Code Clone Analysis Methods for Efficient Software Maintenance

Submitted to
Graduate School of Information Science and Technology
Osaka University

October 2006

Yoshiki Higo

Abstract

Maintaining software systems becomes more difficult as the size and complexity of them increase. One factor that makes software maintenance more difficult is the presence of code clones. A code clone is a code fragment that has identical or similar code fragments to it in the source code. Code clones are introduced by various reasons such as reusing code by 'copy and paste'. If we modify a code clone with many similar code fragments, it is necessary to consider whether or not we have to modify each of them. Especially, for large-scale software, such a process is very complicated and expensive. We tend to overlook some of code fragments which should be modified.

In this study, we propose maintenance support methods for the presence of code clones. This study covers comprehensive maintenance situations that we are suffered from code clones. To sophisticate and enrich our methods as practical ones, we are promoting academic-industrial collaboration. We deliver our tools to industrial people, and they apply the tools to their software development and maintenance contexts, and they send feedback to us. We improve our methods and tools based on the feedback, and deliver the improved tools to them repeatedly. This iterative improvement process enhanced our methods and tools as useful ones in practical software development and maintenance.

We propose methods of visualizing and characterizing code clones to support understanding them in a software system. The methods are very practical since they have a filtering functionality for eliminating uninteresting code clones. Automatic code clone detection by tools tends to detect many uninteresting code clones. Uninteresting code clones are the ones that we are not interested in from a viewpoint of software maintenance. For example, consecutive method invocations and case entries of switch-statements are typically uninteresting. The existence reason of these code clones is not copy and paste but the syntax of the programming language. The visualization method provides a bird's eye view of code clones distribution all over the system. Using this view, we can understand the distribution state of code clones at a glance. The characterization method calculates some metrics of the software entities (code clones and source files). This quantitative characterization enables us to select code clones based on their features (ex. code clones which are very big or occur in many places). We developed a tool, Gemini based on the methods, and applied it to open source and commercial software. The application results show that the filtering works very well and the visualization/characterization methods help the programmer to understand the state of code clones.

We propose a refactoring support method for code clones. *Refactoring* is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Generally, code clone is one of the most noteworthy *Bad Smells* (which are bad code patterns) in the source code. Our proposal consists of two steps. At the first step, refactoring-oriented code clones are detected from the source code. Here, a refactoring-oriented code clone is a code clone whose body is a whole structural unit (ex. class, method, loop). In the second step, each refactoring-oriented code clones is characterized by some metrics. We use two kinds of metrics, coupling and distance metrics. The coupling metric measures how a code clone is coupled with its surrounding code. If the degree of coupling is low, the code clone can be easily moved to other place. The distance metric measures how the code clones that are similar to each other are far in the class hierarchy. If they are in the same class, the code clones will be able to extract as a new method in the class. If they are in different classes having a common parent class, the code clones will be pulled up to the parent class. We develop a refactoring support tool, Aries, and applied it to open source and commercial software. The application results show that the refactoring techniques suggested by Aries are applicable and practical.

We propose a modification support method for code clones. The presence of code clones is a big factor of overlooking some places that should be modified. We propose a refactoring support method as described above, but some code clones cannot or should not be merged. In order to support maintenance against such code clones, the modification support method is helpful. The key idea is very simple. At first, programmer detects a code fragment that should be modified. Secondly, only code clones across fragments and software files are detected. Detecting only these code clone has two advantages. One is that the detection speed is much faster than detecting all code clones, and the other is that the programmer are not confused by the information of unconcerned code clones. We developed a modification support tool, Libra, and applied it to open source software. The result shows that Libra is a good searching tool as well as grep, which is an useful tool of UNIX.

In Chapter 1, the definition of code clone and several relative techniques are introduced. Chapter 2 describes visualization and characterization methods of code clones. Chapter 3 describes a refactoring support method for code clones, and a modification support method is introduced in Chapter 4. In Chapter 5, we summarize the suggestions of this paper and describe future works.

List of Publications

Major Publications

[1-1] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *On Software Maintenance Process Improvement Based on Code Clone Analysis*. Proceedings of the 4th International Conference on Product Focused Software Process Improvement (PROFES 2002), pp.185-197, Rovaniemi, Finland, December 2002.

[1-2] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, Yoshio Kataoka: *On Refactoring for Open Source Java Program*. Proceedings of the 9th IEEE International Software Metrics Symposium (METRICS 2003), pp.247-251, Sydney, Australia, September 2003.

[1-3] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *Refactoring Support Based on Code Clone Analysis*. Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES 2004), pp.220-233, Kyoto-Nara, Japan, April 2004.

[1-4] Yoshiki Higo, Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *On Software Maintenance Process Improvement Based on Code Clone Analysis*. IPSJ Journal, Vol.45, No.5, pp1357-1366, May 2004 (in Japanese).

[1-5] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *ARIES: Refactoring Support Environment Based on Code Clone Analysis*. Proceedings of the 8th IASTED International Conference on Software Engineering and Applications (SEA 2004), pp.222-229, Cambridge, USA, November 2004.

[1-6] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *Refactoring Support Environment Based on Code Clone Aanalysis*. IEICE Journal, Vol.J88-D-I, No.2, pp.186-195, February 2005 (in Japanese).

[1-7] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *ARIES: Refactoring Support Tool for Code Clone*. Proceedings of the 3rd Workshop of Software Quality (WoSQ 2005), pp.53-56, ST. Louis, USA, May 2005.

[1-8] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *Improvement and Implementation of Code Clone Visualization Method based on Academic-Industrial Collaboration*. IPSJ Journal, February 2007 (In Japanese)(to appear).

[1-9] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *Method and Implementation for Investigating Code Clones in a Software System*, Information and Software Technology (to appear).

Related Publications

[2-1] Yasushi Ueda, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *Gemini: Code Clone Analysis Tool*. Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE 2002), Nara, Japan, October 2002.

[2-2] Toru Sasaki, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *A code clone information supplement tool to support program change*. IEICE Journal, Vol.J87-D-I, No.9, pp.868-870, September 2004 (in Japanese).

[2-3] Norihiro Yoshida, Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue: *On Refactoring Support Based on Code Clone Dependency Relation*. Proceedings of the 11th IEEE International Software Metrics Symposium(METRICS 2005), Como, Italy, September 2005.

[2-4] Yoshiki Mitani, Nahomi Kikuchi, Tomoko Matsumura, Satoshi Iwamura, Yoshiki Higo, Katsuro Inoue, Mike Barker, Ken-ichi Matsumoto: *Effects of software industry structure on a research framework for empirical software engineering*. Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE 2006), pp.616-619, Shanghal, China, May 2006.

Acknowledgement

I am most indebted to my supervisor Professor Katsuro Inoue for his continuous support and supervision over the years. Without his help, experience and advice, this thesis would never have reached completion.

I would like to express my gratitude to Professor Toshimitsu Masuzawa, Professor Shinji Kusumoto and Associate Professor Makoto Matsushita for his valuable comments and helpful criticism on this thesis.

I would like to express my gratitude to Doctor Toshihiro Kamiya in National Institute of Advanced Industrial Science and Technology. His comments, criticism and advice have helped me shape the development of this thesis.

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

I would like to express my thanks to Professor Stan Jarzabek of National University of Singapore and Professor Michael W. Godfrey of the University of Waterloo for their insightful comments and valuable discussions on the paper which formed the basis of this thesis.

I wish to express my gratitude to the members of the CCFinder mailing list for their feedback on my software tools.

Thanks are also due to many friends in the Department of Computer Science, especially students in Inoue Laboratory.

Contents

1 Introduction **1**

1.1 Software Maintenance . . . 1

1.2 Code Clone . . . 2

1.2.1 Definition . . . 2

1.2.2 Reasons of the Code Clone Presence . . . 2

1.2.3 Code Clone Detection Tool: CCFinder . . . 4

1.2.4 Related Studies on Code Clone Detection . . . 7

1.3 Refactoring . . . 9

1.4 Research Overview . . . 14

1.4.1 Visualization and Characterization Methods for Comprehension . . . 15

1.4.2 Refactoring Support Method . . . 15

1.4.3 Modification Support Method . . . 16

2 Visualization and Characterization Methods for Comprehension **17**

2.1 Motivation . . . 17

2.2 Proposal Techniques . . . 19

2.2.1 Filtering out Uninteresting Code Clones . . . 19

2.2.2 Scatter Plot of Code Clone . . . 21

2.2.3 Clone Set Metrics . . . 23

2.2.4 File Metrics . . . 24

2.3 Code Clone Visualization Tool: Gemini . . . 25

2.4 Case Study on Open Source Software . . . 26

2.4.1 Target and Configurations . . . 26

2.4.2 Filtering with RNR . . . 27

2.4.3 Scatter Plot Analysis . . . 29

2.4.4 Metric Graph Analysis . . . 32

2.4.5 File List Analysis . . . 33

2.4.6 Evaluation . . . 35

2.5 Case Study on Commercial Software 38
2.5.1 Target and Configurations 38
2.5.2 History Analysis 39
2.5.3 Scatter Plot Analysis 40
2.5.4 Metric Graph Analysis 40
2.5.5 File List Analysis 42
2.6 Related Works and Discussion 42
2.7 Summary 44

3 Refactoring Support Method 45
3.1 Motivation 45
3.2 Proposal Techniques 46
3.2.1 Extraction of Refactoring-Oriented Code Clones 46
3.2.2 Provision of Appricable Refactoring Patterns 47
3.2.3 Example of Refactoring Process 48
3.3 Refactoring Support Tool: Aries 50
3.4 Evaluation on Applicability 54
3.4.1 Result of *Extract Method* 55
3.4.2 Result of *Pull Up Method* 56
3.5 Evaluation on Usefulness 59
3.5.1 Outline 59
3.5.2 Evaluation Criteria 61
3.5.3 Hypothesis 63
3.5.4 Results 63
3.5.5 Discussion 68
3.6 Addaitive Descriptions of Aries 69
3.6.1 Other Refactoring Conditions in Aries 69
3.6.2 Concentrating Target Classes to be Refactored in Aries . . 69
3.7 Related Works 71
3.8 Summary 72

4 Modification Support Method 75
4.1 Motivation 75
4.2 Approach 75
4.3 Debug Support Tool: Libra 76
4.4 Evaluation 77
4.4.1 Canna 78
4.4.2 Ant 79
4.4.3 Discussion 80
4.5 Related Works 80

4.6 Summary 81

5 Conclusions **83**

5.1 Summary of Major Results 83

5.2 Directions of Future Research 84

List of Figures

1.1 Clone Pair and Clone Set 4
1.2 Input Source Code 5
1.3 Detection Processe of CCFinder 6
1.4 An Example of *Extract Method* 11
1.5 An Example of *Pull Up Method* 11
1.6 An Example of *Extract SuperClass* 12
1.7 An Example of *Consolidate Duplicate Conditional Fragments* . . 12
1.8 An Example of *Form Template Method* 13
1.9 An Example of *Replace Conditional with Polymorphism* 14

2.1 Example of output from CCFinder 18
2.2 Example of language dependent code clone 20
2.3 Example of the *Scatter Plot* 22
2.4 Filtering clone sets using the Metric Graph 24
2.5 Snapshot of the File List 26
2.6 Examples of uninteresting code clones 28
2.7 Snapshots of Scatter Plot 30
2.8 Example of code clones in B (branching by using source of events) 31
2.9 Example of code clones in B (methods creating GUI widgets) . . . 32
2.10 One of the fragments making up the clone set whose POP is highest 33
2.11 Transition of Recall, Precision, and F-value 36
2.12 Transition of the duplicated ratio of company Y 40
2.13 A snapshot of Scatter Plot 41

3.1 Example of merging two code fragments 47
3.2 A Snapshot of Main Window 51
3.3 A Snapshot of Clone Set Viewer 52
3.4 Analysis precess using GUI Unit 53
3.5 Example of *Extract Method* in (EG1) 55
3.6 Example of *Extract Method* in (EG2) 55

3.7 Example of *Extract Method* in (EG3) 56
3.8 Example of *Extract Method* in (EG4) 56
3.9 Example of *Pull Up Method* in (PG2) 57
3.10 Example of *Pull Up Method* in (PG3) 57
3.11 Example of *Pull Up Method* in (PG4) 58
3.12 A Snapshot of Class Viewer 70

4.1 A SnapShot of Input Fragment 76
4.2 A SnapShot of Detection Result 77
4.3 An example of Canna's modifications 78
4.4 An example of Ant's modifications 79

List of Tables

2.1 Breakdown of uninteresting code clones 27
2.2 Duplicated Ratios of Files 34
2.3 Size of target software systems and results of executions 37
2.4 Amount of code clones in sub-systems 39

3.1 Number of Detected Clone Sets 59
3.2 Refactoring Evaluations of *Extract Super Class* 64
3.3 Refactoring Evaluations of *Move Method* 66
3.4 Refactoring Evaluations of *Extract Method* 67

4.1 Attributes of target software 77
4.2 Comparison between Libra and grep 80

Chapter 1

Introduction

1.1 Software Maintenance

Software Maintenance is modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [25]. Currently, software maintenance consists of 4 categories, which are defined as follows [26].

Corrective maintenance
The reactive modification for a software product performed after delivery to correct discovered problems.

Adaptive maintenance
The modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.

Perfective maintenance
The modification of a software product after delivery to improve performance or maintainability.

Preventive maintenance
The modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

As the size and complexity of software increase, maintenance tasks become more difficult and burdensome. Arthur states that only one-fourth or one-third of all life-cycle costs are attributed to software development and that some 67% of life-cycle costs are expended in the operation-maintenance phase of the life cycle [3, 55].

Jones states that the work of enhancing an existing system in much more costly than new development work if the base system is not well structured [30]. He provides empirical data to substantiate his claims. Perhaps one of the most salient comments he makes is that organizations lump enhancements and fixing post-delivery bugs together. He states that this is unfortunate because it distorts both development and maintenance, and leads to confusion and mistakes in estimating. He submits that this practice tends to perpetuate the notion that maintenance is fixing bugs and mistakes.

Schneidewind also gives insight into why maintenance is hard [47]. He states that,

1. It is difficult to trace the product or the process that created the product.
2. Changes are not adequately documented.
3. There is a lack of change stability.
4. There is a ripple effect when making changes.

This lack of attention to maintainability requirements during design results in a loss of traceability. Thus, in tracing errors or adding enhancements, it is often impossible to trace back to design specifications and user requirements. As a result, maintenance is hard.

1.2 Code Clone

1.2.1 Definition

A **code clone**, in general, means a code fragment that has identical or similar code fragments to it in source code. However, there is no single and generic definition for code clone. So far, several methods of code clone detection have been proposed, and each of them has its own definition about code clone. Some of these methods are described in Section 1.2.4. In the remaining parts of this paper, we apply the code clone detection feature of CCFinder as the definition of code clone.

1.2.2 Reasons of the Code Clone Presence

Code clones are introduced in source code because of various reasons as follows.

Copy and Paste

So far, many software design methods have been proposed, and they enable us to develop software product with well-designed and high-reusability. Unfortunately there are enormous amount of ad-hoc reuse with copy and paste because reuse with copy and paste is more reliable than writing code from scratch.

Stereotyped Process
There are many stereotyped processes in implementing a software product, and they depend on either programming language or domain of the software product. For example, calculation of salary tax, file open/close and database access are typical stereotyped processes.

Absence of Abstraction Functionality
If abstract data types or local variables cannot be used, we have to write the same logic code many times in different places of the software product. The same logic code tends to be code clones.

Performance Enhancement
In developing a real time software product using a compiler without the inline expansion functionality, we sometimes manually expand code in loops for performance enhancement. Expanded code become code clones.

Tool Generation
Code generated by tools (Code Generator) tends to include many code clones because the tools often use the same template to generate same or similar logic code. Only identifier names of these code clones are different from each other.

Coincidence
Coincidentally, different developers write the same logic code, but the probability of this case is very low.

The presence of code clone is one of the factors that makes software maintenance more difficult. If we modify a code fragment and it has many code clones, it is necessary to consider whether or not we have to modify each of the code clones. Especially, for large-scale software, such processes are very complicated and costly. There are two solutions for this problem.

- Keep making documents of up-to-date code clone information not to overlook some of them in modification process.

- Detect code clones from source code automatically.

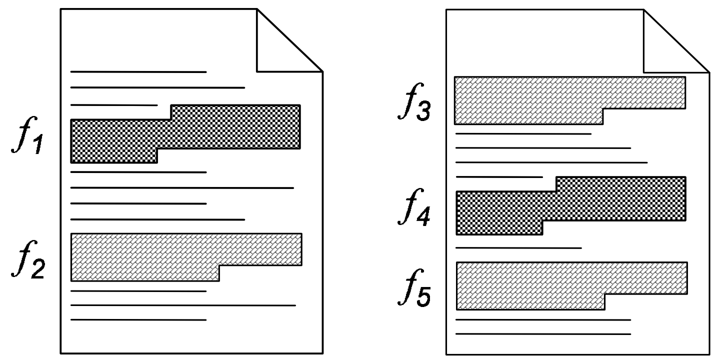


Figure 1.1: Clone Pair and Clone Set

The first solution requires much cost because keep up-to-date code clone information in document is performed manually. In development or maintenance of the large-scale software product, this is unrealistic. On the other hand, the second solution doesn't require much cost because the tool detects code clones automatically. As concrete approaches of it, several methods of code clone detection have been proposed. Some of these methods are described in Section 1.2.4.

1.2.3 Code Clone Detection Tool: CCFinder

In CCFinder [33], a clone relation is defined as an equivalence relation (reflexive, transitive, and symmetric relation) on code fragments. A **code fragment** is a part of a source file, and it can be represented using ID, $Line_{start}$, $Column_{start}$, $Line_{end}$, and $Column_{end}$. For a code fragment f, $ID(f)$ is the numeral ID of the source file where f resided in. CCFinder assigns an unique ID to each of all target source files. $Line_{start}(f)$ ($Line_{end}(f)$) is the start (end) line number of f, and $Column_{start}(f)$ ($Column_{end}(f)$) is the start (end) column number of f respectively. In this definition, it is possible that some code fragments are partially overlapped to each other. Clone relation exist between two code fragments if (and only if) the token sequences of them are identical[1]. For a given clone relation, a pair of code fragments is called a **clone pair** if the clone relation holds between them. An equivalence set of clone relation is called a **clone set**. That is, a clone set is a maximal set of code fragments where a clone relation exits between any pair of them.

Figure 1.1 illustrates an example of clone relation. As shown in this figure,

[1]The sequences are the transformed ones described as below

```
static void foo() throws RESyntaxException {
   String a[] =
          new String [] { "123,400", "abc", "orange 100" };
   org.apache.regexp.RE pat =
          new org.apache.regexp.RE("[0-9,]+");
   int sum = 0;
   for (int i = 0; i < a.length; ++i)
     if (pat.match(a[i]))
       sum += Sample.parseNumber(pat.getParen(0));
   System.out.println("sum = " + sum);
}
static void goo(String [] a) throws RESyntaxException {
   RE exp = new RE("[0-9,]+");
   int sum = 0;
   for (int i = 0; i < a.length; ++i)
     if (exp.match(a[i]))
       sum += parseNumber(exp.getParen(0));
   System.out.println("sum = " + sum);
}
```

Figure 1.2: Input Source Code

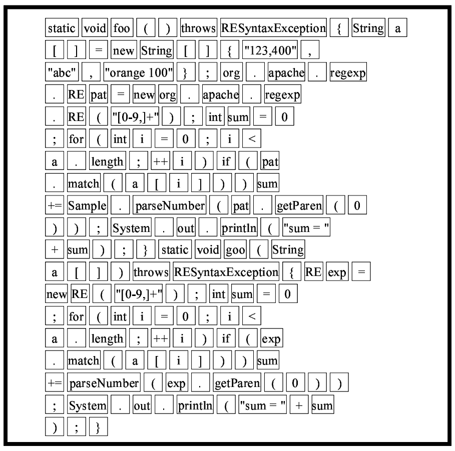
there are five code fragments that have clone relations with other code fragments. Fragment f_1 has a clone relation with code fragment f_4, and fragments f_2, f_3, and f_5 have clone relations with each other. In this case, 4 clone pairs, (f_1, f_4), (f_2, f_3), (f_2, f_5), (f_3, f_5), and 2 clone sets, $\{f_1, f_4\}$, $\{f_2, f_3, f_5\}$ exist.

CCFinder detects code clones from source files, and tells the locations of clone pairs in source files. The minimum code clone length to be detected is set by users in advance. The clone detection of CCFinder is a process in which the input is source files and the output is clone pairs. The process consists of the following steps. Here we use the source code of Figure 1.2 to explain each step, and Figure 1.3 shows how the input source code is treated in each step.

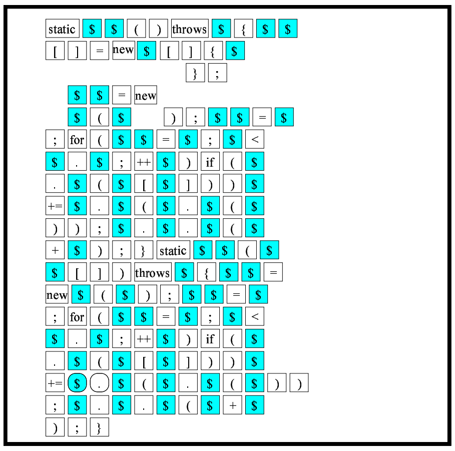
1. **Lexical analysis**: Each line of source files is divided into tokens corresponding to lexical rule of the programming language. The tokens of all source files are concatenated into a single token sequence (Figure 1.3(a)).

2. **Transformation**: The token sequence is transformed, for example, tokens are added, removed, or changed based on the transformation rules that aim at regularization of identifiers and identification of structures (Figure 1.3(b)).

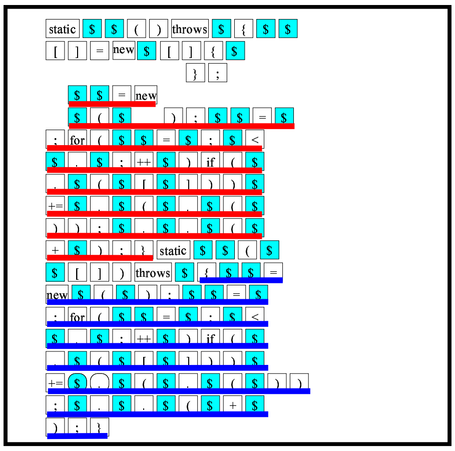
3. **Match Detection**: From all the sub-strings on the transformed token se-



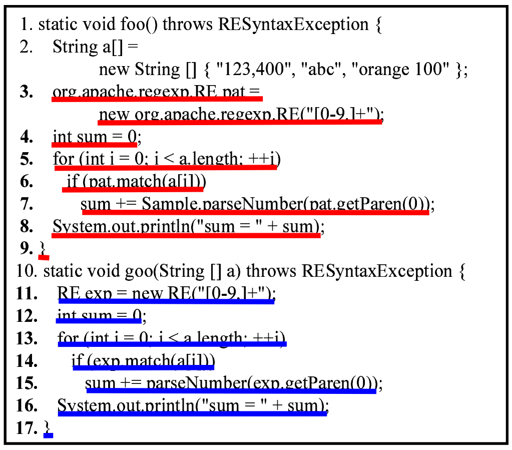
(a) After Devided into Tokens



(b) After Transformation



(c) After Detection on Token Sequence



```
static void foo() throws RESyntaxException {
    String a[] =
            new String [] { "123,400", "abc", "orange 100" };
    org.apache.regexp.RE pat =
            new org.apache.regexp.RE("[0-9,]+");
    int sum = 0;
    for (int i = 0; i < a.length; ++i)
        if (pat.match(a[i]))
            sum += Sample.parseNumber(pat.getParen(0));
    System.out.println("sum = " + sum);
}
static void goo(String [] a) throws RESyntaxException {
    RE exp = new RE("[0-9,]+");
    int sum = 0;
    for (int i = 0; i < a.length; ++i)
        if (exp.match(a[i]))
            sum += parseNumber(exp.getParen(0));
    System.out.println("sum = " + sum);
}
```

(d) After Mapped on Original Source Code

Figure 1.3: Detection Processe of CCFinder

quence, equivalent pairs are detected as clone pairs (Figure 1.3(c)).

