Structural and Semantic Code Analysis for Program Comprehension

January 2016

Jiachen Yang

Structural and Semantic Code Analysis for Program Comprehension

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2016

Jiachen Yang

Abstract

It is important to understand the software during its development and maintenance. Understanding the internal mechanics of a system implies studying its documentation and source code during a given maintenance task. It is well-known that understanding a software is very time-consuming, as reported in the literatures that up to 60% of the software engineering effort is spent on understanding the software system.

Programming comprehension generally utilizes the information about the system from two analyses sources, which are the static information (also recognized as compile time information) and dynamic information (also recognized as run-time information).

Static analyses gather the information from the available source code. While dynamic analyses record the execution trace by viewing the system as a black box, giving no insights in internal aspects, like conditions under which function are called.

There are two dimensions of information that can be obtained from static analyses to aid program comprehension, namely structural information and semantic information. Structural information refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents, The other dimension, semantic information, refers to the domain specific issues (both problem and development domains) of a software system.

This dissertation addresses these two dimensions of program comprehension with two studies. The first one is a study on a classification model of source code clones, which helps understanding the source code from structural similarity. The latter one focuses on an important aspect of semantic information, which is the purity of the code modules.

The first study focuses on code clones, which can represent the structural similarity of programs. A code clone, or simply a clone, is defined as 'a code fragment that has identical or similar code fragments to one another'. The presence of code clones is pointed out as a bad smell for software maintenance. The reason is that: if we need to make a change in one place, we will probably need to change the

others as well, but we sometimes might miss it.

According to a survey we conducted, users of code detectors tend to classify clones differently based on each user's individual uses, purposes or experiences about code clones. A true code clone for one user may be a false code clone for another user. From this observation, we propose an idea of studying the judgments of each user regarding clones. The proposed technique is a new clone classification method, entitled Filter for Individual user on code Clone Analysis (Fica).

Code clones are classified by Fica according to a comparison of their token type sequences through their similarity, based on the "term frequency – inverse document frequency" (TF-IDF) vector. Fica learns user opinions concerning these code clones from the classification results. In a production environment, adapting the method described in this research will decrease the time that a user spends on analyzing code clones.

The second study focuses on understanding the purity and side effects of a given program. It is difficult for programmers to reuse software components without fully understanding their behavior. The documentation and naming of these components usually focus on intent, i.e., what these functions are required to do, but fails to illustrate their side effects, i.e., how these functions accomplish their tasks. For instance, it is rare for API function signatures or documentation to include information about what global and object states will be modified during an invocation.

It is hard to understand and reuse modularized components, because of the possible side effects in API libraries. For instance, it is usually unclear for programmers whether it is safe to call the APIs across multiple threads. In addition, undocumented API side effects may be changed during software maintenance, making debugging even more challenging in the future.

By understanding side effects in the software libraries, programmers can perform high level refactoring on the source code that is using the functional part of the libraries. For instance, the return value of math functions such as "sin" will be the same results if the same parameter is passed, therefore the results can be cached if the same calculations are performed more than once. Moreover, the calculation without side effects are good candidates for parallelization. However, the purity information is usually missing in external libraries, therefore programmers would risk introducing bugs with such refactorings, for example, caching the results of a function which depends on mutable internal states.

In this study, we we present an approach to infer a function's purity from byte code for later use. Programmers can use effect information to understand a function's side effects in order to reuse it. Furthermore, we conducted a case study on purity guided refactoring based on gathered purity information. We focus on refactoring these pure functions and propose a new category of high-level automatic

refactoring patterns, called purity-guided refactoring. As a case study, we applied a kind of the purity-guided refactoring, namely Memoization refactoring on several open-source libraries in Java. We observed improvements of their performance and preservation of their semantics by profiling the bundled test cases of these libraries.

This dissertation is organized as follows.

Chapter 1 gives the background of this work and an overview of this dissertation. It describes the need for program comprehension and two aspects for aiding program comprehension by static analysis, namely the structural aspect and the semantic aspect.

Chapter 2 introduces the related research around these two aspects.

Chapter 3 proposes a classification model based on applying machine learning on code clones. It builds the described system Fica, which is a web-based system, as a proof of its concept. The system consists of a generalized suffix-tree-based clone detector and a web-based user interface that allows a user to mark detected code clones and shows the ranked result.

Chapter 4 presents a study on the purity and side effects of the functions in Java, helping programmers to reuse the software libraries. It proposes a technique to automatically infer the purity and side effect informations from Java bytecode. This chapter also gives a case study on purity-guided refactoring that utilizes the purity information of functions during the automated refactoring on object-oriented languages such as Java.

Finally, Chapter 5 concludes this dissertation with a summary and directions for future work.

List of Publications

Major Publications

1-1 Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto, "Filtering Clones for Individual User Based on Machine Learning Analysis," In Proc. of the 6th International Workshop on Software Clones (IWSC2012), pages 76-77, Zurich, Switzerland, June 2012.

1-2 Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto, "A Method for Identifying Useful Code Clones with Machine Learning Techniques Based on Similarity between Clones," IPSJ Journal, volume 54, number 2, pages 807-819 February 2013. (in Japanese)

1-3 Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto, "Revealing Purity and Side Effects on Functions for Reusing Java Libraries," In 14th International Conference on Software Reuse, volume LNCS 8919, pages 314-329, Miami (Florida), US, January 2015.

1-4 Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto, "Classification Model for Code Clones Based on Machine Learning," In Empirical Software Engineering, volume 20, issue 4, pages 1095-1125 August 2015.

1-5 Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto, "Towards Purity-Guided Refactoring in Java," In 31st International Conference on Software Maintenance and Evolution, pages 521-525, Bremen, Germany, October 2015.

Related Publications

2-1 Keisuke Hotta, Jiachen Yang, Yoshiki Higo, and Shinji Kusumoto, "How Accurate Is Coarse-Grained Clone Detection?: Comparison with Fine-Grained

Detectors," In Proc. of the 8th International Workshop on Software Clones (IWSC 2014), pages 1-18, Antwerp, Belgium, February 2014.

2-2 Keisuke Hotta, Jiachen Yang, Yoshiki Higo, and Shinji Kusumoto, "An Empirical Study on Accuracy of Coarse-Grained Clone Detection," IPSJ SIG Technical Report, volume 2014-SE-183, number 6, pages 1-8 March 2014. (in Japanese)

2-3 Kenji Yamauchi, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto, "Clustering Commits for Understanding the Intents of Implementation," In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pages 406-410, Victoria, Canada, October 2014.

2-4 Keisuke Hotta, Jiachen Yang, Yoshiki Higo, and Shinji Kusumoto, "An Empirical Study on Accuracy of Coarse-Grained Clone Detection," In IPSJ Journal, volume 56, number 2, pages 580-592 Feburary 2015. (in Japanese)

2-5 Naoto Ogura, Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto, "A Study on Relationship between Changing Purity of Java Methods and Inducing/Fixing" In Forum on Information Technology 2015, volume 1, pages 187-188 September 2015. (in Japanese)

Acknowledgements

In this opportunity, I am fortunate to have received advises and helps from many people.

First, I would like to express my sincere gratitude to my supervisor, Professor Shinji Kusumoto, for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. He has spent much time and effort for this work. I could not have imagined having a better advisor and mentor for my Ph.D study.

I would like to thank Dean and Professor Katsuro Inoue for his helpful comments, valuable questions, and kind advice for this work. His broad view of knowledge leads this work to complete.

I also would like to express my sincere appreciation to Professor Toshimitsu Masuzawa for his valuable comments and helpful suggestions on this dissertation. His careful annotations and corrections helps this work in various perspectives.

I would like to express my heartfelt appreciation to Associate Professor Yoshiki Higo for his great contributions throughout this work. His zealous coaching have strongly encouraged me, and his helpful comments and suggestions made it possible to complete this work. His guidance helped me in all the time of research and writing of this thesis. His brilliant ideas corrected me for several times when I met difficulties. I do not believe this work would have been possible without his help.

I would like to thank Assistant Professor Hiroshi Igaki in Osaka Institute of Technology, for his valuable advise for this work. I also would like to thank researchers on software engineering who have given me helpful comments and encouragement.

I would like to express my appreciation to Dr.Keisuke Hotta in Fujitsu, for his great contributions to my studies. He participated in all my researches as a coworker and advisor even after he graduated from our laboratory. He discussed details of my study throughout days and nights and help me with my thesis. This work would be impossible without his advices.

In especially I would like to thank all participants in the survey of this work, including the members of our laboratory and all the friends of researchers in the

software engineering field. They spent valuable efforts on my study and gave feedbacks to my research.

My heartfelt thanks go to Mr.Hiroaki Murakami as a friend as well as a contemporary. He kindly provided advices and suggestions both in my study and in my life in Japan.

I would like to express my appreciation to all my contemporaries during my study in Osaka University, including but not limited in Mr.Tomoya Ishihara, Mr.Shuhei Kimura, Mr.Yukihiro Sasaki, and Mr.Hiroaki Shimba. I learned many things from them, including how to work on research problems and how to enjoy the work.

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

I feel deep appreciations for all the other labmates and graduates of our laboratory, for the sleepless nights we were working together before deadlines, and for all the fun we have had. I think I am very fortunate because I could work with such great colleagues.

I would like to thank my fiends in Arch Linux open source community. Their volunteer contributions provided me a rock solid operating system and an advanced programming environment. They gave me advices on both my study and my life.

Finally but most importantly, I would like to thank my beloved wife, Hong Liu. She married to me at a time when I was in school and had nothing to provide her. She takes care my daily life with her own savings and encourages my research with her excellent English skill. I would like to thank my parents for raising me and their supporting of my study aboard. I inherit their intelligent and hard working.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Program comprehension is an important activity in software engineering with many approaches for understanding different aspects of the software products [1]. It is a major factor in providing effective software maintenance and enabling successful evolution of computer systems [2].

There are five types of tasks are commonly associated with software maintenance and evolution [2], and their types are: adaptive, perfective, and corrective maintenance; reuse; and code leverage. Each task type typically involves certain activities, as described in Table 1.1.

Some activities, such as understanding the system or problem, are common to several task types. To analyze the cognitive processes behind these tasks, researchers have developed several models. These models cover different aspects of the software system to aid the programmer during maintenance process.

One of the most important aspects of software maintenance is to understand the software at hand. Understanding a system's inner workings implies studying such artifacts as source code and documentation in order to gain a sufficient level of understanding for a given maintenance task. This program comprehension process is known to be very time-consuming, and it is reported that up to 60% of the software engineering effort is spent on understanding the software system at hand [3].

Programming comprehension generally utilizes the information about the system from two analyses sources, which are the **static information** (also recognized as **compile time** information) and **dynamic information** (also recognized as **runtime** information) [4].

Static analysis gathers the information from the available source code. While dynamic analysis records the execution trace by viewing the system as a black

box, giving no insights in internal aspects, like conditions under which function are called [5].

This dissertation focuses on the **static** aspect of the information. This aspect can be divided further into two dimensions, which are the **structural** information and **semantic** information. The final goal of this study is to aid programming comprehension with the statical information in these two dimensions.

Table 1.1: Tasks associated with software maintenance and evolution

Task Types	Activities During Software Maintenance
Adaptive	Understand system
	Define adaptation requirements
	Develop preliminary and detailed adaptation
	design
	Code changes
	Debug
	Regression tests
Perfective	Understand system
	Diagnosis and requirements definition for improvements
	Develop preliminary and detailed perfective design
	Code changes/additions
	Debug
	Regression tests
Corrective	Understand system
	Generate/evaluate hypotheses concerning problem
	Repair code
	Regression tests
Reuse	Understand problem, find solution based on close
	fit with reusable components
	Locate components
	Integrate components
Code leverage	Understand problem, find solution based on predefined components
	Reconfigure solution to increase likelihood of reusing components
	Obtain and modify predefined components
	Integrate modified components

1.2 Overview of the Research

There are two dimensions of information that can be obtained from a static analysis to aid program comprehension, namely **structural** information and **semantic** information [6]. **Structural** information refers to issues such as the actual syntactic structure of the program along with the control and data flow that it represents. The other dimension, **semantic** information, refers to the domain specific issues (both problem and development domains) of a software system.

This dissertation addresses these two dimensions of program comprehension with two studies. The first one is a study on a classification model of source code clones, which helps understanding the source code from structural similarity. The latter one focuses on one important aspect of semantic information, which is the purity of the code fragments.

1.2.1 Comprehension with Structural Information

This study focuses on code clones, which can represent the structural similarity of programs. A code clone, or simply a clone, is defined as '*a code fragment that has identical or similar code fragments to one another*' in [7] and [8]. The presence of code clones is pointed out as a *bad smell* for software maintenance [9]. The reason is that: if we need to make a change in one place, we will probably need to change the others as well, but we sometimes might miss it [10].

Great efforts have been made to detect identical or similar code fragments from the source code of software. During development, code clones are introduced into software systems by various operations, namely, copy-and-paste or machine generated source code. Because code cloning is easy and inexpensive, it can make software development faster, especially for "experimental" development. Therefore, various techniques and tools to detect code clones automatically have been proposed by many researchers, such as [11], [12] and [10]. Furthermore, these code clone detectors were studied by [13].

Classification Model for Code Clones Based on Machine Learning

By applying code clone detectors to the source code, users such as programmers can obtain a list of all code clones for a given code fragment. Such a list is useful during modifications to the source code. However, results from code clone detectors may contain plentiful useless code clones, and judging whether each code clone is useful varies from user to user based on the user's purpose for the clone. Therefore, it is difficult to adjust the parameters of code clone detectors to match the individual purposes of the users. It is also a tedious task to analyze the entire

list generated by the code clone detector. Clone detection tools (CDTs) usually generate a long list of clones from source code. A small portion of these clones are regarded as true code clones by a programmer, while the others are considered as false positives that occasionally share an identical program structure.

A previous study [14] suggested that cloned code fragments are less modified than non-cloned code fragments. The results indicate that more clones in existing source code are stable instead of volatile. These stable clones should be filtered from the results of CDTs before applying simultaneous modifications to the volatile ones. Meanwhile, [13] manually created a set of baseline code clones to evaluate several CDTs. While such a set of baseline code clones demanding huge amount of manpower, the accuracy of these baseline code clones depends on the person who makes the judgment, as our research will verified. Manual baseline clones have the following issues.

- They demand a huge amount of manpower.
- Their accuracy depends on the person who made the judgments.

According to a survey we conducted, users of CDTs tended to classify clones differently based on each user's individual uses, purposes or experience about code clones. A true code clone for a user may be a false code clone for another user. From this observation, we propose an idea of studying the judgments of each user regarding clones. The proposed technique is a new clone classification approach, entitled Filter for Individual user on code Clone Analysis (FICA).

Code clones are classified by FICA according to a comparison of their token type sequences through their similarity, based on the "term frequency – inverse document frequency" (TF-IDF) vector. FICA learns the user opinions concerning these code clones from the classification results. In a production environment, adapting the proposed technique will decrease the time that a user spends on analyzing code clones.

From the experimental results, we made several observations:

1. Users agree that false positive code clones are likely to fall into several categories, such as a list of assignment statements or a list of function declarations.
2. Users agree that true positive code clones are more diverse than false positives, which means that "real" code clones are less similar than those "appear-to-be" code clones detected by clone detectors.
3. Users with more experience on code clones are more likely to agree with each other compared to users with less experience.

4. Generally, the minimum required size of the training set grows linearly with the number of categories that clone sets fall into. This is the main reason that the proposed technique will decrease the analyzing time.

1.2.2 Comprehension with Semantic Information

It is difficult for programmers to reuse software components without fully understanding their behavior. The documentation and naming of these components usually focus on *intent*, i.e., what these functions are required to do, but fails to illustrate their *side effects*, i.e., how these functions accomplish their tasks [15]. For instance, it is rare for API function signatures or documentation to include information about what global and object states will be changed during an invocation.

It is difficult in some scenarios to understand and reuse modularized components, because of the possible side effects in API libraries. For instance, it is usually unclear for programmers whether it is safe to call the APIs across multiple threads. In addition, undocumented API side effects may be changed during software maintenance, making debugging even more challenging in the future [16].

By understanding of side effects in the software libraries, programmers can perform high level refactoring on the source code that is using the functional part of the libraries. For instance, return values of math functions such as `sin` will be the same results if the same parameter is passed, therefore the results can be cached if the same calculations are performed more than once. Moreover, the calculations without side effects are good candidates for parallelization [17]. However, the purity information is usually missing in external libraries, therefore programmers would risk introducing bugs with such refactorings, for example, caching the results of a function which depends on the mutable internal state.

There are two studies that focused on the purity aspect as the semantic information in the programs. The first study infers the purity information from the existing program. The second study utilizes the purity information by applying a refactoring called memoriation on the program.

Revealing Purity and Side Effects on Functions for Reusing Java Libraries

This study presents an approach to infer a function's purity from byte code for later use. Programmers can use effect information to understand a function's side effects in order to reuse it. For example, the approach can help to decide whether it is safe to cache or parallel a time-consuming calculation.

The contributions of this research include:

- An extended definition of purity as *stateless* or *stateful* in object-oriented languages such as Java.

- An approach to automatically infer purity and side effects,
- A concrete implementation for Java bytecode,
- A set of method annotations that document the details of effects such as return value dependencies or changing of variable state, for programmers to understand the effects.
- Experiments on well-known open source software libraries with different scale and characteristic.

In our experiments, we found that 24–44% of the methods in the evaluated open source Java libraries were pure. Also, we observed methods that should be pure in theory but not in the implementation, and revealed tricks or potential bugs in the implementation during the experiment of our approach.

We achieved the same percentage of pure functions with the existing study without a manually created white-list, and we revealed which side effects these functions were generating which would not found in the existing studies. We focused on revealing these side effect information on real world software libraries to be used by the programmers and tools.

Furthermore, in this study we conducted a case study on *purity guided refactoring* based on the gathered purity information. Source code refactoring is generally defined as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [18]. Refactoring is one of many important tasks in software development and maintenance. Integrated Development Environments (shortened as IDEs) generally provide supports for common refactoring operations, which improve maintainability, performance or both, during the software development process. Machine-aid refactoring is so important that it is promoted as one of the best practices in both the Extreme programming [19] and Agile Software development [20], so that both of them heavily depend on automatic tools to perform refactoring during the development process.

It is emphasized to preserve the external behavior while performing refactoring so that they can be safely conducted without being evaluated about the breaking changes. However, it is hard to reason about the semantic behavior of a code fragment automatically by refactoring tools, due to the lack of semantic information from the traditional static analysis. Therefore, refactoring tools provided by IDEs generally take more conservative approaches by checking the syntactic structure of a given code fragment only in the pre-conditions of the refactoring operations.

Determining the possible semantic behavior from only the syntactic structure of a given code fragment puts heavy limitations on the possible refactoring patterns. Therefore, the refactoring tools provided by IDEs are limited to low-level

structural restructuring on the code fragment, such as *Rename Method* or *Extract Method*. Although there are more high-level refactoring patterns widely recognized and adopted by programmers, such as *Replace Loop with Collection Closure Method* [21], currently they are not provided automatically by tools or IDEs and are applied manually by programmers.

In the other hand, preservation of semantic behavior can be statically checked for a certain part of the source code, namely the code that use pure functions. Pure functions are the functions that do not have observable side effects during the execution. With the property to be side effect free, whether refactoring changed the semantic behavior of the code fragments that use pure functions can be easily checked by tools. Therefore, more refactoring patterns become available when the purity information is inferred from the source code. For instance, return values of math functions such as `sin` will be the same results if the same parameters are passed, therefore the result can be cached if the same calculation is performed more than once. Moreover, the calculation without side effects are good candidates for parallelization [17].

In this research, we focus on refactoring these pure functions and propose a new category of high-level automatic refactoring patterns, called *purity-guided refactoring*. As a case study, we applied a kind of the purity-guided refactoring, namely *Memoization* refactoring on several open-source libraries in Java. We observed the improvements of the performance and the preservation of semantics by profiling the bundled test cases of these libraries.

1.3 Overview of the Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 describes some of the related works that also focus on program comprehension based on static analyses.

Chapter 3 proposes a classification model based on applying machine learning on code clones. It builds the described system FICA, which is a web-based system, as a proof of concept. The system consists of a generalized suffix-tree-based CDT and a web-based user interface that allows the user to mark detected code clones and shows the ranked results.

Chapter 4 presents a study on the purity and side effects of the functions in Java, helping programmers to reuse the software libraries. It proposes a technique to automatically infer the purity and side effect informations from Java bytecode. Further, this chapter focus on *purity-guided refactoring* that utilize the purity information of functions during the automated refactoring on object-oriented languages such as Java. It conducts an experiment on a refactoring pattern called *Memoiza-*

tion that caches calculation results for pure functions.

Finally, Chapter 5 concludes this dissertation with a summary and directions for future work.

Chapter 2

Related Works

As described in the introduction, this dissertation aims to help statical program comprehension from two major aspects, which are the structural code analysis and semantic code analysis. In this chapter, I will describe the related works corresponding to these two aspects.

2.1 Structural Code Analysis

Some related works have reported on combining machine learning or text mining techniques with code detection to classify or clustering code clones. As a complement to existing code clone detection methods, [22] proposed a method to identify high-level concept clones, such as different implementations of the algorithm of linked lists, directly from identifiers and comments from source code as a new method of clone detection. A similar approach by [23] found semantic topics instead of clones from comments and identities from source code.

Another method proposed by [24] shares some common ideas with the above two methods. They compare code clones by using information retrieved from identifiers; such an approach focuses more on the semantic information or behavior of source code rather than the syntactic or structural information of source code. In contrast, the method described in this paper compares the tokenized source code of code clones, which focuses on the syntactic similarity between code clones. The works by [25] share many general ideas with our work, although their focus is finding clones with bugs. They compare the structure of code clones and thus require an AST-based CDT to extract structure information. Furthermore, although not mentioned in their paper, we contacted the first author and confirmed that the judgment of whether a clone is buggy depends on the opinion of the authors, and thus relies on more subjective judgments.

