

REPRESENTATION, ANALYSIS, AND REFACTORING TECHNIQUES TO
SUPPORT CODE CLONE MAINTENANCE

by

ROBERT AMBROSIUS TAIRAS

BARRETT BRYANT, COMMITTEE CHAIR

JEFF GRAY

NICHOLAS KRAFT

MARJAN MERNIK

BRIAN TOONE

CHENGCUI ZHANG

A DISSERTATION

Submitted to the graduate faculty of The University of Alabama at Birmingham,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

BIRMINGHAM, ALABAMA

2010

Copyright by
Robert Ambrosius Tairas
2010

REPRESENTATION, ANALYSIS, AND REFACTORING TECHNIQUES TO
SUPPORT CODE CLONE MAINTENANCE

ROBERT AMBROSIUS TAIRAS

COMPUTER AND INFORMATION SCIENCES

ABSTRACT

The existence of code cloning, which represents identical or near identical sections of code, has been documented to occur frequently in software systems. The main purpose of cloning is the reuse of a code fragment that performs some functionality by copying and pasting it to a new location in the source code. Code clones embody a unique form of coupling, because their similarity inadvertently connects their representative sections of code together. This inherent duplication requires attention to be given to all related duplicates, even if just one of these duplicates is being evaluated or changed. Failure to consider all related clones when maintaining cloned code can result in errors that are difficult to identify initially.

The introduction of clone detection tools in recent years has provided an automated means to discover clones in code. However, in order to maintain these clones, programmers must first understand the properties of the clones reported by such tools, which in many cases are comprised of clone groups (i.e., clones representing the same duplication) scattered throughout the collection of source files that in turn are contained in reports consisting of a large amount of data. The elimination of the duplication associated with these clones by modularizing the code through refactoring activities reduces the amount of code that needs to be maintained in the future. However, an observed disconnect can still be seen between the detection and analysis of the clones and the subsequent task of eliminating the duplication.

This dissertation focuses on supporting two aspects related to the maintenance of code clones: 1) clone comprehension through its representation and analysis, and 2) clone maintenance with a focus on the removal of the duplication associated with the clones. Pertaining to clone comprehension, the research described in this dissertation contributes several novel clone representations that include clone visualization and localized representation techniques. The research also contributes to the understanding of clones through clone analysis associated with the discovery of higher level relationships among clone groups and the observed relationships between clones reported by a clone detection tool and actual historic refactorings in software artifacts. Pertaining to the second aspect, the research contributes to a more unified process where the phases of clone maintenance with a focus on clone removal (i.e., detection, analysis, and refactoring) are streamlined together within the programmer's working environment. This new process removes tasks that previously required manual intervention.

DEDICATION

To Esther,

To Mami, Papi, and Diana,

To Katherine, Andrew, and Philip,

God is great, God is good.

ACKNOWLEDGEMENTS

My sincerest gratitude and appreciation goes to my family. To my wife Esther, your joy, patience, and encouragement have kept me going in both pleasant times and tough times. I am continuously grateful for the love that we share. To my parents, thank you for your love and non-stop support. Your tireless passion in your endeavors has provided an astounding example for me. To my sister Katherine, you have in many ways been my trailblazer. To my brother-in-law Andrew, I'm grateful that you are part of the family. Thank you both for providing a place devoid of programming and papers where I am able to recharge my batteries. To Diana, we have not been able to spend much time together, but I feel very blessed to have you as my sister. Your cheerfulness is very contagious. I wish you the very best in your future. To my extended family, thank you for always pulling for me and being there to help me as I go through the various stages of my life.

To my mentor and adviser, Dr. Jeff Gray, I'm truly indebted for your involvement in my life these past few years. Not only have you mentored me in research, in teaching, and in participation in the academic community, but I have realized and appreciate your sincere concern for my general well-being. Thank you for motivating me and always being available to help me through this process. To Dr. Barrett Bryant, I appreciate your more involved role on my committee as of late. Thank you for entrusting me with the responsibility and opportunity to teach in the department.

To my committee members, I am grateful for your willingness to take part in the advisement and evaluation of my dissertation research. Dr. Nicholas Kraft – I appreciate our discussions about my work and the field of code clones in general. Thank you also for your assistance in preparing for faculty-related job interviews. Dr. Marjan Mernik – thank you for teaching the informative Domain-Specific Languages class and also for our collaborative work on ontologies. Dr. Brian Toone – I appreciate your willingness to serve on my committee and informally representing the connection to my college alma mater. Dr. Chengcui Zhang – thank you for the time and effort you gave to serve on my committee.

I thank Dr. Frédéric Jouault for teaching the class on Model-Driven Engineering and engaging in conversations about research directions with me. I also acknowledge Dr. Alan Sprague for his feedback regarding the use of biological sequence algorithms in code clone detection.

I felt a bit intimidated when I became a member of the Softcom lab, because the group of students before me had all passed their Level II – Dissertation Research Proposal. However, Faizan Javed, Yuehua “Jane” Lin, Shih-Hsi “Alex” Liu, Suman Roychoudhury, Hui Wu, Xiaoqing “Carl” Wu, and Jing Zhang were all very friendly and helpful right from the start. Thank you all for accepting this rookie into the lab and providing guidance, advice, and a great atmosphere to work in. I continue to wish you the best in your current and future endeavors.

I am also privileged to be associated with the current group of Softcomers: Hyun Cho, Zekai Demirezen, Ferosh Jacob, Qichao Liu, Tomaž Lukman, José Eduardo Rivera, Upendra Sapkota, Dixon Shuttleworth, and Yu Sun. Your enthusiasm and fresh attitude

has provided a catalyst for me to complete my work at UAB. I enjoyed working alongside you and will always remember the jovial times in the lab. I wish all of you the very best and look forward to hearing where life will take you next.

To Kathy Baier, John Faulkner, Janet Tatum, and also the CIS IT staff, thank you for your continuous assistance during my studies at the department. I enjoyed the refreshing volleyball games that we played. To my colleagues at the department, thank you for enriching my experience at UAB.

Finally, I would like to acknowledge the financial support provided by the Department of Computer and Information Sciences at the beginning of my Ph.D. studies. In addition, I also acknowledge subsequent financial support of my studies from the National Science Foundation through a Computing Processes and Artifacts grant. This material is based upon work supported by the National Science Foundation under Grant No. 0702764.

TABLE OF CONTENTS

	<i>Page</i>
ABSTRACT.....	iii
DEDICATION.....	v
ACKNOWLEDGEMENTS.....	vi
LIST OF FIGURES.....	xiii
LIST OF LISTINGS.....	xvi
LIST OF TABLES.....	xviii
LIST OF ABBREVIATIONS.....	xix
CHAPTER	
1 INTRODUCTION.....	1
1.1 Code Clone Research.....	3
1.2 Challenges in Clone Comprehension and Maintenance.....	5
1.2.1 Clone Representation.....	5
1.2.2 Clone Analysis.....	6
1.2.3 Clone Refactoring.....	7
1.3 Research Objectives.....	8
1.3.1 Clone Representation.....	9
1.3.2 Clone Analysis.....	10
1.3.3 Clone Refactoring.....	10
1.4 Outline.....	11
2 BACKGROUND.....	13
2.1 Code Clones.....	13
2.1.1 Clone Detection.....	14
2.1.2 Parsing Clone Detection Tool Results.....	16

2.2	Model-Driven Engineering	18
2.2.1	MDE through AMMA	20
2.3	Information Retrieval	22
2.3.1	Latent Semantic Indexing	23
3	CLONE REPRESENTATION	25
3.1	Visualization of Clone Detection Results	26
3.1.1	Tool Integration and Extension	26
3.1.2	Details of CloViz Plug-in	27
3.1.3	Clone Detection Execution and Display of Detection Results	29
3.1.4	Visualiser View Enhancements	33
3.1.5	Results Representation	34
3.2	Localizing Clone Group Representation	37
3.2.1	Representing Clones in One Location	38
3.2.2	The CeDAR Plug-in and Clone Visualization	39
3.2.3	Detecting Clone Similarities and Differences	42
3.2.4	Similarity Levels	44
3.2.5	Localized Representation in CeDAR	48
3.2.6	Clone Properties Based on Visualizations	54
3.2.7	Evaluation	57
3.2.8	Discussion	60
3.3	CoCloRep: A DSL for Code Clone Representation	62
3.3.1	Overview of CoCloRep	63
3.3.2	Implementation in the AMMA Platform	67
3.3.3	Metamodel Specification	68
3.3.4	Model Transformations	70
3.3.5	Observations	71
3.4	Related Work	72
3.5	Summary	75
4	CLONE ANALYSIS	79
4.1	An Information Retrieval Process to Aid in the Analysis of Code Clones	80
4.1.1	An Information Retrieval Perspective on Code Clone Analysis	81
4.1.2	Corpus Description and Analysis Goals	83
4.1.3	Clone Group Clustering Process	84
4.1.4	Clone Group Clustering Analysis within the Windows NT Kernel	93
4.1.5	Discussion of Analysis Results and Clustering Process	107
4.1.6	Limitations	113
4.2	Sub-clone Refactoring	114