4. **Formatting**: Each location of detected clone pairs is converted into line and column numbers on the original source files (Figure 1.3(d)).

Some research studies have been done with CCFinder. Bruntink et al. [13] evaluated the suitability of clone detection as a technique for the identification of crosscutting concerns. At first, they manually identified four specific concerns (memory error handling, parameter checking, error handling and tracing) in an industrial C application, and analyzed to what extent clone detection is capable of finding these concerns. They used two different tools, which implement different clone detection techniques. One is 'ccdiml', which is an implementation of a variation of Baxter's approach [11], and thus falls in the category of AST-based clone detections. The other is CCFinder [33], a clone detection tool based on tokenized representations of source code. The results show that code belonging to concerns like parameter checking and memory error handling is identified very well by both clone detection tools, while error handling and tracing concerns are more problematic. Kim et al. [36, 37] suggested a method supporting code clone unification based on the history information of the development. They used CCFinder as a clone detection engine. If code fragments of the same clone set are modified simultaneously and repeatedly, they said that removing them probably reduce the maintenance cost of the future.

1.2.4 Related Studies on Code Clone Detection

Several methods and tools have been developed for code clone detection as follows, and some comparative evaluations of them are conducted [14, 46].

Dup [4, 5, 6]

A clone detection tool Dup uses a sequence of lines as a representation of source code and detects line-by-line clones. It performs the following subprocesses:

1. replacement of identifiers of functions, variables, and types into a special identifier (parameter identifier),
2. extraction of matches by a suffix-tree algorithm [22] of $O(n)$ time complexity (n is the number of lines in the input),
3. computation of correspondence (pairing) between parameter identifiers.

The line-by-line method has a weakness in the line-structure modification. In free-format languages such as C, C++, and Java, line breaks in source code have no semantic meaning, they are often placed and relocated based on programmer's preference. The tool cannot detect code clones including such lines.

Duploc [18]

A language dependent clone detection tool Duploc reads source files, makes a sequence of lines, removes white-spaces and comments in lines, and detects match by a string-based Dynamic Pattern Matching (DPM). The output is the line numbers of clone pairs, possibly with gap (deleted) lines in them. The computation complexity is $O(n^2)$ for the input size n and it is practically too expensive. The tool uses an optimization technique by a hash function for string, which reduces the computation complexity.

CloneDR [11]

Baxter et al. proposes a technique to extract clone pairs of statements, declarations, or sequences of them from C source files. The tool parses source code to build an abstract syntax tree (AST) and compares its subtrees by characterization metrics (hash functions). The parser needs a "full-fledged" syntax analysis for C to build AST. Baxter's tool expands C macros (define, include, etc) to compare code portions written with macros. Its computation complexity is $O(n)$, where n is the number of the subtree of the source files. The hash function enables one to do parameterized matching, to detect gapped clones, and to identify clones of code portions in which some statements are reordered. In AST approaches, it is able to transform the source tree to a regular form as we do in the transformation rules. However, the AST based transformation is generally expensive since it requires full syntax analysis and transformation.

Covet [41]

A clone-detecting method proposed by Mayrand et al. uses a representation named Intermediate Representation Language (IRL) to characterize each function in the source code. A code clone is defined only as a pair of whole function bodies that have similar metric values.

SMC [8, 7, 9]

A clone detection tool SMC (Similar method Classifier) uses a hybrid approach of characterization metrics and DPM (dynamic pattern matching). The paper only discusses detection of whole methods, although the approach would be applied to detect partial code portions also. The detection process consists of the following subprocesses:

1. extraction of method bodies from source code of Java,
2. computing characteristic metrics values for each method,
3. identify pairs of methods with similar metric values,
4. compare each pair of token sequences of the similar methods by DPM to identify clone methods, and

5. classifying clone methods into 18 categories.

The computing complexity is $O(n)$, where n is amount of methods in source files. The tool might be easily ported to the languages to which the metrics are applicable, although its parser and hash function have to be constructed and tuned for the input language.

Komondoor's method [38]

Komondoor et al. has proposed a method using program slicing. In this method, a program dependence graph is constructed by analyzing target source codes. Identical or similar parts are detected as code clone. This detection is greatly precise because of considering control and data flow of program. Moreover, it can detect reordered and intertwined clones. But, time complexity of constructing program dependence graph is $O(n^2)$, where n is the number of statement and expression included in target source codes. It is difficult to apply this method to large-scale software.

1.3 Refactoring

Refactoring [20, 43] is the process of changing a software product in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chance of introducing bugs. In essence when the developer refactors he/she is improving the design of the code after it has been written. Refactoring is a tool to be used for several purposes.

Improve the Design of Software
: Without refactoring, the design of the software will decay. As people change code (changes to realize short-term goals or changes made without a full comprehension of the design of code) the code loses its original structure. It becomes harder to see the design by reading the code. Refactoring is rather like tidying up the code. Work is done to remove bits that aren't really in the right place. Loss of the structure of code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps code retain its shape.

Make Software Easier to Understand
: The programmer doesn't remember things about the code that he/she wrote in past times, and he/she often has to modify the code written by other programmers. This means he/she has to understand the code before modifying

it. It maybe takes the programmer a week to make a change that would have taken only an hour if he/she had understood the code. The trouble is that when the programmer is trying to get the program to work, he/she is not thinking about future changes. Refactoring helps the programmer to make his/her code more readable because when he/she is refactoring he/she is thinking about future changes.

Help to Find bugs

Help in understanding the code also helps the programmer spot bugs. By clarifying the structure of the program using refactoring, the programmer clarifies certain assumptions he/she has made, to the point at which even he/she cannot avoid spotting the bugs. Refactoring also helps the programmer be much more effective at writing robust code.

Help to Develop and Maintain Software Rapidly and Efficiently

All the things described above come down to help to develop and maintain the program rapidly and efficiently. A good design is essential for rapid software development. Without a good design, the programmer can progress quickly for a while, but soon the poor design starts to slow him/her down. The programmer spends time finding and fixing bugs instead of adding new function. Changes need more time as you try to understand the program. A good design is essential to maintaining speed in software development. Refactoring helps the programmer develop software more rapidly, because it prevents the design from decaying.

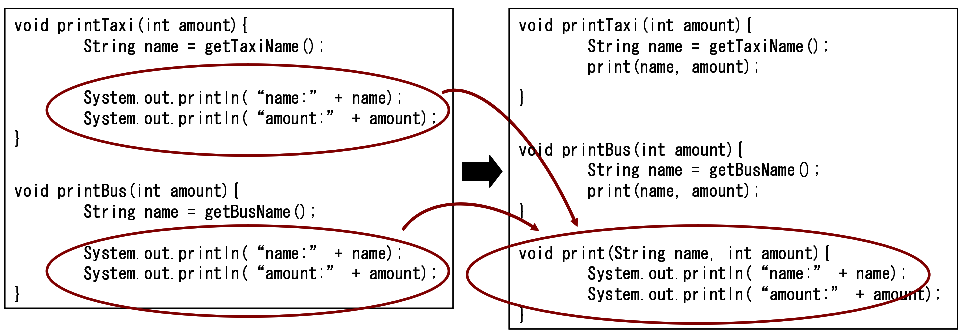
In Fowler's book [20], 22 bad smells (, which are the code patterns that should be refactored) are introduced, and 72 refactoring patterns (, which are modification techniques to remove bad smells) are also introduce to remove bad smells. The book says that number one in the stink parade is duplicated code (code clone). The following patterns can be used to remove code clones.

Extract Method

Originally, *Extract Method* is applied to a too long method or a part of complicated function in order to improve the readability, understandability, and maintainability. It also can be applied to code clones to merge them. Figure 1.4 illustrates a simple example of *Extract Method*. Before the refactoring, the two methods have duplicated instructions. But, after the refactoring, the duplicated parts are extracted as a new method and duplicated code was replaced with a caller statement of the new method.

Pull Up Method

Pull Up Method is that a method in a class is moved to its parent class.



```
void printTaxi(int amount){
        String name = getTaxiName();

        System.out.println( "name:"  + name);
        System.out.println( "amount:"  + amount);
}

void printBus(int amount){
        String name = getBusName();

        System.out.println( "name:"  + name);
        System.out.println( "amount:"  + amount);
}
```

```
void printTaxi(int amount){
        String name = getTaxiName();
        print(name, amount);
}

void printBus(int amount){
        String name = getBusName();
        print(name, amount);
}
void print(String name, int amount){
        System.out.println( "name:"  + name);
        System.out.println( "amount:"  + amount);
}
```

Figure 1.4: An Example of *Extract Method*

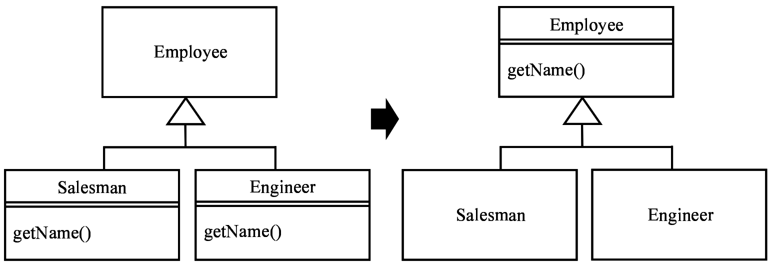


Figure 1.5: An Example of *Pull Up Method*

If several child classes have similar methods, moving them to the common parent class is an effective refactoring. The easiest case of using *Pull Up Method* occurs when the methods have the same body, which implies there's been a copy and paste. Figure 1.5 demonstrates a simple example of *Pull Up Method*. In the figure, before the refactoring, two classes (*Salesman* and *Engineer*) has identical methods (*getName()*). By pull up them to the common parent class (*Employee*), duplicated code in the classes was eliminated.

Move Method

Move Method is similar to *Pull Up Method*. The different is only that duplicated methods are moved to not the parent class but other class which is not extended by the current class.

Extract SuperClass

If two or more classes have many similar functionalities and they don't have a common parent class, *Extract SuperClass* can remove the duplicated code.

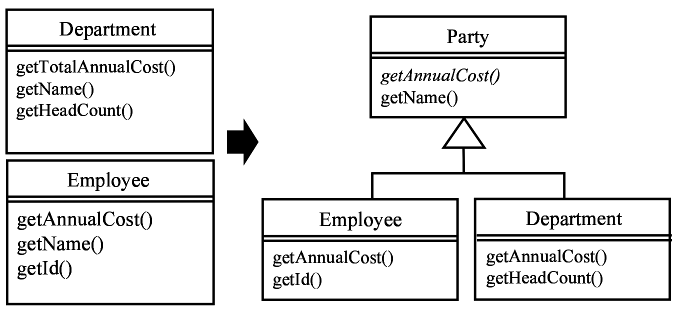
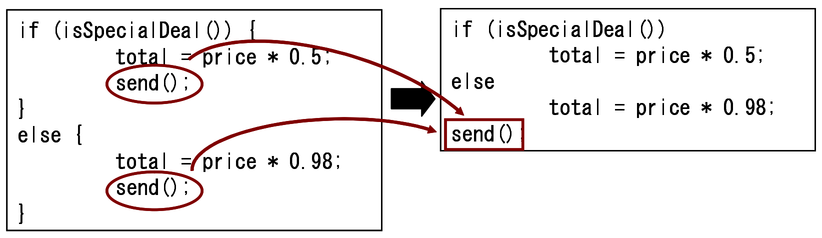


Figure 1.6: An Example of *Extract SuperClass*



```
if (isSpecialDeal()) {
        total = price * 0.5;
        send();
}
else {
        total = price * 0.98;
        send();
}
```

```
if (isSpecialDeal())
        total = price * 0.5;
else
        total = price * 0.98;
send();
```

Figure 1.7: An Example of *Consolidate Duplicate Conditional Fragments*

At first, the programmer creates a new class that is a common parent of the classes. Next, he/she apples *Pull Up Method* to each duplicated method. Figure 1.6 illustrates an example of *Extract SuperClass*. Before the refactoring, two methods (*getTotalAnnualCost()* and *getName()*) in class *Department* and two methods (*getAnnualCost()* and *getName()*) are duplicated. Method *getName()*s are exactly identical to each other, and method *getTotalAnnualCost()* is similar to (not identical to) method *getAnnualCost()*. The different is achieved by overriding the parent's *getAnnualCost()*.

Consolidate Duplicate Conditional Fragments

If the same instructions are in all branches of a conditional expression, moving them outside of the expression is an method of deleting duplicated code. Figure 1.7 illustrates an example of *Consolidate Duplicate Conditional Fragments*. Before the refactoring, in all branches of the if-statement expression, there are the same method invocations. After refactoring, the method invocations were moved to the outside of the conditional expression and the duplicated code was removed.

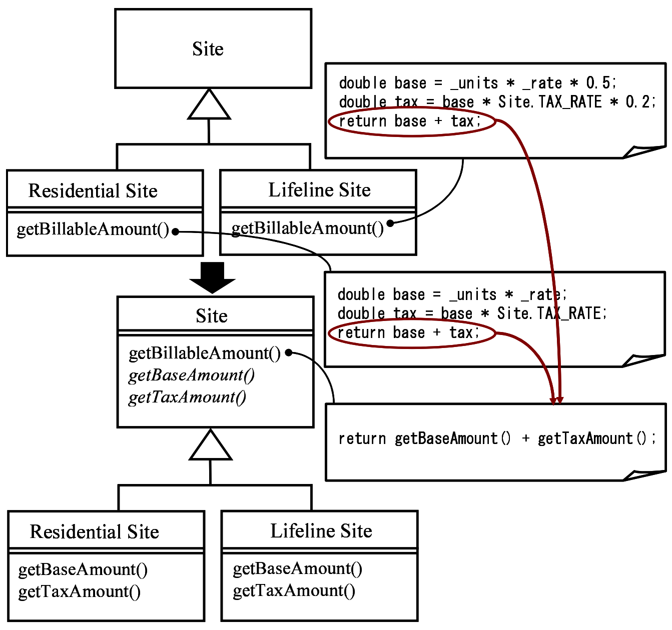


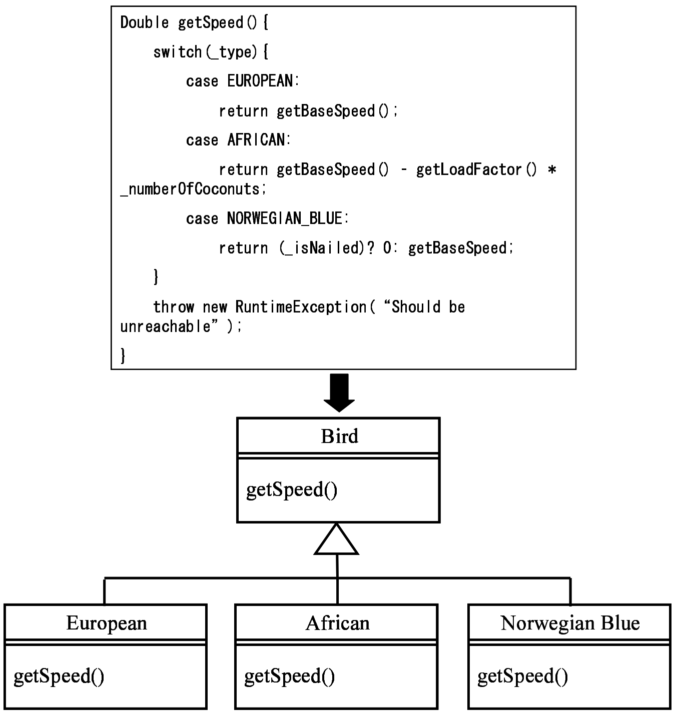
Figure 1.8: An Example of *Form Template Method*

Form Template Method

If there are two or more methods in subclasses that perform similar steps in the same order, the common logic can be merged in the superclass. In this case, the programmer can move the sequence to the superclass and allow polymorphism to play its role in ensuring the different steps do their things differently. This kind of method is called a *template method* [21]. Figure 1.8 illustrates an example of *Form Template Method*. Before the refactoring, the subclasses (*ResidentialSite* and *LifelineSite*) have methods whose logics are similar to each other. By the refactoring, the duplicated logic was pulled up to the superclass and each subclass overrides superclass's method for achiving a little bit of different steps.

Replace Conditional with Polymorphism

If there is a conditional expression that chooses different behavior depending on the type of an object, moving each leg of the conditional expression to an overriding method in a subclass is effective refactoring. The original method becomes abstract. Usually, each case entry of a switch-statement tends to be similar to each other. Using *Replace Conditional with Polymorphism* can remove such code clones. Figure 1.9 illustrates an example of *Replace*



```
Double getSpeed() {
    switch(_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() - getLoadFactor() *
_numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed)? 0: getBaseSpeed;
    }
    throw new RuntimeException( "Should be
unreachable" );
}
```

Figure 1.9: An Example of *Replace Conditional with Polymorphism*

Conditional with Polymorphism. Before the refactoring, there is a switch-statement that chooses how to calculate the speed, and each case entry is similar to each other. After the refactoring, there are three subclasses instead of the switch-statement, and polymorphism has a role of choosing how to calculate the speed. The biggest gain of *Replace Conditional with Polymorphism* occurs when this same set of conditions appears in many places in the program. If the programmer wants to add a new type, he/she has to find and update all the conditionals. But with subclasses, he/she just create a new subclass and provide the appropriate methods.

1.4 Research Overview

In this research, we have discussed about support methods against the presence of code clones. So far, Many support methods have been proposed by other researchers [9, 11, 13, 18, 29, 34, 38, 41, 49]. The feathers of our methods that I want to emphasize are the scalability and the comprehensive support range. Before proposing these methods, we have actively promoted academic-industrial collaboration to get demands of industrial world. Based on the demands, we have improved our methods and tools day by day. As a result, our methods became very practical ones and more than 100 companies are using our tools currently.

1.4.1 Visualization and Characterization Methods for Comprehension

Chapter 2 describes visualization and characterization methods of code clones. The most serious problem of existing visualization methods [18, 45] is that they have no functionality to filter out uninteresting code clones. Automatic code clone detection by tools tends to detect many uninteresting code clones. Filtering of such code clone is essential for practical visualization. Our visualization method includes a filtering method based on code pattern, and it can efficiently eliminate uninteresting code clone.

In characterization, we propose a novel selection way to select code clones having the feature that users are interested in. Using the function, users can select code clones that are very long or appear in many places in the program.

We implemented a tool, Gemini, and applied it to both industrial and open source software. The result shows that the tool is very practical and useful to analysis code clones for comprehension.

1.4.2 Refactoring Support Method

Chapter 3 describes a refactoring support method for code clones. The method consists of two steps. In first step, refactoring-oriented code clones are extracted from the result of code clone detection. We use CCFinder as a code clone detection engine to get code clones quickly. But, as described in Section 1.2.3, CCFinder detects code clones as token sequences, which means code clones detected by CCFinder are not necessarily suitable for refactoring.

In second step, the method suggests applicable refactoring patterns (described in Section 1.3) for each refactoring-oriented code clone. Some metrics are used to characterize code clones.

We implemented a tool, Aries based on the method, and applied it to both industrial and open source software. The results of both applications show that refactorings suggested by Aries is applicable and practical.

1.4.3 Modification Support Method

Chapter 4 describes a modification support method for code clones. We propose code clone removal method for code clone in Chapter 3. But, some code clones cannot or shouldn't be refactored, and a modification method is essential because the programmer sometimes overlooks some of code clones that should be modified. We propose a modification method listing code fragments which should be modified. The key idea is very simple, after the programmer identifies one of code fragment that should be modified, code clones across the fragment and target files are detected.

We implemented a tool, Libra based on the method, and applied it to open source software. We performed imagine debugs using the past bugs information. We compare the recall and precision of Libra and grep, which is a useful tool of UNIX.

Chapter 2

Visualization and Characterization Methods for Comprehension

2.1 Motivation

There are many researches on automatic detection of code clones [4, 5, 6, 8, 7, 9, 11, 18, 31, 28, 38, 39, 40, 41]. We also have developed a code clone detection tool, CCFinder [33], which has been designed as a tool to detect code clones effectively in large-scale software used in the industrial world, and it is still being improved day by day. We have been delivering CCFinder to more than 100 software organizations and the usefulness of it in actual software maintenance is evaluated.

We received much feedback from these organizations, stating that CCFinder extracts too many code clones, hence it is necessary to provide some guidelines to select important code clones from raw output of the tool. Moreover, it is quite difficult for users to investigate code clones only with the output of CCFinder. Figure 2.1 shows an example of the output. In the figure, each line between #begin{clone} and #end{clone} means a clone pair. For example, the fragment starting from line 73 to line 86 in source file (0.2) and the fragment starting from line 124 to line 137 in source file (1.2) contribute to a clone pair.

Depends on size and nature of target program, the amount of code clones detected by CCFinder sometimes can become quite huge. For example, in JDK 1.5, there are approximately 2,500,000 clone pairs (12,000 clone sets) whose fragment length is more than 30 tokens. It is true that CCFinder enables users to quickly obtain code clones from such large-scale software, but it is not realistic to check all of the detected code clones by hand in order to figure out useful information. A

```
#version: ccfinder 7.2.4
#format: pairwise
#langspec: C
#option: -b 30
#option: -e char
#option: -k 30
#option: -r abdfikmnpstuv
#option: -c wfg
#begin{file description}
0.0     249     697     C:¥experiment¥linux-2.6.5¥fs¥autofs¥dirhash.c
0.1     52      93      C:¥experiment¥linux-2.6.5¥fs¥autofs¥init.c
0.2     249     622     C:¥experiment¥linux-2.6.5¥fs¥autofs¥inode.c
0.3     557     1591    C:¥experiment¥linux-2.6.5¥fs¥autofs¥root.c
0.4     30      58      C:¥experiment¥linux-2.6.5¥fs¥autofs¥symlink.c
0.5     205     488     C:¥experiment¥linux-2.6.5¥fs¥autofs¥waitq.c
1.0     272     656     C:¥experiment¥linux-2.6.5¥fs¥autofs4¥expire.c
1.1     42      63      C:¥experiment¥linux-2.6.5¥fs¥autofs4¥init.c
1.2     315     774     C:¥experiment¥linux-2.6.5¥fs¥autofs4¥inode
                :
                :
#end{file description}
#begin{clone}
0.2     73,2,116    86,18,165   47      1.2     124,2,241    137,18,290   47
0.2     152,2,385   165,6,420   34      1.2     221,2,547    237,6,582    34
0.2     171,2,432   201,2,521   81      1.2     248,2,598    282,2,687    81
0.2     80,3,140    88,5,173    32      2.13    193,3,397    201,6,430    32
0.2     75,2,118    88,5,173    52      4.6     702,2,1555   715,6,1610   52
0.2     75,2,118    85,15,161   41      13.10   317,2,659    327,19,702   41
0.2     75,2,118    85,15,161   41      14.11   609,2,1273   619,19,1316  41
0.2     75,2,118    83,47,153   33      15.4    353,2,780    361,44,815   33
                    :
                    :
#end{clone}
```

Figure 2.1: Example of output from CCFinder

simple browsing tool, which displays the source code of clone pairs one by one, is not helpful for large-scale software.