Besides comparing text similarity, other methods have been proposed to filter unneeded code clones from the detection result. [26, 27] proposed a metric-based approach to identify code clones with higher refactoring opportunities. Their method calculates six different metrics for each code clone and then represents the plotted graph of these metrics as a user-friendly interface to allow users to filter out the unneeded code clones. This user-defined metric-based filtering method was further automated by [28]. Koschke is targeting a different objective, which is identification of license violations, but has also used metrics and machine learning algorithms on those metrics to form a decision tree. The result of the decision tree limits the types of metrics to only two, which are PS (Parameter Similarity) and NR (Not Repeat). These metric-based methods all have a limited identification target and thus result in higher accuracy than the method proposed in this paper. More recent research [29] uses a search-based solution to repeatedly change the configurations of several well-known code clone detectors in order to improve the result. They experimented on the true clone dataset of [13]. They tried to improve the general agreement as well as individual agreement results for each software project. We have a different target, which is filtering out false code clones for an individual user's purpose. Therefore, we took a different approach.

Other works on the classification or taxonomy of code clones focus on proposing fixed schemes of common clone categories. [30] proposed a classification scheme for clone methods with 18 different categories. The categories detail what kind of syntax elements have been changed and also how much of the method has been duplicated. [13] defined three different clone types (exact clones, parameterized clones, and clones that have had more extensive edits) for the sake of comparison between different detection tools. Their aim was to test the detection and categorization capabilities of different tools.

Several visualization methods have also been proposed to aid the understanding of code clones. A popular approach that has been implemented in most CDTs is the scatter plot [31, 32]. A scatter plot is useful for selecting and viewing clones on a project scale, but it is difficult to illustrate the relations among clones. Johnson proposed a method in [11] that uses Hasse Diagrams to illustrate clusters of files that contain code clones. Johnson also proposed navigating files containing clone classes by using hyper-linked web pages [33]. The force-directed graph used in the FICA system described in this paper is highly interactive and has all the benefits of a modern web system. Thus, FICA should be useful in the analysis of code clones.

[34] proposed a framework for understanding clone information in large software systems by using a data mining technique framework that mines clone information from the clone candidates produced by CCFinder. First, a lightweight text similarity is applied to filter out false positive clones. Second, various levels of system abstraction are used to further scale down the filtered clone candidates. Finally,

an interactive visualization is provided to present, explore and query the clone candidates, as with the directory structure of a software system. Their method shares some common features with FICA. Compared to their method, FICA depends more on the marks of code clones by users, and thus is more customized for the individual user but also requires more operations by users.

[35] proposed a machine learning based method for predicting the harmfulness of code clones. Their predictor employs a Bayesian Network algorithm and learns three categories of metrics from code clone fragments, namely, the history, code and destination metrics. Their work defines the harmfulness of a clone set based on inconsistent changes among the clones, which is an objective standard. As in the idea of our APPF and APNF, they also define conservative and aggressive scenarios and evaluate their effectiveness accordingly. Although they noted in their paper that each category of metric contributes to the overall effectiveness, they do not illustrate the contribution of the literal similarity of source code. Compared to their work, our FICA system shows that the literal similarity alone is enough to classify the clones for varied purposes.

2.2 Semantic Code Analysis

This research focuses on targeting one paradigm of programming, which is called *statically typed type-safe object-oriented programming* languages. This means that each variable in the program is associated with a statically defined type and that operations on these variables can be checked in compiler-time. Wild pointers are not allowed in these languages, which gives us the ability to consider the reference alias safely. Many industrial programming languages fall into this category, including C#, Java, and Scala. Dynamic-typed languages such as PHP, Python, and Ruby are not covered, where variable types are determined at runtime. Note that although Scala eliminates most of the necessity of defining the variable types, the variable types are statically inferred by the compiler, rather than determined at runtime.

Note also that there are infrastructures that may break type safety in the above languages, such as the `unsafe` keyword in C# or the "dynamic invoke" feature in Java 7.

There are studies of automatic purity analyzers on unmodified syntax. Sălcianu, et al. present a purity analyzer for Java in [36], which uses an inter-procedural pointer analysis [37] and escape analysis to infer reference immutability. Similar to our approach, they verify the purity of functions, but their pointer and escape analysis relies on a whole program analysis starting from a `main` entry point, which is not always available for software libraries. We have compared the purity

result of our approach with their study using the same benchmark in Section 4.7.2. JPure [38] eliminated the need for reference immutability inference by introducing `pure`, `fresh` and `local` annotations, which lead to a more restrictive definition of purity, and loses the exact information for effects.

Both studies focus on analyzing of purity only, and does not expose effects informations outside their toolchain. Compared with these studies, our approach uses lexical state accessor analysis, which will hopefully combine the modularity of JPure by illuminating the need for inter-procedure analysis, and the flexibility of reference immutability with the availability of effect informations. Also neither of these two studies further classify the pure functions into *Stateless* and *Stateful* as we do. Further, we provide a heuristic approach to detect cache semantics in member fields, thus eliminating the need of a manually provided white-list.

Mettler, et al. [39] take a different approach. Instead of extending the syntax of an existing language, they created a subset of Java called Joe-E. As one application of Joe-E, they proposed [40] to verify the purity of functions by only permitting immutable objects in the function signature. Kjolstad, et al. [17] proposed a technique to transform a mutable class into an immutable one. They utilized an escaping and entering analysis similar to [36]. These two studies are similar to each other as they completely eliminate the mutable states from target source codes, which is not always acceptable in general programming scenarios, thus limits their application. Comparing to these two studies, our technique can be performed on the real world software libraries, even without the source code. Therefore we are more suitable for comprehension the legacy code bases.

Xu et al. [41] have been studied dynamic purity analysis and developed a memoization technique during online purity analysis inside JVM. Rito et al. [42] proposed a memoization technique by using software transactional memory to find altered values during execution. These two studies shared a large part of the background with our study. However, we focused on static analysis and aim to source code level refactoring.

There are studies to convert a procedure style program to a more pure functional style program. Mettler, et al. created a subset of Java called Joe-E [39]. As one application of Joe-E, they proposed a method to verify the purity of functions by only permitting immutable objects in the function signatures [40]. Kjolstad, et al. proposed a technique to transform a mutable class into an immutable one [17]. Applying the approaches of these studies will result in more pure functions in the source code, thus they can increase the refactoring candidates of this research.

Chapter 3

Classification Model for Code Clones Based on Machine Learning

3.1 Introduction

Great efforts have been made to detect identical or similar code fragments from the source code of software. These code fragments are called "code clones" or simply "clones", as defined in [7] and [8]. During development, code clones are introduced into software systems by various operations, namely, copy-and-paste or machine generated source code. Because code cloning is easy and inexpensive, it can make software development faster, especially for "experimental" development. However, despite their common occurrence in software development, code clones are generally considered harmful, as they make software maintenance more difficult and indicate poor quality of the source code. If we modify a code fragment, it is necessary to check whether each of the corresponding code clones needs simultaneous modifications. Therefore, various techniques and tools to detect code clones automatically have been proposed by many researchers, such as [11], [12] and [10]. Furthermore, these code clone detectors were studied by [13], who defined the widely used categories of clones based on the following similarities:

Type-1 An exact copy of a code fragment except for white spaces and comments in the source code.

Type-2 A syntactically identical copy where only user-defined identifiers such as variable names are changed.

Type-3 A modified copy of a type-2 clone where statements are added, removed

or modified. Also, a type-3 clone can be viewed as a type-2 clone with gaps in-between.

By applying code clone detectors to the source code, users such as programmers can obtain a list of all code clones for a given code fragment. Such a list is useful during modifications to the source code. However, results from code clone detectors may contain plentiful useless code clones, and judging whether each code clone is useful varies from user to user based on the user's purpose for the clone. Therefore, it is difficult to adjust the parameters of code clone detectors to match the individual purposes of the users. It is also a tedious task to analyze the entire list generated by the code clone detector. Clone detection tools (CDTs) usually generate a long list of clones from source code. A small portion of these clones are regarded as true code clones by a programmer, while the others are considered as false positives that occasionally share an identical program structure.

A previous study [14] suggested that cloned code fragments are less modified than non-cloned code fragments. This result indicates that more clones in existing source code are stable instead of volatile. These stable clones should be filtered from the result of CDTs before applying simultaneous modifications to the volatile ones. Meanwhile, [13] manually created a set of baseline code clones to evaluate several CDTs. Manual baseline clones have the following issues.

- They demand a huge amount of manpower.
- Their accuracy depends on the person who made the judgments.

Several methods have been proposed to filter out unneeded clones, including the metric-based method described in [27]. While these methods can summarize a general standard of true code clones, the users must have professional knowledge, such as what all the metrics mean. Furthermore, it is hard to fit the extraction method of refactoring to individual use cases, such as filtering only the code clones. Tairas and Gray proposed a classification method in [24], which classifies the code clones by identifier names rather than the text similarity of identical clone fragments. The purpose of our study is the same as those of filtering or classification methods: to find true positives from the result of CDTs during software maintenance. However, we take a different approach to find the true positives.

Finding clones that are suitable for refactoring has also been studied. Previous studies such as [43] and [44] suggested that the clone detection process should be augmented with semantic information in order to be useful for refactoring. [45, 46] investigated several code characteristics from different clone detection tools for type-3 clones. Their studies pointed out that clone detection tools may improve the results with feedback from the users, and that the best metrics for different datasets

include source code similarity, token value similarity and identifier similarity, all of which are measured by the edit distance. Our research is based on their observations, though we compare the similarity of the token sequences by a more efficient approach and target the removal of unwanted type-2 code clones.

According to a survey we conducted, users of CDTs tend to classify clones differently based on each user's individual uses, purposes or experience about code clones. A true code clone for one user may be a false code clone for another user. From this observation, we propose the idea of studying the judgments of each user regarding clones. The result is a new clone classification method, entitled Filter for Individual user on code Clone Analysis (FICA).

The code clones are classified by FICA according to a comparison of their token type sequences through their similarity, based on the "term frequency – inverse document frequency" (TF-IDF) vector. FICA learns the user opinions concerning these code clones from the classification result. In a production environment, adapting the method described in this research will decrease the time that a user spends on analyzing code clones.

From the experimental results, we made several observations:

1. Users agree that false positive code clones are likely to fall into several categories, such as a list of assignment statements or a list of function declarations.

2. Users agree that true positive code clones are more diverse than false positives.

3. Generally, the minimum required size of the training set grows linearly with the number of categories that clone sets fall into, which is less than the total number of detected clone sets.

The contributions of this work include:

- A machine learning model based on clone similarity.
- An approach to consider each individual user in code clone analysis.
- An experiment with 32 participants.
- Several important observations from the experiment.

In the paper, Section 3.2 introduces a motivating example that led to this research. Section 3.3 introduces the working process for the proposed method, FICA, and describes how it could help the user to classify code clones. Then, Section 3.4 discusses in detail the proposed method and algorithms that FICA uses. Finally,

```
2717  wcstr = 0;
2718  slen = mbstowcs (wcstr, s, 0);
2719  if (slen == -1)
2720    slen = 0;
2721  wcstr = (wchar_t *)xmalloc (sizeof (wchar_t) * (slen + 1));
2722  mbstowcs (wcstr, s, slen + 1);
2723  wclen = wcswidth (wcstr, slen);
2724  free (wcstr);
2725  return ((int)wclen);
```

Figure 3.1: execute_cmd.c in bash-4.2

Section 3.5 shows the result of an online survey that we conducted to evaluate our proposed method.

3.2 Motivating Example

We conducted a survey with several students[1]. We provided the students with 105 clone sets detected from `bash-4.2` from the result of CDTs on source code in the C language, then asked them whether these code clones were true code clones, based on their own experience and motivation. Table 3.1 shows a part of the result. In this table, a code clone set is marked as `O` if a student thought this clone is a true code clone or as `X` if not. We can see from this table that the attitudes toward these clones varied from person to person.

As an example, the source of a clone with ID 5 is shown in Figures 3.1 and

Table 3.1: Survey of clones in bash-4.2 (O: true code clone, X: false code clone)

Clone ID	Participant			
	Y	S	M	U
1	X	O	O	O
2	X	X	O	O
3	O	X	X	O
4	X	O	X	O
5	X	O	X	O
...	...			
O count	5	24	23	25
X count	100	81	82	80

[1] All students are from the Graduate School of Information Science and Technology, Osaka University

```
1100  if (wcharlist == 0)
1101  {
1102    size_t len;
1103    len = mbstowcs (wcharlist, charlist, 0);
1104    if (len == -1)
1105      len = 0;
1106    wcharlist = (wchar_t *)xmalloc (sizeof (wchar_t) * (len + 1));
1107    mbstowcs (wcharlist, charlist, len + 1);
1108  }
```

Figure 3.2: subst.c in bash-4.2

3.2. These blocks of code convert a multi-byte string to a wide-char string in C code. Because their functions are identical, S and U are considered able to be merged, so these clones are true code clones. Meanwhile, Y and M considered the fact that Figure 3.2 is a code fragment in a larger function having more than 100 LOCs. Because it may be difficult to apply refactoring, these clones should be false positives of the CDT.

Moreover, from Table 3.1 we can see that Y was more strict than the three other students. In the comment to this survey, Y mentioned that only clones that contain an entire body of a C function are candidates. This unique standard was also reflected in all five true clones he chose.

Based on the above motivating survey, we feel that a classifier based on the content of source code is necessary during a clone analysis to meet the individual usage of each user. Therefore, we developed the classification method described in the following sections.

3.3 FICA System

In this section, we introduce the general working process of the FICA system, which ranks detected clones based on the historical behavior of a particular user. As a complement to existing CDTs that filter unexpected clones, FICA is designed as an online supervised machine learning system, which means that the user should first classify a small portion of the input manually, and then FICA gradually adjusts its prediction while more input is given by the user. By adapting this process into a traditional code clone analyzing environment, the desired code clones could be presented to the user more quickly.

3.3.1 Overall Workflow of FICA System

The overall workflow of FICA is described in Figure 3.3 and listed as follows.

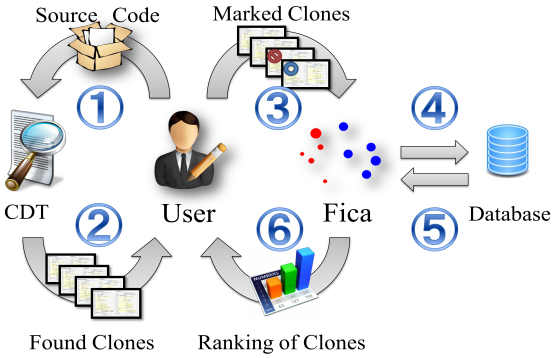


Figure 3.3: Overall workflow of FICA with a CDT

1. A user submits source code to a CDT.
2. The CDT detects a set of clones in the source code.
3. The user marks some of these clones as true or false clones according to her/his own judgment, and then submits these marked clones to FICA as a profile.
4. FICA records the marked clones in its own database.
5. Meanwhile, FICA studies the characteristics of all these marked clones by using machine learning algorithms.
6. FICA ranks other unmarked clones based on the result of machine learning, which predicts the probability that each clone is relevant to the user.
7. The user can further adjust the marks on code clones and resubmit them to FICA to obtain a better prediction. FICA will also record these patterns that it has learned in a database associated with the particular user profile so that further predictions can be made based on earlier decisions.

The main purpose of the FICA system is to find the true code clones from the result of a CDT, or in reverse, to filter out the false code clones, by using a classification method. As we have shown in the motivating example, whether a code clone is true or false largely depends on the user's subjective purpose. Our FICA classifier is designed to consider this subjective factor and to be helpful regardless of different users.

3.3.2 Classification Based on Token Sequence of Clones

FICA utilizes a CDT to obtain a list of code clones from source code. The resulting output from the CDT should contain the following information for each code clone:

1. Positions of the code clone fragment in the original source code.
2. A sequence of tokens included in the code clone fragment.

Most existing token-based CDTs meet this requirement. Text-based CDTs usually output only the positional information of code clones; in this case, code clone fragments need to be parsed through a lexical analyzer to obtain the required lexical token sequences.

FICA compares the exact matched token sequence of reported clone sets from CDTs. These clones are known as type-2 clones in other literature. Type-2 clone instances from a clone set should have the same token sequence. FICA compares the similarity of the token sequences for the following reasons:

- The similarity of the token sequences was pointed out by [45] as one of the best characteristics for filtering out unneeded clones, which are referred to as rejected candidates in that study.
- The types of each token in the token sequence are identical among all instances of a type-2 clone set.
- The required token sequence is obtainable as a side-product of a token-based CDT.

By using the token sequence from CDT, FICA can save the time of reparsing the source code, which should have been done by the CDT. Therefore, FICA can be integrated with the CDT as a complementary system.

3.3.3 Marking Clones Manually

To be used by FICA as an initial training set, only a small set of clones found by CDT is required to be marked manually by the user of FICA. The considered types of marks on clones can be Boolean, numerical, or tags:

- Boolean clones are marked based on whether they are true code clones.
- Range clones are marked with a numerical likelihood to be true code clones.
- Tagged clones are marked by one of several tags or categories by the users based on their use, such as refactoring, issue tracking ID, etc.

As the most simple case, users need to tell FICA what kind of clones should be treated as true code clones. Numerical type marks are an extension of Boolean type marks used in the situation that the user wants to say finding clone A is more useful than finding clone B but less useful than finding clone C. Tag type marks can be considered as a possible extension of Boolean type ones that involve multiple choices. For example, clones marked with the tag `refactoring` are suitable candidates for refactoring, or clones marked with the tag `buggy` are clones that tend to have bugs.

Also, a FICA user is allowed to have multiple profiles in the system, with each profile representing a use case of code clones. Profiles should be trained separately and FICA will treat them as individual users.

3.3.4 Machine Learning

Receiving the clones from the CDT and the marks from the user, FICA studies the characteristic of the marked clones by calculating the similarity of the lexical token sequence of these clones. This step employs a machine learning algorithm that is widely used in natural language processing or text mining. The algorithm used is similar to that used in GMail for detecting spam emails or the CJK Input Methods used in suggesting available input candidates. By comparing the similarity of marked and unmarked clones, FICA can thus predict the probability of an unmarked clone set being a true positive. Details on the machine learning model and algorithm are described in Section 3.4.

3.3.5 Cycle of Supervised Learning

FICA returns the predicted marks for all remaining clones by ranking or calculating the probability of the user considering them as true clones. The user is allowed to correct some of these predictions and resubmit them to FICA to obtain a better prediction. This is known as the cycle of supervised learning. Eventually, FICA is trained to filter all clones according to the interest or purpose of the user. Furthermore, the patterns learned by FICA are also recorded in a database associated with the particular user. As a result, further predictions on clones can be made based on earlier decisions of the same user.

3.3.6 Comparing to Metrics-based Clone Filtering

Before this research, many methods have been proposed to filter unneeded code clones from the detection result using a metric-based approach. [26, 27] proposed a metric-based approach to identify code clones with higher refactoring opportunities. Their method calculates six different metrics for each code clone and then

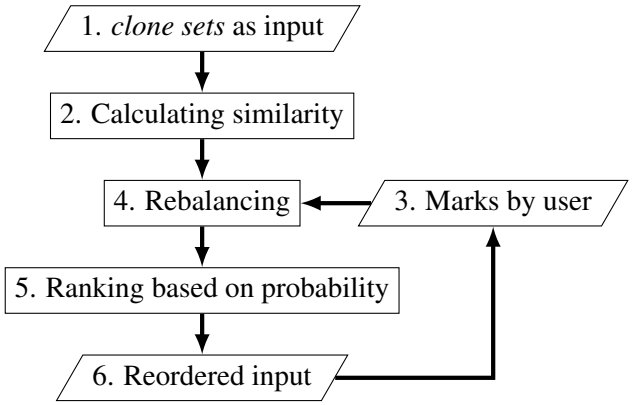


Figure 3.4: Classification process in FICA. Parallelograms represent input data or output data, and rectangles represent the procedures of FICA.

represents the plotted graph of these metrics as a user-friendly interface to allow users to filter out the unneeded code clones. One of the best metrics called Ratio of Non-Repeated code sequence (shortened as RNR) is reported to be useful in filtering unneeded code clones.

This user-defined metric-based filtering method was further automated by [28]. Koschke is targeting a different objective, which is identification of license violations, but has also used metrics and machine learning algorithms on those metrics to form a decision tree. The result of the decision tree limits the types of metrics to only two, which are PS (Parameter Similarity) and NR (Not Repeat).

These metric-based methods all have a limited identification target, thus while they result in higher reported accuracy comparing the method proposed in this paper, the users of these methods are required to have a basic understanding of the meaning of these metrics. For example, filtering clones using RNR is useful only when the users wanted to filtering those code clones with repeated code sequence.

Comparing to these metric-based methods, our proposed method do not require the clone related knowledge from user, and can be used on different targets.

3.4 Machine Learning Method

The classification process in FICA is described in Figure 3.4. FICA can be viewed as a supervised machine learning process with these steps:

1. Retrieving a list of clone sets by a predefined order from a CDT.

2. Calculating the text similarity among those clone sets by their cosine simi-

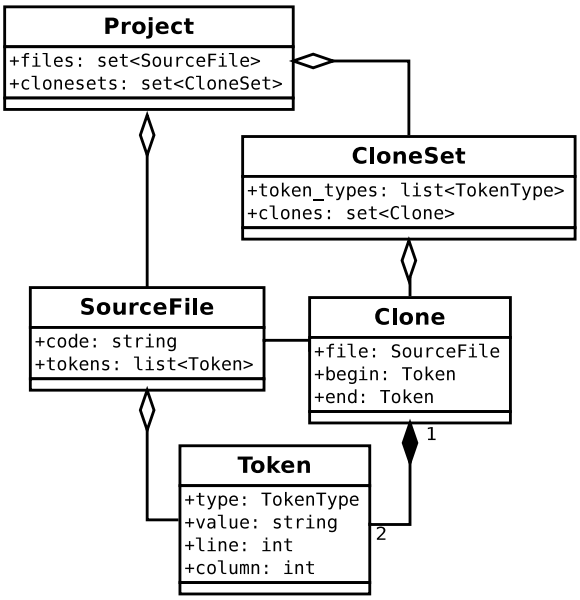


Figure 3.5: Structure of input to FICA

larity in TF-IDF vector space.