4.2.1	Observing Clone Refactoring	115
4.2.2	Observing Sub-clone Refactoring	116
4.2.3	Sub-clone Refactoring of Deckard Clones	121
4.2.4	Discussion	126
4.3	Related Work	127
4.4	Summary	131
5	CLONE REFACTORING	134
5.1	Clone Maintenance Process	135
5.2	Combining Clone Detection, Analysis, and Refactoring	138
5.2.1	Clone Detection Tool Results as Input	138
5.2.2	Analysis for Refactoring Opportunities	140
5.2.3	Clone Refactoring in an IDE	141
5.3	Clone Refactoring Extensions	144
5.3.1	Refactoring Clones in a Single Class	145
5.3.2	Refactoring of Clones in Multiple Classes	153
5.4	Evaluation	155
5.5	Related Work	159
5.6	Summary	160
6	FUTURE WORK	161
6.1	Continued Focus on Clone Maintenance	161
6.1.1	Increasing Refactoring Capabilities	161
6.1.2	Incorporating Visualizations in the Refactoring Task	163
6.1.3	Clone Analysis with MDE	163
6.1.4	Clone Analysis with IR	164
6.2	Broader Application of Work	165
6.2.1	Additional Clone Property Analysis	165
6.2.2	Information Retrieval and Model Analysis	166
7	CONCLUSION	168
7.1	Clone Representation	169
7.2	Clone Analysis	170
7.3	Clone Refactoring	171

LIST OF REFERENCES.....	173
A PHOENIX-BASED CLONE DETECTION USING SUFFIX TREES	185
A.1 Exact Matching Algorithm	186
A.1.1 The Original Suffix Tree	186
A.1.2 Suffix Tree Alteration	188
A.1.3 Potential False Positives.....	189
A.1.4 Sketch of Clone Detection Algorithm.....	191
A.2 Implementation Details.....	191
A.2.1 Microsoft Phoenix	191
A.2.2 Detecting Code Clones.....	193
A.3 Case Study Example	194

LIST OF FIGURES

<i>Figure</i>	<i>Page</i>
1.1 Code clone research.....	4
1.2 Research overview.....	8
2.1 Example of SimScan output	17
2.2 Example of CCFinder output.....	18
3.1 CloViz connections and processes.....	27
3.2 Wizard dialog.....	29
3.3 General information view	30
3.4 Detected clones list view	31
3.5 Visualization of detected clones view	32
3.6 Context menu options	34
3.7 Clone group representations	34
3.8 Ubiquitous clones	36
3.9 Sample code fragment display.....	39
3.10 Localized clone representation in CeDAR	41
3.11 Part of a suffix tree identifying clone similarities and differences	43
3.12 Statement matching levels	45
3.13 Mappings of parameterized differences.....	47
3.14 Pop-up of simple variable differences	51

3.15	Highlighted statement differences	52
3.16	Display of two sub-groups of parameterized clones.....	53
3.17	Summary of clone differences	55
3.18	Small clone class differences.....	56
3.19	Sub-group with higher similarity.....	57
3.20	Model transformation process applied to clone representation	68
3.21	Clone visualization inside source editor view in CeDAR	76
4.1	Hierarchy of clones, clone groups, and clusters	83
4.2	Overview of the clone group clustering process.....	85
4.3	Examples of eliminated clone group subsets.....	88
4.4	Differing location of array initialization.....	103
4.5	File changes based on <i>diff</i> information.....	115
5.1	Clone maintenance process support.....	135
5.2	Clone maintenance processes	142
5.3	Screenshot of CeDAR.....	143
5.4	Filtering of clones with parameterized methods.....	146
5.5	Outline of <code>ExtractMethodRefactoring</code> class in Eclipse	147
5.6	Mapping of parameterized elements.....	148
5.7	Including parameterized fields.....	149
5.8	Removing duplicate variables.....	151
5.9	Filtering clone selection.....	154
A.1	Suffix tree of <i>abcdabe\$</i>	187
A.2	Suffix tree of <i>abgf\$abgf#</i>	188

A.3	Abstract syntax tree nodes	189
A.4	Example functions	190
A.5	Clone detection plug-in for Phoenix.....	192

LIST OF LISTINGS

<i>Listing</i>	<i>Page</i>
3.1 Determining statement similarities	48
3.2 Determining matching statements	50
3.3 Clone sub-groups	54
3.4 Example of two clone instances.....	63
3.5 Clone instances in CoCloRep	64
3.6 A clone group in CoCloRep.....	65
3.7 Example output of <i>variables</i> command.....	66
3.8 A snippet of CoCloRep's abstract syntax	69
3.9 A snippet of CoCloRep's concrete syntax.....	69
3.10 A transformation rule snippet for the <i>expand</i> command	70
4.1 Sample CCFinder textual clone representation	86
4.2 Pseudo-code of setting paged events	96
4.3 Three different implementations of obtaining base address and region size.....	98
4.4 Example code for page initialization	100
4.5 Three examples of obtaining token information length.....	104
4.6 I/O Request Packet (IRP) allocation and initialization general code.....	104
4.7 Clone ranges of clone detection tools and associated refactoring	120
4.8 Incomplete block of refactored code in ArgoUML	124

5.1	Including parameterized fields in argument list	150
5.2	Removing duplicate variables in parameter list.....	152

LIST OF TABLES

<i>Table</i>	<i>Page</i>
1.1 Clone coverage percentage in programs	6
3.1 Clone types identified by CeDAR in programs	59
4.1 Clone group totals of top-five directories in the NT kernel.....	87
4.2 srcML representation of simple declaration	90
4.3 Term and document totals.....	91
4.4 Statistical information on the generated clusters	94
4.5 Clusters of registry functions and file totals	101
4.6 Additional clusters containing clones with slight variations	106
4.7 Clusters where all clone groups are from one directory	106
4.8 Extract Method-related refactorings in JBoss.....	116
4.9 Clone detection tool configuration settings	118
4.10 Coverage of Extract Method-related refactorings	119
4.11 Refactoring coverage and code properties.....	123
5.1 Clone detection tool results availability.....	138
5.2 Additional Extract Method refactorings by CeDAR	157
5.3 Parameterized differences in arguments list of extracted method	158

LIST OF ABBREVIATIONS

AJDT	AspectJ Development Tools
AMMA	Atlas Model Management Architecture
AMW	Atlas Model Weaver
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	Atlas Transformation Language
CeDAR	Clone Detection, Analysis, and Refactoring
CLOC	Cloned Lines of Code
CloViz	Clone Visualization
CoCloRep	Code Clone Representation
ConQAT	Continuous Quality Assessment Toolkit
CSeR	Code Segment Reuse
CVS	Concurrent Versions System
DDMM	Domain Definition Metamodel
DLL	Dynamic Link Library
DSL	Domain-Specific Language
ENBF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework

GPL	General-Purpose Language
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IR	Information Retrieval
IRP	I/O Request Packet
J2EE	Java 2Platform, Enterprise Edition
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDT	Java Development Tools
KM3	Kernel Meta-Meta Model
LOC	Lines of Code
LSI	Latent Semantic Indexing
MB	Megabyte
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT	Query View Transformations
RDz	Rational Development for System Z
SVD	Singular Value Decomposition
TCS	Textual Concrete Syntax
TS	Technical Space

UML	Unified Modeling Language
VE	Version Editor
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

The topic of *cloning* has reverberated through many fields of human endeavor. An event that garnered wide publicity occurred in 1996 when scientists at the Roslin Institute in Scotland successfully cloned the first mammal: a domestic sheep named Dolly that was cloned from an adult somatic cell [Wilmut *et al.*, 1997]. This achievement was widely considered a breakthrough in science as the scientists aimed to use transgenic technology to produce therapeutic human proteins in the milk of clones such as Dolly [Travis, 1997]. Earlier in the century, Henry Ford introduced an assembly line to mass produce, or *clone*, the Model T automobile with a purpose to create a “car for the great multitude” [Lacey, 1986].

The exercise of cloning is also evident in software engineering where a specific type of cloning called *code cloning* exists. Code clones can be defined as identical or almost identical sections of code that represent duplication in multiple locations among a collection of source files. Clones are generated when a section of code that performs some functionality is reused by copying and pasting it to a new location in the source code [Kim *et al.*, 2004]. Cloning can also be initiated by independent development activities that result in clones that perform the same functionality [Juergens *et al.*, 2010]. When clones are generated, their representative sections of code are coupled. The evaluation and changes performed on one clone may necessitate the same action to be

performed on the other clones. Failure to evaluate all related clones when updating cloned code can result in errors that can be difficult to identify in future maintenance activities [Juergens *et al.*, 2009].

The existence of clones in code becomes part of the software development process when such clones need to be maintained. Clone maintenance mirrors the same activities as the maintenance of software systems in general, which are divided into *corrective*, *adaptive*, and *perfective* maintenance [Lientz and Swanson, 1980]. Similar to software maintenance activities, clone maintenance also involves the updating of the clones to fix an error, enhance the associated functionality, or to improve their structure and/or performance. As such, the maintenance of clones can be considered to fall under the spectrum of software maintenance, a phase in software development that can encompass up to 90% of the total effort in software development [Erlikh, 2000]. However, clone maintenance can occur during any software lifecycle phase that involves coding as programmers continue to fix or refine the code that may contain clones. This follows the paradigm of programming as observed by Hunt and Thomas in that all programming is maintenance [Hunt and Thomas, 1999]. In addition, to perform maintenance on clones, they must be understood through the knowledge of their existence, what kind of code is being duplicated, and where the duplicates are located. The comprehension of clones can be considered part of the overall process of program comprehension. When associated with the activity of maintenance, the program comprehension process can consume 50% or more of the maintenance cost [Standish, 1984].