In this chapter, we propose visualization and characterization methods of code clones. By using these methods, users can see how code clones are distributed in the system at a glance or obtain the code clones that have the features that they are interested in. Also, from our previous experience, we figured out that there are many uninteresting code clones in the results of code clone detection. Uninteresting code clone is a code clone that deemed to be not significant in the context of software maintenance. We propose a filtering method to skip this kind of code clones and reduce investigation effort. The filtering method makes the visualization and characterization methods more effective.

2.2 Proposal Techniques

From many our previous experience with CCFinder, we figured out that CCFinder detects many uninteresting code clones. As such, a filtering method is proposed to counteract this problem. In order to realize an effective code clone analysis for large-scale software products, we need some kind of bird's eye view of code clones to grasp the amount of code clones in the system, especially in the initial phase of analysis. Also, we provide an appropriate characterization of entities (code clones, target source files, and functionalities), which enables users to select arbitrary entities based on their features.

In this section, we explain our methods with an example to order to give a complete picture of them. We assume that we detect code clones from 4 source files (F_1, F_2, F_3, F_4) located in 2 directories (D_1, D_2). Each source file consists of the following five tokens (the meaning of superscripting $\star$ will be described in Section 2.2.1).

$F_1\ : a\ b\ c\ a\ b,$

$F_2\ : c\ c^*\ c^*\ a\ b,$

$F_3\ : d\ e\ f\ a\ b,$

$F_4\ : c\ c^*\ d\ e\ f$

Also, we use a label $C(F_i, j, k)$ to represent a fragment. A fragment $C(F_i, j, k)$ starts at the j-th token and ends at the k-th token in source file F_i (j must be less than k).

Here, we assume that at least 2 tokens are needed to be identified as a code clone. With this assumption, the following 3 clone sets are detected from the source files $F_1 \sim F_4$.

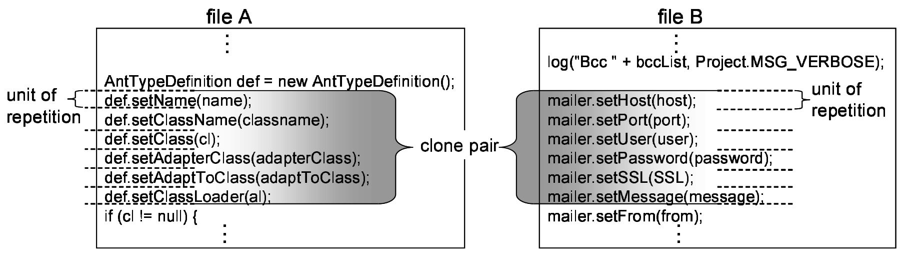
$S_1\ : \{C(F_1, 1, 2), C(F_1, 4, 5), C(F_2, 4, 5), C(F_3, 4, 5)\}$

$S_2\ : \{C(F_2, 1, 2), C(F_2, 2, 3), C(F_4, 1, 2)\}$

$S_3\ : \{C(F_3, 1, 3), C(F_4, 3, 5)\}$

2.2.1 Filtering out Uninteresting Code Clones

Many uninteresting code clones are included in the CCFinder detection results. In this paper, an "uninteresting code clone" is a code clone whose existence information is useless when using code clone information in software development or maintenance.



```
⋮
AntTypeDefinition def = new AntTypeDefinition();
def.setName(name);
def.setClassName(classname);
def.setClass(cl);
def.setAdapterClass(adapterClass);
def.setAdaptToClass(adaptToClass);
def.setClassLoader(al);
if (cl != null) {
⋮
```

```
⋮
log("Bcc " + bccList, Project.MSG_VERBOSE);
mailer.setHost(host);
mailer.setPort(port);
mailer.setUser(user);
mailer.setPassword(password);
mailer.setSSL(SSL);
mailer.setMessage(message);
mailer.setFrom(from);
⋮
```

Figure 2.2: Example of language dependent code clone

We have identified two types of uninteresting code clones. The first is a language-dependent code clone. When a specific programming language is used, a programmer has to repeatedly write some code fragments that cannot be merged into code fragments due to language limitations. A language-dependent code clone consists of such code fragments. The second is an application-dependent code cone. Some application frameworks sometimes require idiomatic code fragments to the application code to interface with the frameworks. For example, a code fragment of database connection is a typical application-dependent code clone. Application-dependent code clone is a code clone that consists of such code fragments. Language-dependent code clones are detected from all software systems written in the same programming language, but not application-dependent code clones, which differ greatly among systems. Therefore, it is much more difficult to filter out application-dependent code clones than language-dependent ones. The presence of uninteresting code clones doesn't negatively influence software development or maintenance. They are stereotyped code and are very stable.

As the first step to filter out uninteresting code clones, we propose a method to filter out language-dependent code clones. For example, consecutive variable declarations, consecutive method invocations, and case entries of switch statements, which become code clones due to the structure of the programming language, are typical language-dependent code clones.

Figure 2.2 is an example of a language-dependent code clone. The highlighted parts are a clone pair between files A and B. Each code fragment of the clone pair is an implementation of consecutive method invocations. The variable and method names in the code fragments are different. As described in Section 1.2.3, CCFinder transforms user-defined names into the same special token. This transform detects the same logic code with different names; for example, after copy and paste some variable names are changed. Unfortunately, CCFinder also detects many language-dependent code clones such as Figure 2.2.

We focused on a repetition structure within a code clone because a language-dependent code clone has repetitive implementations of the same logics. We propose to filter out such code clones with a metric called **RNR(S)** that represents the ratio of *non-repeated code sequence* in clone set S.

Here, we assume that clone set S includes n fragments, f_1,f_2, $\cdots$, f_n. $LOS_{whole}(f_i)$ represents the Length Of *whole* Sequence of fragment f_i, and $LOS_{repeated}(f_i)$ represents the Length Of *repeated* Sequence[1] of fragment f_i, then,

$$RNR(S) = 1 - \frac{\sum_{i=1}^{n} LOS_{repeated}(f_i)}{\sum_{i=1}^{n} LOS_{whole}(f_i)}$$

We defined *repeated code sequence* as repetitions of its adjacent code sequence, and *non-repeated code sequence* as other parts. In the example explained previously, three tokens are considered as *repeated code sequences*, and the superscripting $\star$ is to indicate that its token is in *repeated code sequence*.

In the case of the example,

$$RNR(S_1) = 1 - \frac{0+0+0+0}{2+2+2+2} = \frac{8}{8} = 1.0$$

$$RNR(S_2) = 1 - \frac{1+2+1}{2+2+2} = \frac{2}{6} = 0.\dot{3}$$

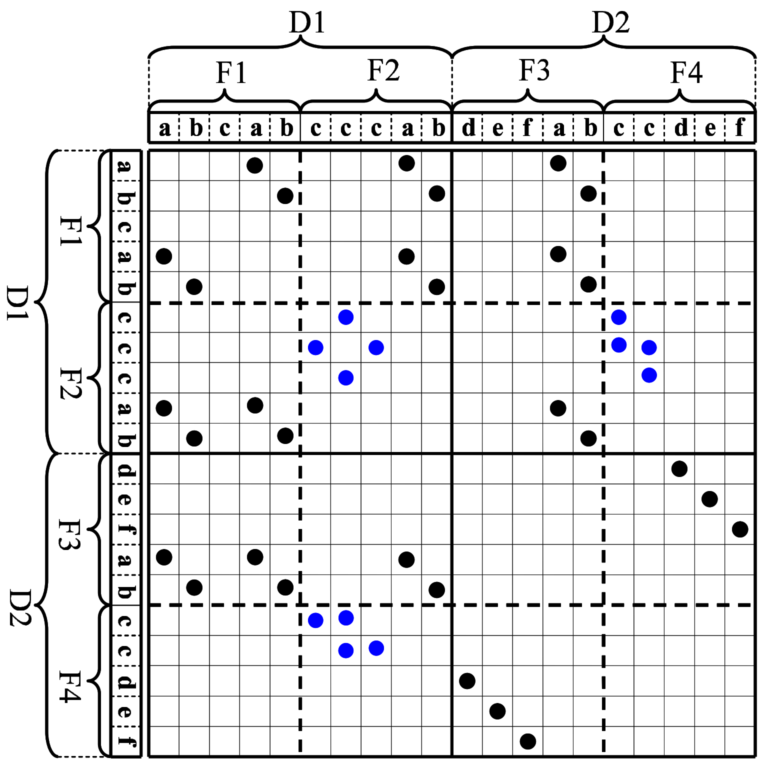
$$RNR(S_3) = 1 - \frac{0+0}{3+3} = \frac{6}{6} = 1.0$$

This metric enables users to identify clone sets such as consecutive variable declarations or consecutive accessor declarations in Java language, or repeated `printf`, `scanf`, and `switch` statements clones in C language. From our experience, '0.5' is deemed appropriate as threshold value of RNR.

2.2.2 Scatter Plot of Code Clone

We have utilized and enhanced Scatter Plot [5, 18, 45] for a bird's eye view visualization method of code clones. Figure 2.3 illustrates Scatter Plot of the example explained previously. Both the vertical and horizontal axes represent tokens of source files. The source files are sorted in alphabetical order of the file path, so that source files in the same directory are close to each other. A clone pair is shown as a diagonal line segment (We previously assumed that a cloned fragment has at least two tokens). Each dot of diagonal line segments means the corresponding tokens on the horizontal and the vertical axes are identical. The dots are spread symmetrically with the diagonal line (from the upper left corner to the bottom right corner). Using Scatter Plot, the distribution state of code clones can be grasped at a glance.

[1]A *repeated* Sequence is a sub-fragment included in fragment f_i. All tokens included in *repeated* Sequence are the same as the previous token (sequence).



F1, F2, F3, F4 : files
D1, D2 : directories
● : matched position detected as a practical code clone
● : matched position detected as a non-interesting code clone

Figure 2.3: Example of the *Scatter Plot*

Also, our Scatter Plot provides the results of filtering with metric RNR to users. Each blue dot represents an element included in a code clone judged uninteresting. By using the results of filtering, users can avoid spending much time on uninteresting code clones, which means the code clone analysis can be done more effectively by using Scatter Plot.

The directory (which means a package, in the case of Java) separators are drawn as a solid line, in order to distinguish from file separators shown as a dotted line. Users can recognize boundaries of directories and understand which directories (packages) contain many code clones, and which directories shares many code clones with other directories.

2.2.3 Clone Set Metrics

In this part, we elaborate on how we quantitatively characterize code clones, and how we visualize them. We defined the following metrics to characterize code clones.

LEN(S) : $LEN(S)$ is the average length of sequences (size of fragments) in clone set S. In the example explained previously, the values of $LEN(S_1)$, $LEN(S_2)$, and $LEN(S_3)$ are 2, 2, and 3 respectively. Using metric LEN, we can see that the fragment size of clone set S_3 is greater than the ones of clone sets S_1 and S_2.

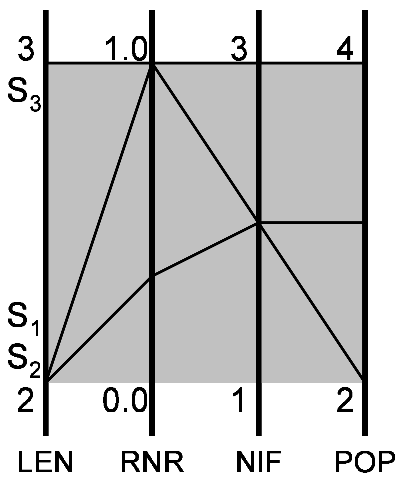
POP(S) : $POP(S)$ is the number of fragments of S. A high value of $POP(S)$ means that fragments of S appear in many places in the system. In the example, the values of $POP(S_1)$, $POP(S_2)$, and $POP(S_3)$ are 4, 3, and 2 respectively. Using metric POP, we can see that the number of occurrences of clone sets S_1 is larger than the ones of S_2 and S_3.

NIF(S) : $NIF(S)$ is the number of the source files that contain any fragments of S. A high $NIF(S)$ value may indicates bad design of the software system, absence of abstraction for fragments of S, or spread code fragments of a cross-cutting concern. In the example, the values of $NIF(S_1)$, $NIF(S_2)$, and $NIF(S_3)$ are 3, 2, and 2 respectively. Using metric NIF, we can see that clone set S_1 involves more source files than S_2 and S_3.

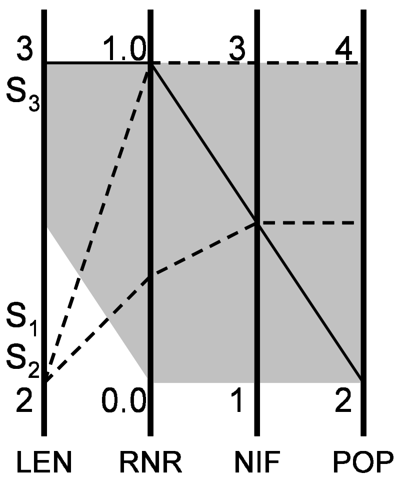
RNR(S) : $RNR(S)$ is described in Section 2.2.1. As mentioned there, using metric RNR, we can see whether each clone set is practical or uninteresting. From our experience, '0.5' is deemed appropriate as threshold value of RNR. In the example, clone set S_2 is judged uninteresting.

Using these simple metrics, we can see which clone sets are discriminative in various aspects.

We propose a visualization/selection method using **Metric Graph** for characterized code clones. We explain Metric Graph using Figure 2.4. In Metric Graph, each metric has a parallel coordinate axis. Users can specify upper and lower limits of each metric. The hatching part is the range bounded by upper and lower limits of them. A polygonal line is drawn per clone set. In this figure, 3 lines of clone sets S_1, S_2, and S_3 are drawn. In the left graph (Figure 2.4(a)), all metric values of all clone sets are in the hatching part. As such, all clone sets are in *selected* state. In the right graph (Figure 2.4(b)), the values of $LEN(S_1)$ and $LEN(S_2)$ are smaller than the under limit of LEN, which makes S_1 and S_2 in *unselected* state. This means that we can get *long* code clones by changing the lower limit of LEN.



(a) before



(b) after

Figure 2.4: Filtering clone sets using the Metric Graph

Thus Metric Graph enables users to make a choice of arbitrary clone sets based on metric values.

2.2.4 File Metrics

We also defined the following metrics to characterize source files. All metrics use only the clone sets whose RNR are threshold th or greater for calculation. Here we use '0.5' as the threshold. These metrics are used for filtering of source files described later in Section 2.3.

NOC$_{th}$**(F)** : $NOC_{th}(F)$ is the number of fragments of any clone sets in source file F, whose RNR values are equal to or greater than threshold th. In the example explained previously, the values of $NOC_{0.5}(F_1)$ and $NOC_{0.5}(F_3)$ are 2, and the ones of $NOC_{0.5}(F_2)$ and $NOC_{0.5}(F_4)$ are 1. Here, by using metric NOC, we can see that source files F_1 and F_3 has more duplicated fragments than source files F_2 and F_4.

ROC$_{th}$**(F)** : $ROC_{th}(F)$ is the ratio of duplication of source file F. In the example, the values of $NOC_{0.5}(F_1)$, $NOC_{0.5}(F_2)$, $NOC_{0.5}(F_3)$, and $NOC_{0.5}(F_4)$ are 0.8, 0.4, 1.0 and 0.6 respectively. Here, by using metric ROC, we can see that source file F_3 is completely duplicated.

NOF$_{th}$(F) : $NOF_{th}(F)$ is the number of the source files that share any code clones with source file F. In the example, the values of $NOF_{0.5}(F_1)$, $NOF_{0.5}(F_2)$, $NOF_{0.5}(F_3)$, and $NOF_{0.5}(F_4)$ are 2, 2, 3, and 1 respectively. Here, by using metric NOF, we can see that source file F_3 share code clones with all the other source files.

2.3 Code Clone Visualization Tool: Gemini

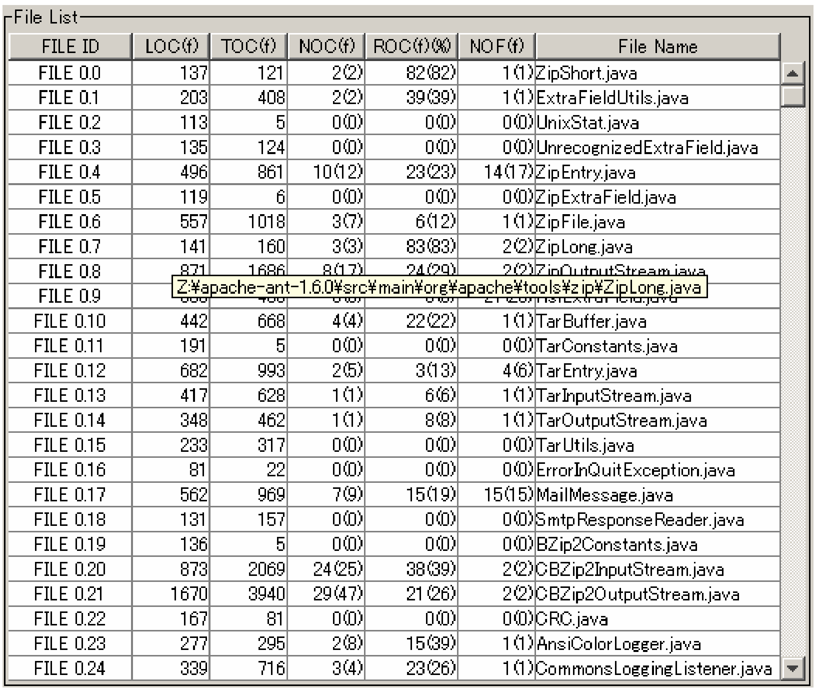
We implemented a code clone visualization tool named **Gemini** based on the proposed visualization and characterization strategies. Gemini supports all programming languages that CCFinder can handle. Gemini provides the following views as the implementations of code clone visualization mechanism and code clone/source file selection mechanisms.

- Scatter Plot
- Metric Graph
- File List

As described in Section 2.2.2, by using Scatter Plot, users can understand the distribution state of code clones at a glance, which is very useful in particular on the early stage of analysis process.

Metric Graph is designed to enable users quantitatively select clone sets. In Scatter Plot, how much clone sets are distinguished depends on their positions. For example, suppose there is a clone set $S_{example}$ which has 100 fragments. If all fragments of $S_{example}$ are in the same source file, they will be distinguished because the positions of each line segment are close to each other. But if they are in different source files, their line segments are scattered, thus will not be distinguished. On the other hand, in Metric Graph, the feature of '100 fragments' is represented as metric $POP(S_{example})$, and users can select $S_{example}$ regardless of their positions.

File List is used to select source files. File List exhibits all source files of the system with the quantitative information, file metrics described in Section 2.2.4 and two size metrics. Users can sort all source files based on any metrics. Figure 2.5 is a snapshot of File List. In this figure, 2 size metrics, $LOC(F)$ and $TOC(F)$, represent the number of lines and the number of tokens in source file F respectively. For metrics NOC, ROC, and NOF, there are 2 values respectively. The values outside parentheses are the metrics on threshold th, and ones in parentheses are the metrics on threshold 0, which means all code clones are used to calculate the metrics. File List is very useful when users want to select source files based on



File List

FILE ID	LOC(f)	TOC(f)	NOC(f)	ROC(f)(%)	NOF(f)	File Name
FILE 0.0	137	121	2(2)	82(82)	1(1)	ZipShort.java
FILE 0.1	203	408	2(2)	39(39)	1(1)	ExtraFieldUtils.java
FILE 0.2	113	5	0(0)	0(0)	0(0)	UnixStat.java
FILE 0.3	135	124	0(0)	0(0)	0(0)	UnrecognizedExtraField.java
FILE 0.4	496	861	10(12)	23(23)	14(17)	ZipEntry.java
FILE 0.5	119	6	0(0)	0(0)	0(0)	ZipExtraField.java
FILE 0.6	557	1018	3(7)	6(12)	1(1)	ZipFile.java
FILE 0.7	141	160	3(3)	83(83)	2(2)	ZipLong.java
FILE 0.8	871	1686	8(17)	24(29)	2(2)	ZipOutputStream.java
FILE 0.9	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
FILE 0.10	442	668	4(4)	22(22)	1(1)	TarBuffer.java
FILE 0.11	191	5	0(0)	0(0)	0(0)	TarConstants.java
FILE 0.12	682	993	2(5)	3(13)	4(6)	TarEntry.java
FILE 0.13	417	628	1(1)	6(6)	1(1)	TarInputStream.java
FILE 0.14	348	462	1(1)	8(8)	1(1)	TarOutputStream.java
FILE 0.15	233	317	0(0)	0(0)	0(0)	TarUtils.java
FILE 0.16	81	22	0(0)	0(0)	0(0)	ErrorInQuitException.java
FILE 0.17	562	969	7(9)	15(19)	15(15)	MailMessage.java
FILE 0.18	131	157	0(0)	0(0)	0(0)	SmtpResponseReader.java
FILE 0.19	136	5	0(0)	0(0)	0(0)	BZip2Constants.java
FILE 0.20	873	2069	24(25)	38(39)	2(2)	CBZip2InputStream.java
FILE 0.21	1670	3940	29(47)	21(26)	2(2)	CBZip2OutputStream.java
FILE 0.22	167	81	0(0)	0(0)	0(0)	CRC.java
FILE 0.23	277	295	2(8)	15(39)	1(1)	AnsiColorLogger.java
FILE 0.24	339	716	3(4)	23(26)	1(1)	CommonsLoggingListener.java

Z:¥apache-ant-1.6.0¥src¥main¥org¥apache¥tools¥zip¥ZipLong.java

Figure 2.5: Snapshot of the File List

the quantitative information. We did not apply a quantitative file selection mechanism using Metric Graph for File List, because it is more useful to use not only quantitative metrics but also their paths or names. As described in Section 2.2.3, Metric Graph is suitable for selections based on numeric values, but not suitable for strings such as file paths or file names. This brings the reason why we use File List, not Metric Graph. Also, File List has a function that sorts source files in ascending or descending order of metrics or in alphabetical order of file names.

2.4 Case Study on Open Source Software

2.4.1 Target and Configurations

We chose Ant [2](version 1.6.0) as the target. Ant 1.6.0 includes 627 source files, and the size is approximately 180,000 LOC. In this case study, we set 30 tokens as the minimum token length of a code clone (intuitively, 30 tokens correspond to about 5 LOC). The value '30' comes from our previous studies of CCFinder [33].

Table 2.1: Breakdown of uninteresting code clones

Kinds of code clones	Number of clone sets
Consecutive accessor declarations	428
Consecutive simple method declarations	224
Consecutive method invocations	177
Consecutive if- or if-else statements	160
Consecutive case entries	30
Consecutive variable declarations	29
Consecutive assign statements	19
Consecutive catch statements	4
Consecutive while-statements	2
Total	1,073

It took less than a minute to detect code clones with CCFinder. As the results of code clone detection, we found 2,406 clone sets (190,004 clone pairs). From the results, we can understand that it is unrealistic to check all detected code clones because of the enormous amount, and it is very important to select discriminative code clones or source files. In this study, we set 0.5 as the threshold of metric RNR. If $RNR(S)$ is less than 0.5, more than half of the tokens in clone set S are in repeated token sequences. The detail of the filtering results is written in Section 2.4.2.

2.4.2 Filtering with RNR

We browsed through the source code of all code clones judged uninteresting by using RNR. Table 2.1 shows the breakdown of clone sets whose RNR are less than 0.5. The number of such clone sets is 1,073, and all of them are consecution of simple implementations. As described in Section 2.1, CCFinder detects code clones after translating all user-defined names into the same special token, and so each code fragment included in the same clone set is an implementation of different contents, as in Figure 2.2.