3. Receiving the true or false clone marks from the user as training sets of machine learning.

4. Rebalancing the input set in order to get identical data points in each set of the training sets.

5. Calculating the probability for each clone set of the different classifications in the marked groups.

6. Ranking and reordering the clone sets based on their probabilities.

The user can adjust the marks that FICA has predicted and submit the marks again to FICA, as in step 3, which then forms the supervised machine learning cycle.

3.4.1 Input Format of FICA

As a support tool for CDTs, FICA needs to extract code clone information from the output of the CDTs. The structure of all the information needed by FICA is represented as the UML diagram in Figure 3.5.

A `project` consists of the original source code and the detected code clones represented as `clone sets`. The source code fragments are tokenized by the CDT. As FICA needs both tokenized source code for calculating the similarity and the original source code for presenting back to the user, these two forms are passed together to FICA. A clone set is a set of identical code fragments. By identical, we mean the tokenized code fragments are literally equal to each other in a `clone set`. We are targeting type-2 code clones; therefore, all the token types in a clone set are identical.

A `clone` in a `clone set` is the tokenized code clone fragment in the original source file in the project, The list of token types in the clone set should be equivalent to the types of tokens in every clone in the clone set. A token has a type and a position in the original source code. An example of how the input of FICA looks like is in Figure 3.6.

3.4.2 Calculating Similarity between Clone Sets

FICA calculates the TF-IDF vector [47] of clone sets, which is widely used in machine learning techniques in natural language processing. In FICA we define `term` t as an n-gram of a token sequence, `document` d as a `clone set`, and all documents `D` as a set of `clone sets` that can be gathered from a single `project` or several `projects`.

A previous study by [46] suggested that the similarity of token sequences is one of the best characteristics for identifying false candidates from type-3 clones. Similar to their study, we also need to calculate the similarity among clones. Therefore, we use a metric from the token sequence to capture syntactic information. In contrast to their study, we compare similar type-2 clones. Therefore, we need a metric that can tolerate structural reordering of source code, such as reordering of statements. Based on these two objectives, we use n-grams of the token sequences.

First, we divide all token sequences in clone sets into n-gram terms. Let us assume we have a clone with the following token types:

STRUCT ID TIMES ID LPAREN CONST ID ID TIMES ID RPAREN ...

If we assume that N for n-gram equals 3, then we can divide this clone into n-grams as:

STRUCT ID TIMES
ID TIMES ID
TIMES ID LPAREN
ID LPAREN CONST
LPAREN CONST ID
CONST ID ID
ID ID TIMES

```
Project: git-v1.7.9
files:
  bloc.c
    const char *blob_type = "blob";
    struct blob *lookup_blob(const ...
    ...
  tree.c
    const char *tree_type = "tree";
    static int read_one_entry_opt( ...
    ...
  builtin/fetch-pack.c
  builtin/receive-pack.c
  builtin/replace.c
  builtin/tag.c
  ...
clonesets:
  1: token_types: STRUCT ID TIMES ID LPAREN CONST ...
     clones:
       blob.c (6:1)-(23,2)
       tree.c (181,1)-(198,2)
  2: token_types: ID RPAREN SEMI RETURN INT_CONST ...
     clones:
       builtin/replace.c (39,48)-(57,10)
       builtin/tag.c (154,45)-(172,10)
  ...
```

Figure 3.6: Example of input format for FICA

...

Then we calculate all the `term frequency` values of n-grams within this document of clone set by Equation 3.1.

$$\text{TF}(t, d) = \frac{|t : t \in d|}{|d|} \tag{3.1}$$

Equation 3.1 states that `term frequency` of `term` t in `document` d is the normalized frequency, where `term` t appears in `document` d. The result of TF of the above `clone set` is as follows:

```
RETURN ID SEMI          :0.00943396226415
ID RPAREN LBRACE        :0.0283018867925
SEMI RETURN ID          :0.00943396226415
ID SEMI IF              :0.00943396226415
ID LPAREN CONST         :0.00943396226415
TIMES ID RPAREN         :0.00943396226415
IF LPAREN LNOT          :0.0188679245283
RPAREN LBRACE ID        :0.00943396226415
LPAREN RPAREN RPAREN    :0.00943396226415
SEMI RBRACE RETURN      :0.00943396226415
RETURN ID LPAREN        :0.00943396226415
ID ID RPAREN            :0.00943396226415
TIMES ID COMMA          :0.0188679245283
...
```

Analogously, the `inverse document frequency` IDF and TF-IDF are calculated by Equations 3.2 and 3.3.

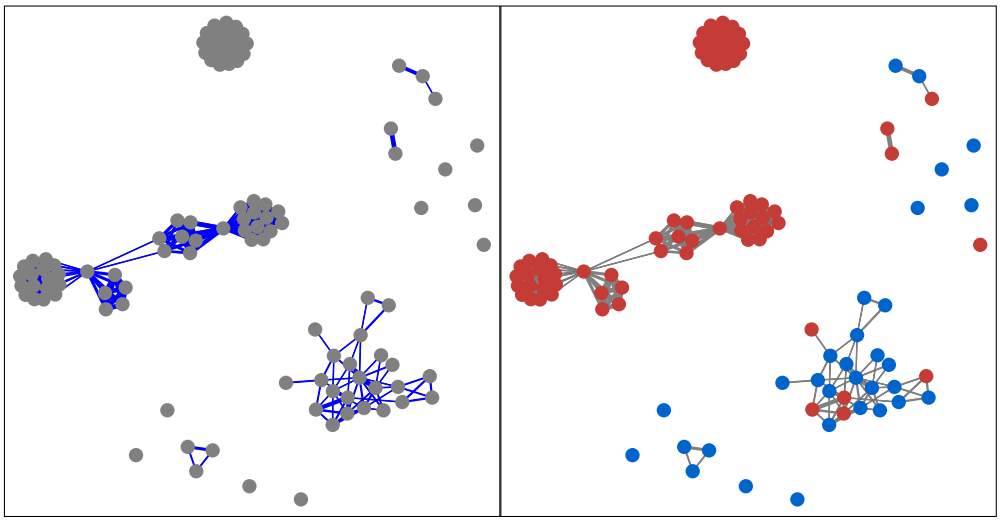
$$\text{IDF}_D(t) = \log \frac{|D|}{1 + |d \in D : t \in d|} \tag{3.2}$$

$$\text{TF-IDF}_D(t, d) = \text{TF}(t, d) \times \text{IDF}_D(t) \tag{3.3}$$

$$\overrightarrow{\text{TF-IDF}_D d} = [\text{TF-IDF}_D(t, d) : \forall t \in d] \tag{3.4}$$

Equation 3.2 states that the `inverse document frequency` *idf* for `term` t in all `documents` D is the logarithm of the total number of `documents` divided by the number of `documents` containing `term` t. In Equation 3.2, we add 1 to the denominator to avoid division by zero. By combining tf and idf as Equation 3.3, we can then calculate the vector space $\overrightarrow{\text{TF-IDF}(d, D)}$ as in Equation 3.4 for each clone set in the overall documents.

By using TF-IDF, we define the cosine similarity $\text{CosSim}_D(a, b)$ of two clone sets, a and b, with regard to a set of documents D, as in Equation 3.5 and 3.6.



(a) clone sets grouped by similarity (b) Colored force-directed graph

Figure 3.7: Force-directed graph of clone sets of bash 4.2. Blue circles mean true code clones and red means false code clones.

$$\text{Sim}_D(a,b) = \overrightarrow{\text{TF-IDF}_D a} \cdot \overrightarrow{\text{TF-IDF}_D b} \tag{3.5}$$

$$\text{CosSim}_D(a,b) = \begin{cases} 0, & \text{Sim}_D(a,b) = 0 \\ \frac{\text{Sim}_D(a,b)}{\left|\overrightarrow{\text{TF-IDF}_D a}\right| \cdot \left|\overrightarrow{\text{TF-IDF}_D b}\right|}, & \text{otherwise} \end{cases} \tag{3.6}$$

After calculating the similarity among all clone sets within a project, we can then plot them based on the force-directed algorithm described in [48] to show the similarity among code clones. As an example, Figure 3.7(a) illustrates the clone sets grouped by their similarity. The details of the force-directed graph (FDG) are discussed in Subsection 3.5.3.

3.4.3 User Profile and Marks on Clone Sets

A profile represents a user's preferences with regard to the classification of clones stored in FICA. A user is allowed to keep multiple profiles for different use cases of FICA. The implementation of FICA does not distinguish a user from the profiles, so for simplicity, the remainder of this paper assumes that every user has a single profile, and FICA needs an initial training set to provide further predictions.

A mark is made on each clone set by the user. Marks, which indicate this clone set has certain properties, are recorded in the user profile. We considered three kinds of marks that users can make on these clone sets. Boolean marks flag a clone set that the user thought to be true or false clones. Numerical marks represent a weight, or importance, assigned by each user, saying that clone set A is more important than clone set B. Tag marks are not exclusive, which is useful in multi-purpose code clone analysis for classifying the clones into multiple overlapping categories.

A training set is a set of clone sets that have been marked by the user. For Boolean marks, the training set M is divided into two sets of clone sets, M_t and M_f, each of which donates a true set and a false set, where $M_t \cap M_f = \emptyset, M_t \cup M_f = M$. For range marks, there is only one group of training sets, M, and each clone set m in M is associated with a weight $w(m)$. For the tag marks, several groups of clone sets are in the form of M_x, with each set donating a different tag. The probability calculation is the same. These groups can overlap each other, and doing so does not affect the result of the calculation.

For each clone set t that has not been marked, FICA calculates the probability that this clone set t should also be marked in the same way as those in clone set group M_x by using Equation 3.7.

$$\text{Prob}_{M_x}(t) = \begin{cases} 1, & \sum_{\forall m \in M_x} w(m) = 0 \\ \dfrac{\sum_{\forall m \in M_x} \text{CosSim}_{M_x}(t, m) \cdot w(m)}{\sum_{\forall m \in M_x} w(m)}, & \text{otherwise} \end{cases} \tag{3.7}$$

Here, $w(m), m \in M_x$ is a weighting function for marked clone set m in clone set group M_x. For Boolean or tag marks, we defined $w(m) = 1$; thus, $\sum_{\forall m \in M_x} w(m) = |M_x|$, which is the size of clone set group M_x. For numerical range marks, the weighting function $w(m)$ is defined as a normalized weight in $[0, 1]$.

As expressed in Equation 3.7, the probability that a clone set t should also be marked as M_x is calculated by the average normalized similarity of t and every clone set in the clone set group M_x. For predicting Boolean marks, FICA can compare the calculated probability of the clone being a true or false clone, and choose the higher one as the prediction. For range marks, the average normalized similarity is multiplied by $w(m)$ from the user's input. For tagged marks, FICA needs to set a threshold for making predictions.

FICA makes predictions based on the probability and ranks all the code clones by the value calculated from $\text{Prob}_{M_x}(t)$. For Boolean marks, the ratio of the probability for the true clone set and the false clone set, $\frac{\text{Prob}_{M_T}(t)}{\text{Prob}_{M_F}(t)}$, is used as a ranking score. For tagged marks, the probabilities for each individual tag are ranked separately, and the tag with the highest probability is chosen as the prediction result. Range marks have only one group of marks. Therefore, the probability $\text{Prob}_M(t)$ is used directly for ranking.

A special case occurs when the sum of the weighting functions is equal to zero; in this case, we define the probability to be 1. This special case occurs when no such data is chosen in clone set group M_x. An arbitrary value is possible in this case as we have no learning data, but we chose 1 to prevent division by zero in calculating the ratio between probabilities.

After the prediction, all clone sets are marked either by the prediction of FICA or by the user's input. An example of all Boolean marked clone sets in the above force-directed graph is shown in Figure 3.7(b). The result of the FICA prediction is presented to the user, so that the user can check its correctness, approve some or all of FICA's prediction or correct other predictions as needed. After correcting part of the result, the user can resubmit the marks to FICA to get a better prediction.

3.5 Experiments

3.5.1 Implementation Details

FICA was implemented as a proof-of-concept system. We implemented the described algorithms for computations of the similarity of code clones, as well as a customized token-based CDT that outputs exactly what FICA requires. Also, we wrapped the CDT part and FICA together by using a web-based user interface, which is referred to as the FICA system from now on[2].

The FICA system manages the profiles of users as normal login sessions, like those in most websites. One of the users uploads an archive of source code to the FICA system. Then FICA unzips the source code into a database and then passes the code to the CDT part of the FICA system as in Figure 3.8.

The CDT part of the FICA system implements the algorithm for constructing a *generalized suffix tree* described by [49] and the algorithm of the detection of clones among multiple files in the *generalized suffix tree*. Code clones and types of token sequences are recorded in a database.

Then the FICA system shows a comparison view of detected clone sets to the

[2]FICA with experimental data can be accessed here: `http://valkyrie.ics.es.osaka-u.ac.jp`

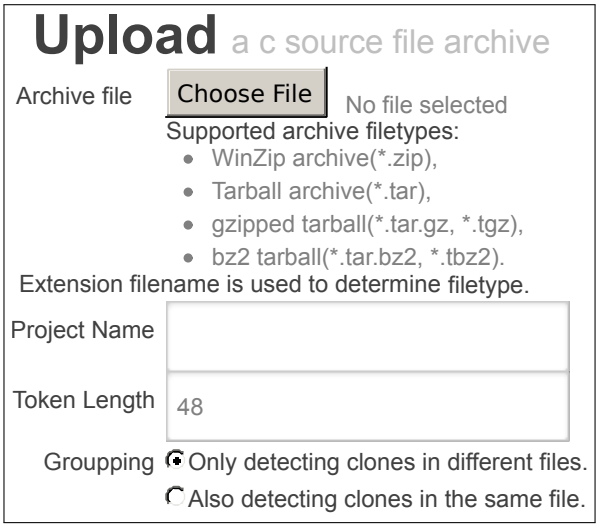


Figure 3.8: Upload Achieve of Source Code to FICA

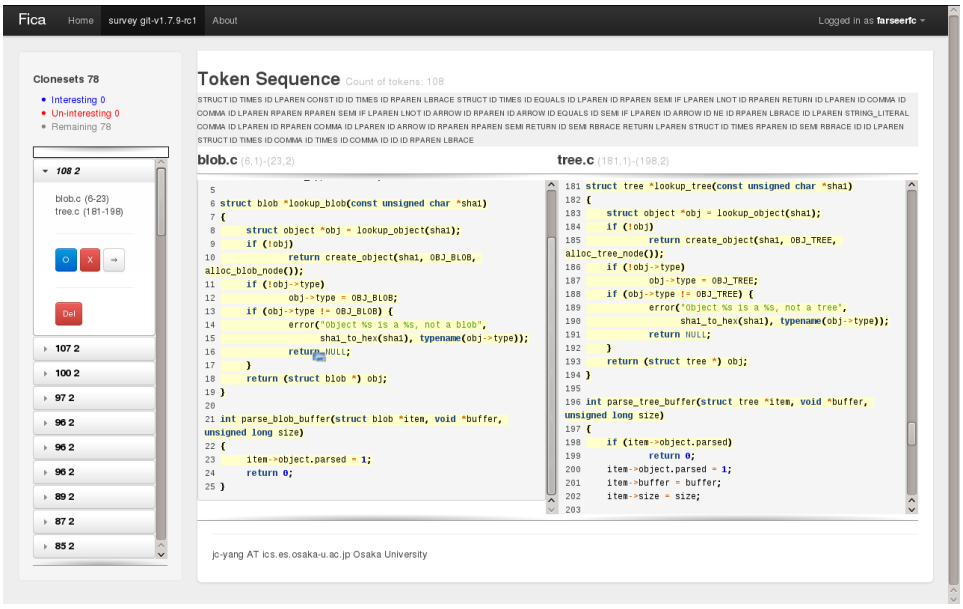


Figure 3.9: FICA showing clone sets

user, as shown in Figure 3.9. The user can mark Boolean tags on these clone sets, and the FICA system stores these marks immediately in the database associated with the user profile. While the user is marking those clone sets, the FICA system calculates the similarity among those clone sets and trains its prediction model by including the user input into its training set on the server side in the background. As a result, the feedback of user inputs in the FICA system can be gained in nearly real time.

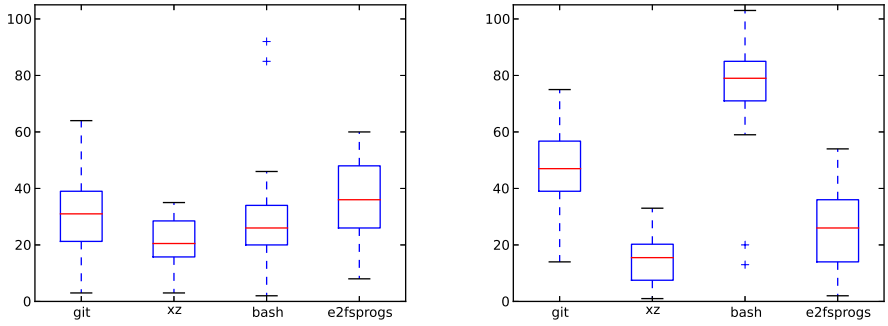
3.5.2 Experimental Setup

To test the validity and user experience of the proposed method, the following experiment was conducted. We uploaded the source code of four open source projects as experimental targets, as shown in Table 3.2. For all the projects in Table 3.2, we only included `.c` files and ignored all other files as unnecessary because all four projects were developed in C language. As the parameters passed to the CDT part of the FICA system, we only detected clone sets that contained more than 48 tokens, and only reported those clone sets from different files. These two parameters were chosen based on experience. Namely, the length limitation less than 48 for C projects is more likely to result in small false positives, as well as for clones from the same source code file. Although too many clone sets are an obstacle to conducting this experiment, those small false positives should be detectable in FICA as well. The CCFinder, one of the well-known CDTs proposed in [50], uses a length of 30 tokens as a default parameter. This length is widely accepted in both industrial and academic studies, while its definition of tokens is compacted. For example, CCFinder treats an `IF` followed with a `LEFT_BRACE` as one compressed token, whereas our CDT considers them as two tokens. Based on our experience, a length limit of 48 tokens generates almost the same amount of code clones as CCFinder.

Altogether, 32 users participated in this experiment. Each user had different experiences about code clone detection techniques. They were required to mark

Table 3.2: Open source projects used in experiments

Project & Version	# of clone sets	LOC	Tokens	Files	# of Participants
git v1.7.9-rc1	78	153,388	829,930	315	32
xz 5.0.3	36	25,873	113,894	113	27
bash 4.2	105	133,547	494,248	248	25
e2fsprogs 1.41.14	62	99,129	442,978	274	25
Total	281	411,937	1,881,050	950	32



(a) # of True Clones

(b) # of False Clones

Figure 3.10: # of true and false clones from the selected data of users

those clone sets found by the CDT built in the FICA system when using Boolean marks as true or false clones. The users were told the steps of the experiments but not the internal techniques used in the FICA system. Consequently, they were not aware that the FICA system compares clone sets by the text similarity of lexical token types. Also, we informed the participants that the experiment would take approximately 2 hours to complete, with approximately 30 seconds per clone set.

As the order of the clone sets affects the prediction result of the system, we disabled the reordering feature of the FICA system. The users were presented with the list of clone sets in a fixed order from the original results of the CDT. This fixed order is determined by the suffix tree algorithm used in the CDT and appears to be random from the user's perspective. The number of true code clones and false code clones selected by the users are shown as box-plots in Figure 3.10.

The experiment was conducted on the Internet through the FICA web interface. Among the 32 participants, 13 were graduate students or professors in our school, 7 were from clone-related research groups in other universities, while the others were in industry and interested in clone related research. All of these participants are or were majors in computer science or software engineering. Regarding their programming language preferences, almost all of the participants used Java as one of their main languages, while 11 also mainly used C. Furthermore, we ensured that none of the participants ever contributed to the development of the four target software projects.

We used 5-grams throughout the experiments. The size of the n-gram was selected based on our experience. It is certain that increasing the N value improves the theoretical accuracy of the algorithm. Theoretically, the upper bound size of

the possible types of n-grams grows exponentially with the N value, although in practice this size is also limited by the size of input. Therefore, increasing the value of N increases the calculation time accordingly. From our experience of the implementation, any N value greater than 3 generates a similar ranking result, while any N value less than 6 enables the prediction of one pass to end in a time limit of 1 second, which is reasonable for interactive usage. Furthermore, we intended to capture the syntactic context of the token sequence and tolerate the reordering of statements. Since 5 is the median length per statement of a C program, we adapted the N value of 5 in our experiments.

3.5.3 Code Clone Similarity of Classification Labels by Users

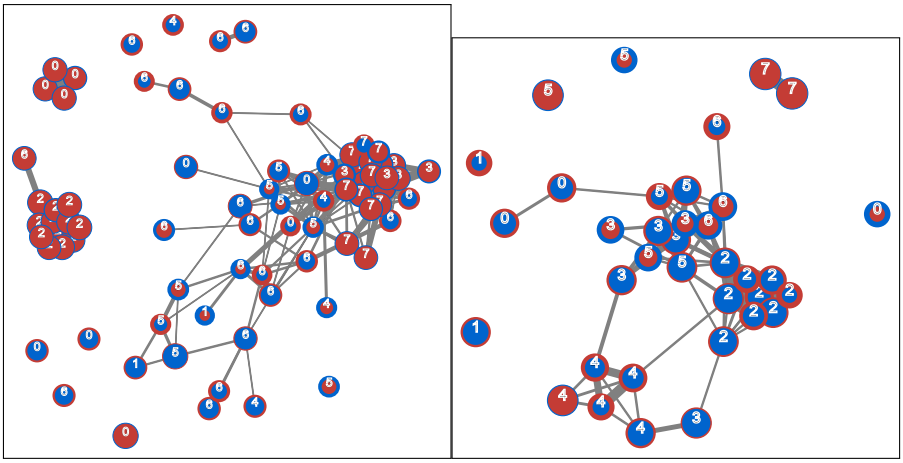
We showed in Figure 3.7 that the clone sets from a given project have a tendency to group into clusters. To further illustrate this phenomenon, we use a semi-supervised learning approach that compares the result of a clustering algorithm with the selection of the users. Note that this semi-supervised clustering approach is used only to show the underneath relations among our dataset of code clones, but it is not part of the FICA machine learning process.