The need to understand and maintain clones has spawned a research field that is briefly outlined in Section 1.1. In Section 1.2, challenges within this research field related to the representation, analysis, and refactoring of clones are given. These challenges motivated the research objectives of this dissertation, which are given in Section 1.3. The research that will be described in this dissertation focuses on supporting two aspects related to the maintenance of code clones: 1) clone comprehension through its representation and analysis, and 2) clone maintenance with a focus on the removal of the duplication associated with the clones.

1.1 Code Clone Research

Figure 1.1 outlines the categories of research related to code clones. Code clone research was mainly initiated by efforts to automatically detect clones. Since the early 1990s, various techniques have been proposed to detect clones in a wide range of code sizes and languages. The detection process looks at various code representations (e.g., string [Ducasse *et al.*, 1999], token [Kamiya *et al.*, 2002], and tree [Kraft *et al.*, 2008]) and utilizes various techniques to find the duplication (e.g., suffix tree [Tairas and Gray, 2006], clustering [Jiang *et al.*, 2007a], and frequent itemsets [Wahler *et al.*, 2004]). These techniques have yielded tools whose automated nature frees the programmer from the need to manually find clones. These tools provide both the original coders and the programmers who are new to the system a means in which to obtain information regarding clones in the code. The majority of the tools provide their reports in textual format. However, efforts to visualize clones include visualizations such as scatter plots

[Kamiya *et al.*, 2002], polymetric views [Rieger *et al.*, 2004], and aspect browser-like views [Tairas *et al.*, 2006].

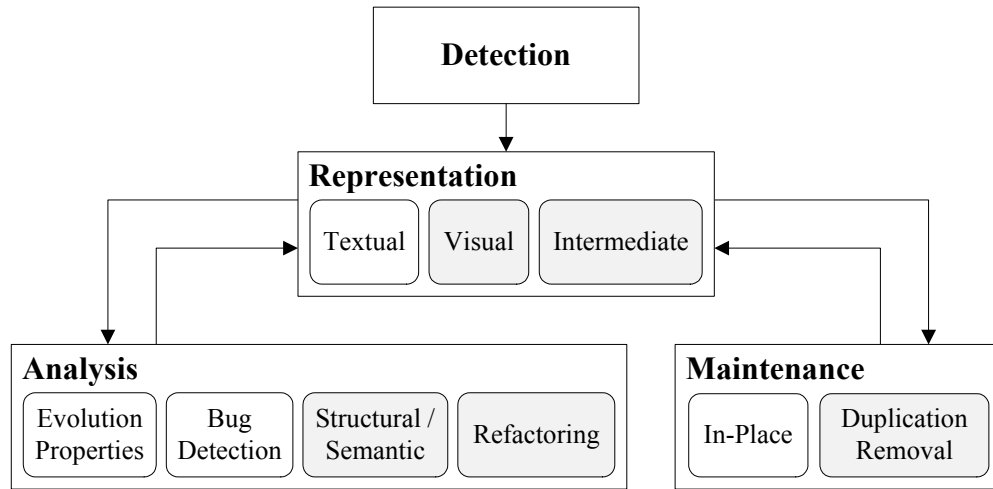


Figure 1.1 – Code clone research.

Research on the detection of clones has provided a stepping stone into subsequent research topics geared toward the further understanding, management, and maintenance of clones. Clone analysis is concerned with determining characteristics of clones based on studies of clones in various software artifacts. The overarching purpose of clone analysis is to provide a better understanding of the clones. Studies related to clone analysis can consist of evaluations of an artifact within a single snapshot of the source code, such as in a web application [Rajapakse and Jarzabek, 2007], in the detection of clone-related bugs [Jiang *et al.*, 2007b], and in an operating system kernel [Tairas and Gray, 2009a]. Other studies evaluate multiple snapshots based on the revision history of the code to determine the genealogical nature of clones [Kim *et al.*, 2005], to observe the

consistency of clones being maintained [Aversano *et al.*, 2007], and to identify actual refactorings of clones [Tairas and Gray, 2010b].

In an effort to assist programmers with the management of clones, further research has also focused on activities such as clone maintenance. This activity arises when a section of duplicated code needs to be updated. In-place clone maintenance [Duala-Ekoko and Robillard, 2007] provides mechanisms for the simultaneous updating of clones where they are located. A separate mechanism assists in removing the actual duplication to allow updates to be performed in one location [Fanta and Rajilich, 1999]. Other mechanisms keep track of copy-and-paste actions for future maintenance activities [Weckerle, 2008].

1.2 Challenges in Clone Comprehension and Maintenance

The previous section summarized code clone research. Within this research field, challenges have been identified related to the understanding of clones at the clone group level and potential large amounts of data that can be retrieved from clone detection tools. Similarly, challenges were found in the maintenance of clones as it relates to the removal of their associated duplication through the process of refactoring. These challenges are further described in the following subsections.

1.2.1 Clone Representation

Clone groups form a collection of clones that represent the same duplication in code. Because of this, they are a critical element in the understanding of cloning in code. Information gathered about each clone in the group can identify characteristics of the

group in general through properties such as clone location and syntactic differences between the clones. In this case, the understanding of the clones is at the level of individual clone groups rather than considering characteristics of cloning in the system-wide level. In order to determine characteristics of a clone group, a programmer must evaluate the sections of code associated with the clones in the group. This presents a challenge for the programmer who must decipher clone group characteristics based on the multiple instances of the clones in the source code.

Table 1.1

Clone coverage percentage in programs

Program	LOC	% of Clones	Source
Linux kernel	4,365K	15%	[Li <i>et al.</i> , 2004]
Java Development Kit (JDK) 1.4.2	2,418K	8%	[Jiang <i>et al.</i> , 2007a]
JDK 1.3.0	570K	9%	[Kamiya <i>et al.</i> , 2002]
Process-Control System	400K	12%	[Baxter <i>et al.</i> , 1998]
JHotDraw 7.0.7	71K	19%	Using [CloneDR, 2010]
JavaGenes 0.7.68	45K	10%	Using [CloneDR, 2010]

LOC = Lines of code

% of Clones = Percentage of LOC that are clones

1.2.2 Clone Analysis

The utilization of clone detection tools has yielded general knowledge of the existence of clones in code. Table 1.1 lists a sampling of clone coverage in programs of various sizes. The table shows that cloning in code occurs fairly consistently in both small and large programs and also in programs written in different languages such as C and Java. The table also shows that a clone detection tool can report a large amount of data related to the cloning in source code. For example, the Linux kernel was reported by

CP-Miner to have over 120K of what is termed “copy-pasted segments,” which represent sections of code that have been duplicated at least once. This large amount of data poses a challenge for programmers in terms of understanding the clones. With the CP-Miner data, the programmer must weed through the 120K of copy-pasted segments to understand the extent of cloning in the program and determine necessary actions related to the cloning.

1.2.3 Clone Refactoring

Removing the duplication represented by clones can reduce the possibility of errors that may occur in the future. The activity of refactoring [Fowler, 1999] (i.e., when the code is structurally changed, but its behavior is not) can be used as a means to remove the duplication associated with the clones. For example, by using refactorings such as *Extract Method*, a section of duplicate code can be modularized and all original sections of code associated with the duplication can be replaced by calls to the newly extracted method. Future maintenance can be performed in a single location, because in part the modularity of the program has been improved. Currently, the task of refactoring clones is mainly delegated to the programmer (i.e., once clones have been selected for refactoring, the actual refactoring activity is delegated to the programmer). The programmer must either manually refactor the clones or forward the information about the clones to a refactoring engine, which may require individual refactoring steps for each clone in the group. This delegation represents a break in the process of clone maintenance that normally starts with the detection of the clones, which is followed by analysis of the clones to determine refactoring opportunities, and finally ending with the actual

refactoring of the clones. This poses a challenge to programmers who want to remove clones in code, because the process contains several steps requiring manual operations, due to the disjoint characteristics of the phases of this type of clone maintenance.

1.3 Research Objectives

The research described in this dissertation focuses on the activity of maintaining clones in code with an objective of supporting the processes related to this activity. Specifically, the research considers the process of clone maintenance related to the removal of the duplication associated with the clones through refactoring activities. In addition, the research considers novel techniques including techniques from other fields (e.g., Information Retrieval – IR) to assist in the process of clone comprehension.