Many consecutive accessor declarations are found as code clones coincidentally however user-defined names used in them are different from each other. As described in Section 1.2.3, CCFinder detects code clones after translating all user-defined names into the same special token, therefore they are detected as code clones.

```
public static boolean isAbstract(int access_flags) {
    return (access_flags & ACC_ABSTRACT) != 0;
}

public static boolean isPublic(int access_flags) {
    return (access_flags & ACC_PUBLIC) != 0;
}

public static boolean isStatic(int access_flags) {
    return (access_flags & ACC_STATIC) != 0;
}

public static boolean isNative(int access_flags) {
    return (access_flags & ACC_NATIVE) != 0;
}
```

(a) Consecutive simple method declarations

```
out.println();
out.println("--------------------------------------");
out.println(" ANT_HOME/lib jar listing");
out.println("--------------------------------------");
doReportLibraries(out);

out.println();
out.println("--------------------------------------");
out.println(" Tasks availability");
out.println("--------------------------------------");
doReportTasksAvailability(out);
```

(b) Consecutive method invocations

```
if (null != storepass) {
    cmd.createArg().setValue("-storepass");
    cmd.createArg().setValue(storepass);
}

if (null != storetype) {
    cmd.createArg().setValue("-storetype");
    cmd.createArg().setValue(storetype);
}

if (null != keypass) {
    cmd.createArg().setValue("-keypass");
    cmd.createArg().setValue(keypass);
}
```

(c) Consecutive if-statements

```
case Project.MSG_ERR:
    msg.insert(0, errColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_WARN:
    msg.insert(0, warnColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_INFO:
    msg.insert(0, infoColor);
    msg.append(END_COLOR);
    break;
case Project.MSG_VERBOSE:
    msg.insert(0, verboseColor);
    msg.append(END_COLOR);
    break;
```

(d) Consecutive case entries

```
private MenuBar iAntMakeMenuBar = null;
private Menu iFileMenu = null;
private MenuItem iSaveMenuItem = null;
private MenuItem iMenuSeparator = null;
private MenuItem iShowLogMenuItem = null;
private Menu iHelpMenu = null;
private MenuItem iAboutMenuItem = null;
```

(e) Consecutive variable declarations

```
src = attributes.getSrcdir();
destDir = attributes.getDestdir();
encoding = attributes.getEncoding();
debug = attributes.getDebug();
optimize = attributes.getOptimize();
deprecation = attributes.getDeprecation();
depend = attributes.getDepend();
verbose = attributes.getVerbose();
```

(f) Consecutive assign statements

```
catch (final ClassNotFoundException cnfe) {
    throw new BuildException(cnfe);
} catch (final InstantiationException ie) {
    throw new BuildException(ie);
} catch (final IllegalAccessException iae) {
    throw new BuildException(iae);
}
```

(g) Consecutive catch statements

```
e = ccList.elements();
while (e.hasMoreElements()) {
    mailMessage.cc(e.nextElement().toString());
}

e = bccList.elements();
while (e.hasMoreElements()) {
    mailMessage.bcc(e.nextElement().toString());
}
```

(h) Consecutive while-statements

Figure 2.6: Examples of uninteresting code clones

Consecutive simple method declarations are found as code clones coincidentally just like the case of consecutive accessor declarations. Figure 2.6(a) is one of such code clones. They implement simple instructions, but not accessors.

Consecutive method invocations are detected as code clones. Figure 2.6(b) is one of such code clones. It is not worthwhile that users see these code clones in the process of code clone analysis because there is nothing they can do about them.

Consecutive if-statements and if-else statements are detected as code clones. Figure 2.6(c) is one of such code clones. These code clones implement verifications of variable states. It is obvious that these code clones are harmless in the context of software maintenance, and users are needless to see them in the process of code clone analysis.

Consecutive case entries are found as code clones coincidentally just like the case of consecutive accessor declarations. Figure 2.6(d) shows one of such code clones. Usually, the programmer implements simple instructions in case entries. Moreover, CCFinder replaces all user-defined names into the same special token. Thus consecutive case entries tend to be detected as code clones, but they are harmless in the context of software maintenance.

Consecutive variable declarations and assign statements are found as code clones coincidentally just like the case of consecutive accessor declarations. Figure 2.6(e) and Figure 2.6(f) is one of such code clones respectively. These coincidences are due to the detection algorithm of CCFinder, and they shouldn't be detected as code clones.

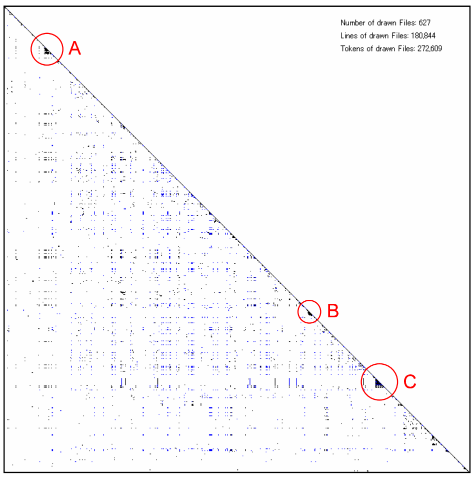
Consecutive catch statements are detected as duplicated fragments. Figure 2.6(g) is one of such code clones. Their existence is due to the specification of Java language, and they shouldn't be detected as code clones.

Consecutive while-statements are detected as code clones. Figure 2.6(h) is one of such code clones. In this case, the logics of each while-statement are very simple, and it is no problem to filter out them. But if their logics are complex, they shouldn't be filtered out.

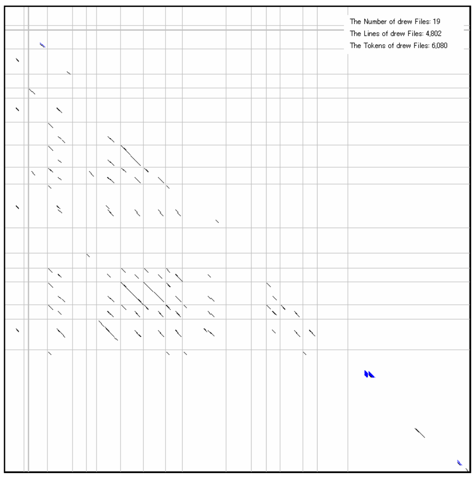
We were able to filter out 44% (1,073 out of 2,406) clone sets by using RNR. All of the clone sets that have been filtered out are either coincidental ones, inevitable duplications by the specification of Java language, or consecutive simple instructions.

2.4.3 Scatter Plot Analysis

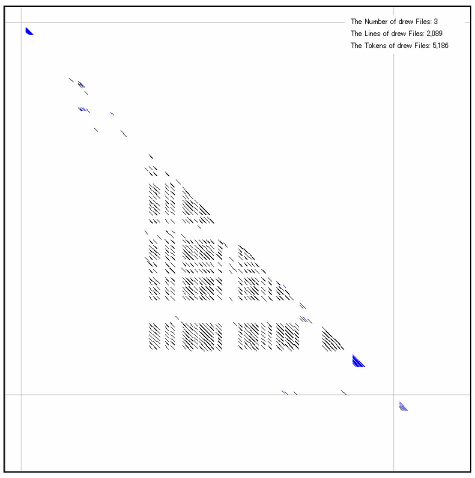
Figure 2.7 are snapshots of Ant's Scatter Plot. In Figure 2.7, clone sets whose RNR are less than 0.5 are drawn in blue, and the others are drawn in black. Each vertical or horizontal line is the border between files or between directories. In Figure 2.7(a), all such lines are being omitted because there are too many lines. We can grasp the distribution state of code clones over the system by using Scatter Plot at a glance. The parts that are distinct in Scatter Plot are the parts where there are many code clones in the system. Finding out whether duplication of such parts in the system is the expected results or not is one of the significant usages of code clone information. We investigated what kinds of implementations were conducted



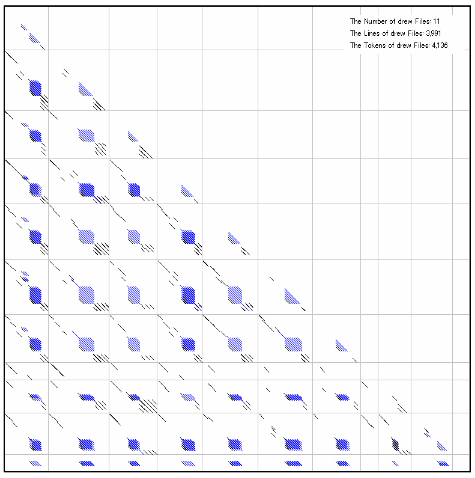
(a) Whole



(b) Zooming “A”



(c) Zooming “B”



(d) Zooming “C”

Figure 2.7: Snapshots of Scatter Plot

in distinct parts of Scatter Plot. Figure 2.7(a) is the entire of Scatter Plot. The following describes A, B, and C, the 3 different parts marked in Figure 2.7(a).

Figure 2.7(b) is a closer view of part A in Figure 2.7(a). The part illustrates source files under directory ant/filters/. These source files implement classes that return a `java.io.Reader` object under various conditions. The following are some of them.

```
if (e.getSource() == VAJAntToolGUI.this.getBuildButton()) {
    executeTarget();
}
if (e.getSource() == VAJAntToolGUI.this.getStopButton()) {
    getBuildInfo().cancelBuild();
}
if (e.getSource() == VAJAntToolGUI.this.getReloadButton()) {
```

Figure 2.8: Example of code clones in B (branching by using source of events)

ConcatFilter.java : Concatenate a file before and/or after the file.

HeadFilter.java : Read the first n-lines of a stream.

LineContains.java : Filter out all lines that don't include all the user-specified strings.

PrefixLines.java : Attach a prefix to every line.

These source files have the following functionalities in common.

1. Read a character from a specified stream. If reached the end of stream, then some operations are performed.
2. Create a new Reader object, and returns it.

The details of the functionalities in these source files were different, but processing flows were duplicated.

Figure 2.7(c) is a closer view of part B in Figure 2.7(a). It shows that source file ant/taskdefs/optional/ide/VAJAntToolGUI.java contains many code clones. This source file implements a simple GUI for providing some build information to Ant or browsing build processes. Most of these code clones were classified into either of the following two types. These code clones are typical processes of GUI.

- If-statements that determine process flow depending on source of events. Figure 2.8 is one of them.
- Method declarations that create GUI widgets. Figure 2.9 is one of them.

Figure 2.7(d) is a closer view of part C in Figure 2.7(a). It corresponds to source files under the directory ant/taskdef/optional/clearcase/. These source files implement several tasks working with ClearCase [17], which is one of the famous version control systems. Each command (for example, Checkin, Checkout, Update...) of ClearCase is implemented as a class. These source files were created by entire file copy, rather than copy and paste of particular parts of text.

```
private Panel getAboutCommandPanel() {
    if (iAboutCommandPanel == null) {
        try {
            iAboutCommandPanel = new Panel();
            iAboutCommandPanel.setName("AboutCommandPanel");
            iAboutCommandPanel.setLayout(new java.awt.FlowLayout());
            getAboutCommandPanel().add(getAboutOkButton(),
                                       getAboutOkButton().getName());
        } catch (Throwable iExc) {
            handleException(iExc);
        }
    }
    return iAboutCommandPanel;
}
```

Figure 2.9: Example of code clones in B (methods creating GUI widgets)

2.4.4 Metric Graph Analysis

We investigated what kind of code clones is quantitatively discriminative by using Metric Graph. The following types of code clones are investigated. Before performing this analysis, we raised the lower limit of RNR to filter out clone sets whose RNR are less than 0.5.

- Clone sets whose POP are high.
- Clone sets whose LEN are high.
- Clone sets whose NIF are high.

Clone sets whose POP are high

Figure 2.10 is one of the fragments making up the clone set that has more fragments than any other ones. The clone set had 31 fragments, and all of them were in source file VAJAntTool.java described in Section 2.4.3. Each fragment begins with the end of a method and ends with the beginning of its next method. This means the center parts of each method are different from each other.

Clone sets whose LEN are high

Two source files WebLogicDeployment.java and WebSphereDeployment.java, under directory ant/taskdefs/optional/ejb/, shared the longest code clones. The fragment size of clone set was 282 tokens (77 lines). Both source files implement tasks working with WebLogic [53] and WebShpere [54], which are famous application servers. Each source file has a method named isRebuildRequired, and both duplicated fragments are in these methods. Some variable names used in

```
        } catch (Throwable iExc) {
            handleException(iExc);
        }
    }
    return iAboutCommandPanel;
}

/**
 * Return the AboutContactLabel property value.
 * @return java.awt.Label
 */
private Label getAboutContactLabel() {
    if (iAboutContactLabel == null) {
        try {
            iAboutContactLabel = new Label();
            iAboutContactLabel.setName("AboutContactLabel");
```

Figure 2.10: One of the fragments making up the clone set whose POP is highest

these methods are different, but other properties (indents, blank lines, comments) are completely identical, which indicates these fragments were made by ‘copy and paste’.

Clone sets whose NIF are high

The clone set involving most source files was implementations of consecutive accessor declarations, which appeared in 19 files (22 places). The accessor’s names were different from each other, but CCFinder ignores differences of user-defined names[2] when detecting code clones. There are both setters and getters in these fragments of clone sets, thus the fragments are not simple consecutive code. RNR value of the clone set was 85.

2.4.5 File List Analysis

We investigated what kind of source files is discriminative by using File List. The following types of source files are investigated. In this analysis, we targeted only the clone sets whose RNR are 0.5 or more.

- Files whose ROC are high.
- Files whose NOC are high.
- Files whose NOF are high.

[2]It is possible to make CCFinder recognize differences of user-defined names

Table 2.2: Duplicated Ratios of Files

Range of Duplicated Ratio($ROC_{0.5}$)	# Files	Percentage
0 % - 10%	207	33 %
11 % - 20%	75	12 %
21 % - 30%	64	10 %
31 % - 40%	61	10 %
41 % - 50%	53	8 %
51 % - 60%	53	8 %
61 % - 70%	33	5 %
71 % - 80%	22	4 %
81 % - 90%	22	4 %
91 % - 100%	37	6 %
Total	627	100%

Files whose ROC are high

Table 2.2 represents the duplicated ratio distribution of source files. As we can see in this table, Ant has many source files with high duplicated ratios. Hence, we describe not only the highest duplicated ratio source file, but also top 10 files. In the following items, the numbers in parentheses are $ROC_{0.5}$ values.

FlatFileNameMapper.java (1.0) : Returns the file name included in a specified java.lang.String.

IdentityMapper.java (1.0) : This source file is a duplication of FlatFileNameMapper.java. Only the class name is different.

DirSet.java (1.0) : Treats a set of directories. This source file is a complete duplication of FileSet.java.

FileSet.java (1.0) : Treats a set of source files. This source file is a complete duplication of DirSet.java.

CCMkbl.java (0.98) : Implements a task working with ClearCase. This source file is duplicated with several source files implementing other ClearCase's tasks.

SOSCheckin.java (0.97) : Implements a task working with SourceOffSite [48]. This source file is duplicated with several source files implementing other

SourceOffSite's tasks.

StringLineComments.java (0.97) : This source file is one of the file filters described in Section 2.4.3 part A. It shares code clones with other filters.

FieldRefCPInfo.java (0.96) : Stores information of a field (for example, field name, type, owner class, ...). This source file is a duplication of InterfaceMethodRefCPInfo.java.

InterfaceMethodRefCPInfo.java (0.96) : Stores information of a method (for example, method name, signature, owner class, ...). This source file is a duplication of FieldRefCPInfo.java.

MSVSSCREATE.java (0.96) : Implements a task working with Visual SourceSafe [51]. This source file is duplicated with several source files implementing other Visual SourceSafe's tasks.

Files whose NOC are high

The source file who has the highest $NOC_{0.5}$ value is VAJAntToolGUI.java described in Section 2.4.3 part B. This source file has 378 code clones, which is overwhelming compared with any other ones.

Files whose NOF are high

The source file who has the highest $NOF_{0.5}$ value is ant/taskdefs/optional/jsp/JspC.java, and most of code clones in the source file are implementations of consecutive accessor declarations. These fragments are the same kinds as ones described in Section 2.4.4. Not only this source file, most of such source files (files with high $NOF_{0.5}$ values) have many code clones of consecutive accessor declarations.

2.4.6 Evaluation

Filtering with RNR

We examined how the RNR filtering worked well. We browsed through the source code of all detected code clones so as to calculate precision, recall and f-value of the filtering. 869 of 2,406 were practical clone sets and 1,537 were uninteresting ones. The definitions of the values are the followings.

$$recall(\%) = 100 \times \frac{\#\ real\ uninteresting\ clone\ sets\ filtered\ out\ by\ RNR}{\#\ clone\ sets\ filtered\ out\ by\ RNR}$$

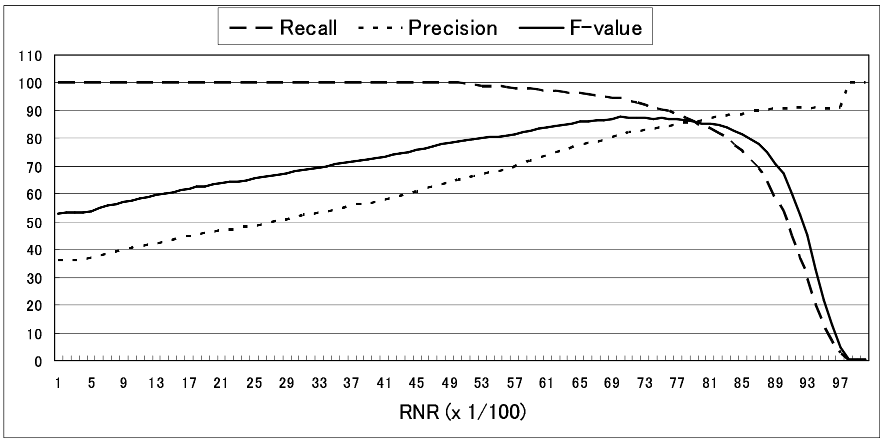


Figure 2.11: Transition of Recall, Precision, and F-value

$$precision(\%) = 100 \times \frac{\#\ clone\ sets\ filtered\ out\ by\ RNR}{\#\ all\ real\ uninteresting\ clone\ sets}$$

$$f - value = \frac{2 \times recall \times precision}{recall + precision}$$

Figure 2.11 illustrates transitions of recall, precision, and f-value when the RNR threshold is between 0 and 1.0. As mentioned above, in this case study, we used 0.5 as the threshold. Under this condition, recall is 100(%), which means that no practical clone set is accidentally filtered out at all. Also, precision is 65(%), which indicates that about one third clone sets judged practical are uninteresting. Using 0.5 as the threshold raised precision from 36(%) to 65(%). Therefore, we can conclude that most part of uninteresting clone clones are filtered out with no false positive by using 0.5 as the threshold.

It might be useful to use the value making f-value its greatest. In this case study, f-value reached its greatest when the threshold was 0.7. Under this condition, recall was 95(%) and precision was 82(%). In other words, one twentieth clone sets filtered out were practical ones and four fifths of real uninteresting clone sets were filtered out. We consider that accidentally filtering out practical code clones should be avoided because filtered clone sets might play an important role in software development and maintenance. Hence, it deems to be better to use 0.5 as the threshold than 0.7.

Table 2.3: Size of target software systems and results of executions

Target	Size		CCFinder		Gemini	
Name	# Files	LOC	Run time	Mem usage	Init time	Mem usage
Ant	627	180,844	55 sec.	30 MBytes	4 sec.	46 MBytes
JDK1.5	6,555	1,883,928	594 sec.	194 MBytes	15 sec.	137 MBytes

Using Gemini in other contexts

In this section, we will discuss the external validity of the case study. The discussion points are the followings.

- Performance and scalability of Gemini
- General versatility of the code clone analysis method described in this case study.
- Required users' skills to perform code clone analysis

First discussion point is the performance and scalability of Gemini. We applied CCFinder and Gemini to a large-scale software system, JDK 1.5 besides Ant for investigating these properties. We used a PC-based workstation[3] to perform the tools. Table 2.3 illustrates the sizes of the target software systems and the results of executions. Note that total time of CCFinder's running and Gemini's initialization is only 10 minutes despites the huge size of JDK 1.5. Additionally, the memory usage of both CCFinder and Gemini is quite reasonable. Therefore, we can conclude that the performance and scalability of these tools are enough to be used in real software development and maintenance. Users can efficiently perform code clone analysis of a large-scale software system with an ordinary PC.

Secondly, we will discuss the general versatility of the code clone analysis method described in this case study. We have already analyzed many other open source and industrial software systems, and the analysis methods for them are almost the same as the one described in this case study. This analysis method can be applied to various software systems independently of their sizes, development patterns, and their domains. From many experiences of code clone analyses, we have learnt that '30' is an appropriate value of the minimum code clone size that CCFinder detects. But infrequently under this condition, especially in the case of

[3]CPU: PentiumIV 3.0 GHz, Memory Size: 2.0 GBytes, OS: WindowsXP

large-scale software, too many code clones are detected, and we cannot efficiently analyze code clones. In such cases, users should change the minimum code clone size to '50' or '100' and re-run CCFinder for efficient analysis. Also, it became clear that industrial software tends to include more code clones than open source software. If users are going to detect and analyze code clones in a large-scale industrial software system, they should use '100' as the minimum code clone size in the first running of CCFinder. In this case study, we evaluated that 0.5 is an appropriate threshold of RNR. Since the target software is written in Java, the threshold value is probably useful for any software systems written in Java. But, for software systems written in another programming language, another value may be more useful.

Finally, we will discuss required users' skills to perform code clone analysis. In the code clone analysis with Gemini, they have to browse through that the source code of code clones and understand the implementations. Hence, they must be familiar with the programming language of the target software. And if the target software was developed by 2 or more people, the higher skill of reading source code is required. But they don't need to know the detail information of the target software; actually we don't have deep knowledge of Ant. If users had such information, they could perform deeper analysis. If users want to do the same kind of analysis as the one described in this case study, they don't need to have such information.

2.5 Case Study on Commercial Software

2.5.1 Target and Configurations

We applied Gemini to the Probe Information System[4] that was developed by 5 vendors on Advanced Software Development(AED) Project [44] of Information Technology Promotion Agency [1]. Venders developed individually and the project manager couldn't see the state of source code, the number of man-hour, and the status of development (ex. outsourcing companies, necessary human resources). On the periodic meetings that the project manager holds, each manager of venders reports only what stage of the development the vender is and how the progress is different from the plan. For helping such a blind management of the project manager, in other words, for grasp the state of black-boxed source code, we performed

[4]Prove Information System is a system that regards a vehicle as a moving sensor. The results of sensing are transformed to the center. The center provides useful information by analysing, accumulating and converting the sensing results.

Table 2.4: Amount of code clones in sub-systems

	After unit test		After combined test	
	# of code clones	Duplicated ratio	# of code clones	Duplicated ratio
V	259	33.9%	259	33.4%
W	369	27.3%	379	26.2%
X	4,483	55.3%	4,768	50.8%
Y	6,747	42.6%	7,628	46.0%
Z	2,450	56.2%	2,505	56.3%

code clone analysis[5].

We applied Gemini two times, after the unit test and after the combined test. The total LOC of the system is about hundreds thousand lines, and the source code after the combined test is 20 thousands greater than the one after the unit test. The system is written in C/C++. The analysis was performed on each vender's source code individually. In this application, we used 30 tokens as the minimum length of code clone that CCFinder detects. We also used 0.5 as the threshold of metric RNR. The analyses described in Section 2.5.3 $\sim$ 2.5.5 are for the source code after the combined test.

