoring detection techniques based on change history analysis. *Computer Software*, 32(1):to appear, 2015. (in Japanese).

[20] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP '06, pages 404–428, 2006.

[21] E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool support for code clone management. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 843–846, 2008.

[22] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching: Research articles. *Journal of Software Maintenance and Evolution*, 18(1):37–58, Jan. 2006.

[23] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching: Research articles. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[24] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings. IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–118, 1999.

[25] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience*, 33(10):933–955, 2003.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.

[27] S. R. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 222–232, 2012.

[28] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[29] K. Fujiwara, K. Fushida, N. Yoshida, and H. Iida. Assessing refactoring instances and the maintainability benefits of them from version archives. In

Proceedings of the 14th International Conference on Product-Focused Software Development and Process Improvement, PROFES 2013, pages 313–323, 6 2013.

[30] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, 2008.

[31] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, 1999.

[32] N. Göde. Clone removal: Fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 33–40, 2010.

[33] N. Göde and J. Harder. Clone stability. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 65–74, 2011.

[34] N. Göde and J. Harder. Oops! . . . I changed it again. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 14–20, 2011.

[35] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 205–214, 2005.

[36] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, 2005.

[37] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets, 2003.

[38] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[39] J. Harder. How multiple developers affect the evolution of code clones. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, ICSM '13, pages 30–39, 2013.

[40] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained version control system for Java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 96–100, 2011.

[41] S. Hayashi, Y. Tsuda, and M. Saeki. Detecting occurrences of refactoring with heuristic search. In *Proceedings of 15th Asia-Pacific Software Engineering Conference*, APSEC '08, pages 453–460, 2008.

[42] S. Hayashi, Y. Tsuda, and M. Saeki. Search-based refactoring detection from source code revisions. *IEICE Transactions on Information and Systems*, E93-D(4):754–762, 2010.

[43] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proceedings of the the 8th IASTED International Conference on Software Engineering and Applications*, SEA '04, pages 222–229, 2004.

[44] Y. Higo, T. Kamiya, S. Kusumoto, K. Inoue, and K. Words. Aries: Refactoring support environment based on code clone analysis. In *Proceedings of the 8th IASTED International Conference on Software Engineering and Applications*, SEA 2004, pages 222–229. ACTA Press, 2004.

[45] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ICSM '2010, pages 1–9, 2010.

[46] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee. Experience of finding inconsistently-changed bugs in code clones of mobile software. In *Proceedings of the 6th International Workshop on Software Clones*, IWSC '12, pages 94–95, 2012.

[47] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[48] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, 2007.

[49] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[50] C. J. Kapser and M. W. Godfrey. ”cloning considered harmful” considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, December 2008.

[51] T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Proceedings of 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE ’11, pages 163–172, 2011.

[52] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE ’11, pages 301–310, 2011.

[53] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE ’11, pages 151–160, 2011.

[54] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: A refactoring reconstruction tool based on logic query templates. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, FSE ’10, pages 371–372, 2010.

[55] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, 2005.

[56] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS ’01, pages 40–56, 2001.

[57] R. Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, 2014.

[58] 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 ’06, pages 253–262, 2006.

[59] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings. Eighth Working Conference on Reverse Engineering*, WCRE ’01, pages 301–309, 2001.

[60] J. Krinke. Is cloned code more stable than non-cloned code. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '08, pages 57–66, 2008.

[61] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[62] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[63] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 106–115, 2007.

[64] R. Mahouachi, M. Kessentini, and M. Ó. Cinnéide. Search-based refactoring detection using software metrics variation. In *Proceedings of the seventh edition of the annual symposium dedicated to Search Based Software Engineering*, SSBSE'13, pages 126–140, 2013.

[65] M. Mandal, C. Roy, and K. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *Proceedings of 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, CSMR-WCRE '14, pages 114–123, 2014.

[66] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, apr 1976.

[67] T. Mende, R. Koschke, and F. Beckwermert. An evaludation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance and Evolution*, 21(2):143–169, 2009.

[68] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1227–1234, 2012.