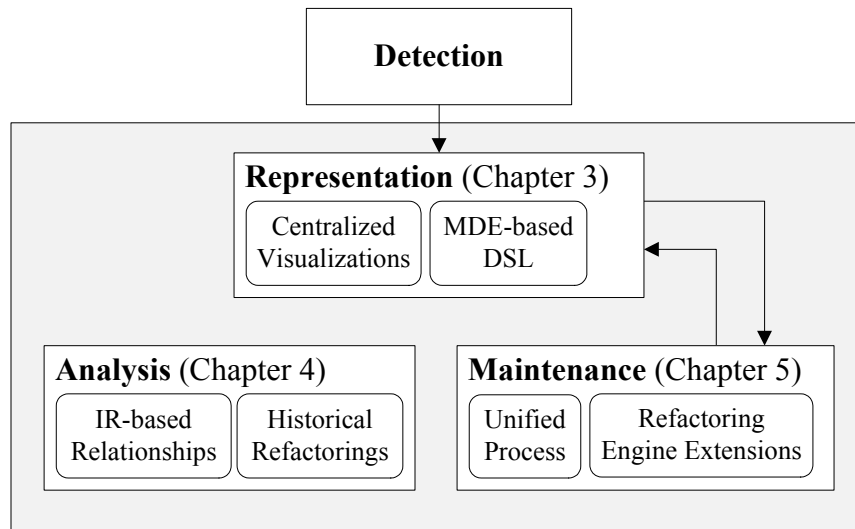


Figure 1.2 – Research overview.

In Figure 1.1, the highlighted research topics (light grey) represent the work related to the research described in this dissertation. Figure 1.2 provides an integrated

view of the research by outlining specific research activities performed in the phases of clone representation, analysis, and maintenance. These research objectives contribute to the three phases specifically related to clone comprehension and maintenance. It should be noted that the research does not focus on the phase of detection, but rather the results from the clone detection phase serve as input into the research activities inside the highlighted box in Figure 1.2. The research utilizes multiple third-party tools that are representative of the current state-of-the-art in clone detection. The work in the analysis phase does not directly feed into the other phases, but both research activities (i.e., “IR-based Relationships” and “Historical Refactorings”) consider clone analysis as it relates to clone maintenance. The research objectives are summarized in the following subsections.

1.3.1 Clone Representation

The research addresses the challenge of comprehending individual clone groups by contributing novel visualizations of clone groups that are incorporated within an integrated development environment (IDE). First, the visualization technique used to show the scattering of concerns as it relates to Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 1997] is used to show the location of clones within the source files. Second, a centralized or localized representation of clone groups displays differences between the clones in the group in a single location. These novel representations of clones contribute to the understanding of individual clone groups by simplifying the task of evaluating properties of the clones in the group. The research also considers utilizing techniques from the field of Model-Driven Engineering (MDE) [Schmidt, 2006] to

represent clone groups as models that are realized through a Domain-Specific Language (DSL) [Mernik *et al.*, 2005] called CoCloRep (Code Clone Representation), which in turn presents the opportunity of transforming these models for purposes of clone analysis.

1.3.2 Clone Analysis

The research addresses the challenge of overwhelming clone data (i.e., clone detection tool reports that can consist of hundreds or even thousands of identified clone groups) by proposing a mechanism of discovering relationships among clone groups. This provides programmers with a higher level grouping of clones above the clone group level. The approach analyzes non-structural properties of the clone groups by utilizing a technique from the field of IR. A separate objective of this research related to clone analysis is to observe the relationship of clones reported by a clone detection tool and actual refactorings obtained from comparisons of consecutive versions of open source software artifacts. In both cases, the research contributes to the further understanding of clone characteristics as it relates to the clone maintenance process.

1.3.3 Clone Refactoring

To address the challenge of the clone maintenance process, this research extends the capabilities of an IDE in an effort to unify the phases of clone detection, analysis, and refactoring. The extensions are realized through an Eclipse IDE plug-in called CeDAR (Clone Detection, Analysis, and Refactoring). The extensions include the incorporation of results from multiple clone detection tools. The refactoring engine of the IDE is also extended to allow for a clone group reported by a clone detection tool to serve as the

input for the refactoring activity. Refactoring transformations for additional types of clones are considered to increase the instances of clone refactoring support. The contribution of the research is a more unified process where the phases of clone maintenance (i.e., detection, analysis, and refactoring) are streamlined together within the programmer's working environment (i.e., IDE). This new process removes tasks that before required manual intervention. However, it should be noted that the research will not produce a totally automatic process and will include user confirmation in the appropriate stages of the process.

1.4 Outline

This chapter has summarized code clone research and the challenges that exist related to clone comprehension and the clone maintenance process of eliminating duplicated code associated with clones. Research objectives that address these problems have been outlined. Chapter 2 describes background information related to the research of this dissertation.

Chapter 3 presents representations of code clones through two visualization techniques. The first visualization [Tairas *et al.*, 2006] adopts the popular AOP visualization technique offered in the AspectJ Development Tools (AJDT) [AJDT Visualiser, 2010]. The second visualization considers a representation of a clone group where the information of clone differences is displayed in a localized manner [Tairas, 2009]. This chapter also describes a representation using a DSL called CoCloRep [Tairas *et al.*, 2007], which is defined using DSLs that are part of the Atlas Model Management Architecture Platform (AMMA) [Kurtev *et al.*, 2006].

Chapter 4 details two analysis studies on clones. The first study considers the semantic properties of the clones by observing the identifier names contained in the code associated with the clones [Tairas and Gray, 2009a]. The IR technique called Latent Semantic Indexing (LSI) is used to generate vector representations of the clone groups. These vectors in turn are used to cluster the clone groups to determine relationships within the clone group level. The study evaluated clones detected by CCFinder in the Windows NT Research Kernel. The second study evaluates the history of code changes as they relate to the clones reported by a detection tool [Tairas and Gray, 2010b]. Clones detected in one version of the source code are observed in a subsequent version to identify characteristics of their refactorings. Several Java open source software artifacts were studied. Clones from these artifacts were reported by several clone detection tools: CCFinder, CloneDR, Deckard, Simian, and SimScan.

Chapter 5 describes the CeDAR Eclipse plug-in, which represents an effort to unify the process of clone maintenance [Tairas and Gray, 2009b]. This plug-in can obtain clone detection results from several tools and incorporates the visual representations presented in Chapter 3. Extensions to the Eclipse refactoring engine are described that allow more types of clones to be supported for automated refactoring after a programmer has confirmed a clone group for refactoring. The extensions are evaluated by observing the increase in the number of automated refactoring instances.

Chapter 6 outlines future work of the research described in the previous chapters. Chapter 7 concludes the work of this dissertation and summarizes its contributions.

CHAPTER 2

BACKGROUND

This chapter provides background information relevant to the research of this dissertation. First, a general description of code clones is given with additional information related to automated clone detection techniques and how the contributions of this dissertation are informed by results from these techniques. This section will also outline the code clone-related terms that will be used for the remainder of the dissertation. Second, background information will be given for the topic of MDE including MDE-related technologies that are utilized in CoCloRep. Third, the field of IR is described in general with additional emphasis on the IR technique called LSI.

2.1 Code Clones

Code clones represent two or more sections of code that are duplicates of each other. A collection of clones that represent the same duplication are called a clone group, clone class, or clone set. The exact definition of what constitutes a clone has eluded code clone researchers. This can be seen by an effort to find different names that are more appropriate to represent the activity of cloning [Walenstein, 2006]. A widely accepted categorization was proposed in [Bellon *et al.*, 2007] where clones are distinguished based on their levels of similarity. Bellon *et al.* divide code clones into three types. *Type I* clones represent sections of code that are identical or exactly match each other. *Type II*

clones represent syntactically identical copies that allow differences among the identifiers such as variables, types, and functions. *Type III* clones allow differences within the statements of the code including the addition or deletion of statements among the clones. A fourth type (i.e., *Type IV*) represents semantically identical code that performs the same functionality, but can differ syntactically [Roy *et al.*, 2009].

For the remainder of this dissertation, the following terminology related to code clones will be used: The term “clone” will refer to a code clone unless otherwise stated. In addition, this term will also refer specifically to the section of duplicated code that it represents. In various research publications, the collection of clones of the same duplication has been called a clone group, clone class, or clone set. This dissertation will use the term “clone group.”

2.1.1 Clone Detection

As stated in Section 1.3, the research described in this dissertation does not focus on the phase of clone detection. The results of various clone detection techniques are instead utilized as input to the process described in this dissertation. This allows for more focus to be placed on improving the representation of clones detected by such techniques and performing further analysis of these clones. In addition, the research focuses on the utilization of the results from clone detection tools in the process of eliminating the duplication associated with the clones. It should be noted that the original interest in code clones started with the author’s Master’s project that developed a clone detection plug-in for the Microsoft Phoenix framework [Microsoft Phoenix, 2010]. This work is detailed in Appendix A.

Research related to the detection of clones has received much focus in the past decade. In their qualitative evaluation of detection techniques, Roy *et al.* list 38 unique methods of finding clones that have been proposed [Roy *et al.*, 2009]. These methods utilize various representations of code and search techniques in order to find the clones. Determining the best method of clone detection has not been easy. An evaluation of six clone detection tools representing different techniques did not find a clear winner [Bellon *et al.*, 2007]. Each technique consists of both advantages and disadvantages. A tool that evaluates the string representation of the code can execute quickly, but fails to find clones with trivial differences such as whitespace and comments. Token-based tools focus more on meaningful parts of the code (i.e., the tokens recognized by the definition of a programming language), but may report clones that are not syntactically meaningful (e.g., clones spanning across method bodies). Tree-based tools report the most structurally defined clones, but can require significantly more time to execute, because of the need to parse the code.

For this dissertation, several clone detection tools are used in different research activities. This provides a broad representation and utilization of current state-of-the-art techniques of clone detection within the research activities. The research related to the analysis of clones as described in Chapter 4 has utilized tools representative of different code representation techniques. Further discussion related to the performance of these tools is given in the chapter. The refactoring work described in Chapter 5 utilizes a tree-based tool because the focus is on refactoring a syntactically meaningful block of code.