2.5.2 History Analysis

We analyzed how the amount of code clones after the unit test is different from the one after the combined test. Table 2.4 illustrates the amount of code clones and the duplicated ratio on the source code after the unit test and after the combined test. In the sub-system developed by company Y, the number of code clones after the combined test is greater than the one after the unit test. Usually, after unit test, no new function is added to the system. Thus we predicted that the amounts of code clones between them were not different. Figure 2.12 illustrates a graph of the relation between duplicated ratio and the number of file. We can see that the number of high duplicated files is greatly increased. We interviewed the developers of company Y. They said that these files were added just before the combined test to implement some new functions. They are library code managed in company Y, and used in many software developments. They contain many code clones but they are very stable because they are managed in many projects.

[5]This analysis was performed in the secluded room, which was established under the agreement of all vendors

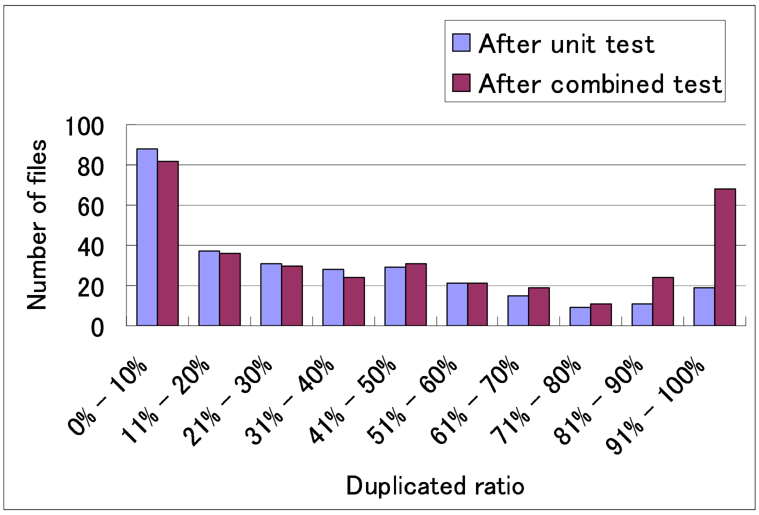


Figure 2.12: Transition of the duplicated ratio of company Y

2.5.3 Scatter Plot Analysis

Figure 2.13 illustrates a snapshot of the sub-system that a vender developed. In part A, there are many code clones filtered out by metric RNR. Actually, we browsed through source code of them, and we knew that they are code of outputting information for debug (consecutive printf-statements) or checking the validity of data (consecutive if-statements), which are not interesting as described above.

Also in part B, there are many code clones. These code clones are shared by different directories hence they are across black-lines. These directories treat position information of vehicles and each directory is for a kind of vehicles. The above means each directory treats different information but the logics are the same, which is detected as code clones.

2.5.4 Metric Graph Analysis

Before this analysis, we eliminated code clones whose RNR value is less than 0.5.

Clone sets whose LEN is high

In the sub-system of a vender, we detected a clone set whose LEN value was 441 (154 lines). This clone set consists of two code fragments, one is in AAAXXXBBB.cpp and the other is in AAAYYYBBB.cpp. In the code fragment of AAAYYYBBB.cpp,

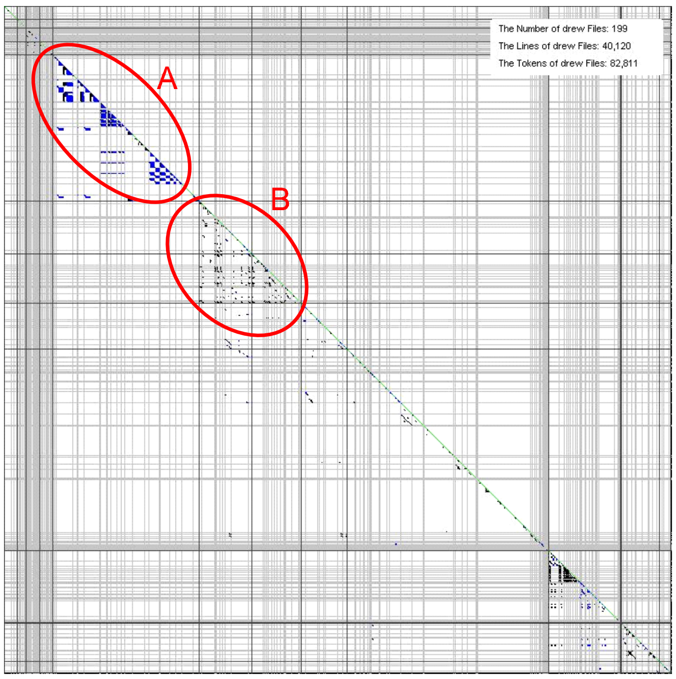


Figure 2.13: A snapshot of Scatter Plot

some method names and comments included string XXX. This implies that a copy and paste from AAAXXXBBB.cpp to AAAYYYBBB.cpp is performed, and forgot to modify some of names.

Clone sets whose POP is high

In all sub-systems, the clone set whose POP value was highest was data validity check (checking by if-statement, and if not valid output error). The kinds of data are different from each vender, but the processes of validity checking were the same logic.

Clone sets whose NIF is high

In the sub-system of a vender, we detected a clone set whose NIF value was 8 (the clone set involves 8 files). This code clone checked whether string ends with NULL. If there is no NULL, added NULL to the end. This clone set seemed to be merged easily hence each code clone in this clone sets is a whole function.

2.5.5 File List Analysis

In this analysis, we used 50 as the thureshold of RNR.

Files whose NOC are high

In the sub-system of a vender, there was a file containing 358 code clones. Code clones were scattered all over the file but concentrated on a part. Code clones are both within-file-clone and across-files-clone. These code clones seemed not to be problematic but the maintainability of the file deems to be not good.

Files whose ROC are high

The duplicated ratio of two files included in a sub-system was 96%. One is for online process, and the other is for offline process. We interviewed the developers of the system. They decided to separate implementations of the two processes before coding, and so they know the existence of the code clones.

Files whose NOF are high

A file included in a sub-system shared code clones with other 13 files. The file included various input/output processes, and each of which is duplicated with a part of other files. As a result of these duplications, the NOF value became large. Code clones are well-understood logics in the target software and the logics were simple. We determined that the code clones are not problematic.

2.6 Related Works and Discussion

Kapser et al. implemented a visualization tool named CLICS for comprehension of cloning [34]. CLICS displays the structures in the source files and the system architecture with code clone information, which makes it possible for users to easily get the information of code clones that he/she is interested in. Also, CLICS is facilitated with query support feature to display code clones satisfying the conditions of queries. CLICS doesn't implements Scatter Plot because of its limited scalability.

We believe that our enhanced Scatter Plot is scalable and useful to understand the state of code clones over software system. In our case study, the Scatter Plot works smoothly on source code of JDK 1.5, whose size is 1.8 million LOC, including 2,497,433 clone pairs in 12,522 clone sets, under the condition of $LEN \geq 30$. Our Scatter Plot displays the results that already filtered out the uninteresting code clones, which differentiates it from previous works [5, 18, 45]. This enhancement comes from our previous experiences, where we applied Scatter Plot to large-scale software systems and found an enormous amount of such code clones. Kapser et al. have the same opinion as ours [34], and their tool, CLICS has implemented some filtering features. Also, in our Scatter Plot, the directory (package) separators are differently shown from the file ones, which makes it possible for users to know the boundaries of directories, and then finds out directories (packages) that contain many code clones and directories that shares many code clones with other directories. Users can see the state of practical code clones in the package hierarchy of a software system by using our Scatter Plot since uninteresting code clones are filtered out.

Also Kapser et al. describes functionalities to support navigation and understanding of cloning in a software system [34]. The functionalities are as below.

1. *Facilities to evaluate overall cloning activity.*

2. *Mechanisms to guide users toward clones that will be most effectively used in their task.*

3. *Methods for filtering and refining the analysis of the clones.*

We think of these functionalities in the context of our tool, Gemini. Scatter Plot provides a bird's eye view, therefore it accomplishes the first functionality. We believe that Metric Graph and File List implement the second functionality because we can easily get arbitrary clone sets based on their quantitative features by using them. The third functionality is realized through metric RNR. RNR enables users to filter out uninteresting code clones. With all these characteristics mentioned above, Gemini is definitely useful and appropriate as a code clone visualization tool.

Rieger et al. suggested and implemented some diagrams to visualize code clone information [45]. Their diagrams are based on the principle of Polymetric View, and provide abstracted code clone information on various granularities to users. They also said that, in the large-scale software too many code clones are detected, and some filtering functionalities are essential.

Johnson suggested a navigation method using HTML [29]. The hyperlink functionality of HTML enables users to jump freely between source files having clone relations with each other or fragments included in the same clone set (in his paper, term Hash is used instead of clone set). We do agree hyperlink properties are very nice to navigate users, but there is no functionality to see the state of code clones over the system.

Basit et al. suggested a method detecting structural clones [10]. A structural clone is a pattern of cloned code fragments, and it indicates the presence of design-level similarities. They also implemented a tool detecting such clones based on the "market basket analysis" technique of the Data Mining domain. Providing structural clone information is a great support of program comprehension, but how the information is expressed is a big challenge.

Walenstein et al. reported that judgment of code clones varies among experts [52]. In one of their experiments, for more than 60% of automatically detected clones, three experts disagreed whether the fragments are really code clone or not. Our metric system does not settle the controversy, but it helps users to make a choice of what kind of codes should be regarded as code clones.

2.7 Summary

We proposed visualization and characterization methods of code clones in a software system, and implemented them as a tool, Gemini. Gemini provides valuable mechanisms as described below.

- Mechanism for filtering out uninteresting code clones.
- Mechanism for viewing the state of code clones over a system.
- Mechanism for navigating users to code clones which have the features that they are interested in.

As described in Sections 2.4 and 2.5, our proposed methods worked well and we were able to figure out various cloning activities in Ant.

Chapter 3

Refactoring Support Method

3.1 Motivation

One of the promising approaches to get the high maintainable software is refactoring. *Refactoring* is defined as a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [20]. It is generally accepted that refactoring is useful to improve the maintainability of software. However, it is difficult to introduce refactoring to actual software development. The developer cannot afford to apply refactorings because they are too busy implementing given requirements and it's difficult to identify refactoring candidates, determine what refactoring patterns should be applied and understand the actual effects of refactorings to software maintenance. Thus, it is necessary to support the difficulties of refactoring.

Code clone is considered as one of the typical *Bad Smells* in refactoring process [20], so that the code clone detection is identification of code fragments to be refactored. From a practical standpoint, it is very hard to identify which code clones should be merged. Some code clones are not simply able to be merged, or at least not appropriate to be merged since the merged code will make the source code less understandable. Usually, large-scale software have complicated logics intertwining each other, which make the maintainer confused judging which code clones can be merged and how to merge them.

To conduct effective refactoring for code clone, we have to extract code clones that can be refactored. Since the detection speed of CCFinder is very fast, the tool can be used to conduct practical refactoring. However the code clones detected by CCFinder are sequences of tokens as described in Section 1.2.3, they are not necessarily appropriate to be directly merged into one module.

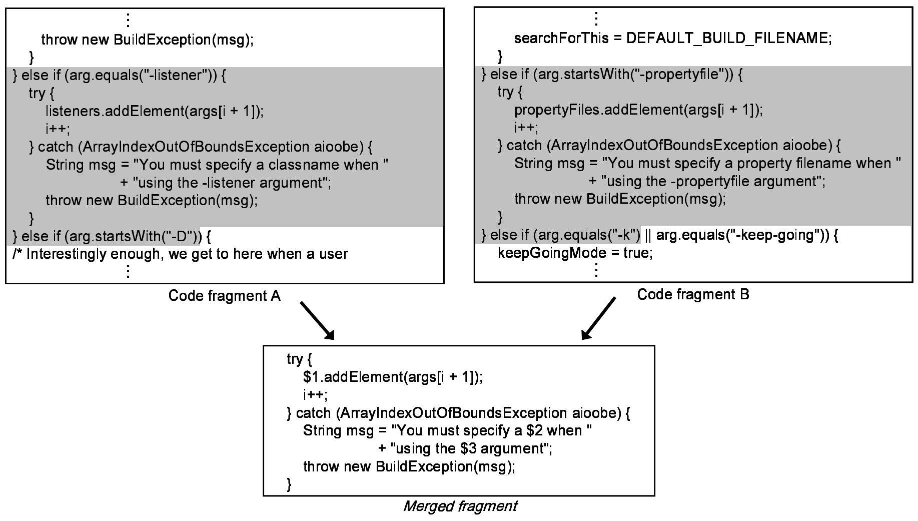
In this Chaper, we propose a refactoring support method for code clones. Our method consists of two steps. In the first step, the method extracts refactoring-oriented code clones from the results of CCFinder. In the second step, the method calculates some metrics of refactoring-oriented code clones to suggest how they can be refactored. In other words, the method tells users which code clones can be removed and how to remove them. we also describe about the applicability and usefulness of refactorings suggested by our method.

3.2 Proposal Techniques

Here, we describe a refactoring support method for code clones. Usually, refactorings are performed in the following steps [20, 42]. (1) *Identify where the software should be refactored*, (2) *determine which refactoring patterns should be applied to the identified locations*, (3) *confirm that the refactoring doesn't change the external behavior of the software*, (4) *modify the source code*, (5) *assess the effect of the refactoring on the software quality characteristics*, and (6) *maintain consistency between the refactored program code and other software artifacts (or vice versa).* In refactoring process, the steps (1) and (2) are very complicated tasks, especially, in large-scale software. Our approach identifies where can be refactored, and tells users which refactoring patterns can be applied. So, users can perform refactorings effectively. The method consists of two phases. At first, the method extracts refactoring-oriented code clones from ones detected by CCFinder, which corresponds to the step (1). Secondly, the method provides appropriate refactoring patterns of each extracted code clone to users, which corresponds to the step (2).

3.2.1 Extraction of Refactoring-Oriented Code Clones

As described above, firstly, refactoring-oriented code clones are extracted from ones detected by CCFinder. Here, we regard structural code clones as refactoring-oriented ones. Figure 3.1 shows an example. In this figure, there are two fragments A and B from a program, and the fragments with hatchings are token-based code clone between them. In fragment A, operations on *class name* are performed, and in fragment B, operations on *property file name* are performed. The try-catch blocks in A and B have a common logic that handles a java.util.Vector data structure. There are, however, sentences before and after try-catch blocks, which are not necessarily related with the try-catch blocks from the semantic standpoint. Such semantically unrelated sentences often obstruct refactoring. In other words, extracting only try-catch blocks as code clones is more preferable from refactoring viewpoint in this example. As shown in this example, each logical block is similar to the scope of programming language. So our proposed method extracts structural



```
            ⋮
    throw new BuildException(msg);
  }
} else if (arg.equals("-listener")) {
  try {
    listeners.addElement(args[i + 1]);
    i++;
  } catch (ArrayIndexOutOfBoundsException aioobe) {
    String msg = "You must specify a classname when "
               + "using the -listener argument";
    throw new BuildException(msg);
  }
} else if (arg.startsWith("-D")) {
/* Interestingly enough, we get to here when a user
            ⋮
```

```
            ⋮
    searchForThis = DEFAULT_BUILD_FILENAME;
  }
} else if (arg.startsWith("-propertyfile")) {
  try {
    propertyFiles.addElement(args[i + 1]);
    i++;
  } catch (ArrayIndexOutOfBoundsException aioobe) {
    String msg = "You must specify a property filename when "
               + "using the -propertyfile argument";
    throw new BuildException(msg);
  }
} else if (arg.equals("-k") || arg.equals("-keep-going")) {
  keepGoingMode = true;
            ⋮
```

```
try {
  $1.addElement(args[i + 1]);
  i++;
} catch (ArrayIndexOutOfBoundsException aioobe) {
  String msg = "You must specify a $2 when "
             + "using the $3 argument";
  throw new BuildException(msg);
}
```

Figure 3.1: Example of merging two code fragments

blocks inside code clones

3.2.2 Provision of Appricable Refactoring Patterns

Secondly, appricable refactoring patterns of extracted code clones are provided to users. So far, many refactoring patterns have been proposed [20], and some of them can be used for merging code clones as shown in Section 1.3.

The method judges which refactoring patterns can be used to each code clone. Merging ways of code clones can be divided into two types, (a) extract a code fragment as a new module, and (b) move an existing module to other place.

We show an example of type (a) using *Extract Method* pattern. Originally, *Extract Method* is applied to a too long method or a part of complicated function in order to improve the readability, understandability, and maintainability. It also can be applied to code clones to merge them. To apply this pattern, it is desirable that each code fragment of the clone set has low coupling with its surrounding code. In other words, the less the variables defined outside the code fragment are used (referred and assigned), the easier we move it to other place. If such variables are used, it is necessary to provide them as parameters for the extracted method. Therefore, to measure the amount of such variables, we defined two metrics **NRV(S)** (*the Number of Referred Variables*) and **NAV(S)** (*the Number of Assigned Variables*). Here, we assume that a clone set S includes code fragments $f_1,\ f_2,\ \cdots,\ f_n$. The code fragment f_i refers s_i variables defined externally, and assigns to t_i variables defined externally. Then,

$$NRV(S) = \frac{1}{n}\sum_{i=1}^{n} s_i, \quad NAV(S) = \frac{1}{n}\sum_{i=1}^{n} t_i,$$

Intuitively, $NRV(S)$ represents the average number of externally defined variables referred in the code fragments of clone set S, $NAV(S)$ represents the average number of externally defined variables assigned to in the code fragments of S.

Next, we demonstrates an example of type (b) using *Pull Up Method* pattern. *Pull Up Method* means that a method in a class is moved to its parent class. If several child classes have identical methods, moving them to the common parent class is an effective refactoring. Naturally, classes including code clones have to be descendants of the common parent class. Therefore, to measure the positional relationship of code clones in the class hierarchy, we defined a metric **DCH(S)** (*the Dispersion in Class Hierarchy*). As described above, a clone set S includes code fragments $f_1,\ f_2,\ \cdots,\ f_n$. C_i denotes the class that includes the code fragment f_i. Then, if classes $C_1,\ C_2,\ \cdots,\ C_n$ have some common parent classes, C_p is defined as a class which lays the lowest position in the class hierarchy among the parent classes $C_1,\ C_2,\ \cdots,\ C_n$. Also, $D(C_k,\ C_h)$ represents the distance between class C_k and class C_h in the class hierarchy. Then,

$$DCH(S) = max\ \{D(C_1,\ C_p),\ D(C_2,\ C_p),\ \cdots,\ D(C_n,\ C_p)\}$$

The value of $DCH(S)$ becomes large as the degree of the dispersion of S becomes extended. If all code fragments of S are in the same class, the value of $DCH(S)$ is set as 0. If all code fragments of S are in a class and its direct child classes, the value of $DCH(S)$ is set as 1. Exceptionally, if some of classes have no common parent class, the value of $DCH(S)$ is set as ∞. In detail, this metric is measured for only the class hierarchy where the target software exists because it is unrealistic that users pull up some 'method's which are defined in the target software classes to library classes like JDK.

3.2.3 Example of Refactoring Process

We demonstrate two examples of filtering conditions using two refactoring patterns *Pull Up Method* and *Extract Method*.

Pull Up Method

If users want to perform *Pull Up Method*, the following conditions should be considered for example.

PC1 : The target is the method unit.

PC2 : The value of $DCH(S)$ is 1 or more (not ∞).

Usually, *Pull Up Method* is performed on existing methods, so (PC1) is considered. Furthermore, all classes sharing code fragments (methods) of the same clone set have to inherit a common parent class, so (PC2) is considered. By using the conditions, clone sets are categorized as described below.

PG1 : Clone sets that can be merged only by moving each of the code fragments to the common parent class.

PG2 : Clone sets that can be merged by moving each of the code fragments to common parent class and adding parameters for each variable which is defined outside of them. Existing methods which include the pull-uped code clones can be deleted or changed so that they call the new method from the inside. If they are deleted, it is necessary to change all its caller places, because the signature was changed.

PG3 : Clone sets that can be merged by moving the code fragments to the common parent class and adding parameters for each variable which is defined outside and adding a return-statement. As well as (PG2), existing methods can be deleted or changed from the same reason.

PG4 : Clone sets that need much contrivance to be merged.

Extract Method

If users want to perform *Extract Method*, a typical set of conditions will be as follows.

EC1 : The target is statement unit.

EC2 : The value of $DCH(S)$ is 0.

EC3 : The value of $NAV(S)$ is 1 or less.

Since *Extract Method* is directed to a part in a method, (EC1) is considered. If all code fragments of a clone set S are in the same class, it is easy to merge them, so, (EC2) is considered. The reason to consider (EC3) is that, if some values are assigned to variables defined externally, it is necessary to make them parameters of the new extracted method, and to return them to its caller place to reflect the values of them. It is necessary to contrive like making a new data class if two or more values are assigned to. By using these conditions, clone sets are categorized as described below.

EG1 : Clone sets that can be merged only by extracting them and making a new method in the same class.

EG2 : Clone sets that can be merged by extracting them and making a new method with setting the externally defined variables as parameters of it, because such variables are referred in the code clone.

EG3 : Clone sets that can be merged by extracting them and making a new method with setting the externally defined variables as parameters of it and adding a return-statement to deliver the results of the assignments to the caller place.

EG4 : Clone sets that can be merged but need much effort.

3.3 Refactoring Support Tool: Aries

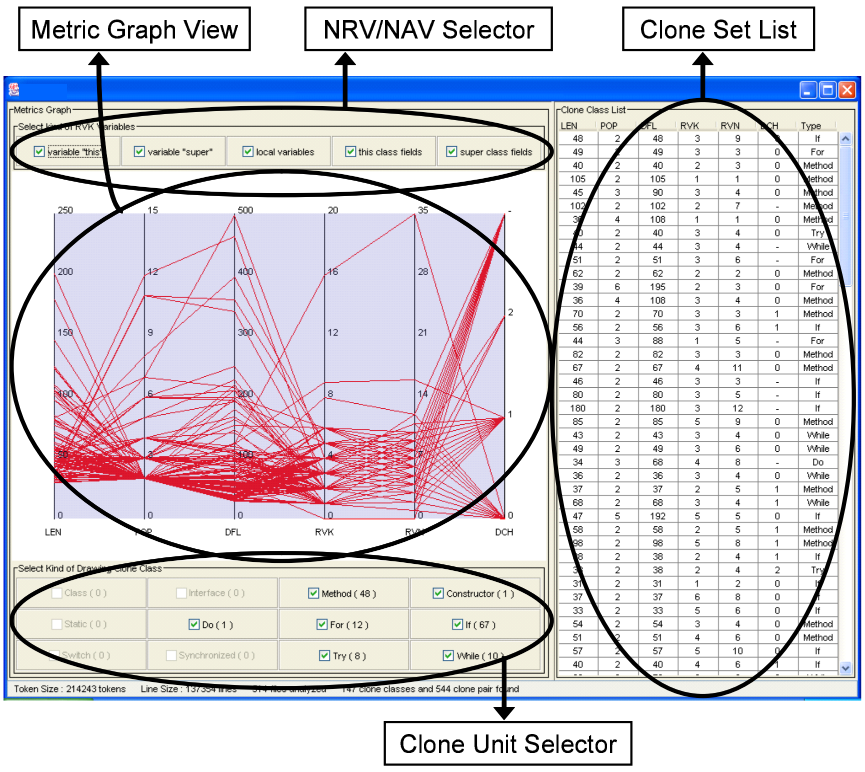
We implemented a refactoring support tool, Aries based on the refactoring pattern recommendation method. Currently, Aries supports only Java language, but can be extended to other languages. Aries consists of Extraction Unit and GUI Unit.

Extraction Unit extracts the following types of code clones from source code as refactoring-oriented ones.