We applied the KMeans clustering algorithm [51] on the TF-IDF vector of our datasets with $K = 8$ as the target cluster number for each software project. As a result, the KMeans algorithm labeled each clone set with a cluster id from 0 to 7, and all four software projects converged in at most four iterations. The K value of KMeans is determined by trial, and we experimented with values from 5 to 10. In practice, the K value grows with the scale of the targeted software project.

Then we overlaid the labels from the KMeans algorithm onto the FDG, as in Figure 3.11. For each project in FDG, a clone set is represented as a circle drawn in a mixture of red and blue. Blue circles mean true code clones and red means false code clones. Partially red and blue circles indicate clone sets for which people had different opinions, and the relative area occupied by the two colors reflects the relative number of opinions. The central color always represents the majority opinion. The numerical text in each circle is the label of the clustering result by the KMeans algorithm.

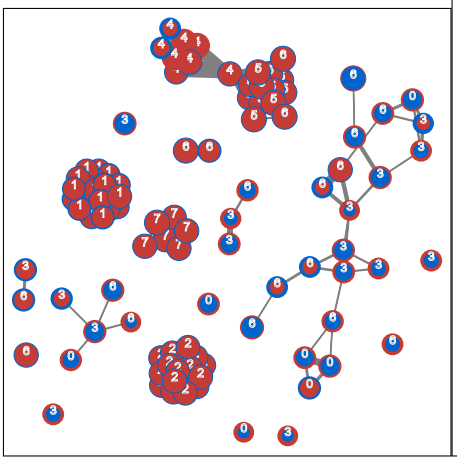
For each pair of clone sets, a link exists between the circles if the similarity between the clone sets is greater than a threshold, which is adjustable from the web interface in real time.The values of similarity were assigned as the force strength on the links between the circles of clone sets. We used the `d3.js` javascript library by [52] to generate this FDG. Some other technical details about this graph are listed in Table 3.3.

To show the distance among the clones in each cluster by KMeans, we calculated the average normalized cluster distance, as in Table 3.4. The distances in

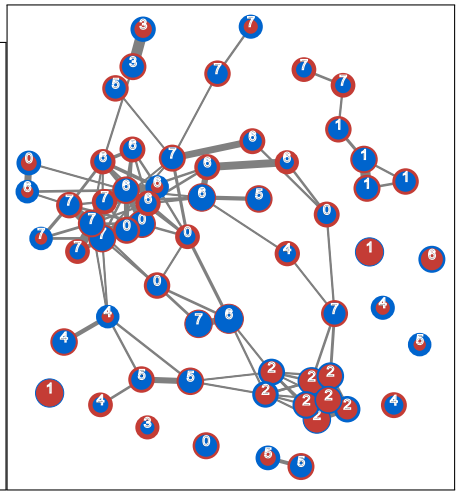


(a) Code clone similarity of `git`

(b) Code clone similarity of `xz`



(c) Code clone similarity of `bash`



(d) Code clone similarity of `e2fsprogs`

Figure 3.11: Code Clone Similarity with classification by users

this table were normalized from 0 to 1, where the larger number indicates a larger distance among the clone sets and the most distance clone sets have a score of 1.

Figure 3.11 shows that groups of false positives are more similar than groups of true positives. Thus, these false positives are closer to each other in the figure and are clustered with the same label by KMeans. For example, in the `git` project, the cluster with ID 2 has an average normalized distance of 0.21; this means that the false clone sets appear to be closer than the cluster with ID 6, whose average normalized distance is 0.9. This result indicates that the probability of false clone sets calculated by Equation 3.7 is more accurate than the probability of true clone sets, because false clones are grouped into categories more tightly.

We calculated the homogeneity and completeness scores defined by [53]. These two scores compare the labels of the clustering result with the label of ground truth, which comprises the selected marks by the users in our dataset. These two scores reveal the different characteristics of clustering:

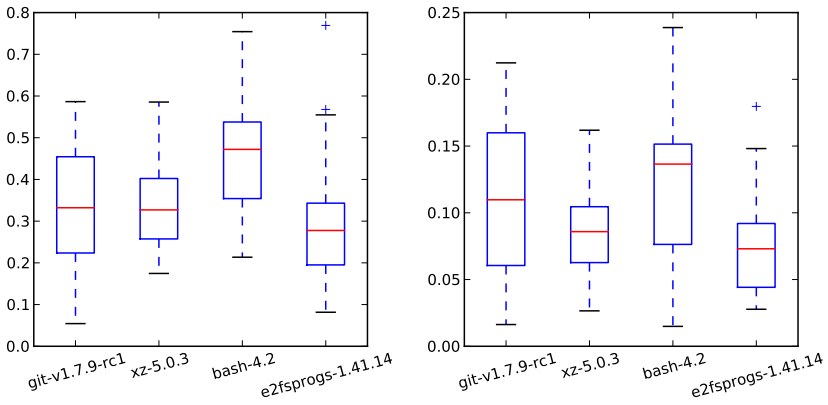
Table 3.3: Parameters for force-directed graph

Parameter	Value
force.size	800
force.charge	100
link.linkDistance	$1/link.value$
link.linkStrength	$link.value/16 + 0.2$
link.stroke-width	$10 \cdot \sqrt{link.value}$
node.r	$13 \cdot major/(minor/1.5 + major)$ 3.8
node.stroke-width	$8 \cdot minor/(minor/1.5 + major)$ 3.9

$$major = \max\{|\texttt{true}|, |\texttt{false}|\} \tag{3.8}$$
$$minor = \min\{|\texttt{true}|, |\texttt{false}|\} \tag{3.9}$$

Table 3.4: Average normalized distance of clusters by KMeans

	0	1	2	3	4	5	6	7
git	0.78	0.46	0.21	0.36	0.73	0.78	0.9	0.55
xz	0.64	0.46	0.37	0.66	0.58	0.77	0.45	0.0
bash	0.79	0.18	0.19	0.87	0.25	0.06	0.9	0.14
e2fsprogs	0.76	0.74	0.42	0.33	0.71	0.76	0.79	0.79



(a) Homogeneity (b) Completeness

Figure 3.12: Homogeneity and completeness between users' marks and clusters by KMeans

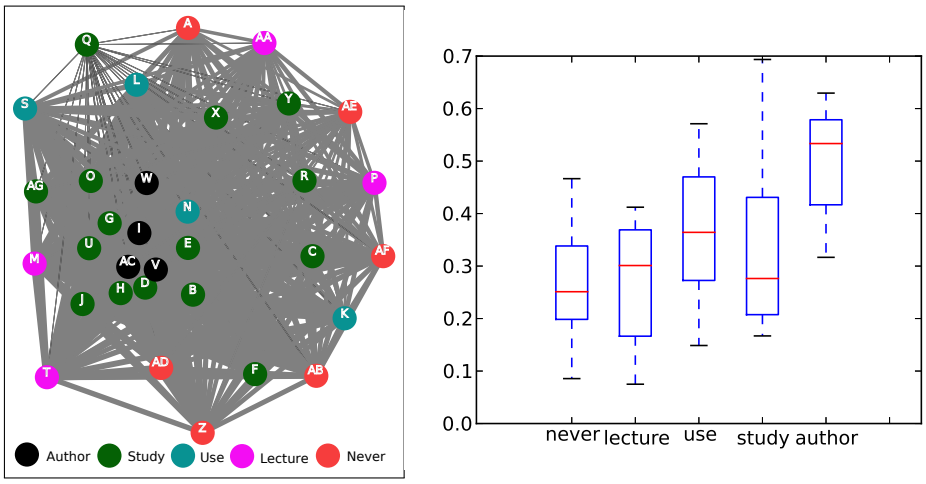
homogeneity scores high when each cluster contains only members of a single class.

completeness scores high when all members of a given class are assigned to the same cluster.

As we have eight clusters by the KMeans algorithm and two classes from the marks of users, the homogeneity is expected to be high and the completeness is expected to be low. For each software project, we compared the marks of each user with the KMeans labels and used a boxplot to show these two scores in Figure 3.12. We can see the co-relationship of users' marks with the clusters from the KMeans algorithm.

The internal reason for this result is the fact that almost all false clone sets fall into categories of meta-clones with certain obvious characteristics of their token types, such as a group of `switch-case` statements, or a group of assignments. Meanwhile, the true clones are harder to classify into similar categories. In general, true code clones usually contain rare sequences of tokens. Thus, we have observations 1 and 2, regardless of the subjective judgment of participants,

Observation 1 (Categories of False Clones)**.** *False code clones are more likely to fall into several categories of meta-clones by comparing the literal similarity of their token types.*



(a) Similarity (b) Homogeneity

Figure 3.13: Similarity and Homogeneity of Users' Selection with Their Experience

Observation 2 (Diversity of True Clones)**.** *True code clones composed of sequences of tokens are more diverse than false clones by comparing the literal similarity.*

3.5.4 Similarity among Users' Selection

We also draw a FDG of users to illustrate the similarity among their selection and homogeneity of them for each experience category, as in Figure 3.13. Different colors in the FDG represent the user's experience with code clone detection technology, which is manually selected by these participants during the on-line survey, with options given as in Table 3.5 in that order. The levels with higher experiences are supposed to cover the levels with lower experiences. For example, the author of a clone detection tool is expected to also be studying on the topic of code clones. Therefore the user can only select one category from these options.

As we can see from Figure 3.13, selections from participants with more knowledge with code clone techniques (those who selected *author* or *study*) are more similar to each other, while selections by participants with less experience with code clone tends to vary differently.

3.5.5 Ranking Clone Sets

To measure the quality of supervised machine learning of FICA, we adapted a measure called *Average true Positive Found* (referred to as ATPF) proposed by [25]. Their study is similar to ours in that they also reordered the clone list according to the structure similarity. Here, the word *positive* refers to the clones predicted by FICA that have a high likelihood of being true clones, rather than the result from the CDTs.

The measure of ATPF is visualized as a graph capturing the cumulative proportion of true positives found as more clones are inspected by the users. In the cumulative true positives curve, a larger area under the curve indicates that more true positives were found by the users early; this means the refinement process effectively re-sorts the clone list.

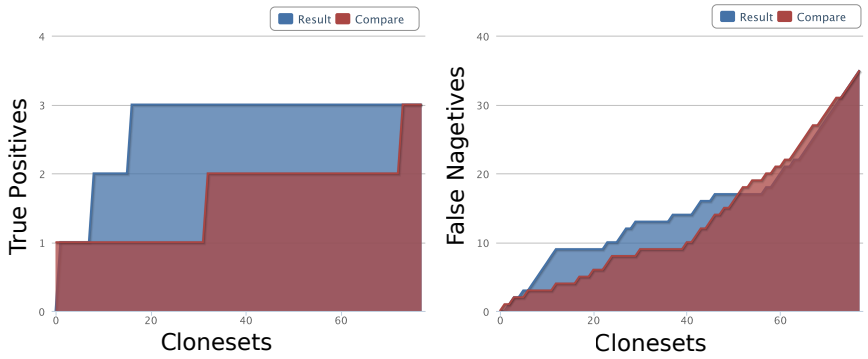
An example of an ATPF graph is shown in Figure 3.14(a). In the ATPF graph, both the X-axis and the Y-axis are the number of clone sets, where the value of the X-axis is the number of clone sets presented to the user while the value of the Y-axis is the number of clone sets that the user considers as true clones. The graph has two cumulative curves. The *"result"* curve in blue is the result after reordering by our tool, while the *"compare"* curve in red is the original order presented by the CDTs. For our embedded CDT, the output order of clones is in alphabetical order of the token sequence because of the suffix tree algorithm.

In Figure 3.14(a), 78 clone sets are found by the CDT, but only 3 of them are regarded as true clones by the user. As we can see from the *"compare"* curve, these 3 true clone sets are distributed in the very beginning, the middle, and near the tail of the clone set list. In the *result* curve, our method successfully rearranged the order of the clone list, so that all three true clone sets appeared at the front part of the clone set list. In a real working environment, this will significantly increase the efficiency of the user.

As the user can also change the sorting order to filter out the false clone sets,

Table 3.5: User Experience of Code Clone Categories

Experience Category	Means
Author	"I Implemented a Clone Detection Tool"
Study	"Code clone is one of my research topics"
Use	"Using code clone detectors in works"
Lecture	"Heard a lecture of code clone"
Never	"Never heard of code clone"



(a) Average True Positives Found

(b) Average False Negatives Found

Figure 3.14: Average True Positives Found and Average False Negatives Found for different users on git project

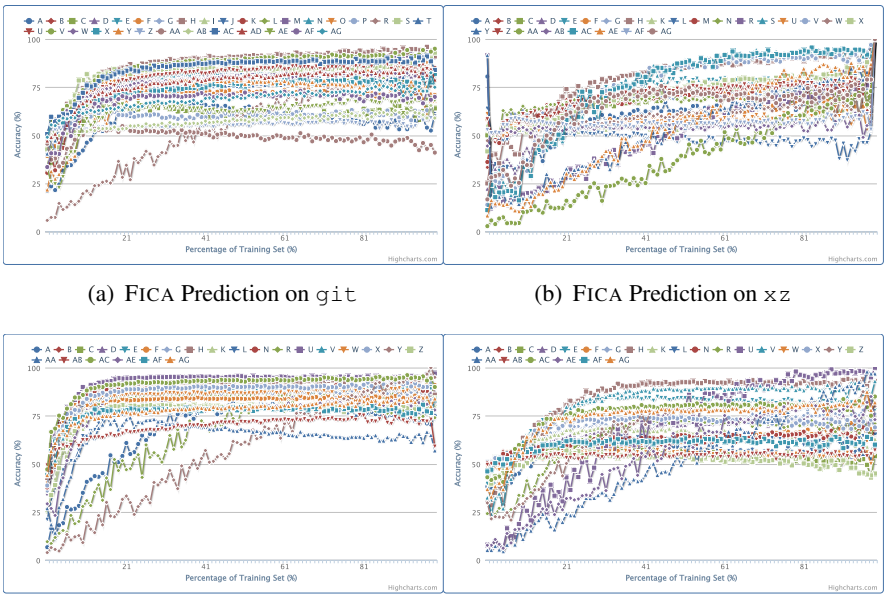
we defined *Average False Negative Found* (referred to as AFNF), which is similar to the definition of ATPF shown in Figure 3.14(b). Analogously, the word *negative* refers to the clones that have a high likelihood to be false clones by FICA. As we can see in the AFNF graph, the algorithm generates more false negatives in the beginning, and then after approximately 64% of the clone list is processed, the result fails to be better than the compared result. This behavior of the algorithm is expected. By reordering the list of clones, moving the desired clone to the end of the list is a small probability if the clone is similar to those previously marked as not desired by the user.

3.5.6 Predictions for Each Project

To measure the accuracy of predictions made by FICA with the marked labels from the user, we define a metric called "accuracy" as the percentage of how many predictions by FICA are equal to the selections of the user, that is, the percentage of both true positives and true negatives in all clone sets.

We trained the FICA system by using the marked data of eight users on all clone sets for each project. The accuracy of our prediction model is shown in Figure 3.15. The horizontal axis of the figure is the percentage of training sets and the vertical axis is the previously defined "accuracy" of prediction. Three steps were required to perform a prediction:

1. The FICA system randomly selected a part of all clone sets from a project as the training set and the remaining were used as the comparison set.



(a) FICA Prediction on `git`

(b) FICA Prediction on `xz`

(c) FICA Prediction on `bash`

(d) FICA Prediction on `e2fsprogs`

Figure 3.15: Accuracy of machine learning model by FICA

2. For the division of training and comparison sets, FICA trained its machine learning model with the training set and calculated a prediction for each clone set in the comparison set.

3. FICA compared the predicted result with the mark made by a user, and calculated the accuracy.

We repeated these three steps 256 times for each training set size, and the plotted values of accuracy in the Figure 3.15 are the averaged results of the 256 times prediction.

We can see from Figure 3.15 that the prediction results change with the users and the target projects. For all four projects and nearly all users, we can see that the accuracy of the prediction model grows with the portion of the training set. As the training set grows, in the beginning the accuracy of the prediction increases quickly until it reaches a point, between 10% to 30%, where it increases more slowly.

Among all four projects, the `bash` project shows the most desirable result, namely, most of the prediction accuracies are over 80% when the training set is larger than 16%, which is approximately 17 of all 105 clone sets. The results of the `git` project and the `e2fsprogs` project largely depend on the user, for example,

the result of user H always achieves more than 90%. Meanwhile, the results of user A and C converge to approximately 60% and even decrease when the training set is growing. The reason why the result did not converge to 100% and the accuracy dropped is discussed in Section 3.5.9.

By combining the Observation 1 with the details from the result of Figure 3.15, we got our Observation 3.

Observation 3 (Minimum Size of Training Set)**.** *The minimum required size for the training set grows roughly linearly with the number of categories that the clone sets fall into, which is less than the total number of detected clone sets.*

We can also observe from Figure 3.15 and 3.16 that the different prediction results of users are separated into different levels. The predictions for some particular users, namely user R and user V, are always low, which means less literal similarity is found among the clone sets they marked as true clones. Also, the prediction for user H is high for all projects. This result shows the consistency of user behaviors.

Note that experience of the user is not necessary affecting the accuracy of prediction. This is just suggesting that the results from experienced users tend to be more similar. Some of the participants, such as user Q, share little in common with other users, while still conducted a reasonable prediction accuracy.

3.5.7 Cross Project Evaluation

To illustrate that the training data of selections by the users can be applied across different projects, we merged all clone sets from the four projects into a single project and repeated the above experiment again. The result is shown in Figure 3.16. The result is actually better than the result for a single project, except for the `bash` project.

To simulate a real cross-project analysis, we also did a cross-project evaluation. For each user and each pair of the four projects, we trained the model with the data gathered from one project and evaluated our method on the other. We used the same accuracy definition and drew the boxplot of the users in Figure 3.17. The result varies with the project: `e2fsprogs/bash` is a desirable result and `xz/git` is an undesirable result. We expect this kind of variation in the results, because each project has its own characteristics for code clones. In practice, the user of the FICA system is expected to cross-analyze similar projects when finding similar code clone categories in both projects.

3.5.8 Recall and Precision of FICA

We also measured the recall and precision for separated true positives and false negatives of FICA as support for our definition of accuracy. Firstly we defines the

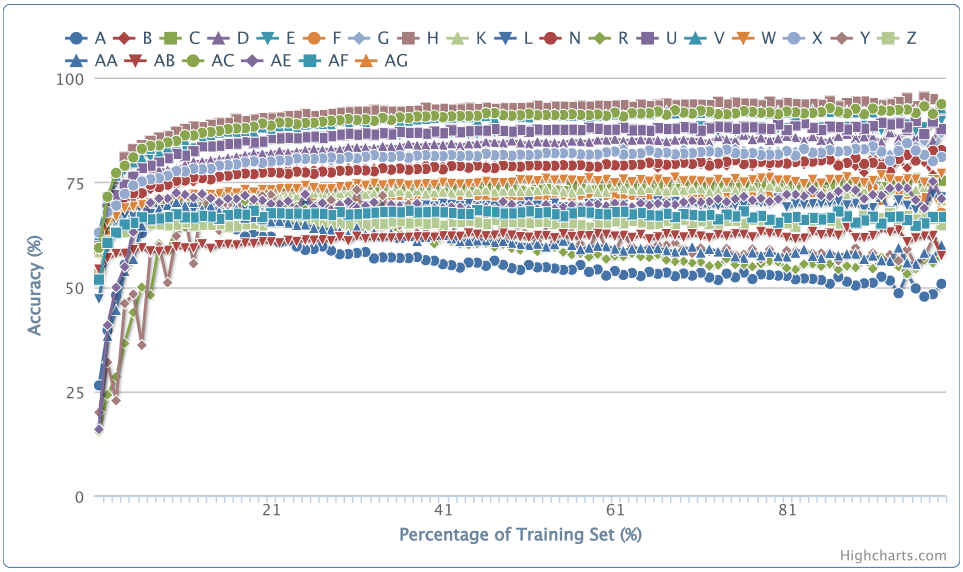


Figure 3.16: Merged result of all projects

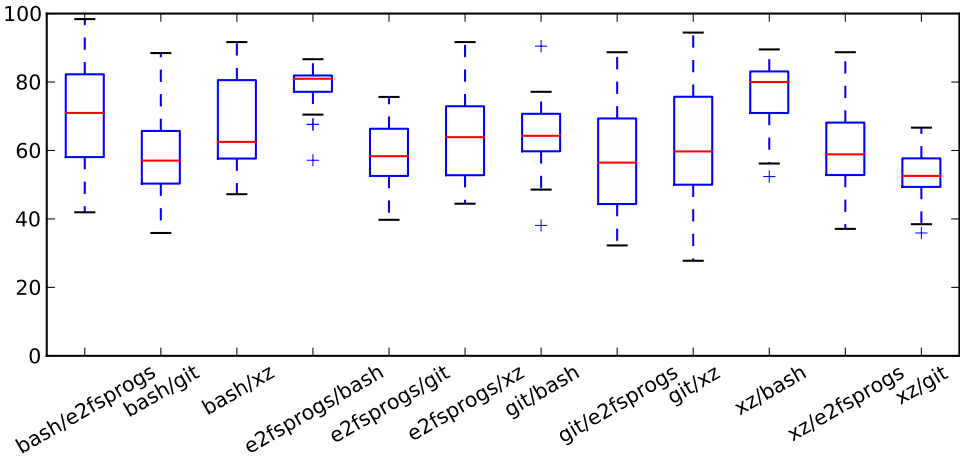


Figure 3.17: Accuracy of predictions with training/evaluation projects

true clones as the clones that are considered interesting by the user, and the *false* clones as the clones that are considered un-interesting. Then we defines the *positive* clones as the clones that are predicted as interesting by FICA and *negative* clones as otherwise.