2.1.2 Parsing Clone Detection Tool Results

Among the work listed in [Roy *et al.*, 2009], a few clone detection tools have been made available to the public for free use in a non-commercial research setting. These tools include CCFinder [Kamiya *et al.*, 2002], Deckard [Jiang *et al.*, 2007a], Simian [Simian, 2010], and SimScan [SimScan, 2010]. A separate commercial tool called CloneDR [Baxter *et al.*, 1998] was purchased and made available for this research in addition to the aforementioned non-commercial tools.

The initial step of the research activities described in subsequent chapters is the parsing of textual clone detection tool outputs in order to retrieve pertinent information regarding the location and grouping of the clones. The tools selected for the research include in their reporting mechanisms a textual-based output containing information about the detected clones. In these reports a clone is generally identified by three properties: the location of the file containing the clone and the clone's starting and ending lines within that file. Thus, a clone, c , can be represented by $c = (f, s, e)$, where f is the file containing the clone, and s and e are the line numbers signifying the range of the clone. In addition, clones that represent the same duplication are grouped together in the report using different delimiters.

Figure 2.1 provides a snippet of the textual output of the SimScan clone detection tool. Information for each clone follows a standard format as do the grouping of clones. In this case, clones are delimited by a comma and clone groups are delimited by a new line (the snippet in Figure 2.1 truncates a single line into separate lines by each comma). The standard layout for this and other tools' output enables a fairly straightforward process of parsing the clone information. The textual output can also contain extraneous

content that is not parsed. For example, the numbers before the listing of each source file in the SimScan output are not utilized in subsequent steps.

Source File	Starting Line	Ending Line	
76397-C:\...\CMPFieldMetaData.java:	134-	145,	
76296-C:\...\CMPFieldMetaData.java:	117-	129	
433729-C:\...\UsersRolesLoginModuleTest.java:	64-	68,	Clone
420696-C:\...\LoginModulesTest.java:	312-	316	
164262-C:\...\ServerDataCollector.java:	230-	265,	
231230-C:\...\Scheduler.java:	552-	587	
248103-C:\...\EJBVerifier11.java:	448-	480,	Clone Group
249898-C:\...\EJBVerifier11.java:	1073-	1109,	
250532-C:\...\EJBVerifier11.java:	1297-	1337	
...			

Figure 2.1 – Example of SimScan output.

CCFinder requires additional processing, because the reported clones are identified with their token ranges rather than line ranges. In addition, only clone pairs are reported and hence a clone group in CCFinder must be determined by performing a union among associated clone pairs. In Figure 2.2, a snippet of the textual output of CCFinder is shown. Clones are associated with each other in pairs inside the `clone_pairs` block. Information for each clone consists of the source file ID and token ranges. A clone pair relation ID is included to relate clone pairs of the same group.

The parsing of the results of several clone detection tools is available in the CeDAR plug-in. Currently, CeDAR can parse the textual outputs of CCFinder, CloneDR, Deckard, Simian, and SimScan. Parsing the textual outputs provides the input for the representation, analysis, and refactoring of clones in CeDAR, which will be discussed in the subsequent chapters of this dissertation. The textual outputs of the tools can overwhelm a programmer if he or she were to process these outputs manually. For

example, the textual output of Simian for JDK 1.5.2 is approximately one megabyte (MB) in size [Simian, 2010]. By parsing the results of tools that only provide textual outputs, CeDAR offers an alternative means of presenting the results to the user within an IDE. This allows for such features as providing connections to the actual code to become available.

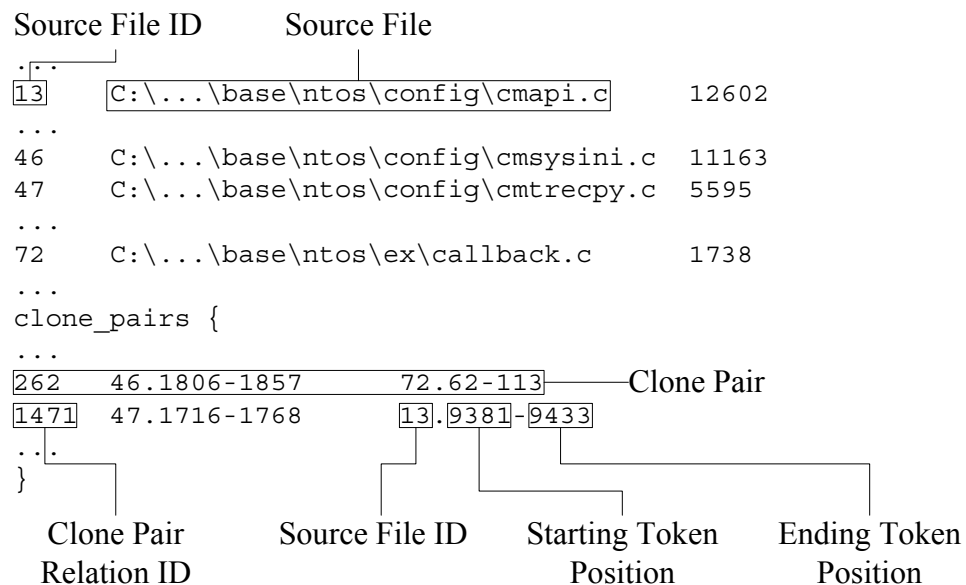


Figure 2.2 – Example of CCFinder output.

2.2 Model-Driven Engineering

The advancement of software development techniques have resulted in the introduction of various platforms, tools, and frameworks that can improve the software development process. However, the proliferation of these various solutions has also increased the problems of communication, interoperability, and selection of a single solution for software development [OMG, 2003]. To alleviate these shortcomings, the Object Management Group (OMG) proposed an initiative called Model-Driven

Architecture (MDA) [OMG, 2001], which is concerned with the development of platform-independent models (PIMs) for software solutions that are “language-, vendor-, and middleware-neutral” [OMG, 2000]. Using transformations, these general application models can produce models targeted for a specific platform. In this case, the transformations take a PIM as the input or source model and transforms or translates it into a platform-specific model (PSM) as the output or target model. This allows users to develop platform-independent constructs that can subsequently be transformed, in some cases automatically, into a platform-specific solution for a particular language, vendor, or middleware.

The utilization of models and their associated transformations in a broader sense falls under the category of MDE. MDA can be considered an MDE initiative proposed by OMG. MDE is concerned with raising the abstraction level of software development by utilizing models to specify the application. This can be seen in MDA through the initiative of raising the development abstraction with PIMs. The specification of models in MDE utilizes DSLs that are tailored to a particular domain [Schmidt, 2006]. In this case, DSLs are used rather than general-purpose languages (GPLs), because of their improved expressiveness of the domain [Mernik *et al.*, 2005]. In addition, the utilization of a DSL allows for non-programmers to contribute in the development of domain-specific models.

Model transformations are developed to produce output based on the need of the user. One specific use of model transformations is to automatically synthesize or generate source code based on the properties of the models [Schmidt, 2006]. The availability of model transformations allows for the development of a system at a higher level of

abstraction through models and the generation of, for example, object-oriented code through transformations of these models. Similar to the paradigm of “Everything is an object,” as promoted by the object-oriented methodology, MDE in turn promotes the notion of “Everything is a model.” Related to this notion, models in MDE become a means to produce implementations of the system and not merely a type of static documentation of a system [Bézivin, 2005].

Chapter 3 includes an investigation into the utilization of MDE techniques to represent and analyze clones through DSLs. This investigation considers the representation of clone groups as models and their analysis performed via model transformations. The Atlas Model Management Architecture Platform was used to develop a DSL for clone representation that can be projected into the MDE space as models. AMMA was used because it provides support for model transformations through an implementation that is inspired by the specification of model transformations adopted by OMG, namely Query/View/Transformations (QVT) [OMG, 2002]. AMMA is described in the following subsection.

2.2.1 MDE through AMMA

The AMMA Platform is a model-based framework for the development of DSLs [Kurtev *et al.*, 2006]. DSLs can be defined using components available in AMMA and manipulation of the models associated to the DSLs can be performed through model transformations. In AMMA, the components that define the DSLs and model transformations are themselves DSLs. For example, the abstract syntax of a DSL can be defined using a DSL called Kernel Meta-Meta Model (KM3) [Jouault and Bézivin,

2006]. In general MDE terms, this abstract syntax is also called the metamodel. For model-based DSLs, the abstract syntax can also be considered as a Domain Definition Metamodel (DDMM). The metamodel provides the important abstractions of a domain and the relations between these elements. These abstractions in turn are represented by a model and thus the metamodel becomes the reference for all models conforming to it [Kurtev *et al.*, 2006].