Declaration	:	class { }, interface { }
Function	:	method, constructor, static-initializer
Statement	:	if, for, while, do, switch, try, synchronized

Also, Extraction Unit quantitatively characterizes refactoring-oriented code clones by using 6 metrics. These metrics are $NRV(S)$, $NAV(S)$, $DCH(S)$, which are described in Section 3.2.2, and $LEN(S)$, $POP(S)$, $DFL(S)$, which are described in Section 2.2.3. Refactoring-oriented code clone information with the metrics are stored into a file with the XML format.

GUI Unit includes two windows, Main Window and Clone Set Viewer. Users select code clones that he/she is going to refactor on Main Window after he/she gives the XML file to GUI Unit. Figure 3.2 is a snapshot of Main Window. Users can get detail information of the code clones that they selected on Clone Set



Select kind of RVK Variables

[✓] variable "this"   [✓] variable "super"   [✓] local variables   [✓] this class fields   [✓] super class fields

Select Kind of Drawing Clone Class

[ ] Class (0)	[ ] Interface (0)	[✓] Method (48)	[✓] Constructor (1)
[ ] Static (0)	[✓] Do (1)	[✓] For (12)	[✓] If (67)
[ ] Switch (0)	[ ] Synchronized (0)	[✓] Try (8)	[✓] While (10)

Token Size : 214243 tokens   Line Size : 137354 lines   574 files analyzed   147 clone classes and 544 clone pair found

Clone Class List

LEN	POP	DFL	RVK	RVN	DCH	Type
48	2	48	3	9	[illegible]	If
49	2	49	3	3	0	For
40	2	40	2	3	0	Method
105	2	105	1	1	0	Method
45	3	90	3	4	0	Method
102	2	102	2	7	-	Method
36	4	108	1	1	0	Method
40	2	40	3	4	0	Try
44	2	44	3	4	-	While
51	2	51	3	6	-	For
62	2	62	2	2	0	Method
39	6	195	2	3	0	For
36	4	108	3	4	0	Method
70	2	70	3	3	1	Method
56	2	56	3	6	1	If
44	3	88	1	5	-	For
82	2	82	3	3	0	Method
67	2	67	4	11	0	Method
46	2	46	3	3	-	If
80	2	80	3	5	-	If
180	2	180	3	12	-	If
85	2	85	5	9	0	Method
43	2	43	3	4	0	While
49	2	49	3	6	0	While
34	3	68	4	8	-	Do
36	2	36	3	4	0	While
37	2	37	2	5	1	Method
68	2	68	3	4	1	While
47	5	192	5	5	0	If
58	2	58	2	5	1	Method
98	2	98	5	8	1	Method
38	2	38	2	4	1	If
38	2	38	2	4	2	Try
31	2	31	1	2	0	If
37	2	37	6	8	0	If
33	2	33	5	6	0	If
54	2	54	3	4	0	Method
51	2	51	4	6	0	Method
57	2	57	5	10	0	If
40	2	40	4	6	1	If

Figure 3.2: A Snapshot of Main Window

Viewer. Figure 3.3 is a snapshot of Clone Set Viewer. Here, we explain each component of GUI Unit.

Metric Graph View

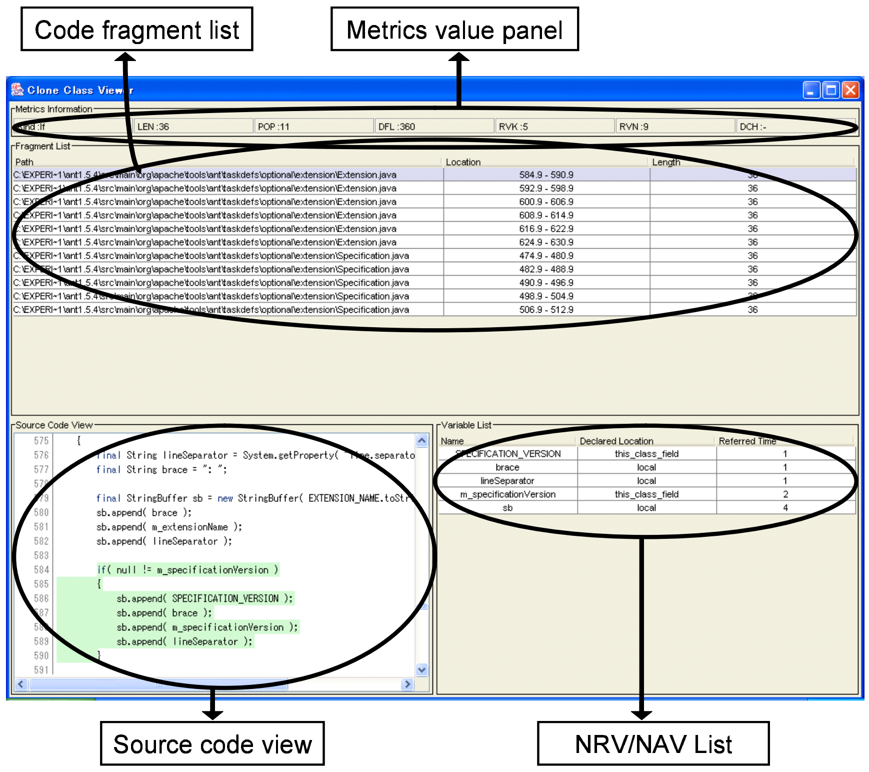
Aries's Metric Graph is identical to Gemini's one except metric types. How Metric Graph works is described in Section 2.2.3.

Clone Unit Selector

In Clone Unit Selector, users can choose the target unit of refactoring. For example, in performing *Parameterized Method*, only method unit should be selected.

NRV/NAV Selector

Variable types used for calculating $NRV(S)$ and $NAV(S)$ can be chosen in NRV/NAV Selector. The variable types are as follows.



Metrics Information

Kind :If	LEN :36	POP :11	DFL :360	RVK :5	RVN :9	DCH :-

Fragment List

Path	Location	Length
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	584.9 - 590.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	592.9 - 598.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	600.9 - 606.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	608.9 - 614.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	616.9 - 622.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Extension.java	624.9 - 630.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Specification.java	474.9 - 480.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Specification.java	482.9 - 488.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Specification.java	490.9 - 496.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Specification.java	498.9 - 504.9	36
C:\EXPERI~1\ant1.5.4\src\main\org\apache\tools\ant\taskdefs\optional\extension\Specification.java	506.9 - 512.9	36

Source Code View

```
    {
        final String lineSeparator = System.getProperty( "line.separato
        final String brace = ": ";

        final StringBuffer sb = new StringBuffer( EXTENSION_NAME.toStr
        sb.append( brace );
        sb.append( m_extensionName );
        sb.append( lineSeparator );

        if( null != m_specificationVersion )
        {
            sb.append( SPECIFICATION_VERSION );
            sb.append( brace );
            sb.append( m_specificationVersion );
            sb.append( lineSeparator );
        }

```

Variable List

Name	Declared Location	Referred Time
SPECIFICATION_VERSION	this_class_field	1
brace	local	1
lineSeparator	local	1
m_specificationVersion	this_class_field	2
sb	local	4

Figure 3.3: A Snapshot of Clone Set Viewer

- field-members-of-its-class,
- field-members-of-parent-class,
- field-members-of-interface,
- local-variables.

For example, in performing *Extract Method* within a class, only *local-variables* need to be considered, because other variable types can be accessed wherever in the same class. In performing *Pull Up Method*, we think that *field-members-of-parent-class* don't need to be considered, but *field-members-of-its-class* have to be considered. Like this, users can choose which types are counted according to each refactoring pattern.

Clone Set List

Clone Set List shows clone sets which are *selected state* in Metric Graph View. The list has a function to sort clone sets in ascending or descending order of arbitrary metrics. Double-clicking a clone set on the list is a trigger

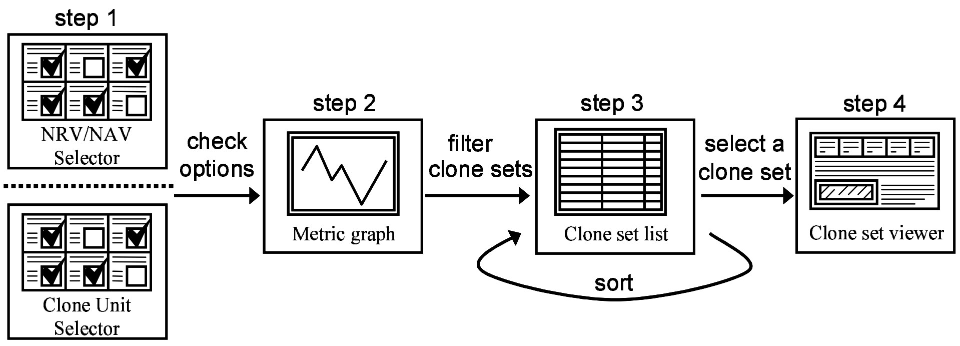


Figure 3.4: Analysis precess using GUI Unit

to display Clone Set Viewer. It provides more detail information of the selected clone set to users.

Metrics Value Panel
Metrics Value Panel displays metrics values of the selected clone set.

Code Fragment List
Code Fragment List displays the list of code fragments included in the selected clone set. Each element of the list has three kinds of information, a path to each file including the code fragment, the location of the code clone in the file, and the size of the code fragment.

Source Code View
Source Code View works cooperatively with Code Fragment List. Users can browse the actual source code corresponding to the code fragment selected in Code Fragment List. The code clone is emphatically displayed.

NRV/NAV List
NRV/NAV List displays the list of variables which are used and defined externally in the code fragment selected in Code Fragment List. Each element of the list has three kinds of information, the variable name, the variable type and the number of usage.

Figure 3.4 illustrates the model of the analysis process on GUI Unit. In Step 1, users decide which variable types are used to calculate metrics and which unit types are target that they are going to refactor by using NRV/NSV Selector and Clone Unit Selector.

In Step 2, users filter code clones by changing the lower and upper limits of metrics. The code clones that satisfying the metrics conditions are listed in Clone Set List. Currently, Metric Graph View has pre-defined conditions for the following refactoring patterns. By only choosing a refactoring pattern, users can get clone sets which can be merged using the pattern.

- *Extract Class*,
- *Extract Method*,
- *Extract SuperClass*,
- *Form Template Method*,
- *Move Method*,
- *Parameterize Method*,
- *Pull Up Constructor*, and
- *Pull Up Method*.

In Step 3, clone sets filtered in Metric Graph View are displayed on the Clone Set List with metrics values, and this list can sort clone sets in ascending or descending order of arbitrary metrics. Users select a clone set in this list.

Finally, in Step 4, users get detail information of the selected clone set by using Clone Set Viewer. The information includes all metric values and all code fragments of the selected clone set. And for each code fragment, variables which are used to calculate $NRV(S)$ and $NAV(S)$ are displayed with the number of references and assignments. Also, users can browse source code of the code fragments selected in Code Fragment List. Users can conduct refactorings effectively using the information.

3.4 Evaluation on Applicability

In this Section, we describe the case study that we conducted to evaluate applicability of refactorings suggested by Aries.

Target and Configuration

We chose Ant [2](version 1.6.0) as our target because of two reasons. Firstly, Ant is written in Java language. As previously mentioned, CCFinder can deal with several popular programming languages (i.e., C/C++, Java, COBOL, Fortran, ...), but Aries can deal with only Java language. Secondly, the Ant package includes many test cases which can be used to confirm that Ant's external behavior doesn't change by refactorings.

Ant 1.6.0 includes 627 source files, and the size is about 180,000 LOC. In this case study, we set 30 tokens as the minimum token length of code clone (intuitively, 30 tokens correspond to about 5 LOC). The value 30 comes from our previous studies of CCFinder [33].

We applied *Extract Method* and *Pull Up Method* refactoring patterns to code clones. It took 2 minutes to detect refactoring-oriented code clones, and we got 154 clone sets from Ant. Fifty-nine of them satisfied the conditions of *Extract Method*, and twenty of them satisfied the conditions of *Pull Up Method*. The conditions of *Extract Method* and *Pull Up Method* are the same as ones described in Section 3.2.3. In Sections 3.4.1 and 3.4.2, we describe the details of refactoring using Aries. Also, after merging each clone set, we performed regression tests to confirm the behavior of Ant. In the regression test process, we used totally 220 test cases included in the Ant package. These test cases were conducted to use JUnit [32], which is one of unit testing frameworks. So, we could easily perform all test cases and took about 4 minutes to perform all of them.

3.4.1 Result of *Extract Method*

As described above, we got 59 clone sets as the result using the conditions of *Extract Method*. Then, we browsed and examined source code of all clone sets, and classified them into 4 groups (EG1) $\sim$ (EG4), which are described in Section 3.2.3. After classifying, we merged all code clones in groups (EG1) $\sim$ (EG3).

```
if (!isChecked()) {
    // make sure we don't have a circular reference here
    Stack stk = new Stack();
    stk.push(this);
    dieOnCircularReference(stk, getProject());
}
```

Figure 3.5: Example of *Extract Method* in (EG1)

Three clone sets were classified into (EG1). Figure 3.5 shows a code fragment included in them. In this if-statement clone, no externally defined *local-variable* was used. So, it was very easy to extract each code fragment as a new method in the same class.

```
if (javacopts != null && !javacopts.equals("")) {
    genicTask.createArg().setValue("-javacopts");
    genicTask.createArg().setLine(javacopts);
 }
```

Figure 3.6: Example of *Extract Method* in (EG2)

Thirty-four clone sets were classified into (EG2). Figure 3.6 shows a code fragment included in them. In this if-statement clone, variable javacopts was a *field-member-of-its-class*, and variable *genicTask* was a *local-variable*. So, it was necessary to set genicTask as a parameter of a new method to extract each code fragment in the same class.

```
if (iSaveMenuItem == null) {
   try {
       iSaveMenuItem = new MenuItem();
       iSaveMenuItem.setLabel("Save BuildInfo To Repository");
   } catch (Throwable iExc) {
       handleException(iExc);
   }
}
```

Figure 3.7: Example of *Extract Method* in (EG3)

Fifteen clone sets were classified into (EG3). Figure 3.7 shows a code fragment included in them. In this if-statement clone, variable iSaveMenuItem was externally defined. Moreover, there was an assignment to the variable in the code fragment. So, it was necessary to make iSaveMenuItem a parameter of the new method and add a return-statement to reflect the results of the assignment to the caller code.

```
if (name == null) {
    if (other.name != null) {
        return false;
    }
} else if (!name.equals(other.name)) {
    return false;
}
```

Figure 3.8: Example of *Extract Method* in (EG4)

Seven clone sets were classified into (EG4). Figure 3.8 shows a code fragment included in them. In this if-statement clone, two return-statements existed. So, much effort would be necessary to extract it. In this case study, we didn't merge these seven clone sets because we thought that merging them would be strongly dependent on the skill of each programmer.

3.4.2 Result of *Pull Up Method*

Here, we describe the result of applying *Pull Up Method*. As described above, we got 20 clone sets as the results using the conditions of *Pull Up Method*. Then, we browsed and examined source code of all code clones, and classified them into 4 groups (PG1) $\sim$ (PG4), which are described in Section 3.2.3. After classifying, we merged all code clones in groups (PG1) $\sim$ (PG3).

In this case study, no clone set was classified into (PG1).

```
private void getCommentFileCommand(Commandline cmd) {
    if (getCommentFile() != null) {
        /* Had to make two separate commands here because
           if a space is inserted between the flag and the
           value, it is treated as a Windows filename with
           a space and it is enclosed in double quotes (").
           This breaks clearcase.
        */
        cmd.createArgument().setValue(FLAG_COMMENTFILE);
        cmd.createArgument().setValue(getCommentFile());
    }
}
```

Figure 3.9: Example of *Pull Up Method* in (PG2)

Ten clone sets were classified into (PG2). Figure 3.9 shows a code fragment included in them. In this method clone, variable this was omitted at calling method getCommentFile, since it was defined in the same class. Variables this and FLAG_COMMENTFILE, which were *field-members-of-its-class*, were externally defined. So, we pulled up them to the common parent class after adding two parameters.

```
public void verifySettings() {
    if (targetdir == null) {
        setError("The targetdir attribute is required.");
    }
    if (mapperElement == null) {
        map = new IdentityMapper();
    } else {
        map = mapperElement.getImplementation();
    }
    if (map == null) {
        setError("Could not set <mapper> element.");
    }
}
```

Figure 3.10: Example of *Pull Up Method* in (PG3)

Two clone sets were classified into (PG3). Figure 3.10 shows a code fragment included in them. In this method clone, variable map was externally defined, and some values were assigned to it. Method setError was defined in the common parent class. So, in order to pull up this clone set to the common parent class, it was necessary to add a parameter and a return-statement for variable map.

```
public void execute() throws BuildException {
    Commandline commandLine = new Commandline();
    Project aProj = getProject();
    int result = 0;

    // Default the viewpath to basedir if it is not specified
    if (getViewPath() == null) {
        setViewPath(aProj.getBaseDir().getPath());
    }

    // build the command line from what we got. the format is
    // cleartool checkin [options...] [viewpath ...]
    // as specified in the CLEARTOOL.EXE help
    commandLine.setExecutable(getClearToolCommand());
    commandLine.createArgument().setValue(COMMAND_CHECKIN);

    checkOptions(commandLine);

    result = run(commandLine);
    if (Execute.isFailure(result)) {
        String msg = "Failed executing: " +
                     commandLine.toString();
        throw new BuildException(msg, getLocation());
    }
}
```

Figure 3.11: Example of *Pull Up Method* in (PG4)

Eight clone sets were classified into (PG4). Figure 3.11 shows a code fragment included in them. This method called method checkOptions, which was defined in the same class. Methods getProject, getViewPath and getLocation were defined by using the common parent class. Also, variable commandLine, which was a parameter of checkOptions, was defined and used in this code fragment. Method checkOptions was defined in each class having a code clone of the clone set, and each code clone called different method checkOptions. It prevented them from merging with *Pull Up Method*. But, we thought that we would be able to apply *Form Template Method* pattern on them. That is, we move the code clone to the common parent class. Then, we define an abstract method named checkOptions in the common parent class.

Table 3.1: Number of Detected Clone Sets

Unit	# of Clone Sets	Refactoring Pattern
Declaration	4	Extract Super Class
Function	13	Move Method
Statement	49	Extract Method

3.5 Evaluation on Usefulness

3.5.1 Outline

In this case study, we evaluate whether refactorings suggested by Aries are useful[1]. The target is a web-based system developed in Hitachi Systems & Services, Ltd, which is a software company in Japan. The total LOC of the system is about 310,000. Some parts of the system (about 70,000 (309 classes) LOC) were developed from scratch and we applied Aries to them.

In this study, we used 50 as the minimum clone length that Aries detects. In the case study of applicability, which is described in Section3.4, we uses 30. But in this case study, we evaluate the usefulness of refactorings suggested by Aries. We think that the size of code clones affects the usefulness of refactoring, so that we used a slightly larger value, 50. In the default settings, Aries ignores differences of all user-defined names (variable names, type names, object types, and so on). But, in this study, we configured that Aries did not ignore differences of type names, method names, and literals. Because it is difficult and costly to refactor code clones with such differences.

With the above settings, Aries detected 66 clone sets. We applied different refactoring patterns to each unit(*Declaration*, *Function*, and *Statement*) of clone sets. Table 3.1 classifies the detected clone sets.

After the detection of code clones, we filtered clone sets by using some conditions. And, a maintainer of the system evaluated whether refactorings of filtered clone sets are effective. Next, we explain some conditions to filter the clone sets.

Pattern 1: Extract Super Class

Extract Super Class is creating a common super class of some classes, and moves the common features to the super class. In this study, we used the following three conditions to get clone sets which can be refactored by *Extract Super Class*.

[1]In Section 3.4, we evaluated whether refactorings suggested by Aries can be performed. Here, we evaluate whether refactorings suggested by Aries help future maintenance.

(ESC1) The unit of code clone is class,

(ESC2) All code clones (classes) do not use any *field-members-of-super-class*, and

(ESC3) All code clones (classes) have no common super class.

Obviously, since *Extract Super Class*'s target is class, the condition (ESC1) is required. It is difficult to apply this pattern to classes depending on the class hierarchy since applying this pattern changes the class hierarchy, so the condition (ESC2) is required. (ESC2) is represented as $(NRV(S) = 0) \cap (NSV(S) = 0)$. Also, this pattern creates a common super class and so we have to filter out code clones (classes) that already have a common super class using the condition (ESC3). (ESC3) is represented as $DCH(S) = \infty$.

If the code clones (classes) are completely identical, it would be appropriate to delete all code clones (classes) except one without applying *Extract Super Class* pattern.

Pattern 2: Move Method

Move Method is moving a method to another class. Originally, the location might be the best where the method had been defined. But, repeated modification and extending the features sometimes change the best location of the method. Then, we use *Move Method*. In this study, we used the following two conditions to get clone sets that can be moved to utility classes.

(MM1) The unit of code clone is method, and

(MM2) All code clones (methods) don't use any *field-members-of-its-class*.

Since obviously *Move Method*'s target is method, the condition (MM1) is used. Also, to get methods that did not implement some features of the class (did not use any fields of the class), we use the condition (MM2). (MM2) is represented as $(NRV(S) = 0) \cap (NSV(S) = 0)$.

Pattern 3: Extract Method

Extract Method is turning a code fragment into a method whose name explains the purpose of it. Originally, *Extract Method* is applied to a part of a too long method or a part of complicated function in order to improve the readability, understandability, and maintainability. In this study, we used the following conditions to get clone sets that can be refactored by *Extract Method* pattern.

(EM1) The unit of code clone is statement,

(EM2) There is only one or no assignment for externally defined variables,

(EM3) All code clones (statements) are in the same class, and

(EM4) There are three or more code clones (statements) in the same clone set.

Since *Extract Method* is applied to a part of a method, the condition (EM1) is required. If some values are assigned to some externally defined variables, it is necessary to make them arguments of the new extracted method, and to return them to its caller places to reflect them. It is necessary to contrive like making a new data class if two or more values are assigned to. If there is only one assignment, we just have to add a return-statement to the extracted method. So, the condition (EM2) is required. (EM2) is represented as $NSV(S) \leq 1$. And, if all code clones (statements) are in the same class, it is easy to merge them. So, (EM3) is required and is represented as $DCH(S) = 0$. At last, we filtered clone set consisting of only two clones. Refactorings of such clone sets may not be effective, because the size of statement is much smaller than one of *declaration* and *function*,

3.5.2 Evaluation Criteria

Here, we describe evaluation criteria in the case study. The criteria consist of *(A) State of Clone*, *(B) Effectiveness of Refactoring*, *(C) Cost of Refactoring*, and *(D) Comprehensive Evaluation*.

(A) State of Clone

We evaluated *(A) State of Clone* from four viewpoints. For each point, the maintainer of the system determined whether each clone set *(a) has a bad influence*, or *(b) has a little influence on the system*.

(A1) *Size of Software*
(A2) *Design of Software*
(A3) *Cohesion of Class*
(A4) *Coupling of Classes*

The first point is *(A1) Size of Software*. Here, 'size' means the LOC of class or method. The second point is *(A2) Design of Software*. Here, 'design' means the class hierarchy or Encapsulation. The third point is *(A3) Cohesion of Class*. Here, 'cohesion' means whether each class has responsibility of a function or not. If a class implements two or more functions, its 'cohesion' should be considered as low (bad). The last point is *(A4) Coupling among Classes*. Here, 'coupling' means that a class uses method and fields of other classes. If some methods and fields are not defined in the proper class, 'coupling' should be considered as high (bad).

(B) Effect of Refactoring

We evaluated *(B) Effect of Refactoring* from the following six viewpoints. For each point, the maintainer evaluated refactorings for each clone set as it *(a) improves the point*, *(b) prevents future problems*, *(c) has no impact*, or *(d) has a bad influence*.