Based on these definitions, we referred to *true positive* as *tp*, *true negative* as *tn*, *false positive* as *fp*, *false negative* as *fn*. We then defined the recall and precision for *tp* and *fn* as in Equations 3.10 to 3.13.

$$recall_{tp} = \frac{tp}{tp + fn} \tag{3.10}$$

$$precision_{tp} = \frac{tp}{tp + fp} \tag{3.11}$$

$$recall_{fn} = \frac{tn}{tn + fp} \tag{3.12}$$

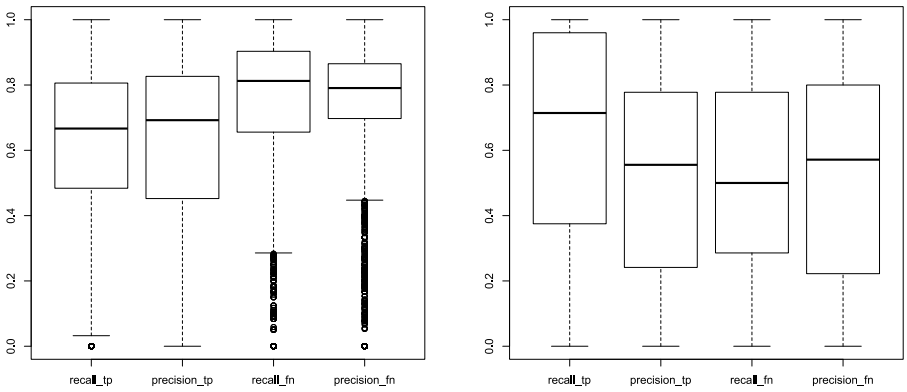
$$precision_{fn} = \frac{fn}{fn + tn} \tag{3.13}$$

We took 20% of all the clone sets as the training set, the remaining as the evaluation set, and then repeated the experiment 100 times. The result is shown as the boxplot in Figure 3.18. From the boxplot we can see a similar result to that of Figure 3.15, as the `git` and `bash` projects show good results while the results for `xz` and `e2fsprogs` are not so comparable. These recall and precision charts show a trend similar to the accuracy graph in Figure 3.15; that is, when the project has clustered clone categories, the result is more appropriate.

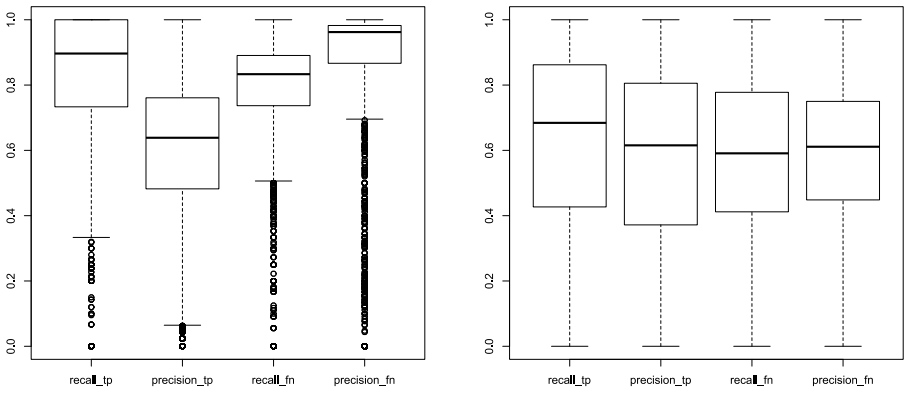
3.5.9 Reason for Converging Results

For all projects in Figure 3.15 and 3.16, the accuracy of predictions made by FICA converges to approximately 70% to 90% and it is difficult to achieve 100%. In this section, we discuss some code fragments to show the reason.

The first example comes from the `xz` project in Figure 3.19. These three code fragments have two code clone pairs. The code from Figure 3.19(a) in lines 130 to 142 and the code from Figure 3.19(b) are the code fragments of the first clone pair, referred to as clone α. The code from Figure 3.19(a) in lines 136 to 145 and the code from Figure 3.19(c) are the code fragments of the second clone pair, referred to as clone β. As we can see from the source code, each instance of clone α consists of a complete function body. Clone β consists of only two half parts of functions. Thus, from the viewpoint of the ease of refactoring, clone α is much easier than is clone β. On the other hand, the calculated similarity between clone α and β is greater than 43% by Equation 3.5. This is a very high percentage among



(a) `git` (b) `xz`



(c) `bash` (d) `e2fsprogs`

Figure 3.18: Recall and precision of FICA in each project

```
130         }
131         return LZMA_PROG_ERROR;
132     }
133     static void
134     block_encoder_end(lzma_coder *coder, lzma_allocator *allocator)
135     {
136         lzma_next_end(&coder->next, allocator);
137         lzma_free(coder, allocator);
138         return;
139     }
140     static lzma_ret
141     block_encoder_update(lzma_coder *coder, lzma_allocator *allocator,
142             const lzma_filter *filters lzma_attribute((__unused__)),
143             const lzma_filter *reversed_filters)
144     {
145         if (coder->sequence != SEQ_CODE)
```

(a) src/liblzma/common/block_encoder.c

```
61      }
62      return LZMA_OK;
63  }
64  static void
65  alone_encoder_end(lzma_coder *coder, lzma_allocator *allocator)
66  {
67      lzma_next_end(&coder->next, allocator);
68      lzma_free(coder, allocator);
69      return;
70  }
71  static lzma_ret
72  alone_encoder_init(lzma_next_coder *next, lzma_allocator *allocator,
73          const lzma_options_lzma *options)
```

(b) src/liblzma/common/alone_encoder.c

```
474     else
475         lzma_free(coder->lz.coder, allocator);
476     lzma_free(coder, allocator);
477     return;
478 }
479 static lzma_ret
480 lz_encoder_update(lzma_coder *coder, lzma_allocator *allocator,
481         const lzma_filter *filters_null lzma_attribute((__unused__)),
482         const lzma_filter *reversed_filters)
483 {
484     if (coder->lz.options_update == NULL)
```

(c) src/liblzma/lz/lz_encoder.c

Figure 3.19: Example of source code in xz

all the other similarities between clone sets, because these two clones share a large amount of identical code fragments. As the result, 7 out of 8 users thought that clone pair α were true clones and 6 out of 8 users thought that clone pair β were false clones, while FICA always thought they belonged to the same group.

Another example comes from the `e2progs` project shown in Figure 3.20. It is clear that two code clone pairs are found in Figure 3.20. The code from Figure 3.20(a) and that from Figure 3.20(b) form the first code clone pair, referred to as clone γ, and the code from Figure 3.20(c) and Figure 3.20(d) forms the second, referred to as clone δ. Users can tell these two clone pairs are different because clone γ consists of two function bodies and can be merged into one function, while clone δ consists of two lists of function declarations. However, the calculated similarity between clone γ and δ is greater than 6.4%, which is large enough to affect the overall result. The reason why FICA regarded them as similar is that they share many common N-grams, such as `STATIC ID ID LPAREN CONST` or `CONST ID TIMES ID COMMA`, which results in the high value of similarity.

Based on the above discussion, we learned from the examples that comparing code clones by only their literal similarity has some limitations. We will continue to learn how much this limits the result and whether we can improve the accuracy by combining other methods, such as the hybrid token-metric based approach.

3.5.10 Threats to Validity

The internal and external validity of our methodology faces several threats.

First, we focused on whether a code clone is a true clone, which depends on the subjective judgment of the particular user. Therefore, the subjective nature of the experiment leads to a major threat to the internal validity of our work. We sent out invitations to participate in the experiment through mailing lists and twitter; therefore, most of the authors participated in the experiment and most of them shared an academic background. We tried to reach software developers in industry, but failed to collect enough amount of data to be useful in this study.

Second, while conducting the experiment, we could not enforce a general rule for the participants, so we depended on the participants to maintain consistency during the experiment, which is not a trivial task.

Third, the accuracy and precision of our work were evaluated against the set of true clones, which can be affected by the subjective emotions of the participants. We also recorded the time spent by each participant during the selections, and found that the participant with the worst prediction accuracy spent the longest time. However, we could not find a similar correlation among other participants.

Moreover, the implementation of FICA and the way we conducted the experiment may also be a threat to internal validity. Although FICA did not enforce the

```
48      return 0;
49  }
50  static int parse_block(
51    const char *request,
52    const char *desc,
53    const char *str,
54    blk_t *blk)
55  {
56    char *tmp;
57    *blk = strtoul(str, &tmp, 0);
58    if (*tmp) {
59      com_err(request, 0,
60        "Bad %s - %s",
61        desc, str);
62      return 1;
63    }
64    return 0;
65  }
66  static int
67  check_brel(char *request)
```

(a) tests/progs/test_rel.c

```
44      return 1;
45  }
46  static int parse_inode(
47    const char *request,
48    const char *desc,
49    const char *str,
50    ext2_ino_t *ino)
51  {
52    char *tmp;
53    *ino = strtoul(str, &tmp, 0);
54    if (*tmp) {
55      com_err(request, 0,
56        "Bad %s - %s",
57        desc, str);
58      return 1;
59    }
60    return 0;
61  }
62  void
63  do_create_icount(int argc,
64    char **argv)
```

(b) tests/progs/test_icount.c

```
68  static errcode_t test_write_blk64(io_channel channel,
69      unsigned long long block, int count, const void *data);
70  static errcode_t test_flush(io_channel channel);
71  static errcode_t test_write_byte(io_channel channel,
72      unsigned long offset, int count, const void *buf);
73  static errcode_t test_set_option(io_channel channel,
74      const char *option, const char *arg);
75  static errcode_t test_get_stats(io_channel channel, io_stats *stats);
76  static struct struct_io_manager struct_test_manager = {
```

(c) lib/ext2fs/test_io.c

```
102 static errcode_t unix_write_blk(io_channel channel,
103     unsigned long block, int count, const void *data);
104 static errcode_t unix_flush(io_channel channel);
105 static errcode_t unix_write_byte(io_channel channel,
106     unsigned long offset, int size, const void *data);
107 static errcode_t unix_set_option(io_channel channel,
108     const char *option, const char *arg);
109 static errcode_t unix_get_stats(io_channel channel, io_stats *stats)
110 ;
111 static void reuse_cache(io_channel channel, ...
```

(d) lib/ext2fs/unix_io.c

Figure 3.20: Examples of source code in e2fsprogs

exact order in which the participants viewed the code clones, it is quite possible that these participants were affected by the order in which FICA listed the clones. They could gain experience during the former part of the experiment and apply the experience during the latter part.

Our experiment with participants assumed that these participants were independent. We sent out invitations to participate in the online experiment through email and twitter. It is quite possible that many of these anonymous participants were from the same clone-related research group in our university. Therefore, they may share some common knowledge that was not common for all participants. We tried to reach more participants outside our research group, for example, experts with an industrial background. However, it was very hard to collect data from programmers who had no interest in code clone research.

With regard to external validity, our current implementation and experiments only covered a small part of our overall methodology. We combined FICA with a token-based CDT without proving that it can work with other CDTs, although the method should be CDT neutral. Also, FICA only experimented with Boolean marks of the code clones. Whether other kinds of marks such as range marks or tag marks will affect the result is uncertain.

Chapter 4

Revealing Purity and Side Effects on Functions for Reusing Java Libraries

4.1 Introduction

It is difficult for programmers to reuse software components without fully understanding their behavior. The documentation and naming of these components usually focuses on *intent*, i.e., what these functions are required to do, but fails to illustrate their *side effects*, i.e., how these functions accomplish their task [15]. For instance, it is rare for API function[1] signatures or documentation to include information about what global and object states will be modified during an invocation.

It is hard to reuse the modularized components, because of the possible side effects in API libraries. For instance, it is usually unclear for programmers whether it is safe to call the APIs across multiple threads. In addition, undocumented API side effects may be changed during software maintenance, making debugging even more challenging in the future [16].

By understanding of side effects in the software libraries, programmers can perform high level refactoring on the source code that is using the functional part of the libraries. For instance, the return value of math functions such as `sin` will be the same result if the same parameter is passed, therefore the result can be cached if the same calculation is performed more than once. Moreover, the calculation without side effects are good candidates for parallelization[17]. However, the purity

[1] We interchange the term *function* with the term *method* throughout this paper referring to the same thing. We use *function* to refer the ideas that originate from the functional paradigm, and *method* to refer the ideas that originate from object-oriented paradigm such as Java.

information is usually missing in external libraries, therefore programmers would risk introducing bugs with such refactorings, for example, caching the result of a function which depends on the mutable internal state.

In this paper, we present an approach to infer a function's purity from byte code for later use. Programmers can use effect information to understand a function's side effects in order to reuse it. For example, the approach can help to decide whether it is safe to cache or parallel a time-consuming calculation.

The contributions of this research include:

- An extended definition of purity as *stateless* or *stateful* in object-oriented languages such as Java.
- An approach to automatically infer purity and side effects,
- A concrete implementation for Java bytecode,
- A set of method annotations that document the details of effects such as return value dependencies or variable state modifications, for programmers to understand the effects.
- Experiments on well-known open source software libraries with different scale and characteristic.

In our experiments,we found that 24–44% of the methods in the evaluated open source Java libraries are pure. Also, we observed methods that should be pure in theory but not in the implementation, and revealed tricks or potential bugs in the implementation by a case study of our approach.

We achieved the same percentage of pure functions with the existing study without a manually created white-list, and we revealed which side effects these functions were generating which would not found in the existing studies. We focused on revealing these side effect information on real world software libraries to be used by the programmers and tools.

Furthermore, we focus on refactoring these pure functions and propose a new category of high-level automatic refactoring patterns, called *purity-guided refactoring*. In the following part of this paper, we will discuss the purity-guided refactoring. As a case study, we applied a kind of the purity-guided refactoring, namely *Memoization* refactoring on several open-source libraries in Java. We observed the improvements of the performance and the preservation of semantics by profiling the bundled test cases of these libraries.

The following sections cover the details of these contribution in separate sections.

4.2 Purity and Side Effect

In this section, we firstly discuss how to adopt the ideas of pure functions into a traditional object-oriented(OO) language. Secondly we define a formal syntax of targeted language as the basis of our discussion. Thirdly we discuss our definition of the purity and side effects based on the targeted language, with a set of annotations to document these informations. Lastly we discuss the rules that should be followed by these annotations.

4.2.1 Stateless & Stateful Purity of Functions

The notion of purity on functions does not match well with other OO paradigm concepts. In OO languages, program states are usually encapsulated within objects, which use well-defined boundary functions called methods to interact with each other. This is the opposite of a pure functional paradigm where states of the program are passing through function arguments.

Moreover, we noticed that most objects have a life span pattern of creation, use and destroy. Many objects will not change their states after properly created, and the methods called on them simply query these internal states. We would like to distinguish these state-querying methods from those methods that modify the states. Through our research we have observed that OO libraries can contain around 24-44% of functional code that does not modify the program's state.

Revealing the functional part of a program enables to perform refactoring by programmers and tools, such as parallelizing the computation, which is difficult without the knowledge of these side effects.

Based on the above observation, we defined a function as *pure* if it does not generate side effects such as modifying the state outside the object. Note that this definition is slightly different from the traditional definition of pure functions by return value dependencies [54]. Meanwhile, many existing studies such as [36, 38, 55] share the same purity definition with us. To illustrate the difference of two definitions, we divide our definition of a *pure* function into *stateless* and *stateful* functions:

Definition 4.2.1 (Stateless)**.** *If the return value of a pure function is only determined by the state of its arguments.*

Definition 4.2.2 (Stateful)**.** *If the return value of a pure function is also determined by the states of member fields.*

All other non-*pure* functions generate side effects. Although the notion of a stateful pure function may seem like a contradiction, we can view the state of field

members as extra arguments, so that they can be converted into the mathematical form of a pure function. An example of a stateful pure function is `equals` method in Java, which compares the value equality of two objects. Although they depend on an object's state, well-formed `equals` methods do not change the state.

4.2.2 Formal Syntax of a Core Language

Firstly we need to clarify what kind of languages we are targeting. We formalize the syntax of our targeted language as the *core language* in our discussion. This research focuses on Java-like programming languages. More formally, we are targeting *statically-typed type-safe OO* languages. This means that each variable in a program is associated with a statically defined type, and operations on these variables can be checked during compiling time. Wild pointers are not allowed in these languages, which gives us the ability to reason about the reference alias safely. Many industrial programming languages fall into this category, including C#, Java and Scala. Dynamic-typed languages such as PHP, Python and Ruby are not covered, where types of variables are determined at runtime. Note that although Scala eliminates most of the necessity on defining the type of variables, those types are statically inferred by the compiler, rather than determined at runtime.

Note that there are infrastructures that may break type-safety in the targeted languages, such as the `unsafe` keyword in C#. Studies suggest that unexpectedly breaking type-safety may be a consequence of implementation limitations [56]. Besides, there are studies that suggesting unexpected breakage of type-safety because of limitations on the implementations such as literature [56]. Our research does not address these problems and will generate false results if the target source code utilizes some of these infrastructures. For example, the program may modify some state through `unsafe` pointers, which is not possible to check statically.

Our approach differentiates the types of variables in the programs between reference types and value types. Variables with the reference types behave like the pointers in C++, i.e. they may point to the same object thus share the same state, meanwhile variables with the value types can not share the same state. We use the same definition with C# and Java. Value types in Java is limited to primitive types, while in C# programmers can introduce new value types with `struct` keyword in addition to primitive types. For the brevity, we also ignore the use of special reference facilities provided by libraries in these languages such as unsafe pointers, as they may break the type safety of these languages.

For brevity, we follow a style similar to prior work [57], restricting our formal definitions to core calculus in an A-Normal Form with the syntax in Figure 4.1. This core language abstractly models the syntax of a Java-like language. Our implementation extends this model and handles the general case of a normal Java

cd	::=	**class** C **extends** D $\{\overline{fd}\,\overline{md}\}$	$class$
fd	::=	τ f ;	$field$
md	::=	$\alpha\,\tau$ m($\overline{\tau\,\text{x}}$)$\{\overline{\tau\,\text{y}}\,b\}$	$method$
b	::=	$\overline{s}$	$block$
s	::=	**return** v ; | v=v ; |	$statement$
		if(v)$\{b\}$ | **for**(s; v; s)$\{b\}$	
v	::=	$vn\|vp\|vl\|vt\|vs\|vf\|$	$value$
		$vu\|vb\|va\|vc\|vm$	
vn	::=	**new** τ ($\overline{\tau\,\text{x}}$)	$new\,object$
vp	::=	x	$parameter$
vl	::=	y | c	$local\,variable$
vt	::=	**this**.f	$this\,field$
vs	::=	C.f	$static\,field$
vf	::=	v.f	$member\,field$
vu	::=	unary v	$unary\,operator$
vb	::=	v binary v	$binary\,operator$
va	::=	$v[v]$	$array\,access$
vc	::=	$(\tau)v$	$cast$
vm	::=	v.m(v)	$function\,call$

Figure 4.1: The formal syntax of our core language, where C and D are class names, τ is a type name, x is a parameter name, y is a local variable name, c is a literal constant, f is a field name, m is a method name, and α represents method annotations defined in Figure 4.3.

program.

4.2.3 Lexical State Accessors and Side Effects

The main purpose of this research is to reveal the side effects of functions. Therefore we need to define what is an effect and what is a side effect of a function.

Definition 4.2.3 (Effect)**.** *We define the effects of a function as the modifications to the states of the program, including the return value.*

Definition 4.2.4 (Side Effect)**.** *We define the side effects of a function as the modifications to the states of the objects or performing I/O operations.*

The effects of a functions are all the side effects plus the return value. According to the *single response principle* in [58], a function should have exactly one

effect, either calculating a value and return it, or doing one kind of modification to the state of the program. Disobeying this practice usually leads to problematic, unmaintainable coding style.

Definition 4.2.5 (Lexical State Accessor)**.** *We define a* lexical state accessor *to be any variable that is directly accessible within a function's lexical scope before the execution.*

In statically-typed OO languages such as Java, lexical state accessors of a function include the possible `this` pointer, the arguments, the member fields within the same class, and the static fields in any arbitrary classes. Note that local variables defined inside a function are excluded in the definition of lexical accessors, because they do not exist outside the function's body. We focus on lexical variables because they can be easily identified and understood from the function definition by programmers.

All possible modifications to the state of a program are achieved by accessing the aforementioned lexical state accessors. There are two forms of modification: changing the values of these accessors directly, or modifying indirectly though the use of lexical state accessors. These modifications are considered to be the side effects of executing the function. Additional side effects include directly or transitively calling system routines to perform I/O operations.

For example, in the Huffman algorithm source code in Figure 4.2, there are four classes with several member functions that may call each other.

The constructors of class `Leaf` and `Node` demonstrate side effects, as they changed the member fields. The methods `freq` in three classes demonstrate the data dependency on lexical state accessors, because they return the state from their internal member fields. These `freq` methods do not modifies the object states, therefore they are *stateful* pure functions by our definition. The overridden method `freq` in class `Tree` will be covered in the Section 4.2.5.

`A.changeArg` modifies the state of its argument through a copy of reference, which should be recognized as a side effect. `A.getM` exposes the state of a member field `m`, which makes it possible to modify the member outside the function. `A.modifyM` calls both `A.getM` and `A.changeArg`, transitively applying the side effect of `A.changeArg` to the exposed state from `A.getM` by modifying a member field. In contrast to these functions, `A.create` creates a new array and returns its reference, then `A.modifyTemp` modifies the temporary array created from it by calling `A.changeArg`, which does not change any existing object state in the lexical scope. Despite the fact that `A.changeArg` can have side effects, neither `A.modifyTemp` nor `A.create` modify the state of any existing objects.