[69] M. Mondal, C. K. Roy, and K. A. Schneider. Automatic identification of important clones for refactoring and tracking. In *Proceedings of 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 11–20, 2014.

[70] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. In *Proceedings of IEEE 21st International Conference on Program Comprehension*, ICPC '13, pages 93–102, 2013.

[71] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[72] W. F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois, 1992.

[73] J. Pérez and Y. Crespo. Exploring a method to detect behaviour-preserving evolution using graph transformation. In *Proceedings of the Third International ERCIM Symposium on. Software Evolution*, EVOL '07, pages 114–122, 2007.

[74] K. Prete, N. Rachatasumrit, and M. Kim. A catalogue of template refactoring rules. Technical Report UTAUSTINECE-TR-041610, The University of Texas at Austin, 2010.

[75] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 International Conference on Software Maintenance*, ICSM '06, 2010.

[76] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, ICSM '12, pages 357–366, 2012.

[77] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530, August 2012.

[78] D. Rattan, R. K. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[79] C. Roy. Detection and analysis of near-miss software clones. In *Proceedings of IEEE International Conference on Software Maintenance*, ICSM '09, pages 447–450, 2009.

[80] C. Roy and J. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, 2008.

[81] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-541, QUEENS UNIVERSITY*, 541, 2007.

[82] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[83] P. Runeson, C. Andersson, and M. Höst. Test processes in software product evolution: A qualitative survey on the state of practice. *Journal of Software Maintenance*, 15(1):41–59, 2003.

[84] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, SCAM '10, pages 87–96, 2010.

[85] R. Saha, C. Roy, and K. Schneider. An automatic framework for extracting and classifying near-miss clone genealogies. In *Proceedings of 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 293–302, 2011.

[86] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[87] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *Proceedings of the 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 164–173, Sept 2011.

[88] G. Soares, R. Gheyi, E. Murphy-Hill, and B. Johnson. Comparing approaches to analyze refactoring activity on software repositories. *Journal of Systems and Software*, 86(4):1006–1022, 2013.

[89] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE Software*, 27(4):52–57, July 2010.

[90] Q. D. Soetens, J. Pérez, and S. Demeyer. An initial investigation into change-based reconstruction of floss-refactorings. In *Proceedings of 2013 29th IEEE International Conference on Software Maintenance*, ICSM '13, pages 384–387, 2013.

[91] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.

[92] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 377–380, 2007.

[93] S. Thangthumachit, S. Hayashi, and M. Saeki. Understanding source code differences by separating refactoring effects. In *Proceedings of 2011 18th Asia Pacific Software Engineering Conference*, APSEC '11, pages 339–347, 2011.

[94] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle. A multidimensional empirical study on refactoring activity. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, pages 132–146, 2013.

[95] W. Wang and M. W. Godfrey. Investigating intentional clone refactoring. *ECEASST*, 63, 2014.

[96] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 231–240, 2006.

[97] Z. Xing and E. Stroulia. UMLDiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, 2005.

[98] Z. Xing and E. Stroulia. Refactoring detection based on UMLDiff change-facts queries. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 263–274, 2006.

[99] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an Eclipse case study. In *Proceedings of the 22th IEEE International Conference on Software Maintenance*, ICSM '06, pages 458–468, 2006.

[100] Z. Xing and E. Stroulia. Differencing logical UML models. *Automated Software Engineering*, 14(2):215–259, 2007.

[101] Y. Yamanaka, E. Choi, N. Yoshida, and K. Inoue. A high speed function clone detection based on information retrieval techniques. *IPSJ Journal*, 55(10):2245–2255, October 2014.

[102] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano. Applying clone change notification system into an industrial development process. In *Proceedings of the IEEE 21st International Conference on Program Comprehension*, ICPC '13, pages 199–206, 2013. (in Japanese).

[103] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. In *Proceedings of the Third SIAM International Conference on Data Mining*, SDM '03, pages 166–177, 2003.

[104] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Proceedings of 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 23–32, 2011.

[105] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1123–1130, 2013.

[106] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider. Genealogical insights into the facts and fictions of clone removal. *ACM SIGAPP Applied Computing Review*, 13(4):30–42, December 2013.