(B1) *Size of Software*
(B2) *Design of Software*
(B3) *Cohesion of Class*
(B4) *Coupling of Classes*
(B5) *Readability of Source Code*
(B6) *Reusability of Source Code*

(B1) $\sim$ *(B4)* are the same as ones described in *(A) State of Clone*. *(B5) Readability of Source Code* is whether the refactoring improves readability of the source code. If the refactoring improves the readability, the maintainer need less time to understand the source code when he has to modify it. *(B6) Reusability of Source Code* is whether the refactoring makes it easier to reuse the source code. If the reusability is improved, the refactoring reduces the future cost of the same or different software development/maintenance.

(C) Cost of Refactoring

We evaluated *(C) Cost of Refactoring* from the following two viewpoints. For each point, the maintainer evaluated whether each task *(a) can end immediately*, *(b) is a little costly*, or *(c) is very complicated*.

(C1) *Modification of Source Code*
(C2) *Regression Test*

(C1) is the cost of modifying source code. If the refactoring is big, the maintainer has to modify different parts of software. *(C2)* is the cost of performing regression tests. As described in Section 1.3, refactoring must not changes the external behavior of software. After modifying source code, regression test is required to confirm the behavior. If tests of the modified parts use some test framework like JUnit [32], the regression tests probably be able to be performed just by inputting some command or clicking some buttons of GUI.

(D) Comprehensive Evaluation

Taking into account all things of refactorings, the maintainer totally judged the effectiveness of refactoring each clone set on the followings: it *(a) should be done* *immediately*, *(b) needs to be done in the future*, *(c) doesn't need to be done*, or *(d) must not be done*.

(D1) *Refactoring*

3.5.3 Hypothesis

Here, we define our hypotheses in this case study. The units of refactorings are *declaration*, *function*, and *statement*. Since there is a whole wide spread in the size of those units, we considered that all aspects of refactorings are different dependent on the unit.

(A) State of Clone

We made a hypothesis that the bigger units of clones are, the worse effects they have for software maintenance. In other words, declaration-clones has the worst effect, and statement-clones has less bad effect.

(B) Effect of Refactoring

We made the same hypothesis as *(A) State of Clone*. The worse effect clones have the effective it is to remove them.

(C) Cost of Refactoring

We hypothesized that, the bigger units of clones are, the more costly their refactorings are. Because a big refactoring needs complicated modifications of the source code.

(D) Comprehensive Evaluation

We could not hypothesize which unit of refactorings is the most effective. Because we predicted that refactorings of big unit (declaration-clone) has big effects but need much cost. On the other hand, refactorings of small unit (statement-clone) has small effects but need a little cost. The trade-off between effects and cost is important to determine whether the refactorings should be done or not.

3.5.4 Results

We detected 66 clone sets comprising refactoring oriented code clones by using the extraction method. Then, we filtered the code clones by using the conditions (ESC1 $\sim$ 3), (MM1 $\sim$ 2), and (EM1 $\sim$ 4), As a result, 4, 5 and 12 clone sets satisfied the conditions of each refactoring pattern.

Table 3.2: Refactoring Evaluations of *Extract Super Class*

(a) State of Clones

	(A1)	(A2)	(A3)	(A4)
(a) have a bad influence	4	4	0	0
(b) have no impact	0	0	4	4

(b) Effect of Refactorings

	(B1)	(B2)	(B3)	(B4)	(B5)	(B6)
(a) improve	2	2	0	0	1	1
(b) prevent future problems	2	2	0	0	3	3
(c) have no impact	0	0	4	4	0	0
(d) have a bad influence	0	0	0	0	0	0

(c) Cost of Refactorings

	(C1)	(C2)
(a) can end immediately	1	1
(b) is a little costly	3	1
(c) is very complicated	0	2

(d) Comprehensive Evaluation

	(D1)
(a) must be done immediately	2
(b) need to be done in the future	2
(c) does't need to be done	0
(d) must not be done	0

Pattern 1: Extract Super Class

Table 3.2 shows the evaluation of *Extract Super Class* pattern. Table 3.2(a) says that all clone sets *have a bad influence* on *(A1) Size of Software* and *(A2) Design of Software*. But, no clone set *have a bad influence* on *(A3) Cohesion of Class* and *(A4) Coupling of Classes*.

Table 3.2(b) shows *Effect of Refactorings*. From this table, we found that refactorings of all clone sets *improve* or *prevent future problems* from the viewpoint of *(B1) Size of Software*, *(B2) Design of Software*, *(B5) Readability of Source Code* and *(B6) Reusability of Source Code*. The maintainer judged that there was no impact on (B3) and (B4).

Table 3.2(c) shows *Cost of Refactorings*. From the viewpoint of *(C1) Modifying Source Code*, all refactorings are judged *can end immediately* and *is a little costly*. The result is different from our hypothesis. From the viewpoint of *(C2) Regression Test*, the judgments are divisive. Refactoring a clone set which calculates something about 'date' was judged *can end immediately* in both *(C1) Modification of Source Code* and *(C2) Regression Tests*. The classes are included in different packages and in the same clone set. The difference was only their package names. This clone set can be easily removed by deleting all classes except only one class. Also, other three clone sets depend on the framework that the software uses. But, the maintainer commented that we should introduce some interfaces to avoid such clone sets in the future.

Table 3.2(d) shows *Comprehensive Evaluation*. The maintainer judged that refactorings of all clone sets as *must be done immediately* or *need to be done in the future*. From these evaluations, we can conclude that Aries can effectively specified clone sets that should be refactored.

Pattern 2: Move Method

Table 3.3 shows the evaluation results of *Move Method* pattern. From Table 3.3(a), we can see that all clone sets *have a bad influence* on *(A1) Size of Software* and *(A2) Design of Software*. But, no clone set *have a bad influence* on *(A4) Coupling of Classes*.

Table 3.3(b) shows *(B) Effect of Refactorings*. From this table, we can see that refactorings *improve* or *prevent future problems* from the viewpoint of *(B1) Size of Software*, *(B2) Design of Software*, *(B3) Cohesion of Class*, *(B5) Readability of Source Code*, and *Reusability of Source Code*. Improving cohesion means that location of cloned methods are not appropriate, and we were able to identify such ones by Aries. On the other hand, all refactoring *had no impact* on *(B4) Coupling of Classes*.

From Table 3.3(c), we can see that all refactorings *can end immediately* both *(C1) Modification of Source Code* and *(C2) Regression Test*. We consider that it is due to the strict condition **MM2** (All code clones (methods) don't use any *field-members-of-its-class*), and the simplicity of *Move Method* (just move a method to another class).

Table 3.3(d) shows *Comprehensive Evaluation*. The maintainer judged that refactorings of all clone sets *must be done immediately* or *need to be done in the future*. From these evaluations, we can say that Aries can effectively identify the method candidates of refactorings.

Table 3.3: Refactoring Evaluations of *Move Method*

(a) State of Clones

	(A1)	(A2)	(A3)	(A4)
(a) have a bad influence	5	5	4	0
(b) have no impact	0	0	1	5

(b) Effect of Refactorings

	(B1)	(B2)	(B3)	(B4)	(B5)	(B6)
(a) improve	5	5	5	0	4	4
(b) prevent future problems	0	0	0	0	1	1
(c) have no impact	0	0	0	5	0	0
(d) have a bad influence	0	0	0	0	0	0

(c) Cost of Refactorings

	(C1)	(C2)
(a) can end immediately	5	5
(b) is a little costly	0	0
(c) is very complicated	0	0

(d) Comprehensive Evaluation

	(D1)
(a) must be done immediately	4
(b) need to be done in the future	1
(c) does't need to be done	0
(d) must not be done	0

Pattern 3: Extract Method

Table 3.4 shows the refactoring evaluations of *Extract Method* pattern. Table 3.4(a) says that most of clone sets *have no impact* on software quality.

Table 3.4(b) shows *Effect of Refactorings*. From the viewpoint of *(B1) Size of Software*, *(B2) Design of Software*, *(B5) Readability of Source Code*, and *(B6) Reusability of Source Code*, some refactorings *improved* the quality but others yield opposite effect. Aries could not effectively identify clone sets that *Extract Method* should be applied. Also, all refactorings *have no impact* on other properties.

From Table 3.4(c), wee can see that some refactorings need high cost. The

Table 3.4: Refactoring Evaluations of *Extract Method*

(a) State of Clones

	(A1)	(A2)	(A3)	(A4)
(a) have a bad influence	2	2	0	0
(b) have no impact	10	10	12	12

(b) Effect of Refactorings

	(B1)	(B2)	(B3)	(B4)	(B5)	(B6)
(a) improve	0	0	0	0	0	2
(b) prevent future problems	50	3	0	0	0	2
(c) have no impact	4	3	12	12	3	4
(d) have a bad influence	3	6	0	0	5	4

(c) Cost of Refactorings

	(C1)	(C2)
(a) can end immediately	3	2
(b) is a little costly	5	6
(c) is very complicated	4	4

(d) Comprehensive Evaluation

	(D1)
(a) must be done immediately	0
(b) need to be done in the future	3
(c) does't need to be done	3
(d) must not be done	6

maintainer commented that, it is troublesome to extract a part of existing methods as a new method although Aries provided information where and how we can refactor them.

Table 3.4(d) shows *Comprehensive Evaluation*. Half of clone sets were judged that their refactorings must not to be done. Most of such clone sets depend on the application framework, and they do not have bad impact on software quality. Also, refactorings of ‘statement-clones’ are a little effective, yet need much cost. That is a big factor of bad comprehensive evaluation.

3.5.5 Discussion

From the above evaluations, with respect to 'declaration-clones' and 'function-clones', all clone sets satisfying filtering conditions had bad impact on *(A1) Size of Software* and *(A2) Design of Software*, and the maintainer judged that refactorings for them improve software qualities. Especially, refactorings of all clone sets satisfying *Move Method* conditions are regarded as improving *(B2) Design of Software* and *(B3) Cohesion of Class*, and don't need much cost.

On the other hand, with respect to the 'statement-clones', most of clone sets satisfying *Extract Method* conditions have no impact of software quality. There are some clone sets whose refactoring might have a bad influence. One of the reasons the result is that, such clone sets are very small elements of software (they are *statement*s.), and so have a little impact on software quality. Moreover, most of them depend on the application framework and so it is not appropriate to simply remove them. Also, the costs of 'statement-clones' are higher than 'function-clones' and 'declaration-clones'. The maintainer said that it is troublesome to extract manually a part of existing methods as a new method even if we get the information where and how we can refactor them.

In this case study, we applied Aries to just a software system in a specific context. So, the results might be significant only to the context. However, the followings would be generalized to software developed in other context.

1. In case that the target system uses any application frameworks, it is not appropriate to remove code clones depending on the frameworks. Such code clones are prone to be stereotyped, not ad-hoc implementations. Generally, stereotyped code is very stable and not to be refactored.

2. Refactorings of 'declaration-clones' and 'function-clones' are more effective than ones of 'statement-clones'. Most refactorings of 'declaration-clones' and 'function-clones' need only simple operations like just moving or just deleting. But, refactorings of 'statement-clones' require complicated operations like renaming variables or adding parameter. Doing such operations manually is prone to be troublesome and costly.

In this case study, a maintainer of the target system subjectively judged the effectiveness of refactorings. To suggest the effectiveness of refactorings automatically, it is essential to characterize code clones quantitatively and show the effect of the refactoring. For example, the coupling metrics among methods proposed by Kataoka, et al. [35] would be used. We measure the metrics both before and after refactorings, and compare them. If the values are quite different, we could say that the refactorings have greatly changed the quality. Also, if we would use the history information of the development, the method suggested by Kim et al. [36, 37] may be useful. If we can identify clone sets whose fragments simultaneously and repeatedly, removing them maybe reduce the maintenance cost of the future. By using this method, refactorings of small clones like *statement* might be also effective.

3.6 Addaitive Descriptions of Aries

3.6.1 Other Refactoring Conditions in Aries

In Sections 3.2.3 and 3.5.1, we have given some conditions for *Extract Method*, *Extract SuperClass*, *Move Method* and *Pull Up Method*. But, of course, there are many other conditions used to filter clone sets. For example, in performing *Extract Method*, it is considered that users use not only $NAV(S)$ but also $NRV(S)$. Using the conditions, he/she can get clone sets which can be extracted with less than a certain number of parameters. It maybe be useful to use $POP(S)$, which is described in Section 2.2.3, to filter clone sets consisting of 10 or more code fragments. If some bugs are found in clone sets having many code fragments, modifying each of them without overlooking is complicated task. Refactoring of such clone sets should be practical. That is, users can filter clone sets with any conditions of all metrics.

3.6.2 Concentrating Target Classes to be Refactored in Aries

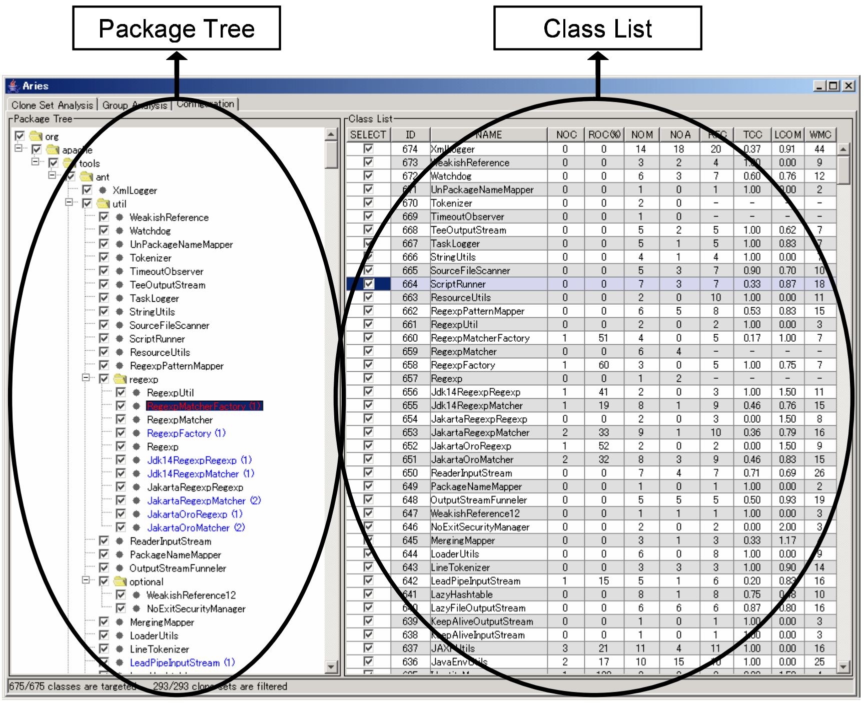
In Section 3.2.1, we explained how to identify the refactoring-oriented code clones. But, if users have already identified some classes which should be refactored, Providing only code clones in such classes is very effective. Aries has a function to do that. Figure 3.12 is a snapshot of the window that users select classes.

The left side is Package Tree. If users have already decided which classes he/she refactors, Package Tree is useful to select classes. Package Tree represents classes on the package hierarchy. Each element of the tree has a checkbox. Only code clones in checked classes are shown in Main Window, whose snapshot is Figure 3.2.

The right side is Class List. Class List represents classes with several metrics. The following are brief explanations of each metrics.

NOC(C) : $NOC(C)$ is the number of code clones included in a given class C.

ROC(C) : $ROC(C)$ is the ratio of duplication of class C.



```
Package Tree
☑ org
  ☑ apache
    ☑ tools
      ☑ ant
        ☑ ● XmlLogger
        ☑ util
          ☑ ● WeakishReference
          ☑ ● Watchdog
          ☑ ● UnPackageNameMapper
          ☑ ● Tokenizer
          ☑ ● TimeoutObserver
          ☑ ● TeeOutputStream
          ☑ ● TaskLogger
          ☑ ● StringUtils
          ☑ ● SourceFileScanner
          ☑ ● ScriptRunner
          ☑ ● ResourceUtils
          ☑ ● RegexpPatternMapper
          ☑ regexp
            ☑ ● RegexpUtil
            ☑ ● RegexpMatcherFactory (1)
            ☑ ● RegexpMatcher
            ☑ ● RegexpFactory (1)
            ☑ ● Regexp
            ☑ ● Jdk14RegexpRegexp (1)
            ☑ ● Jdk14RegexpMatcher (1)
            ☑ ● JakartaRegexpRegexp
            ☑ ● JakartaRegexpMatcher (2)
            ☑ ● JakartaOroRegexp (1)
            ☑ ● JakartaOroMatcher (2)
          ☑ ● ReaderInputStream
          ☑ ● PackageNameMapper
          ☑ ● OutputStreamFunneler
          ☑ optional
            ☑ ● WeakishReference12
            ☑ ● NoExitSecurityManager
          ☑ ● MergingMapper
          ☑ ● LoaderUtils
          ☑ ● LineTokenizer
          ☑ ● LeadPipeInputStream (1)
675/675 classes are targeted    293/293 clone sets are filtered
```

Class List

SELECT	ID	NAME	NOC	ROC(%)	NOM	NOA	RFC	TCC	LCOM	WMC
✓	674	XmlLogger	0	0	14	18	20	0.37	0.91	44
✓	673	WeakishReference	0	0	3	2	4	1.00	0.00	9
✓	672	Watchdog	0	0	6	3	7	0.60	0.76	12
✓	671	UnPackageNameMapper	0	0	1	0	1	1.00	0.00	2
✓	670	Tokenizer	0	0	2	0	-	-	-	-
✓	669	TimeoutObserver	0	0	1	0	-	-	-	-
✓	668	TeeOutputStream	0	0	5	2	5	1.00	0.62	7
✓	667	TaskLogger	0	0	5	1	5	1.00	0.83	7
✓	666	StringUtils	0	0	4	1	4	1.00	0.00	[illegible]
✓	665	SourceFileScanner	0	0	5	3	7	0.90	0.70	10
✓	664	ScriptRunner	0	0	7	3	7	0.33	0.87	18
✓	663	ResourceUtils	0	0	2	0	10	1.00	0.00	11
✓	662	RegexpPatternMapper	0	0	6	5	8	0.53	0.83	15
✓	661	RegexpUtil	0	0	2	0	2	1.00	0.00	3
✓	660	RegexpMatcherFactory	1	51	4	0	5	0.17	1.00	7
✓	659	RegexpMatcher	0	0	6	4	-	-	-	-
✓	658	RegexpFactory	1	60	3	0	5	1.00	0.75	7
✓	657	Regexp	0	0	1	2	-	-	-	-
✓	656	Jdk14RegexpRegexp	1	41	2	0	3	1.00	1.50	11
✓	655	Jdk14RegexpMatcher	1	19	8	1	9	0.46	0.76	15
✓	654	JakartaRegexpRegexp	0	0	2	0	3	0.00	1.50	8
✓	653	JakartaRegexpMatcher	2	33	9	1	10	0.36	0.79	16
✓	652	JakartaOroRegexp	1	52	2	0	2	0.00	1.50	9
✓	651	JakartaOroMatcher	2	32	8	3	9	0.46	0.83	15
✓	650	ReaderInputStream	0	0	7	4	7	0.71	0.69	26
✓	649	PackageNameMapper	0	0	1	0	1	1.00	0.00	2
✓	648	OutputStreamFunneler	0	0	5	5	5	0.50	0.93	19
✓	647	WeakishReference12	0	0	1	1	1	1.00	0.00	3
✓	646	NoExitSecurityManager	0	0	2	0	2	0.00	2.00	3
✓	645	MergingMapper	0	0	3	1	3	0.33	1.17	[illegible]
✓	644	LoaderUtils	0	0	6	0	8	1.00	0.00	9
✓	643	LineTokenizer	0	0	3	3	3	1.00	0.90	14
✓	642	LeadPipeInputStream	1	15	5	1	6	0.20	0.83	16
✓	641	LazyHashtable	0	0	8	1	8	0.75	0.48	10
✓	640	LazyFileOutputStream	0	0	6	6	6	0.87	0.80	16
✓	639	KeepAliveOutputStream	0	0	1	0	1	1.00	0.00	3
✓	638	KeepAliveInputStream	0	0	1	0	1	1.00	0.00	3
✓	637	JAXPUtils	3	21	11	4	11	1.00	0.00	16
✓	636	JavaEnvUtils	2	17	10	15	10	1.00	0.00	25

Figure 3.12: A Snapshot of Class Viewer

NOM(C) : $NOM(C)$ is the number of methods defined in class C. Accessors are included, but inherited methods are not included.

NOA(C) : $NOA(C)$ is the number of attributes defined in class C. Inherited attributes are not included.

TCC(C) ; $TCC(C)$ [12] means a cohesion of class C. Here, we assume that $NDC(C)$ is the number of method pairs which share some attributes, and $NP(C)$ is the number of all method pairs. Then,

$$TCC(C) = \frac{NDC(C)}{NP(C)}$$

RFC(C) : $RFC(C)$ [16] means the cardinality of the response set for class C, which is a set of methods that can be potentially executed in response to a message received by an object of that class. Mathematically it can be defined using elements of set theory, as : $RS = M \bigcup_{i=1}^{n} R_i$ where R_i is the set of methods called by method i and M is the set of all methods in the class, and n is the number of methods defined in C. Then, mathematically $RFC(C)$ = $|RS|$.

LCOM(C) ; $LCOM(C)$ [23] means a lack of cohesion. Consider the set M of m-th methods M_1, $\cdots$, M_m, and the set A of a-th data attributes A_1, $\cdots$, A_a accessed by M. Let $\mu(A_k)$ be the number of methods that access data attribute A_k where $1 \leq k \leq a$. Then,

$$LCOM(C) = \frac{(\frac{1}{a}\sum_{j=1}^{a} \mu(A_j)) - m}{1 - m}$$

Class List has a sort function, so that users can sort classes in ascending or descending order of arbitrary metrics. By using Class List, users can see which classes have inappropriate metrics values, and can select them to refactor. Each element of the list has a checkbox. Only code clones in checked classes are shown in Main Window, whose snapshot is Figure 3.2.

3.7 Related Works

Jarzabek et al. suggested a clone unification method [27]. The method unifies a set of cloned code fragments (or patterns of them; they call them *design-level similarity patterns*) in the source code as a meta-component. The method was implemented as a tool, XVCL (XML-based Variant Configuration Language). XVCL is language-independent in that it can be applied on top of any programming language. XVCL provides a generic solution for the problem that language-level unification methods fail to provide.

Kataoka et al. suggested a method measuring the effectiveness of refactorings [35]. They use coupling among methods as the indicator of refactorings effects. The coupling is calculated from three attributes, *return-value*, *arguments*, and *shared-variables*. They compare the values of before and after refactorings, and the degree of difference represents the effectiveness of the refactoring. We consider that their method can be applied to refactorings of code clones because we use existing refactoring patterns to merge code clones.

Rysselberghe et al. argued which of the following detection techniques are appropriate from a refactoring perspective: 1. string-based detection techniques. 2. token-based detection techniques. 3. metric fingerprint techniques [46]. The metric fingerprint technique was judged the most appropriate one from the viewpoint of the unit of code clones. The metric fingerprints inherently identify duplicated methods or scope blocks which can readily be refactored using refactoring like *Extract Method*, *Pull Up Method* and *Extract Class*. String-based detection technique is the best confidence since exact matches are targeted. Token-based detection techniques also have a good confidence if it uses a strict one-to-one parameterization. Metric fingerprint technique decreases accuracy. From all techniques studied, the metric fingerprint technique returns the most false positive. In our method, code clones are detected using a token-based detection technique, but blocks in them are extracted based on the programming language structure. Moreover our method has a big scalability, and so we believe that our method is adequately appropriate for refactorings.