The concrete syntax of a DSL can be defined using a DSL in AMMA called Textual Concrete Syntax (TCS) [Jouault *et al.*, 2006]. In TCS, the concrete representations of elements in a DSL can be connected to their associated abstractions as defined in a corresponding metamodel. The constructs of TCS consist mainly of syntactic templates based on the DSL syntax of TCS. The language designer specifies his or her DSL using these templates, and a parser is automatically generated. The models conforming to a metamodel originate from textual descriptions of the engineered DSL. This presents a division of two different levels of technologies or technical spaces [Kurtev *et al.*, 2006], namely grammar-level and model-level. A popular name for the grammar-level is “Grammarware” [Klint *et al.*, 2005]. As the name suggests, grammarware is concerned with software systems that are based or dependent on grammars. These include parsers for textual programming languages that are defined using a grammar such as the Extended Backus-Naur Form (EBNF). As the source models are originally written in the textual DSL (i.e., grammar-level) and manipulations of the models are performed in the MDE space (i.e., model-level), transitions to and from the two technical spaces are needed. *Injections* are performed to move from the grammar level to the model level and *extractions* are performed for the reciprocal task.

Model transformations represent the key activity in MDE that is used to produce an output expected by the user. The output generated by the transformation varies widely based on the specific needs of the user. As described in the previous section, one output is synthesized code. This code could be produced through a process of transforming PIMs to PSMs. However, this is only one possible result of model transformations. In the next chapter, we utilize model transformation to perform analysis on clones based on the representative model generated from a DSL called CoCloRep. Utilizing another DSL of AMMA called Atlas Transformation Language (ATL) [Jouault *et al.*, 2008], we define a set of rules that will transform the clone representation in CoCloRep into analysis results of the clones. In this case, CoCloRep is considered as the source model and the results of analysis are viewed as the target model generated from the transformation.

2.3 Information Retrieval

The field of information retrieval focuses on the extraction of useful information from a corpus of data. The activity of information retrieval is mostly associated with the scenario where a user generates a query that is used to obtain a selection of documents within a corpus of data that satisfies the query put forth by the user [Rijsbergen, 1979]. In this case, the documents that are retrieved are deemed *relevant* in some manner by the information retrieval technique or system that is utilized. This relevance is normally associated with calculations based on the unstructured terms or elements that exist within the documents. This is different from database searching styles that require a specific query value (i.e., order ID) that is then evaluated on specific elements within a structured layout of database tables [Manning *et al.*, 2008]. In addition, the retrieval of documents is

based on a process that attempts to collect related documents that are more automated compared to the process of browsing documents manually through the corpus [Baeza-Yates and Ribeiro-Neto, 1999].

Retrieving documents based on a query is just one scenario that is considered in the information retrieval process. A separate activity is the clustering of documents where the grouping is based on the contents of the documents (i.e., the terms in the documents) [Manning *et al.*, 2008]. In this case, relevant documents are clustered together and form a grouping consisting of semantically associated documents. This type of scenario is used in the analysis of clones as described in Chapter 4. LSI is used to cluster clone groups to determine further relationships of clones within the clustered groups. In Chapter 4, the utilization of LSI was inspired by a previous work that detected high-level concept clones that also used LSI [Marcus and Maletic, 2001]. Moreover, LSI has been shown to perform very well compared to other related techniques [Zukas and Price, 2003]. The LSI technique is introduced in the following subsection.

2.3.1 *Latent Semantic Indexing*

LSI is concerned with obtaining relationships among the documents that divide a corpus and the terms in these documents. The relationships are discovered through an analysis of the terms and documents using matrix manipulation techniques [Deerwester *et al.*, 1990]. The technique generates a semantic space through steps that are initiated by the counting of terms in each document in the corpus. The term totals are recorded in a matrix called a term-document matrix. Subsequent steps include calculating a lower rank representation of the original term-document matrix in order to remove noise from the

data. In this semantic space, documents are represented as vectors where relevant documents are positioned near each other and can be clustered together based on their cosine similarity values. These relationships are based on an underlying “latent” structure within the matrix.

An important part of the LSI process is obtaining the singular value decomposition (SVD) [Strang, 1993] of the term-document matrix. An SVD of a matrix, say $X_{m \times n}$, produces three matrices; i.e., $X_{m \times n} = U_{m \times n} S_{n \times n} V_{n \times n}^T$, where S is a diagonal matrix containing singular values, which are ordered from largest to smallest along the diagonal. By selecting a number of retained singular values, say k , a new matrix $\hat{X}_{m \times n}$ can be obtained; i.e., $\hat{X}_{m \times n} = U_{m \times k} S_{k \times k} V_{n \times k}^T$. This matrix is an approximate (but still valuable) representation of the original matrix X with the added property that the lower dimensionality reduces the noise from the original matrix. Given \hat{X} , cosine similarity calculations can be performed between the documents that are represented by column vectors of \hat{X} [Deerwester *et al.*, 1990].

In Chapter 4, the process of document clustering is used to cluster clone groups based on the terms inside the clones that represent the groups. The terms in this case are the identifier names extracted from each clone.

CHAPTER 3

CLONE REPRESENTATION

Clone groups play a crucial role in the understanding of clones as they identify the relationship of individual clones consisting of the same duplication. After clone detection using an automated tool has been performed, the programmer will focus on a clone group and the clones within that group. This can involve evaluating properties of the clones in the group where the clones can be scattered in multiple files within the collection of source files. Representations of clones that include visualization techniques have been proposed to provide a means to understand the properties of clones. However, these representations deal mostly with cloning in a system-wide perspective. This chapter introduces our work in representations that are more focused on individual clone group properties. Section 3.1 provides a visualization of clone groups based on the location of the clones within the collection of source files. Section 3.2 introduces a localized representation technique for clone groups in which the properties of the clones in the group are displayed in a single location. Section 3.3 describes an investigation on the use of MDE to represent clone groups and thus allow analysis of the groups through model transformations.

3.1 Visualization of Clone Detection Results

Previous research has shown how scatter plots [Ducasse *et al.*, 1999] can be used to render a graphical representation of clone detection results where duplicate sections of code are identified as a sequence of connected dots in a graph. This section describes the integration of a stand-alone clone detection tool into Eclipse and a corresponding alternative visualization of clone detection results [Tairas *et al.*, 2006]. An Eclipse plug-in called CloViz (Clone Visualization) is described that displays the results of CloneDR. The visualization of the results is implemented as an extension to the AspectJ Development Tool (AJDT) Visualiser plug-in, which is primarily used to view crosscutting concerns in aspect-oriented programs.

The connection between clones and crosscutting concerns can be seen through the notion of dominant decomposition within AOP. In this case, one functionality or concern dominates another [Tarr *et al.*, 1999]. These two concerns are said to crosscut each other and the code for the “weak” functionality must be scattered throughout the program. The scattering of the weaker concern produces similar sections of code that are duplicated in multiple locations of the source code exhibiting the same properties as clones. This connection motivates the utilization of extending the AJDT Visualiser to in turn visualize clones through CloViz.

3.1.1 Tool Integration and Extension

CloneDR serves as an external pre-existing clone detection tool that is called by CloViz and executes independently. The results of CloneDR are retrieved and parsed by CloViz and reported to the user. The visualization feature is implemented through an

Eclipse plug-in extension point provided by the AJDT Visualiser plug-in. The developers of the Visualiser have opened the plug-in for adaptation by providing several extension points to allow other types of information to utilize its visualization features. Some examples of other tools that use the Visualiser include applications for Google search results and Concurrent Versions System (CVS) file histories.

3.1.2 Details of CloViz Plug-in

CloViz plays the role of an interface between CloneDR and the user. The plug-in performs the duties of setting up the configuration file, executing CloneDR as an external task, and displaying the clone detection results to the user. A diagram of the connections and processes related to CloViz is shown in Figure 3.1.

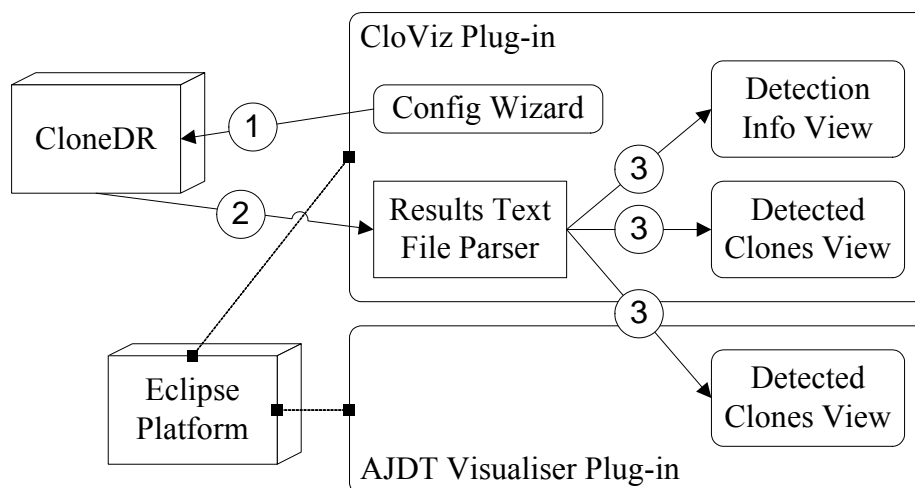


Figure 3.1 – CloViz connections and processes.

When the process is started (i.e., when the user clicks the clone detection button on the toolbar), CloViz will display a configuration wizard that assists the user in

determining the type of configuration for the clone detection procedure. Upon completion of this wizard, CloneDR will be called and given the location of the file containing the configuration settings (step 1). CloneDR will then execute and produce a text file containing its clone detection results. These results are parsed by CloViz (step 2) and sent to three Eclipse views that will display the results of the clone detection procedure (step 3). One of these views is an extension of the AJDT Visualiser that produces a graphical representation of the results of the clone detection.