The last function in class `A`, `change`, overrides the function in its super class `Base`, which will be covered in the Section 4.2.5.

```
class Tree extends Comparable{
  int freq() { return 0; }
  int compareTo(Tree t) {
    return freq() - t.freq();
  }
}
class Leaf extends Tree {
  char v; int f;
  Leaf(int fr, char va) { f = fr; v = va; }
  int freq() { return f; }
}
class Node extends Tree {
  Tree l, r;
  Node(Tree le, Tree ri) { l = le; r = ri; }
  int freq() { return l.freq() + r.freq(); }
}
class Main{
  Tree build(int[] chrs) {
    PriQueue q = new PriQueue();
    for(int i = 0;i < chrs.length; i = i + 1)
      if (chrs[i] > 0)
        q.offer(new Leaf(chrs[i], (char)i));
    for (;q.size() > 1;)
      q.offer(new Node(q.poll(),q.poll()));
    return q.poll();
  }
  void main(String[] args) {
    String test = "this is an example";
    int[] chrs = new int[256];
    for(int i = 0;i < test.length(); i = i + 1)
      chrs[test.getChar(i)]=chrs[test.getChar(i)]+1;
    Tree tree = build(chrs);
  }
}
```

Figure 4.2: Effects in Huffman Algorithm

$$
\begin{array}{rcl}
\alpha & ::= & \overline{\alpha_r(dp)}\ \overline{\alpha_m(dp, from, target)} \\
\alpha_r & ::= & \texttt{@Depend} \mid \texttt{@Expose} \\
\alpha_m & ::= & \texttt{@Field} \mid \texttt{@Static} \mid \texttt{@Argument} \\
dp & ::= & [\texttt{dependFields} = \{s[, s]*\},] \\
 & & [\texttt{dependStatic} = \{s[, s]*\},] \\
 & & [\texttt{dependThis} = \texttt{true},] \\
 & & [\texttt{dependArguments} = \{s[, s]*\}] \\
from & ::= & [\texttt{from} = \{s[, s]*\}] \\
target & ::= & \texttt{name} = s, \texttt{type} = t, \texttt{owner} = t \\
t & ::= & \tau\texttt{.class}
\end{array}
$$

Figure 4.3: Syntax for proposed annotations. τ is a type name. s is a string value.

Functions `B.thisAdd` and `B.staticAdd` demonstrate the data dependency of return value, where the return value of `B.thisAdd` depends on an argument and a member field, and the return value of `B.staticAdd` depends on a member field and a static field. These return value dependencies are effects but not side effects of the corresponding functions. The function `B.setS` changed the value of the static field. The function `B.changeArg` calls a function on a `Base` object, which is possible to modify the state of this object. And finally, the function `B.change` calls several modifier functions.

4.2.4 Effect Annotations

We introduce a set of method annotations to indicate the effects that can arise during invocation. For each method, several annotations can be prepended, each representing a side effect that for example modifies one member field. We define the syntax of these annotations in Figure 4.3. The proposed annotations express the effects such as direct or transitive modifications to lexical state accessors, with the possible data dependency between these effects and other lexical state accessors from the function. The data dependencies are captured in annotation records such as `dependThis`, `dependArguments`, `dependFields` and `dependStatic`, with detailed informations such as types and owner classes of the fields. Although the `this` pointer is not a mutable variable in the context of a target function, it is possible to compare the identity by using `this` pointer to other pointers or expose `this` pointer as return value of the function, hence the `this` pointer is included in data dependency.

- α_r: return value annotations. These two annotations capture the dependency

```
class Tree extends Comparable {
  @Depend(dependThis=true,
    dependFields= {"Tree Node.r", "int Leaf.f", "Tree Node.l"}
    from = {"int Leaf.freq()","int Node.freq()"})
  int freq() { return 0; }
  @Depend(dependThis=true,
    dependArguments= {"Tree tree"},
    dependFields= {"Tree Node.r", "int Leaf.f", "Tree Node.l"})
  int compareTo(Tree t) { return freq() - t.freq(); }
}
class Leaf extends Tree {
  char v; int f;
  @Field(type=int.class, owner=Leaf.class,
    name="f", dependArguments= {"int fr"})
  @Field(type=char.class, owner=Leaf.class,
    name="v", dependArguments= {"char va"})
  Leaf(int fr, char va) { f = fr; v = va; }
  @Depend(dependThis=true, dependFields= {"int Leaf.f"})
  int freq() { return f; }
}
class Node extends Tree {
  Tree l, r;
  @Field(type=Tree.class, owner=Node.class,
    name="l", dependArguments= {"Tree le"})
  @Field(type=Tree.class, owner=Node.class,
    name="r", dependArguments= {"Tree ri"})
  Node(Tree le, Tree ri) {
    l = le; r = ri;
  }
  @Depend(dependThis=true,
    dependFields= {"Tree Node.r", "Tree Node.l"})
  int freq() { return l.freq() + r.freq(); }
}
      class Main{
        @Depend(dependArguments= {"int[] chrs"})
        Tree build(int[] chrs) {
          PriQueue q = new PriQueue();
          for(int i = 0;i < chrs.length; i = i + 1)
            if (chrs[i] > 0)
              q.offer(new Leaf(chrs[i], (char)i));
          for (;q.size() > 1;)
            q.offer(new Node(q.poll(),q.poll()));
          return q.poll();
        }
        void main(String[] args) {
          String test = "this is an example";
          int[] chrs = new int[256];
          for(int i = 0;i < test.length(); i = i + 1)
            chrs[test.getChar(i)]=chrs[test.getChar(i)]+1;
          Tree tree = build(chrs);
        }
      }
```

Figure 4.4: Annotated Huffman Algorithm

of a return value:
`@Depend` specifies the dependency of a function's return value of the function.
`@Expose` specifies the exposed state of lexical variables returned from a function.

- α_m: modification annotations. These three annotations capture the modifications to lexical state accessors. Multiple annotations of the same kind are possible when there are multiple modifications within the function.
  `@Field` represents the modification to member fields.
  `@Static` represents the modification to static fields.
  `@Argument` represents the modification to arguments, which should be a reference type so that pass-by-value semantics can affect the state of real arguments.
  The members of these modification annotations are:
  `name` represents the name of the field.
  `owner` is the name of the class that defined the field.
  `type` is the type name of the field.

Two return value annotations α_r capture the dependency of return value, whether it depends on other state accessors or exposes the state of other state accessors. Three modification annotations α_m capture the modifications on the lexical state accessors. Multiple annotations of the same kind are possible if there are multiple modifications in the function.

Taking the source code from Figure 4.2, the annotated version of the same source code is represented in Figure 4.4. We can see how the function effects discussed in the previous subsection directly map to the example annotations. The schema of Java annotations allows us to directly use meta-class objects such as `Tree.class`, rather than using a string representation.

Although `@Argument` annotations may be more suitable on the corresponding arguments, the syntax would become ugly. In this simple example, the annotation occupies more lines than the source code on the constructors, but in actual code they are less distracting. As the *single response principle* suggests, the side effects of a function should be as little as possible. For example, the function `Main.build` in Figure 4.4 is annotated with only one `@Depend` annotation, which is fewer lines than its function body.

The effect annotations are intended to be used by both programmers and tools that process the program, as a contract describing the given function. This contract can be viewed as a complimentary to the function signature and exception specification that imposes restrictions on the implementation of the function.

An example application of these annotations is for a programmer to annotate a function to restrict a function's possible side effects, and then use a static checker to ensure this restriction. For example, programmers can ensure that the `build` function in Figure 4.4 is a *pure* functions and its return value only depends on the argument. If there were any other side effects occur in the `build` function, a static checker can notify them. Another kind of application would be to use the static checker to infer these annotations automatically from the given program, thus helping programmers to understand the function's side effects. We will discuss about the process of such kind of static checker in Section 4.3.

4.2.5 Reverse Inheritance Rule

Effect annotations are viewed as part of the method signature to indicate its behavior. However, it is impossible to know in general exactly which method will be statically called due to dynamic dispatch in OO languages. Like the inheritance rules for checked exceptions in Java expressed in a `throws` clause, purity annotations on subclass methods must be more restrictive than the methods they override. Therefore we merge the annotations from methods in the subclasses and annotate the merged result on the overridden methods in the super classes. We defined this rule as the *reverse inheritance rule*. Reverse inherited annotations in a super class are specially marked as "from" to distinguish them from annotations directly on the method.

As an example, in Figure 4.4, both `Node.freq` and `Leaf.freq` override the method in class `Tree`, so the method `Tree.freq` should inherit all the effects from these methods. Note that the name of arguments could be changed in overriding methods, so the positions of arguments are used to convert these names.

4.3 Automatic Inference of Purity and Side Effects

In this section, we present our approach to automatically infer the purity and side effects. And then, we will describe how to utilize our approach during the reusing of software components.

4.3.1 Call Graph and Data Analysis

The analyzer identifies method targets by using a class diagram and call graph. The class diagram records the inheritance relationship of classes (including interfaces) and the overriding relationship between methods in a class hierarchy. The call graph records the invocation instructions inside the method body, which points

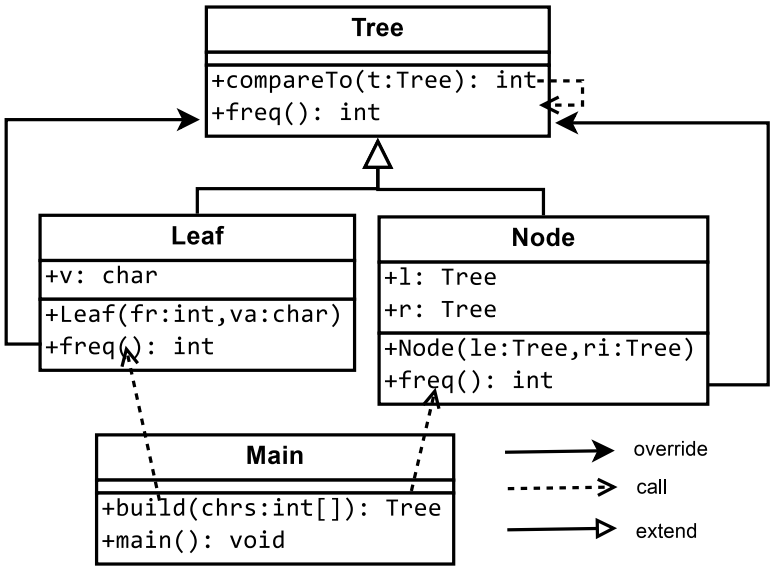


Figure 4.5: Class Diagram with Call Graph

to another method defined in the class diagram. An example class diagram is shown in Figure 4.5.

Our analyzer traverses all of the methods in the class diagram, inferring possible effects including side effects. We capture only the dependencies of lexical state accessors, during these three analysis stages:
data flow analysis estimates the return value dependency.
reference alias analysis identifies possible modifications to lexical state accessors that are side effects.
control flow analysis supports data dependence calculations on conditional branches.

There are three kinds of lexical state accessors as defined in Section 4.2.3, which are the *static field*s (shortened as S) of a class, the *member field*s (shortened as F) of an object, and the *argument*s (shortened as A) passed to the function.

Definition 4.3.1 (Data Dependency Set)**.** *We define a data dependency as the value of a lexical state accessor before a function executes, and a* dependency set *(DS) as the set of data dependencies such that $DS \subset \{x | x \in S \cup F \cup A\}$.*

The above definition of *dependency set* is used in both our data flow analysis and reference alias analysis. The difference between the *dependency sets* used in these two analyses is that we only consider reference type dependencies in reference alias analysis, and value type dependencies in data flow analysis. All depen-

dencies suitable in reference alias analysis are also suitable in data flow analysis, but not vice versa. We define two *dependency sets* used in these two stages of analysis as:

reference dependency (rd) is a DS of the possible reference aliases.
value dependency (vd) is a DS that affects the value.

Our analyzer interpret the code, follow the instructions in the given function, and apply the aforementioned three analysis. The analyzer begins its interpretation by breaking the code of a given function into statement *block*s using control flow analysis, where we define a *block* to be a sequence of statements. The *block* can be associated with a value of its condition if it is nested in a *if* or *while* statement. Next, the analyzer interprets each *block*'s instructions to evaluate the value dependencies and obtain a list of effects. During the interpretation stage, each value is represented as a triplet of its static type, a reference-dependency set, and a value dependency set ($V = (\text{type}, rd, vd)$).

At the beginning of the interpretation of the given function, the argument values are assigned with value and reference dependencies of themselves. Next we interpret each instructions of the function by following the transfer functions in Table 4.1. The input of a transfer function is V before the execution of the instruction, and the output is the new V after the execution. Besides the reference and value dependency sets in this table, the static types of these values should also be calcu-

Table 4.1: Transfer Functions for Values and Instructions

Instuctions[a]	Code Pattern	Reference Dependency [b]	Value Dependency
new object	**new** τ	$\emptyset$	$\emptyset$
parameter	x	$\{x\}$	$\{x\}$
local variable	y	$\emptyset$	$\emptyset$
member field	`this.field`	{field}	{field}
static field	`Class.field`	{field}	{field}
object field	V`.field`	V_{rd}	V_{vd}
unary operation	$op\,V$	$\emptyset$	V
binary operation	$V_1\ op\ V_2$	$\emptyset$	$V_{1vd} \cup V_{2vd}$
array access	$V_1[V_2]$	V_{1rd}	$V_{1vd} \cup V_{2vd}$
type cast	$(\tau)\,V$	V_{rd}	V_{vd}
assignment	$V_1 = V_2$	V_{1rd}	V_{2vd}
return value	`return` V	$\emptyset$	$\emptyset$
merge		$V_{1rd} \cup V_{2rd}$	$V_{1vd} \cup V_{2vd}$

[a] Defined in the core language syntax in Figure 4.1.
[b] If the corresponding type is a reference type, otherwise $\emptyset$

```
boolean f(int[] a, int b) {
  if(a.length > 0){ // condition depends on arg a
1:  int [] local = a;  // copy reference
2:  a = new int[1];    // overwrite reference
3:  a[0] = local[0];   // not modification
4:  local[0] = b;      // modify arg a
5:  b = a[0];          // not modification
6:  return true;
  }else{
    return false;      // depend on arg a
  }
}
```

Figure 4.6: Example of Data and Control Analysis

lated as defined in the language specifications. Note that the "merge" instruction in this table merges the branches of statements during the interpretation. Besides the instructions listed in the table, there is another important kind of instructions, the function invocations, described in Section 4.3.2.

During interpretation, possible function effects are collected when processing assignment instructions. We initially mark two kinds of dependencies: *modification behavior* for reference dependencies and *return statement* for value dependencies. Both dependencies are merged with the value dependency set for the current block.

An example of the interpretation stage is represented in Figure 4.6. At the beginning of interpretation, the reference dependency of `a` is assigned as argument `a`, and the value dependency of `a` and `b` are assigned as corresponding argument names. There are two blocks in this code which are associated with the branch condition `a.length > 0`. Since the value dependency of this condition is argument `a`, both two blocks depend on the state of `a`. Then, during the interpretation of the first block:

1. The reference of `a` is copied into `local`, which implies that the reference dependency of `local` is {`a`}
2. The reference dependency of `a` is now $\emptyset$
3. A *modification behavior* is performed on the reference dependency of `a`, which is $\emptyset$, and thus has no side effects.
4. A *modification behavior* is performed on the reference dependency of `local`, with a value dependency of {`b`}. An `@Argument` effect on `a` is generated with a data dependency on `b` and a control flow dependency on `a`.
5. A *modification behavior* is performed on $\emptyset$.

6. A *return statement* generates a `Depend` effect with a value dependency of $\emptyset$ and an value dependency of the constant `true`, which is then merged with the control dependency on `a`.

The analysis on the else block generates the same `Depend` effect, and these two `Depend` effect are then merged.

4.3.2 Effects from Function Invocations

There is one kind of important instruction that is not covered in Table 4.1, which will be discussed in this subsection. We refer to the function containing an invocation as a *caller*, and the function being called as a *callee*. When the analyzer sees a function invocation instruction during interpretation, it generates possible effects by examining the data flow across the invocation boundaries. Fortunately, this cross-function analysis is possible with the generated effect information on the callee, so that we do not need to examine the codes of the caller and callee at the same time.

When the effect annotations on the callee are not available during analysis of a caller, the analyzer simply ignores the invocation, pretends callee has no effects, and then refreshes the result when the annotations on the callee become available.

There are two kinds of invocation instructions in Java: static and dynamic dispatch. Dynamic dispatch is used to call virtual methods, and static dispatch is used to call non-virtual methods and special cases such as calling overridden methods defined in a super class.

During cross-function analysis, we use a different set of effect annotations according to the type of invocation, based on whether it includes the reverse-inherited annotations or not.

All of the invocation instructions share the same form as V_{obj} `.function(`$\overline{V_{arg}}$`)`. All side effects on static fields are transferred from callee to caller. If there are argument effects generated on the callee method, i.e., when the callee is modifying the state of a passed argument, then the analyzer will generate a modification behavior on the reference dependencies of corresponding position, as if the modification occurred inside the caller method.

The V_{obj} is the object that owns the method, which could be `this`, `ClassName` or a certain dynamically calculated value during the interpretation. Static member methods on `ClassNames` are guaranteed not to generate modification side effects on member fields. If a reference dependency of V_{obj} is `this`, all the modification side effect information on member fields will be copied, otherwise a single modification effect on the reference dependency of the current V_{obj} will be recorded. This behavior of analyzer follows the definition of lexical state accessors described

in Section 4.2.3 to distinguish between directly and transitively accesses of these accessors.

Finally, if the interpreted invocation expression returns a value, we need to determine the reference and value dependency of its return value. The reference dependency of the invocation expression is the reference dependency of return value from callee, and the value-dependency of this expression is the merged value dependencies of all V_{arg}.

With the effect information on the functions, we can simply determine whether a function is a *pure function*, and further, whether it is *stateful* or *stateless*. A function that has no modifications is considered to be a *pure function*. A pure function whose return value depends only on arguments is considered to be a *stateless* pure function.

4.3.3 Iteration to a Fix-point of Class Diagram

A function's effects depend on the effects of its callees as well its overriding functions, potentially causing a function to be analyzed several times. In addition, recursive functions may also be analyzed multiple times. We continue analyzing until the effects are inferred. We set a flag in each function on the class diagram to indicate whether the effects for this function need to be inferred or updated.

We also differentiate two sets of effects: *static effects* and *dynamic effects*, because we differentiate between static and dynamic dispatch invocations.

In the syntax of annotation defined in Section 4.2.4, the *dynamic effects* are recorded with a `from` clause, whereas the *static effects* have no `from` clause.

Firstly, we initialize all methods in the class diagram with both *static effects* and *dynamic effects* as $\emptyset$. Next we mark the flags for all of these methods as "need to be analyzed". Then, for each method whose flag is marked, the analyzer:

1. Merges the *static effects* with the result of the data analysis on this method.
2. Sets the *dynamic effects* to be the merge of *static effects* and all *dynamic effects* of the overridden methods.
3. Clears the flag on this method.
4. If the effects have changed since last analysis, marks the flags of all methods that depend on this method.

We continue iterating until none of the methods in the class diagram are marked, which means a fix-point of the analysis is reached. Note that during the execution of this algorithm, the size of both *static effects* and *dynamic effects* only increases and never decreases. There is an upper limit on the size, which is the sum of

numbers of all possible modifications to the fields and arguments in the program. With the monotone increasing property and the upper bound of the algorithm, we can guarantee that it will halt.

4.3.4 Applications in Reusing Software Components

We have described how our analyzer infer the effect information. Next, we will briefly introduce how to use our analyzer from a programmer's point of view.

Suppose a programmer is facing a reusable software component, either in distributed binary form or in source code form, and the programmer would like to know whether it is safe to reuse this component in his new code. The programmer can apply our analyzer on the candidate component, together with all its dependent libraries, to obtain a list of side effects on each functions from the component. The programmer can then decide whether it is safe to reuse the component based on the side effects.

For example, if the programmer is writing a multi-threaded program, and the candidate component is accessing some global states, then the programmer may need to introduce a thread lock to synchronize the accesses to these global states. As another example, if the candidate function is a pure function reported by our analyzer, then it is usually safe to reuse this function in the new source code without introducing hidden data dependency.

Moreover, the output of our analyzer can help the debugging and understanding of the behavior of software components. It is reported [59] that some bug will appear only if the programmer execute the unit test separately. Understanding the side effects could reveal these bugs even before executing the test cases.

4.4 Purity Analyzer Implementation Details

We discuss some of the implementation details of our analyzer in this section. We chose Java bytecode defined by the Java Runtime Environment (shortened as JRE) version 6 as our target language, and implemented the described analyzer based on the widely used ASM library [60]. There are several advantages in targeting an intermediate language rather than source code. First, the analyzer is syntax neutral, so we can automatically analyze all languages targeting the Java Virtual Machine. Second, the analyzer can be applied on binary libraries without source code. Finally, the type safety is assured by the JRE's compiler and bytecode verifier.

4.4.1 Dynamic Building Class Diagram

Our analyzer locates the bytecode of classes by utilizing classloaders in the JRE. Therefore, the target class files to be analyzed are found in `CLASSPATH` directories, and the `jar` archive file can be handled automatically by the classloader. Our analyzer builds the whole class diagram in several passes, and classes are loaded into the class diagram dynamically as needed.

The analyzer first takes a list of prefixes on the fully qualified class name matching against the manifests defined in `CLASSPATH`. These matched classes form the initial class diagram. Representations of their super classes are also loaded. These target classes together with their super classes are analyzed in the first pass.

Each class and function in the class diagram is visited exactly once, if its flag is marked, during one pass of analysis. The classes that contain called functions, together with their super classes, are loaded and analyzed in the next pass. Class diagram building iterates until no more classes are needed, and the whole analyzer stops when the effect annotations on every function have converged.

4.4.2 Details of Bytecode Processing

There are certain cases when the class file of a given non-abstract methods mentioned by some loaded methods can not be found in the `CLASSPATH` directories. For example, there are usually several types of logging facilities supported, while only one of them will be dynamically enabled during runtime, and the program is guaranteed to function normally when none of them are found. When this occurs, a special `@Call` effect annotation is introduced to indicate this unsolved calling.