Baxter et al. developed a tool CloneDR [11]. The tool presents templates which show how clone sets can be merged, after detecting them. Users can modify code clones by using the templates, but the tool doesn't show where the merged code should be located. Our tool Aries presents the location by using the metric DCH. If the DCH is 0, the modified code should be located in the original class. If the DCH is 2, the modified code should be locate in the direct parent of the original class. This information allows users to refactor code clones without detracting design intentions or unnecessarily increasing the complexity.

3.8 Summary

We have proposed a refactoring support method against code clones. The proposed method consists of extraction and provision steps. In the extraction step, code clones which are suitable for refactoring are extracted from the source code. In the provision step, code clones are quantitatively characterized by using coupling and distance metrics. The removal ways, which are existing refactoring patterns, of characterized code clones is decided automatically. A tool, Aries was implemented based on the proposal. Moreover we applied the tool to open source and commercial software to evaluate applicability and usefulness of refactorings suggested by the tool.

In the case study to evaluate usefulness, a maintainer of the system evaluated the effectiveness of refactorings suggested by Aries. For *declaration* and *function*, all refactorings of code clones satisfying the conditions were judged very effective. But, for *statement*, most refactorings of clones were judged as not effective, because the cost of refactorings is very expensive in comparison with the effectiveness of ones, or most of code clones depend on the frameworks used by the software. One of the actions for such code clones is to add a function omitting specified code from candidates of refactorings to Aries. And, specifying code depending on the frameworks will increase the effectiveness of refactorings suggested by Aries. For supporting complicated modifications of ‘statement-clones’, re-implementing Aries as a plug-in of Eclipse [19] may be effective. If Aries is a plug-in of Eclipse, Aries can cooperate with other plug-ins and can support various

Chapter 4

Modification Support Method

4.1 Motivation

Code clone makes the modification process more complicated. For example, when we find a bug and modify a code fragment (code clone) including the bug, it is necessary to determine whether or not we have to modify other code clones in the same clone set. Furthermore, it is likely to overlook some of them. So, it is important to have a search feature for code clones of a specified code fragment.

In this Chapter, we examine a modification support method for code clones. The method supports modification of source files by showing code clones of the code fragment that users are going to modify as a debugging, an adaptation, or an enhancement of a software product. It can prevent users from overlooking code fragments of the source files in the modification process, and enable them to maintain the system effectively.

4.2 Approach

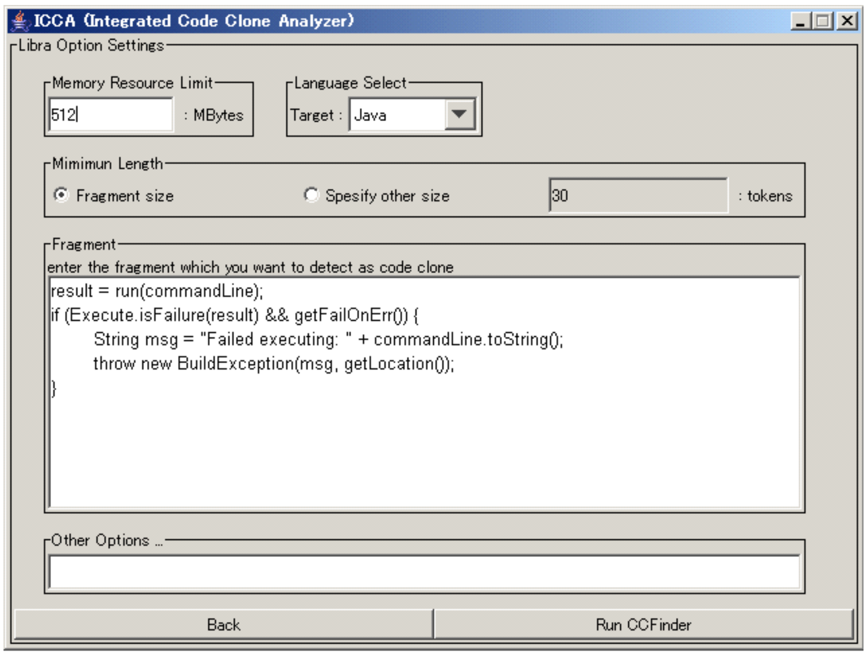
To detect only code clones across the input fragment and target files, we use the options of CCFinder. CCFinder has the following options.

-cg- : not detect code clones across groups.

-cf- : not detect code clones across files.

-cw- : not detect code clones within a file.

Here, a group means a file set. Users can freely construct groups before CCFinder's detection. Usually, users make a group from files which are in the same directory or


ICCA (Integrated Code Clone Analyzer)

Libra Option Settings

Memory Resource Limit: 512 : MBytes

Language Select: Target : Java

Mimimun Length: [✓ Fragment size] Spesify other size 30 : tokens

Fragment

enter the fragment which you want to detect as code clone

```
result = run(commandLine);
if (Execute.isFailure(result) && getFailOnErr()) {
	String msg = "Failed executing: " + commandLine.toString();
	throw new BuildException(msg, getLocation());
}
```

Other Options ...

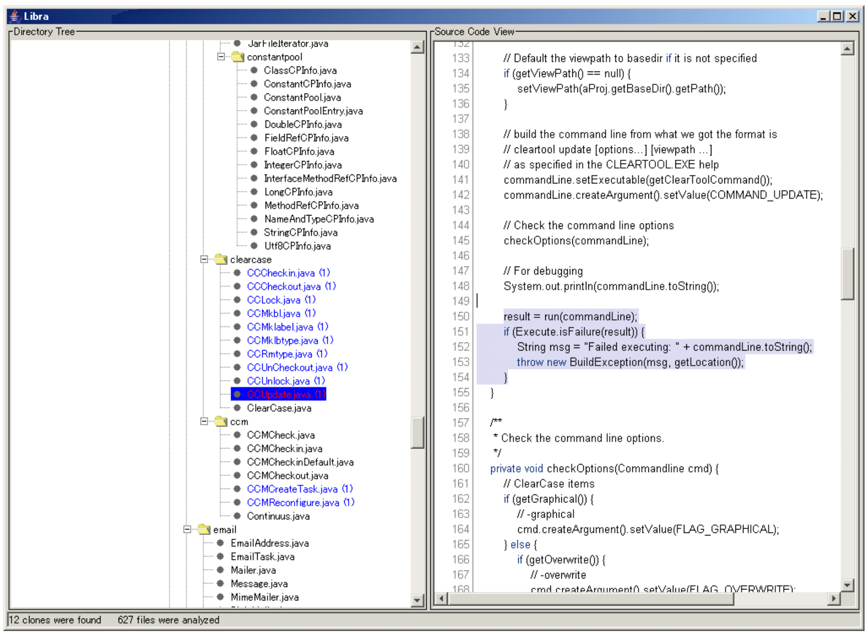
Figure 4.1: A SnapShot of Input Fragment

module. Conducting groups enables them to measure the similarities of directories or modules.

With the default settings, CCFinder detects all code clones (across groups, across files, within a file). But using above options, users can get only code clones across groups. In the proposed method, we assign the input fragment to group 1, and target files to group 2. After that, CCFinder runs with -cf- and -cw- options. Also, the proposed method count the token number of the input fragment, and set it as minimum token length of code clones which CCFinder detects. This setting prevents CCFinder from detecting redundant code clones.

4.3 Debug Support Tool: Libra

We implemented a tool, **Libra**, based on the above modification support method. At first, users input a code fragment to be modified and specify target files. Figure 4.1 demonstrates the window that users input a code fragment. Next, Libra executes CCFinder internally with the options previously. Figure 4.2 demonstrates the window of the detection result. In the left side, target files are listed as the di-



```
JarFileIterator.java
constantpool
    ClassCPInfo.java
    ConstantCPInfo.java
    ConstantPool.java
    ConstantPoolEntry.java
    DoubleCPInfo.java
    FieldRefCPInfo.java
    FloatCPInfo.java
    IntegerCPInfo.java
    InterfaceMethodRefCPInfo.java
    LongCPInfo.java
    MethodRefCPInfo.java
    NameAndTypeCPInfo.java
    StringCPInfo.java
    Utf8CPInfo.java
clearcase
    CCCheckin.java (1)
    CCCheckout.java (1)
    CCLock.java (1)
    CCMkbl.java (1)
    CCMklabel.java (1)
    CCMklbtype.java (1)
    CCRmtype.java (1)
    CCUnCheckout.java (1)
    CCUnlock.java (1)
    CCUpdate.java (1)
    ClearCase.java
ccm
    CCMCheck.java
    CCMCheckin.java
    CCMCheckinDefault.java
    CCMCheckout.java
    CCMCreateTask.java (1)
    CCMReconfigure.java (1)
    Continuus.java
email
    EmailAddress.java
    EmailTask.java
    Mailer.java
    Message.java
    MimeMailer.java
```

```
        // Default the viewpath to basedir if it is not specified
        if (getViewPath() == null) {
            setViewPath(aProj.getBaseDir().getPath());
        }

        // build the command line from what we got the format is
        // cleartool update [options...] [viewpath ...]
        // as specified in the CLEARTOOL.EXE help
        commandLine.setExecutable(getClearToolCommand());
        commandLine.createArgument().setValue(COMMAND_UPDATE);

        // Check the command line options
        checkOptions(commandLine);

        // For debugging
        System.out.println(commandLine.toString());

        result = run(commandLine);
        if (Execute.isFailure(result)) {
            String msg = "Failed executing: " + commandLine.toString();
            throw new BuildException(msg, getLocation());
        }
    }

    /**
     * Check the command line options.
     */
    private void checkOptions(Commandline cmd) {
        // ClearCase items
        if (getGraphical()) {
            // -graphical
            cmd.createArgument().setValue(FLAG_GRAPHICAL);
        } else {
            if (getOverwrite()) {
                // -overwrite
                cmd.createArgument().setValue(FLAG_OVERWRITE);
```

12 clones were found   627 files were analyzed

Figure 4.2: A SnapShot of Detection Result

Table 4.1: Attributes of target software

Name	Version		Total LOC	Number of Files	Language
	pre	post			
Canna	3.6	3.6p1	100,000	132	C
Ant	1.6.0	1.6.1	180,000	627	Java

rectory tree of the target system. Each file containing code clones of the input code fragment is shown in the highlighted color with a number of code clone contained in it. When users select a file in this view, the right side displays the source code of it.

4.4 Evaluation

We have applied Libra to a imaginary debug process as a case study. We chose Ant [2](version 1.6.0) and Canna [15](version 3.6) as our target. Table 4.1 illustrates

```
    ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );

#   buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
#   buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);

    ir_debug( Dmsg(10, "req->context = %d\n", Recuest.type3.context) );
```

(a) Before modification (version 3.6)

```
    ir_debug( Dmsg(10, "ProcWideReq3 start!!\n") );

+   if (Request.type3.datalen != SIZEOFSHORT * 2)
+   return( -1 );
+
    buf += HEADER_SIZE;  Request.type3.context = S2TOS(buf);
    buf += SIZEOFINT;    Request.type3.buflen = S2TOS(buf);

    ir_debug( Dmsg(10, "req->context = %d\n", Recuest.type3.context) );
```

(b) After modification (version 3.6p1)

Figure 4.3: An example of Canna's modifications

some attributes of target software. Because many versions of them are published, and so it enables us to conduct Libra's evaluations. Since Libra is a tool to search the similar code fragments that are created by 'copy and paste' and need to modify simultaneously in maintenance process, we have evaluated whether Libra can find out such code fragments.

We compared the Libra's search results to the grep's ones, which is a search tool of UNIX. In this case study, the length of a code fragment by users was set as the minimum token length of code clone.

4.4.1 Canna

Canna 3.6 includes 92 .c files and 40 .h files, and the size is about 100,000 LOC. We used an actual modification between version 3.6 and version 3.6p1. In version 3.6p1, a buffer overflow checking is inserted before each process using buffer, which is 21 places. Figure 4.3(a) represents a code fragment before modified, and Figure 4.3(b) represents its modified version. The lines beginning with '+' were added in version 3.6p1.

We assumed that a maintainer had detected a code fragment of them, and input it into Libra. We investigated whether Libra can detect the 21 code fragments. Two lines beginning with '#' were input to Libra, and 17 code fragments were detected as identical or similar to it. All the detected code fragments were modified ones in version 3.6p1, but Libra couldn't detect other 4 modified code fragments.

As the input of grep, we choose a variable Request.type which is a part

```
    commandLine.setExecutable(getClearToolCommand());
    commandLine.createArgument().setValue(COMMAND_CHECKIN);

    checkOptions(commandLine);

#   result = run(commandLine);
#   if (Execute.isFailure(result)) {
#       String msg = "Failed executing: " + commandLine.toString();
#       throw new BuildException(msg, getLocation());
#   }
```

(a) Before modification (version 1.6.0)

```
    commandLine.setExecutable(getClearToolCommand());
    commandLine.createArgument().setValue(COMMAND_CHECKIN);

    checkOptions(commandLine);

+   if (!getFailOnErr()) {
+       getProject().log("Ignoring any errors that occur for: "
+       + getViewPathBasename(), Project.MSG_VERBOSE);
+   }

    result = run(commandLine);
!   if (Execute.isFailure(result) && getFailOnErr()) {
        String msg = "Failed executing: " + commandLine.toString();
        throw new BuildException(msg, getLocation());
    }
```

(b) After modification (version 1.6.1)

Figure 4.4: An example of Ant's modifications

of variable name used in the buffer process. As a result, grep detected 58 Request.types, and 20 modified code fragments were included[1].

4.4.2 Ant

Ant 1.6.0 includes 627 .java files, and the size is about 180,000 LOC. We used an actual modification between version 1.6.0 and version 1.6.1. This modification was additions of log instructions to 10 places. Figure 4.4(a) represents a code fragment before modified, and Figure 4.4(b) represents its modified version. The lines beginning with '+' were added and the one with '!' was changed.

We assumed that a maintainer had detected a code fragment of them, and input it into Libra. We investigated whether Libra can detect the 10 code fragments. Five lines beginning with '#' were input to Libra, and 12 code fragments were detected as identical or similar to it. All 10 modified code fragments were included in them,

[1]grep output 234 lines, and 134 lines of them are related with any of modified code fragments.

Table 4.2: Comparison between Libra and grep

Target	Libra			grep		
Name	recall	precision	run-time	recall	precision	run-time
Canna	81%	100%	8 secs.	95%	34%	less than 1 sec.
Ant	100%	83%	10 secs.	100%	40%	less than 1 sec.

and the other 2 code fragments were not modified in version 1.6.1. This modification is not bug fix but additions of logging instruments, so we can't judge whether the 2 code fragments should be modified or not. However, pointing out such code fragments by Libra maybe be helpful for the maintainer in some situations.

As the input of grep, we choose a variable Execute.Failure which is a part of the if-statement condition. As a result, grep detected 25 Execute.Failures, and all modified code fragments were included.

4.4.3 Discussion

Table 4.2 illustrates precisions, recalls and execution times of Libra and grep. While recalls between them are almost the same, precisions of Libra are much better than ones of grep. This means results of Libra include less extra information than ones of grep. In other words, the maintainer can avoid expending times to check code fragments which don't need to be modified.

We also compared the execution times between Libra and grep. The execution time of grep was less than a second. On the other hand, Libra took 10 seconds. Libra's execution time was more than 8(10) times of grep's one, but the 8(10) seconds is no problem considering the precision.

4.5 Related Works

Toomim et al. [49] suggested a modification support method. In their method, there is a database of code clone information in the backend of the editor that users use. If a fragment including any clone sets is modified, its code clones are also simultaneously modified. From our case study, though it is obvious that not all code clones could/should be merged, their method would be very useful for simple code clones.

4.6 Summary

In this Chapter, we proposed a modification support method using code clone information. We have implemented a modification support tool, Libra based on code clone analysis. Libra supports modifying code clones. Users can detect all code clones which should be modified without omission. That is, in debugging or adding new functions, only a specified code and its code clones are easily detected. We also applied the tool to open source software, and compared it to grep, which is a search tool of UNIX. Libra finds code fragments including different identifiers(ex. variable name, method name) as code clones, which reduces maintainer's chances of overlooking some places that must be modified. So, we can say that Libra is one of good searching tool as well as grep.

Chapter 5

Conclusions

5.1 Summary of Major Results

In this paper, we have proposed three methods to use code clone information for effective software development and maintenance.

First, we have proposed methods of visualizing and characterizing code clones for comprehension. The proposed method has the following functions that are required as comprehension support tool.

- The function to provide a bird's eye view of code clones all over the system.
- The function to guide users to code clones which have the features that he/she is interested in.
- The function to eliminate code clone information which is not required in users' task.

We implemented a tool, Gemini based on the methods, and applied it to open source software. By using Gemini, we could see various activities of code clones.

Second, we have proposed a refactoring support method for code clones. The method provides information which and how code clones can be refactored. Based on the proposed method, we implemented a tool, Aries, and applied it to open source and commercial software. Through the case studies, we can conclude that refactorings suggested by Aries are applicable and useful.

Third, we have proposed a modification support method using code clone information. The method prevents users from overlooking some of code fragments which should be modified. Also, we implemented a tool, Libra based on the method, and applied it to open source software. In the case studies, we compared the tool with grep, which is a search tool of UNIX. The result showed that the tool was a good search tool as well as grep[1].

We have made the above tools a code clone analysis environment, **ICCA** (Integrated Code Clone Analyzer) [24]. The environment is used by domestic/overseas organizations and individuals.

5.2 Directions of Future Research

From many experiences of code clone detection, we learned that sometimes gaps occur between the original code fragments and modified ones since the developers are usually modifying the copied-and-pasted code fragments. We call modified the code fragments including some gaps *Gapped code clone* [50]. By treating the gapped code clones as well, our methods become more effective.

As described in Section 2.2.1, the proposed filtering method only deals with language dependent code clones. We are going to improve the method to filter out application dependent code clones. Features of application dependent code clones are so strikingly different among software systems that such metric-based filtering as RNR is probably unpractical. We believe that pattern-based filtering works well. For example, users input code patterns that they are not interested in. After that, Gemini filters out all code clones whose contents correspond to the code patterns. Using such a filtering approach, we can filter out any kind of code clones dependent on the domain or the framework of a system.

We are going to perform more detail analyses for code clones to evaluate the effectiveness of refactorings. Currently, we filter code clones based on the judgment whether or not they *can* be merged. If we can judge whether the code clones *should* be merged or not, the supporting of the refactoring will become more effective. Further, since the current Aries subsystem can deal with only Java programs, extension is necessary so that it can be applied to other programming languages.

Categorization of code clones may be helpful to analyze them. As described in Section 2.2.1, code clone detection by tools prone to detect many code clones, and some approaches mining them are required. Categorizing code clones before users analyze them should prevent them from spending much time on code clones which are not related with their tasks.

[1]As shown in Table 4.2, the detection accuracy of the tool is higher than grep's one. but the detection speed of grep is much faster than the tool's one.

Bibliography

[1] I. T. P. Agency(IPA). http://www.ipa.go.jp/index-e.html.

[2] Ant. http://ant.apache.org/.

[3] L. Arthur. *Software Evolution: The Software Maintenance Challenge*. Wiley, 1988.

[4] B. S. Baker. A program for identifying duplicated code. In *Proc. the 24th Symposium of Computing Science and Statistics*, pages 49–57, Mar 1992.

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. of the 2nd Working Conference on Reverse Engineering*, pages 86–95, Jul 1995.

[6] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, Oct 1997.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proc. of the 6th IEEE International Symposium on Software Metrics*, pages 292–303, Nov 1999.

[8] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Partical redesign of java software system based on clone analysis. In *Proc. of the 6th IEEE International Working Conference on Reverse Engineering*, pages 326–336, Oct 1999.

[9] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. of the 7th IEEE International Working Conference on Reverse Engineering*, pages 98–107, Nov 2000.

[10] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–165, Sep 2005.

[11] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proc. International Conference on Software Maintenance 98*, pages 368–377, Mar 1998.

[12] J. Bieman and B. Kang. Cohesion and reuse in an object-oriented system. In *Proc. of the 1995 Symposium on Software reusability*, pages 259–262, Aug 1995.

[13] M. Bruntink, T. A. V. Deursen and, and R. V. Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Sep 2004.

[14] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proc. of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, Oct 2002.

[15] Canna. http://canna.sourceforge.jp/.

[16] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 25(5):476–493, Jun 1994.

[17] ClearCase. http://www-306.ibm.com/software/awdtools/clearcase/.

[18] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proc. of the International Conference on Software Maintenance 99*, pages 109–118, Aug 1999.

[19] Eclipse. http://www.eclipse.org/.

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

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[22] D. Gusfield. *Algorithm on Strings, Trees, and Sequences*. Campridge University Press, 1997.

[23] B. Henderson-Sellors. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[24] ICCA. http://sel.ist.osaka-u.ac.jp/icca/.

[25] IEEE. *Standard for Software Maintenance*. IEEE Standard 1219, 1998.

[26] ISO/IEC. *Software Engineering - Software Maintenance*. ISO/IEC 14764, 1999.

[27] S. Jarzabek, P. Basset, H. Zhang, and W. Zhang. Xvcl: Xml-based variant configuration language. In *Proc. of the 25th International Conference on Software Engineering*, pages 810–811, May 2003.

[28] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proc. of International Conference on Software Maintenance 94*, pages 120–126, Sep 1994.

[29] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *Proc. of the 1996 Conference of Centre for Advanced Studies on Collaborative Research*, pages 7–16, Nov 1996.

[30] C. Jones. *Programming Productivity*. McGraw-Hill, 1986.

[31] JPlag. http://www.ipd.uni-karlsruhe.de/jplag/.

[32] JUnit. http://www.junit.org/.

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

[34] C. Kapser and M. Godfrey. Improved tool support for the investication of duplication in software. In *Proc. of the 21st International Conference on Software Maintenance*, pages 305–314, Sep 2005.

[35] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. of the 18th IEEE International Conference on Software Maintenance*, pages 576–585, Oct 2002.

[36] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *Proc. of the 2nd International Workshop on Mining Software Repositories*, pages 17–21, May 2005.

[37] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 187–196, Sep 2005.

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

[39] B. Lague, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating fucntion clone detection in a development process. In *Proc. of Intenational Conference on Software Maintenance 97*, pages 314–321, Oct 1997.

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

[41] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proc. of the International Conference on Software Maintenance 96*, pages 244–253, Nov 1996.

[42] T. Mens and A. Deursen. Refactoring: Emerging trends and open problems. In *Proc. of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects*, Nov 2003.

[43] R. H. Page. http://www.refactoring.com/.

[44] A. S. D. Project. http://www.ipa.go.jp/english/sec/third.html.

[45] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *Proc. the 11th Working Conference on Reverse Engineering*, pages 100–109, Nov 2004.

[46] F. Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proc. of the 19th IEEE International Conference on Automated Software Engineering*, pages 336–339, Sep 2004.

[47] N. Schneidewind. The state of software maintenance. *IEEE Transactions on Software Engineering*, 13(3):303–310, Mar 1987.

[48] SourceOffSize. http://www.sourcegear.com/sos/.

[49] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 173–180, Sep 2004.

[50] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On detection of gapped code clones using gap locations. In *Proc. of the 9th Asia-Pacific Software Engineering Conference*, pages 327–336, Dec 2002.

[51] VisualSourceSafe. http://msdn.microsoft.com/vstudio/previous/ssafe/.

[52] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proc. of the 10th Working Conference on Reverse Engineering*, pages 285–294, Nov 2003.

[53] WebLogic. http://www.beasys.com/products/weblogic/.

[54] WebSphere. http://www-306.ibm.com/software/websphere/.

[55] S. W. L. Yip and T. Lam. A software maintenance survey. In *Proc. of the 1st Asia-Pacific Software Engineering Conference*, pages 70–79, Dec 1994.