A file containing the configuration settings must be provided for every execution of CloneDR. The user is allowed to set five configuration settings. The values entered by the user set the attributes that the tool will use to determine whether two sections of code are clones. Examples of these attributes include the similarity of code sections, the maximum number of parameters contained in each clone, and the starting depth of the sub-tree to be evaluated. In addition to the five configuration settings, the configuration file also contains the list of all source files that will be searched for clones. A wizard dialog is available to assist the user with the configuration setup, which contains a checkbox listing of files and directories of the Eclipse project that is currently selected and textboxes for the configuration settings. The detection of clones usually includes all files for a project or program, but this setup allows users to select specific files. Figure 3.2 shows a screenshot of the wizard dialog.

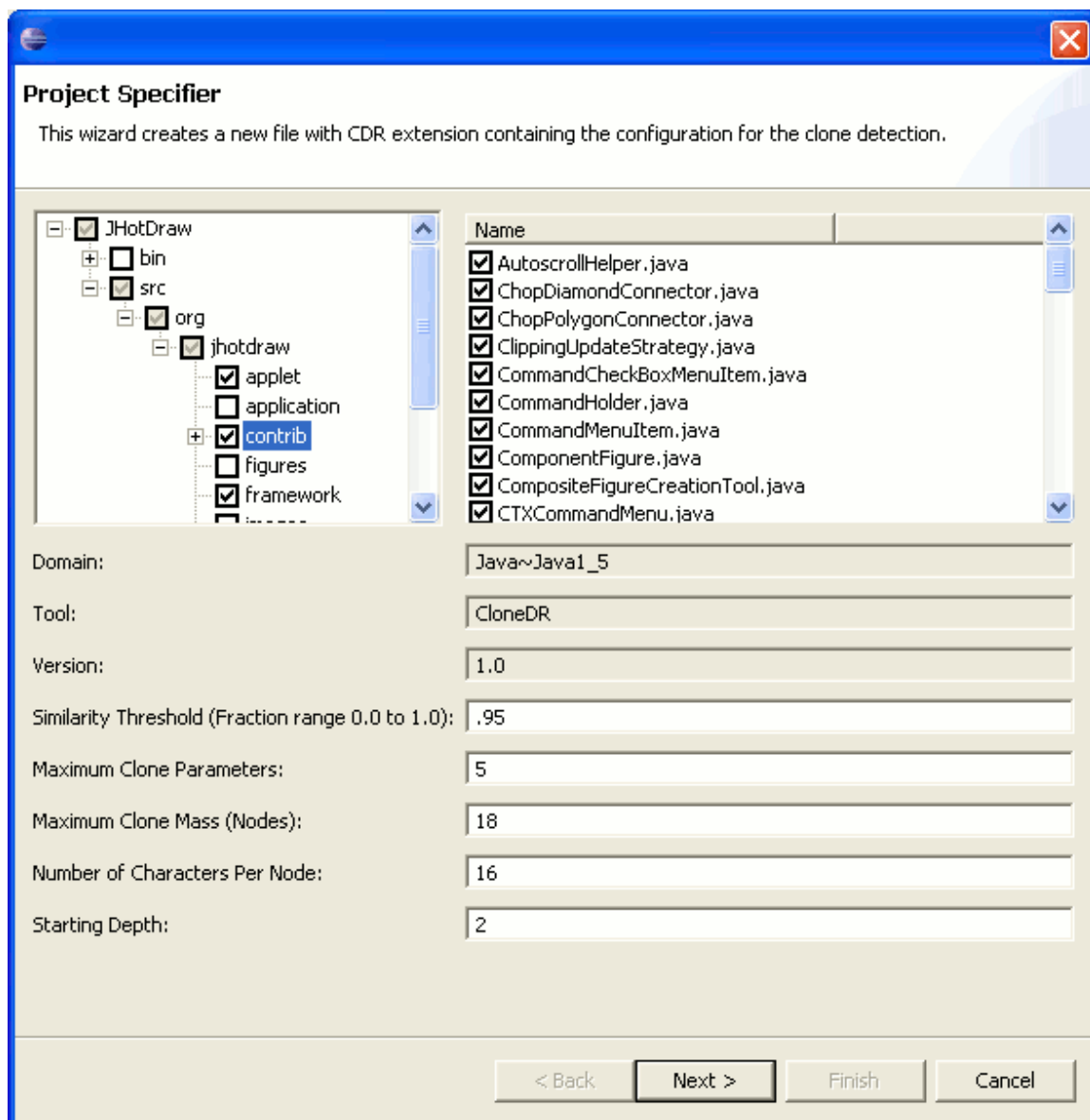


Figure 3.2 – Wizard dialog.

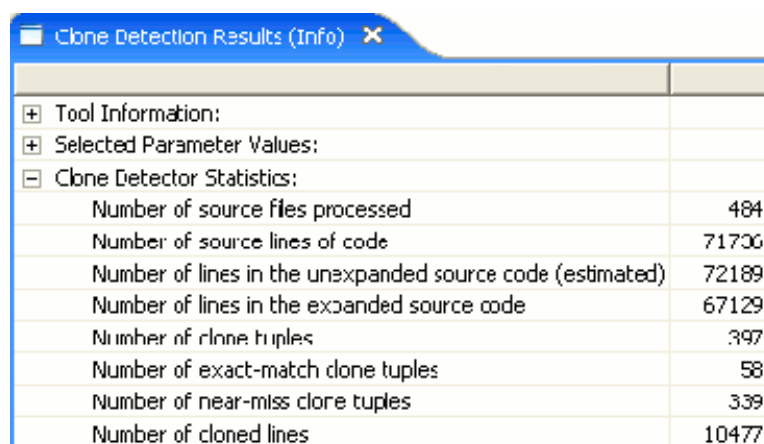
3.1.3 Clone Detection Execution and Display of Detection Results

CloViz invokes CloneDR as a separate command-line process. The arguments that are passed to the command include the location of the configuration file and the location of the file that will contain the results of the clone detection procedure. The results file is parsed to extract the information about the detected clones.

After the clone detection procedure and parsing are completed, CloViz will display the results in three customized views:

- General information view
- Detected clones list view
- Visualization of detected clones view

The last two views essentially display the same information with the only difference in appearance – one is textual and the other is graphical. Both views update their states through listeners that are triggered when information about a clone is parsed and retrieved from the results file. A more detailed description of each view is given in the following paragraphs.




Clone Detection Results (Info)	
+ Tool Information:	
+ Selected Parameter Values:	
- Clone Detector Statistics:	
Number of source files processed	484
Number of source lines of code	71706
Number of lines in the unexpanded source code (estimated)	72189
Number of lines in the expanded source code	67129
Number of clone tuples	397
Number of exact-match clone tuples	58
Number of near-miss clone tuples	339
Number of cloned lines	10477

Figure 3.3 – General information view.

General information view. This view provides a summary of general information related to the clone detection procedure. This information includes the values that were selected by the user for the various configuration settings and statistical information such as the number of source files that were processed, the number of clones detected, and the lines of cloned code. An example of this view is given in Figure 3.3.

Detected clones list view. The detected clones are displayed under their respective clone groups. An example of this view is given in Figure 3.4. In the view, each clone is identified by the source file where it is located and its starting and ending lines. Double-clicking on one of the clones will display the clone (with its corresponding lines highlighted) in the editor view.



	Start	End
[-] Group 7 (Clones: 2 Lines: ~28 Parameters: 0 Similarity: 1.000)		
AbstractFigure.java	344	371
NullFigure.java	155	183
[-] Group 8 (Clones: 10 Lines: ~4 Parameters: 1 Similarity: 0.988)		
SelectAllCommand.java	64	67
PasteCommand.java	82	85
ChangeConnectionHandle.java	232	235
BoxHandleKit.java	118	121
TriangleRotationHandle.java	120	123
PolygonScaleHandle.java	141	144
UngroupCommand.java	71	74
RadiusHandle.java	88	91
GroupCommand.java	58	61
BorderTool.java	91	94

Figure 3.4 – Detected clones list view.

Visualization of detected clones view. Extending the AJDT Visualiser plug-in offers a graphical view of the clones that were detected in the source files. The view consists of three parts: bars, stripes, and kinds. In the original AJDT Visualiser, a bar represents a source file. A stripe represents a crosscutting concern and a kind represents concerns of the same type. In CloViz, a bar also represents a source file, but a stripe now represents a clone and a kind represents a clone group. In the AJDT Visualiser, stripes can occur in more than one bar and each stripe can be associated with one or more kinds, where each

kind is distinguished by a different color. In our case, stripes are only associated with one kind because a clone is only associated with one clone group. Before the clones are retrieved from the results file, the bars for the view are generated by traversing through the Eclipse project and selecting resources that are source files. After the bars have been generated, stripes can be added to display the clones in each source file. Similar to the detected clones list view, double-clicking on a stripe will display the clone (represented by that stripe) highlighted in the editor view. Figure 3.5 shows the visualization of detected clones view that is part of the CloViz plug-in.