It is possible for the compiler to generate functions that do not exist in the corresponding source code. These generated functions are called bridge functions by the Java compiler, and are usually marked as `synthetic` in Java bytecode. Some of them have a normal name if the compiler can generate one, others have unusual names like `access$100`, which does not typically occur naturally in Java source code. These bridge functions are analyzed normally by our analyzer, but the annotations on them will not appear in the output as they do not exist in the corresponding source code.

Another notable detail is the possible inconsistencies between function signature definitions between the Java language and the Java Virtual Machine. We need a definition of function signatures to determine the overriding relationship between the functions in super and derived classes. In the Java Virtual Machine (JVM), a function signature includes the specific return value type, whereas the Java lan-

guage excludes the return type from the definition of a function signature to support the *covariant return type* language feature (see Section 8.4.2 of [61]). For example, a class that contains both `int foo()` and `void foo()` definitions are not allowed in the Java language, because they have the same signature `foo()`. But they are allowed and recognized as a function overloading by the JVM, because JVM considers different return value in the function signature. Our analyzer follows the Java language definition. This choice eliminated the possibility to apply our tool on some bytecode files that are not generated by a Java compiler, usually from a different language. But this case is quite rare in practical.

4.4.3 Details of Interpretation

Our analyzer directly maps the instructions from bytecode directly to the operations defined in Figure 4.1. We need to differentiate between the access to member field of `this` pointer as state accessor and other kind of access through variables in lexical scope. Therefore we reasoned about whether the reference-dependency is `this` pointer. However, we can only identify those access of `this` pointer if it is expressed explicitly in the bytecode. It is possible to store `this` pointer in member field or static field, which will generate a side effect, then access it from these fields, which will be treat as accessing a variable other than `this` pointer. We consider this result as appropriate, as effects are still generated to indicate the modification on one of the lexical state accessors of the method. The same mechanism is used in judging whether a method is invoked on `this` object or object through variables from the lexical scope, which have been discussed in Section 4.3.2.

4.4.4 Manually Provided White-list Functions

When implementing our analyzer, we found that our purity analysis had a higher true-negative rate than expected. More precisely, we found that over 60% of the `equals` functions were annotated as having side effects by our analyzer. Recall that `equals` functions should be *stateful pure functions* as described in Section 4.2.1. Upon further study, we found that caching the calculation result inside a member field of an object leads to this high true-negative rate.

We found that the implementation of `HashMap.equals` modifies its member field `HashMap.entrySet`, and the implementation of `String.hashCode` caches the result in its member field `String.hash`. By our definition, these functions change the state of internal member fields, and thus are no longer pure functions. As a result of these two functions not being pure, callers of these functions were also marked as generating side effects.

Whether this caching behavior could still be count as pure or not depends on

```
class String{
  /** Cache the hash code for the string */
  private int hash; // Default to 0
  ...
  public int hashCode() {
    int h = hash;
    if (h == 0 && value.length > 0) {
      char val[] = value;
      for (int i = 0; i < value.length; i++) {
        h = 31 * h + val[i];
      }
      hash = h;
    }
    return h;
  }
}
```

Figure 4.7: Example of Cache Semantic in `java.lang.String`

how programmers would use our analysis tool. For example, the programmer may want to serialize all the internal states of these objects and preserve the serialized state during comparison by the `equals` function. As our result suggested, the internal states may have been changed during the invocation, therefore the serialized representation is not guaranteed to be identical. For this kind of usage, the functions should not be treated as pure. On the other hand, the programmer may only interested in the observable states by invoking well-defined public member functions, and for these programmers, the functions with caching ability should be treated as pure functions. We leave the options to the users of our tool to decide their purity.

To allow the user to override this setting so that the output annotations match their expectations, we enable our analyzer to take a white-list of pure functions from the user (regardless of whether our analysis identifies them as pure). For the white-list, we only calculate data dependencies on the return value. Users can select whether the functions in the white-list are inheritable, i.e. whether the functions override them should also be in the white-list.

4.4.5 Detection of Cache Semantics

Although the described analysis works well for identifying modification behaviors in theory, we find a difficulty to apply it in practice when member fields are used solely to cache the calculation results. We refer to the member fields that are used to cache the calculation results as having cache semantics. We found that the implementation of `HashMap.equals` modifies its member field `HashMap.entrySet`, and the implementation of `String.hashCode` caches the result in its member

field `String.hash`, as shown in Figure 4.7. By our definition, these methods change the state of internal member fields, and thus are no longer pure functions. As a result of these two methods not being pure, callers of these methods were also marked as generating side effects.

One example of the caching semantic can be found in the `String` class in the implementation of the standard JRE, as shown in Figure 4.7. As we can see from the source code, `hashCode` caches its calculation result in a member field `hash` at the first time of calling, and return the cached result afterwards. The `hashCode` function will generate a modification to member field behavior during the previous data analysis, although changing the caching field `hash` do not modify the calculation result.

This caching semantic is not only found by us, but also described in previous literatures such as [36]. A widely accepted solution to this problem was to accept a white-list of functions from the user (called *special* methods in [36]), indicating that they are proven to be pure by the user manually. For the reason that the selection of the white-list will impose great impact on the precision of the analyzing result, and they involve human judgments, we do not consider this as an ideal solution.

To preciously and automatically analyze this kind of methods that have caching semantics, we extend our analyzer to detect the cache semantics using a heuristic approach. More precisely, we consider a member field of a class having the cache semantic if all the following preconditions are true:

P1 The field is assigned either by a constant value, or in only one member function.

P2 The non-constant assignment on the field occurs within a branch block.

P3 The right-hand value of the non-constant assignment is only depended on other fields.

P4 The branch condition of the block checks that the value of the modified member field is a constant value.

We consider the following values as constant values: constant literals, null pointers and values of `static final` member fields that have a primitive type. The assignment with a constant value is considered as re-initializing the state of the cache field. The checking with a constant value is considered as checking the initialized state. In either cases, the value of the field is determined by other fields, therefore, it cannot be used to store a mutable state of the object.

In the example of `String`, the member field `hash` is assigned by `hashCode` with a calculation result and by its constructor with a constant value, therefore P1

is true. The assignment to `hash` occurs in a `if` condition block, therefore P2 is true. The value of the assignment is depend on the member field `value`, therefore P3 is true. Lastly in the condition block, the value of `hash` is assured to be zero by the condition check `h==0`, therefore P4 are true. The member field `hash` meets all the preconditions, therefore it is considered to be a caching field by our analyzer.

The modification behavior on the detected caching fields are suppressed from the effects, and the return value dependencies on these caching fields are ignored.

4.5 Purity-Guided Refactoring

Source code refactoring is generally defined as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [18]. Refactoring is one of many important tasks in software development and maintenance. Integrated Development Environments (shortened as IDEs) generally provide supports for common refactoring operations, which improve maintainability, performance or both, during the software development process. Machine-aid refactoring is so important that it is promoted as one of the best practices in both the Extreme programming [19] and Agile Software development [20], so that both of them heavily depend on automatic tools to perform refactoring during the development process.

It is emphasized to preserve the external behavior while performing refactoring so that they can be safely conducted without being evaluated about the breaking changes. However, it is hard to reason about the semantic behavior of a code fragment automatically by refactoring tools, due to the lack of semantic information from the traditional static analysis. Therefore, refactoring tools provided by IDEs generally take more conservative approaches by checking the syntactic structure of a given code fragment only in the pre-conditions of the refactoring operations.

Determining the possible semantic behavior from only the syntactic structure of a given code fragment puts heavy limitations on the possible refactoring patterns. Therefore, the refactoring tools provided by IDEs are limited to low-level structural restructuring on the code fragment, such as *Rename Method* or *Extract Method*. Although there are more high-level refactoring patterns widely recognized and adopted by programmers, such as *Replace Loop with Collection Closure Method* [21], currently they are not provided automatically by tools or IDEs and are applied manually by programmers.

In the other hand, preservation of semantic behavior can be statically checked for a certain part of the source code, namely the code that use pure functions. Pure functions are the functions that do not have observable side effects during the execution. With the property to be side effect free, whether refactoring changed

```
public class PreciseDateTimeField ... {
  @Depend(dependFields="long PreciseDateTimeField.iUnitMillis")
  public long getUnitMillis(){
    return iUnitMillis;
  }
  @Depend(dependArguments={"long instant"},
    dependFields={"int PreciseDateTimeField.iRange",
      "long PreciseDateTimeField.iUnitMillis"})
  public int get(long instant) {
    if (instant >= 0) {
      return (int)(instant/getUnitMillis())%iRange;
    } else {
      return iRange-1+(int)(((instant+1)/getUnitMillis())%iRange);
    }
  }
}
```

Figure 4.8: An example of `@Depend` effect annotations by *purano*

the semantic behavior of the code fragments that use pure functions can be easily checked by tools. Therefore, more refactoring patterns become available when the purity information is inferred from the source code. For instance, the return value of math functions such as `sin` will be the same result if the same parameter is passed, therefore the result can be cached if the same calculation is performed more than once. Moreover, the calculation without side effects are good candidates for parallelization [17].

4.5.1 Inference of Purity Information

We use the *purano* analyzer described in the previous section to infer the previously described effect annotations. *Purano* analyzes the provided bytecode of given Java software, together with all the binary dependency libraries. The output of *purano* is a set of annotations that record the purity and side effect information of all the methods in the given Java software.

An example of the output result from *purano* can be found in Figure 4.8. From the figure, we can see that *purano* is capable of identifying the *get* method in PreciseDateTimeField as a pure function, because its return value depends on both the value of its argument and the state of two member fields `iRange` and `iUnitMillis`.

More importantly, *purano* collects the data dependencies transitively across the function invocation boundaries. As shown in the example, the dependency on member field `iUnitMillis` is passed from the member method `getUnitMillis`. Besides the demonstrated `@Depend` annotation in the example, *purano* will also generate side effect annotations described in the previous subsection if the function may generate observable side effects during the execution.

4.5.2 Purity Queries During Refactoring

With the purity information at hand, refactoring tools can query several semantic properties on a given code fragment that is not available in its syntactic structure. These semantic properties are useful during both pre-condition checking and code restructuring. To name a few, refactoring tools can ask the following queries:

Q1 **Is this function pure?** The tools can query whether the function has modification annotations. Whether the given function is *stateless* or *stateful* can be further checked by querying whether the return `@Depend` on the member fields.

Q2 **Are the objects of this class immutable?** The tools can query whether the class includes a member function except for constructors that generate `@Field` side effects or returns an `@Exposed` member field.

Q3 **Which methods modify the value of this field?** The tools can query on all member methods whether the method generate a side effect of type `@Field` on the field or `@Expose` the reference to the field.

This list of queries is rather ad-hoc and far from complete. We only listed the queries that we use during the case study that will be described in the following section.

4.6 Case Study: *Memoization* Refactoring

Memoization refactoring (also referred as memorization) is a refactoring pattern that caches the result of a given function and returns the same result when the function gets called afterward with the same set of arguments. The implementation will store the result of the given function into a key-value storage with the set of the arguments as the key. The purpose of this refactoring is to reduce duplicated calculations and trade the CPU time with the memory.

The idea of the *Memoization* refactoring is simple and widely recognized. But it is often error-prone to apply the refactoring in practice, as the return value of the function may depend on internal states that may change between the function calls. Traditionally it requires the programmers to manually check these internal states and identify the locations that change them, which largely prevents the programmers from adopting this refactoring. Therefore, automatically tracking these internal states is essential for this refactoring.

By understanding the purity and side effects of functions in given code fragments, memoization can be safely applied. Tracking the internal states is no longer required for pure functions as they have been ensured to be side-effect-free.

In this research, we apply the *Memoization* refactoring on Java member methods, although we believe the same approach could be easily extended to other object-oriented languages. The target of the *Memoization* refactoring is always a member function in Java. We will describe the pre-conditions need to be checked, the general restructuring pattern, and some optional optimizations in detail. Further, we will discuss our considerations on the preservation of the purity.

4.6.1 Pre-conditions

The following pre-conditions should be hold for the target:

P1 **The function is pure.** This can be checked by **Q1**. Also, this implies that the return type can not be `void`.

P2 **The types of the arguments are immutable.** This can be checked by **Q2** for object types and ensured for primitive types in Java. Immutability for arguments is required to store them as keys in a key-value storage. Further, we rely on the correct implementation of `hashCode` in the classes of the arguments, but currently we do not check the correctness in our refactoring implementation.

P3 **The return value depends on no static fields, no public member fields nor publicly `@Exposed` member fields.** The refactoring tools can check this by retrieving the data dependencies from a `@Depend` annotation and querying **Q3** on the `dependFields`. This ensures that only member methods in the same class can affect the return value of the target function.

4.6.2 Refactoring Steps

Our refactoring tool applies the following steps on the target functions that meet all the pre-conditions:

1. Creates a private member field that served as the key-value storage in the class of the function. The key type of the field is a tuple of all the types of the arguments. The value type of the field is the type of the return value of the function. In the implementation, we use `HashMap` as the concrete key-value type and use the tuple types from a third-party library called *javatuples*.

2. At the entry point of the function:

   (a) Defines a tuple variable that captures all arguments.

(b) Checks whether the tuple is already stored in the key-value storage. If it contains the tuple, return the corresponding value stored in it.

3. At every exit point of the function:

   (a) Puts the calculated value into the key-value storage, with the captured tuple as the key.

   (b) Returns the value.

4. For each member function that modifies one of the `dependFields` in `@Depend`:

   (a) Empties the key-value storage right after the modification happens, so that the result will be re-calculated next time.

4.6.3 Optimizations

Depending on the target functions, optimizations can be applied to avoid unnecessary overheads and improve the maintainability of the generated source code.

O1 If there are no arguments, the key-value storage can be replaced with a simple member field whose type is the type of the return value. We use a `null` pointer to represent the empty state of the key-value storage.

O2 If there is only one argument, the `Unit` tuple used as the key can be replaced as the argument directly.

4.6.4 Preservation of the Purity on Functions

After refactoring, the target functions inevitably modify the key-value storage as a side effect. Therefore, the purity of the target function changed from pure to impure, by the definition of a pure function in related studies [36, 40]. However, the observable purity of the function is actually preserved, as still the return value of the function is determined by the state of its arguments and member fields excluding the newly introduced key-value storage.

One advantage of our purity analyzer *purano* is that it adopts a heuristic approach to automatically identify this kind of the cache semantics, and automatically exclude the fields that are used in caching from the purity analysis. As a result, the purity of the target function is still reported as pure by our analyzer. Therefore, we can say the purity of the target function is preserved during the refactoring. The preservation of purity is also held for other functions involving in the refactoring

because the modification on the key-value storage is introduced if and only if there are modifications on other member fields.

We value the preservation of purity throughout our study because we view the purity as not only the output result of our analyzer but also an inherently designed property of the function. It can serve as a metric to indicate the "functional" aspect of the software implementation.

4.7 Experiments

We implemented our analyzer with name *purano*[2], and evaluated it on real world software components in terms of accuracy, performance, and the distribution of different kinds of effects in different scale of software components. During the experimentation, we expected to answer the following research questions:
RQ1 What is the distribution of pure and side effect methods in the software libraries?
RQ2 How is the accuracy of our analysis comparing with an existing study? How is the heuristic approach in the detection of cache semantic compared to the white-list approach?
RQ3 How to utilize the revealed information during reusing the software components?

Firstly, we will answer the 2 research questions by experiments. Then we will demonstrate how would our study help programmers in RQ3 as a case study.

Lastly, we conducted an experiment on the proposed *Memoization* refactoring to evaluate our approach and to demonstrate the possibility of the *purity-guided refactoring* in general.

4.7.1 R1: Distribution of Effects

To show the distribution of purity and side effects of the methods in real world software libraries, we experimented on 4 target software projects, listed in Ta-

Table 4.2: Experiment Target and Analysis Performance

Software	Analyzed Classes	Target Classes	Target Func.	Time	#Pass
purano	2,942	253	2,372	148s	16
htmlparser	5,795	156	1,645	112s	17
tomcat	7,673	772	8,824	186s	18
argouml	11,608	2,545	20,167	233s	22

[2]We have published *purano* at `https://github.com/farseerfc/purano`.

ble 4.2. These experiments were executed on an octo-core Xeon E5520 with a 2GB heap size limitation. *purano* is the implementation of the analyzer of this paper, which includes a modified version of the ASM library. Both *htmlparser*, *tomcat* and *argouml* are well-known open source Java projects, and we used their latest stable binary distributions. Note that all of these software projects were analyzed together with the JRE standard libraries, because the analyzer need the purity and side effect information for all functions being called including the ones in the libraries. This lead to the much greater number of analyzed classes than the number of the target classes. According to the Javadoc for JRE 7, there are 3,793 public classes altogether, and more private ones in the JRE library. The analysis time of *argouml* was around 4 minutes, which is reasonable for large scale software. The number of analysis passes ranged from 16 to 22, which was depended on the longest invocation and overriding chain in all analyzed methods. Based on the analysis times in Table 4.2, we can conclude that the performance of our analyzer is reasonable within a daily programming environment, although it could be further optimized by caching the result of the standard libraries.

The purity of functions of the experimental result is listed in Table 4.3. The number of functions with side effects are listed in Table 4.4.

From the output, we find that around 24%–44% of the methods in these software projects were marked as *stateless* or *stateful* pure functions. We manually confirmed the generated result for *purano* to make sure it matched our expectation. The *argouml* project contains many non-pure graphical code percentage and the

Table 4.3: Percentage of Effects

Software	Pure Functions		Side Effects
	Stateless	Stateful	
purano	382 (16.1%)	192 (8.0%)	1,798 (75.9%)
htmlparser	363 (22.1%)	358 (21.8%)	924 (56.2%)
tomcat	1,260 (14.3%)	1,861 (21.1%)	5,703 (64.6%)
argouml	5,019 (24.9%)	1,744 (8.6%)	13,404 (66.5%)

Table 4.4: Breakdown of Side Effects

Software	Modifying		
	Member Fields	Static Fields	Arguments
purano	1,548	1,087	485
htmlparser	679	462	143
tomcat	4,346	3,990	1,288
argouml	7,057	11,849	3,255

htmlparser project have more pure functional code percentage.

4.7.2 R2: Comparison with an Existing Approach

While there are none of existing studies to identify the side effect informations within our knowledge, there are studies that only infer the purity of the functions based on different approaches. Therefore, we compare our purity result with one of the existing studies to examine the accuracy of our analysis. We ran our tool against the JOlden benchmark used in [36]. The result from the benchmark is shown in Table 4.5, comparing with the result from their study. Also we run our analyzer in two different configurations. One configuration is using a white-list which is similar to the configuration of [36], with the detection of cache semantic disabled. Another configuration is using the detection of cache semantics.

Their approach relies on a whole program analysis starting from a `main` entry point, and thus they covered fewer functions than our tool. They chose a set of functions for the white-list by viewing all the source code manually in advance, a time-consuming task in practice, while our approach automatically identifying the cache semantics. We were unable to compare precision and recall due to challenges

Table 4.5: Comparison on JOlden Benchmark. Function numbers are different because our approach analyzes all functions while Sălcianu's approach analyzes only the functions invoked transitively from the `main` entry point.

Application	Our (White-list)				Our (Cache Semantic)			Sălcianu's	
	Total	Stateless	Stateful	Pure	Stateless	Stateful	Pure	Total	Pure
BH	73	14	17	31	13	13	26	59	28
BiSort	15	6	0	6	5	0	5	13	5
Em3d	23	7	3	10	5	2	7	20	8
Health	29	8	1	9	8	0	8	27	13
MST	36	8	11	19	5	9	14	31	17
Perimeter	50	28	11	39	28	9	37	37	33
Power	32	2	4	6	2	4	6	29	9
TSP	16	5	1	6	4	1	5	14	5
TreeAdd	12	3	1	4	2	1	3	5	2
Voronoi	73	11	31	42	12	33	45	70	50

in executing their tool in our environment. Therefore we compared with their result from the published literature [36]. As we can see from the result table, we achieved a similar result on the number of pure functions. In addition to the number of pure functions shown in the result, we identified all the side effects and the type of purity, which is the main purpose of our study and cannot be found in their result.

Moreover, the compared existing study only experimented on the artificial benchmarks, while we applied our analysis on real world open source projects in the previous experiment. The answer to RQ2 will be that we identified a similar number of pure functions with the existing study while we generate more precious information about what kind of the side effect are generated from the libraries.

Comparing our result with different configurations, we can see that the detection of cache semantics result to a slightly lower pure percentage than the white-list approach. This is excepted, as the heuristic detection approach cannot find all the fields that are used for caching purpose without increasing the false positive rate. For example, we cannot detect the cached result within an entry of a hashmap instead of a single field. We consider the heuristic detection approach is more applicable for the existing software libraries because the programmers usually do not have a clue of which API functions are the libraries using and whether they are pure functions. Revealing this information is the main purpose of the purity analysis in the first place. An automatic technique like our approach will break the chicken or the egg dilemma and enable the purity analysis to be adopt in practice.

4.7.3 RQ3 A Case Study: Purity of `equals` and `hashCode`

Different programmers may use our tool for their own usages. Therefore, we conducted a case study to illustrate one possible usage of our tool. We examined the inferred effects on two methods, namely `equals` and `hashCode`. These two methods are related with the value equality of objects in Java, and they are used by collection classes such as `HashMap`. The programmer must ensure that the return values of these methods reflect their value equalities, and hence these return values should depend on the state of the objects. Therefore, we expect these methods to be *stateful* pure functions if they contain member fields. The purity types of these two methods are listed in Table 4.6.

To further understand the result, firstly we focused on the existence of *stateless* pure functions in Table 4.6 by manually examining their source code. Most of these methods are defined in interfaces or abstract classes. There were also 2 `equals` and 6 `hashCode` methods defined in the classes that do not have member fields. There were 9 `equals` that compares referential identities defined in classes, while these classes have member fields that are not accessed in the `equals`. These were used in unusual cases when comparing by referential identity rather than value

```
package javax.swing.text;
public class DefaultCaret extends Rectangle ... {
  /* Compares this object to the specified object.
   * The superclass behavior of comparing
   * rectangles is not desired, so this is changed
   * to the Object behavior.
   */
  public boolean equals(Object obj) {
    return (this == obj);
  }
}
```

Figure 4.9: A Special Design in `DefaultCaret`

```
package java.io;
public final class FilePermission ... {
  public boolean equals(Object obj) {
    ...
    return (this.mask == that.mask) &&
      this.cpath.equals(that.cpath) &&
      (this.directory == that.directory) &&
      (this.recursive == that.recursive);
  }
  public int hashCode() { return 0; }
}
```

Figure 4.10: A Potential Problem in `FilePermission`

identity is desired. An example of this kind of special design can be found in `DefaultCaret.equals`, shown in Figure 4.9, where the author explicitly documented in the Javadoc as *"The superclass behavior of comparing rectangles is not desired, so this is changed to the Object behavior"*. In addition, most of these classes are inner classes in Java with their names containing a "`$`" character. These inner classes are supposed to be used internally, where programmers control the creation of all objects. We found 3 `hashCode` that return a constant, whereas their corresponding `equals` compared the states of member fields. An example is shown in Figure 4.10, that `FilePermission.hashCode` will always return 0. The user of these classes must be aware of their respective behaviors, in order to avoid putting them in a `HashSet` or `HashMap`, or comparing them using equals. With the introduction of effect annotations as documentation, the user of these classes is able to notice the special behavior.

Next we examined the functions in Table 4.6 that generate side effects. Some classes such as `Date` and `Calendar` normalized their internal representation before comparing equality or calculating the hash code. Classes used in reflection at runtime, such as `java.lang.reflect.Class`, used a lazy loading technique to optimize general performance, which is similiar to the caching technique but

will change the observable state of the object. Due to the limitation of our heuristic approach in the detection of caching semantics, there are still some member fields used as the cache field that can not be automatically detected by our analyzer.

All of these implementation details revealed by our analyzer require special care in both development and maintenance of the software. We hope our research can aid the development in the situations like we have studied in this case study.

4.7.4 Experiment on *Memoization* Refactoring

We applied *Memoization* refactoring on 3 open-source software projects. These projects are collected from the maven repository and we use their latest available source code release. All these 3 projects can be directly built with `mvn install` command and tested with `mvn test` command by their bundled test cases. The version and statistic metrics of these 3 projects are listed in Table 4.7, where the code coverage is measured by block unit. HTMLPARSER [3] is a library to parse HTML. JODA-TIME [4] is a replacement for the Java date and time classes. PCOLLECTIONS [5] is a persistent and immutable analogue of the Java Collections Framework. We chose these projects for their relatively high coverage by the test cases, as shown in Table 4.7. The code coverage are measured by block unit.

In a real-world development process, refactoring is generally initiated by the programmer. The programmer may select the target functions by his knowledge and execute desired refactoring operations on them. Then refactoring tools will check the pre-conditions and perform the refactoring on the selected functions.

Table 4.6: Purity of `equals` and `hashCode`

Software		All	Pure Functions		Side Effects
			Stateless	Stateful	
purano	`equals`	518	19 (3.7%)	165 (31.9%)	334 (64.5%)
	`hashCode`	499	14 (2.8%)	176 (35.3%)	306 (61.9%)
htmlparser	`equals`	359	14 (3.9%)	141 (39.3%)	204 (56.8%)
	`hashCode`	355	10 (2.8%)	147 (41.4%)	198 (55.8%)
tomcat	`equals`	477	65 (13.6%)	282 (59.1%)	132 (27.7%)
	`hashCode`	473	52 (11.0%)	245 (51.8%)	176 (37.2%)
argouml	`equals`	426	55 (12.9%)	219 (51.4%)	152 (35.7%)
	`hashCode`	416	55 (12.2%)	214 (51.4%)	162 (28.9%)

[3] `http://htmlparser.sourceforge.net/`
[4] `http://www.joda.org/joda-time/`
[5] `http://pcollections.org/`

However, we do not have the deep knowledge on the target software as their developers in this experiment. Instead, we use a profiler to provide the knowledge about the performance of the source code. We use the `hprof` java-agent library provided with the standard JDK as the profiler.

In addition, we assumed that the programmer is provided with a purity analyzer like our *purano* by the IDE so that the purity information is available for the programmer. With these conditions, we conducted the experiment in the following steps for each software target:

1. Run the test cases by the profiler with the original code.
2. Get the top-20 *hot methods* from the profiling result that take most accumulated execution time.
3. Check the pre-conditions on all *hot methods* to get a list of candidates.
4. Apply the *Memoization* refactoring on all the candidates.
5. Run the test cases by the profiler with refactored code.

We checked the pre-conditions only on the top-20 hot methods because they have the most influence on the performance. We applied refactoring on 2 candidates in HTMLPARSER, 2 candidates in JODA-TIME and 1 candidate in PCOLLECTIONS.

The total execution time and heap usage before and after the refactoring provided by the profiler is shown in Table 4.8. We can see from the table that the performance improvements vary from 3.1% to 31.3% in these 3 projects. We measured JVM peak heap usage of test cases for these projects. The memory usage

Table 4.7: Experiment Targets

Software & Version	Class	Function	Pure	Test	Coverage
HTMLPARSER 2.2	157	1,643	707	472	67%
JODA-TIME 2.8.1	247	4,411	1,334	4,157	90%
PCOLLECTIONS 2.1.3	31	282	77	22	74%

Table 4.8: Profiling Results

	Performance			Heap Usage (MB)	
Software	Before	After	Improve	Before	After
HTMLPARSER	1m44s	1m31s	12.5%	13	16
JODA-TIME	60m10s	58m17s	3.1%	10	10
PCOLLECTIONS	16s	11s	31.3%	15	15

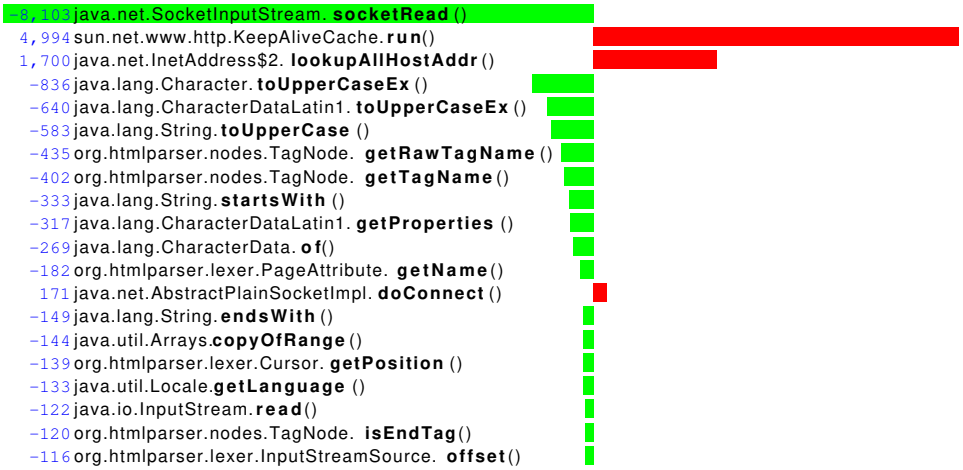


Figure 4.11: HTMLPARSER top-20 changed functions

increased 23% for HTMLPARSER, but not changed much for JODA-TIME and PCOLLECTION, because the objects doing caching are short-lived during the test cases. Meanwhile, we observed more objects are created during the test cases. For example, there are 27,991 more `String` objects created for HTMLPARSER and 13,893,640 more `Long` objects created for JODA-TIME.

We also compare the profiling results before and after the refactorings using *hpjmeter* and draw bar graphs on top-20 most changed functions in Figure 4.11 to 4.13. In these bar graphs, each bar represents a function. The length of the bar represents the changed accumulated time between two executions. The text titles of the bars are the differences of the accumulated execution time in milliseconds and the fully-qualified names. For example, we can see from the bar graph of JODA-TIME that the execution time of `IslamicChronology.isLeapYear` is reduced by 282s and the execution time of `Long.hashCode` is increased by 59s.

The execution time of each function in these test cases is generally short. Therefore, as we can observed from the bar graphs, the overhead of introducing *Memoization* is relatively large. In fact, the refactoring patterns applied on HTMLPARSER are optimized with **O1** while the refactoring patterns applied on JODA-TIME and PCOLLECTIONS are optimized with **O2**. As we can see, the main overhead in JODA-TIME and PCOLLECTIONS are introduced with `HashMap` operations.

Nevertheless, we observed the improvements in performance for all 3 targets. As all these projects are well-tested and focused on speed, we considered these

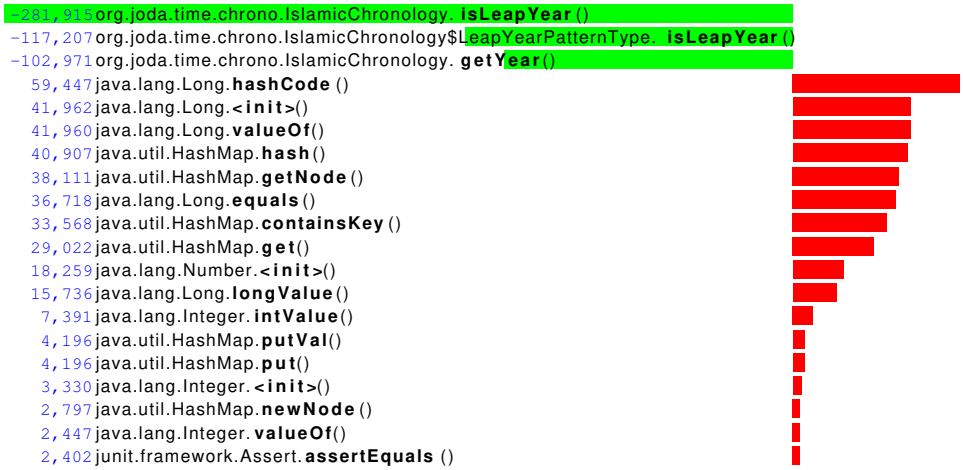


Figure 4.12: JODA-TIME top-20 changed functions

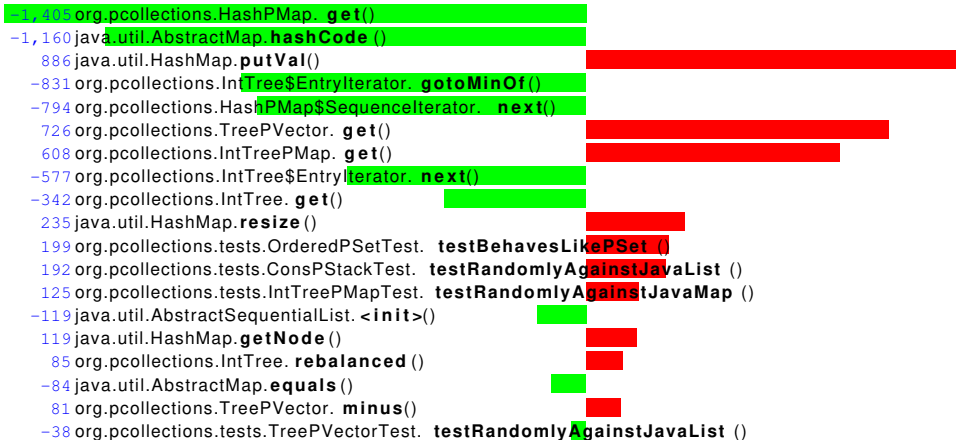


Figure 4.13: PCOLLECTIONS top-20 changed functions

results still remarkable.

As for the preservation of semantics, the results for test cases are not changed before and after the refactoring. In fact, only a single test case for HTMLPARSER failed, both before and after the refactoring, because the test case retrieved a URL on the Internet and the content have been changed since the test case was written. All other test cases passed without errors.

The experiment depends largely on the correctness of our purity analyzer *purano*, which is only used and tested in our research group currently. There may

exist bugs in the implementation of *purano*, and there are known limitations due to the decisions we made and the heuristic approaches we adopted, which are discussed in our previous paper [62]. However, we believed those decisions directly result from the goal of this research, which is to guide refactoring by the purity of the methods found in object-oriented languages.

We only experimented one kind of the *purity-guided refactoring*, namely *Memoization* refactoring, on 3 open-source Java libraries, and only tested them using test cases rather than real workloads. We are aware that the refactorings happen in real-world developments may differ from the experimental environments we have made. Meanwhile, as we tested on well-tested speed-oriented open-source projects and still observed improvements on performance, we believe that the refactorings in the real development process would have more opportunities to apply the proposed refactoring.

Chapter 5

Conclusion

This dissertation focus on using the **static** techniques to help program comprehension. It divide this the techniques into two dimensions, which are the **structural** information and **semantic** information. The final goal of this study is to aid programming comprehension with the statical information at hand in these two dimensions.

5.1 Conclusions on Comprehension based on Structural Information

We have shown that users of CDTs have different opinions on whether a code clone is indeed a true clone, which means "useful" or "interesting" according to their particular purpose. This observation suggested that filtering code clones should take user judgments into consideration to generate more useful list of code clones.

With this observation, we proposed a classification model based on applying machine learning on code clones. We built the described system FICA, which is a web-based system, as a proof of concept. The system consists of a generalized suffix-tree-based CDT and a web-based user interface that allows the user to mark detected code clones and shows the ranked result.

We conducted an experiment on the FICA system with 32 participants. Our classification model showed more than 70% accuracy on average and more than 90% accuracy for particular users and source code projects.

We analyzed the experiment in detail, and obtained several observations from the experiments about true code clones:

1. Users agree that false positive code clones are likely to fall into several categories, such as a list of assignment statements or a list of function declarations.

2. Users agree that true positive code clones are more diverse than false positives.

3. The minimum required size for the training set generally grows linearly with the number of categories that the clone sets fall into, which is less than the total number of detected clone sets.

The contributions of this work include:

- A machine learning model based on clone similarity.
- An approach to consider each individual user in code clone analysis.
- An experiment with 32 participants.
- Several important observations from the experiment.

5.2 Conclusions on Comprehension based on Semantic Information

The current implementation of our analyzer works on Java bytecode rather than source code. Besides all the advantages described, this decision is also made to ease the development, because it is easy to generate bytecode from source code by a compiler but not vice versa. However, targeting source code format is still important for integrating as an IDE plugin. We plan to add a source code analyzer in the future.

Moreover, we plan to further evaluate the usability of the generated effect information, by programmers as well as by analysis tools. Currently we output the effect information as annotations. The format of these annotations needs to be more readable and understandable to be used by programmers. We will also further investigate the applications of these effect annotations other than identification of pure functions. We will apply this approach to more software projects for further evaluation.

To conclude, in this paper we presented a study on the purity and side effects of the functions in Java, helping programmers to reuse the software libraries. We proposed a technique to automatically infer the purity and side effect informations from Java bytecode. We implemented and experimented the proposed analyzer on real world Java software libraries, and found that around 24%–44% of all the methods of a Java libraries are made of pure functions. We compared the accuracy of distribution of pure functions with an existing study. Also, we demonstrated how programmers will use our technique to understand the behavior of library APIs by a case study.

Further, to demonstrate the possible use-case of the gathered purity information, we discussed a case study on the topic of *purity-guided refactoring*. We conducted an experiment on a refactoring pattern called *Memoization* that caches the calculation result for pure functions. We tested the refactoring pattern on 3 open-source Java libraries and observed improvements in performance and preservation of semantics by running a profiler on the bundled test cases on these libraries.

We are still in an early stage of this research, continually evaluating new refactoring techniques that require purity information at hand. In the future, we will develop more refactoring approaches, for example, one converts a single-threaded sequential program to a thread-pool based multithreading program or an event-driven asynchronized program. We are planning to testify our methodology on larger programs or apply the refactorings during the development process.

5.3 General Conclusions of This Dissertation

This dissertation focuses on the **static analysis** to aide program comprehension. Static analysis can be divided further into two dimensions, namely the **structural** information and the **semantic** information. The final goal of this study is to aid programming comprehension with the static information in these two dimensions.

In this dissertation, I looked into these two aspects with two studies. The first one is a study on a classification model of source code clones, which helps understanding the source code from structural similarity. The latter one focuses on one important aspect of semantic information, which is the purity of the code fragments. I **developed tools** to help programmer in understanding their software.

In the future, our integrated development environments can be further improved to adopt the methodology that I described in this dissertation.

I firmly believe that our IDEs should understand our code more precious than us human beings, so that common development tasks can be automated by simple directions rather than repeatedly micro-operations from human.

Bibliography

[1] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[2] Anneliese Von Mayrhauser et al. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[3] Thomas A Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[4] Margaret-Anne Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 181–191. IEEE, 2005.

[5] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 602–611. IEEE, 2001.

[6] Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society, 2001.

[7] Y. Higo, S. Kusumoto, and K. Inoue. A survey of code clone detection and its related techniques. *IEICE Transactions on Information and Systems*, 91-D(6):1465–1481, June 2008. (in Japanese).

[8] 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.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] Z. Li, S. Myagmar, S. Lu, and Y.Zhou. Cp-miner : Finding copy-paste and related bugs in large-scale software code. *IEEE Transcations on Software Engineering*, 32(3):176–192, Mar. 2006.

[11] J.H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.

[12] I. Baxter, A. Yahin, M. Anna L. Moura, and L. Bier. Clone detection using abstract syntax trees. In *Proc. of the 14th International Conference on Software Maintenance*, pages 368–377, Mar. 1998.

[13] 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.

[14] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of the 8th International Working Conference on Source Code Analysis and Manipulation*, pages 57–66, Sep. 2008.

[15] Brian Goetz. Java theory and practice: I have to document that? `http://www.ibm.com/developerworks/java/library/j-jtp0821/index.html`, 2002.

[16] Chen Raymond. The importance of error code backwards compatibility. `http://blogs.msdn.com/b/oldnewthing/archive/2005/01/18/355177.aspx`, 2005.

[17] Fredrik Kjolstad, Danny Dig, Gabriel Acevedo, and Marc Snir. Transformation for class immutability. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM.

[18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[19] Kent Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[20] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[21] Jay Fields, Shane Harvie, Martin Fowler, and Kent Beck. *Refactoring: Ruby Edition*. Pearson Education, 2009.

[22] A. Marcus and J.I. Maletic. Identification of high-level concept clones in source code. In *16th Annual International Conference on Automated Software Engineering*, pages 107–114. IEEE, 2001.

[23] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[24] Robert Tairas and Jeff Gray. An information retrieval process to aid in the analysis of code clones. *Empirical Software Engineering*, 14:33–56, 2009. 10.1007/s10664-008-9089-1.

[25] Lucia, D. Lo, L. Jiang, A. Budi, et al. Active refinement of clone anomaly reports. In *34th International Conference on Software Engineering*, pages 397–407. IEEE, 2012.

[26] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, 49(9):985–998, 2007.

[27] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[28] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 309–318. IEEE, 2012.

[29] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. *Compare*, 31(58.5):81–1, 2013.

[30] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Sixth International Software Metrics Symposium*, pages 292–303. IEEE, 1999.

[31] K.W. Church and J.I. Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Journal of Computational and Graphical Statistics*, pages 153–174, 1993.

[32] Y. Higo. *Code clone analysis methods for efficient software maintenance*. PhD thesis, Osaka University, 2006.

[33] J. Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pages 16–. IBM Press, 1996.

[34] Z.M. Jiang and A.E. Hassan. A framework for studying clones in large software systems. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 203–212. IEEE, 2007.

[35] Xiaoyin Wang, Yingnong Dang, Lu Zhang, Dongmei Zhang, Erica Lan, and Hong Mei. Can i clone this piece of code here? In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 170–179. ACM, 2012.

[36] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Verification, Model Checking, and Abstract Interpretation*, pages 199–215. Springer, 2005.

[37] Alexandru Sălcianu. *Pointer analysis and its applications for Java programs*. PhD thesis, Citeseer, 2001.

[38] David J Pearce. Jpure: a modular purity system for java. In *Compiler Construction*, pages 104–123. Springer, 2011.

[39] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *Network and Distributed Systems Symposium, Internet Society*, volume 10, pages 357–374, 2010.

[40] Matthew Finifter, Adrian Mettler, Naveen Sastry, and David Wagner. Verifiable functional purity in java. In *Proc. of the 15th ACM conference on Computer and communications security*, pages 161–174. ACM, 2008.

[41] Haiying Xu, Christopher JF Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 75–82. ACM, 2007.

[42] Hugo Rito and João Cachopo. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, pages 89–98. ACM, 2010.

[43] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In Markku Oivo and Seija Komi-Sirvi, editors, *Product Focused Software Process Improvement*, volume 2559 of *Lecture Notes in Computer Science*, pages 185–197. Springer Berlin Heidelberg, 2002.

[44] Filip Van Rysselberghe and Serge Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 336–339. IEEE Computer Society, 2004.

[45] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An assessment of type-3 clones as detected by state-of-the-art tools. In *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*, pages 67–76. IEEE, 2009.

[46] Rebecca Tiarks, Rainer Koschke, and Raimar Falke. An extended assessment of type-3 clones as detected by state-of-the-art tools. *Software Quality Journal*, 19(2):295–331, 2011.

[47] K.S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[48] S.G. Kobourov. Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*, 2012.

[49] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[50] 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.

[51] Jonathan Drake and Greg Hamerly. Accelerated k-means with adaptive distance bounds. In *5th NIPS workshop on optimization for machine learning*, 2012.

[52] Michael Bostock. D3.js, Data-Driven Documents. `http://d3js.org/`, 2012. [Online; accessed 1-May-2012].

[53] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *EMNLP-CoNLL*, volume 7, pages 410–420, 2007.

[54] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[55] Gary T Leavens, Albert L Baker, and Clyde Ruby. Preliminary design of jml. Technical report, Technical Report 96-06p, Iowa State University, 2001.

[56] Vijay Saraswat. Java is not type-safe. `http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html`, 1997.

[57] Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for java. In *ECOOP 2009–Object-Oriented Programming*, pages 445–469. Springer, 2009.

[58] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Prentice Hall, 2008.

[59] Jonathan S Bell and Gail E Kaiser. Unit test virtualization with vmvm. 2013.

[60] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[61] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsysterns, Inc, 1996.

[62] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Revealing purity and side effects on functions for reusing java libraries. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, pages 314–329. Springer, 2014.