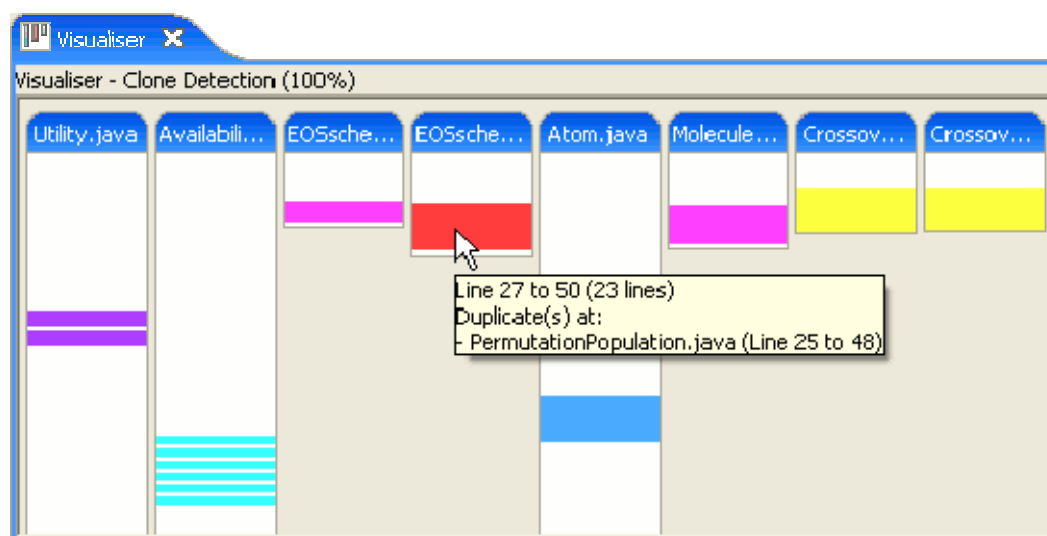


Figure 3.5 – Visualization of detected clones view.

Figure 3.5 also displays an example of a pop-up text that provides more information about the clone when the stripe representing the clone is moused over. The information given includes the starting and ending line numbers of the clone in the source file and the names of other source files that also contain the stripe (or clone) with their respective starting and ending line numbers.

3.1.4 Visualiser View Enhancements

Taking advantage of the visualization offered by the AJDT Visualiser plug-in through its extension points allows the clones to be displayed in a more visual manner. However, some limitations were found in the current implementation of the AJDT Visualiser. This is specifically true with the limitation of not being able to add customized actions in the context menu of the AJDT Visualiser view. One predefined action is given in the context menu, which emerges if the user right-clicks on a bar in the view. This action will display all bars that contain at least one of the stripes that is in the currently selected bar. An alternate filtering method was added in CloViz's visualization view that displays all bars that contain the same type (kind) of stripe that is specifically selected. This is a useful feature in terms of clone detection results because only source files (or bars) that contain clones (or stripes) that are part of the same clone group will be displayed. Clones that are scattered throughout the program can be separated out and viewed together. Additional context menu options that open the source file of a specific clone or all files containing the same clone were also included. Figure 3.6 displays all available options upon right-clicking on a stripe in the view. The context menu for the AJDT Visualiser view was not extendable because it was not registered as an extension point. This required the source code of the AJDT Visualiser plug-in to be edited directly to allow the additional actions in the context menu. The changes are realized in the CloViz plug-in.

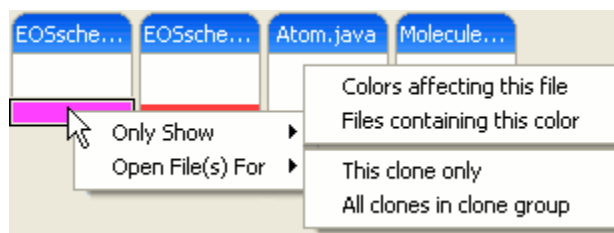


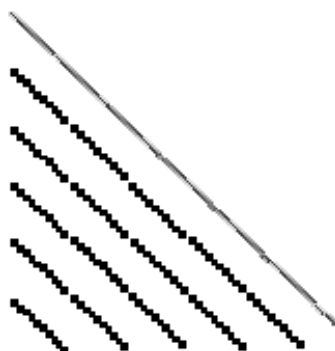
Figure 3.6 – Context menu options.

3.1.5 Results Representation

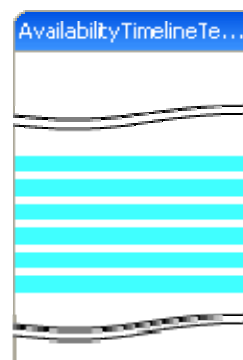
We applied CloViz on two open source Java programs, JavaGenes and JHotDraw. JavaGenes is an open source NASA program that has been made available to the public [JavaGenes, 2010]. JHotDraw is an open source GUI framework that is used frequently in software engineering tool evaluation [JHotDraw, 2010].

Group 1 (Clones: 6 Lines: ~5 Parameters: 5 Similarity: 0.982)		
AvailabilityTimelineTest.java	143	147
AvailabilityTimelineTest.java	149	153
AvailabilityTimelineTest.java	155	159
AvailabilityTimelineTest.java	161	165
AvailabilityTimelineTest.java	167	171
AvailabilityTimelineTest.java	173	177

(a) Text listing



(b) Scatter plot



(c) CloViz visualization view

Figure 3.7 – Clone group representations.

Although CloneDR was selected for this project, the focus of this section is not about the quality of the clone detection results, but rather an investigation into the types of clone detection reporting methods. We consider two types of clone detection result representations: an estimation of how a scatter plot would display the clones and how CloViz would display the clones.

Clones in the same class. Both JavaGenes and JHotDraw contain an example where one group of clones is located in multiple locations in the same source file. In JHotDraw, `CTXWindowMenu.java` contains five sections of code that are the same clone. Similarly, the file named `AvailabilityTimeLineTest.java` in JavaGenes contains six sections of code that are clones in the same file. Figure 3.7 provides a sample of how the clones of the latter group (i.e., listed in Figure 3.7(a)) are displayed in: (b) a scatter plot and (c) a CloViz visualization view. The clone group represented in Figure 3.7 corresponds to methods that contain identical statements with varying degrees of time values. The scatter plot consists of dots where each dot represents two lines that are duplicates of each other. When multiple dots generate a semblance of a diagonal line that is parallel to the main diagonal line, the sections of code that is represented by the dots are clones. In Figure 3.7(b), each clone is represented by a sequence of diagonal lines. Because the clones are right after each other, the diagonal lines form their own sequence from top to bottom or from left to right. However, the longest sequence of diagonal lines consists of only five diagonal dotted lines, because one clone is represented by the main diagonal (i.e., solid line). Moreover, additional diagonal dotted line sequences are drawn, because each clone

is a duplicate of every other clone in the group. This forms a triangle of diagonal dotted lines, which must be interpreted by the user as a representation of the sequence of six clones in the same file. In contrast, CloViz provides a straightforward representation of the same clones. The stripes are drawn in sequence in the approximate location of where the clones are located in the source file. The width of the stripes is an estimated representation of the size of the clones in terms of lines of code and the same color is used for each stripe to signify that the clones they represent are in the same clone group.

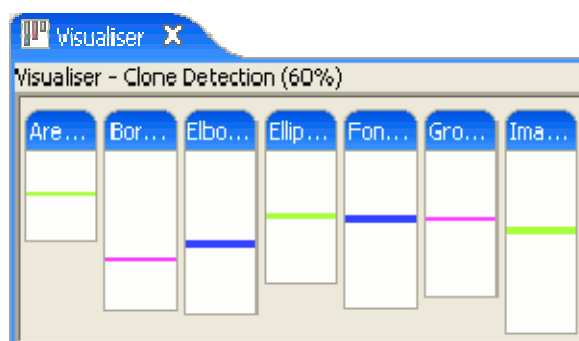


Figure 3.8 – Ubiquitous clones.

Ubiquitous clones. As shown in Figure 3.8, several clones are ubiquitous in that they are present throughout multiple source files in JHotDraw. These ubiquitous clones in particular are short methods that perform a specific task, such as returning a new `Rectangle` object, drawing two ovals (including setting their colors), and setting the undo and redo flags for the drawing view. A feature in CloViz can filter the bars representing the source files to show only those containing clones of selected clone groups. Figure 3.8 shows three clone groups that are clearly distinguished by unique

colors. Current implementations of scatter plots are based on one color making it more difficult to distinguish which diagonal lines are a part of which clone group.

The visual representation of clones provided by CloViz is based on the position of the clones in the source files. The next section considers properties closer to the actual representation of clones. In this case, the similarities and differences of the actual source code elements of the clones provide the properties needed to summarize and visualize the clones within a centralized location.

3.2 Localizing Clone Group Representation

The utilization of refactoring tools, including those that are part of an IDE, has been documented to be under-utilized [Murphy-Hill and Black, 2008]. One of the reasons is the need to configure the refactoring activity through multiple modal dialog boxes that forces a separation between the activities of program editing and the actual refactoring task. That is, the programmer must switch from the activity of editing the source code in a source editor to answering configuration questions in dialog boxes, thus making the source editor unfocused. The refactoring tool Refactor! Pro proposes a solution to reduce the need for dialog boxes by visualizing refactoring changes directly in the source editor [Refactor! Pro, 2010].

The principle of keeping most programming activities within the source editor can also be applied to the representation of code clones. After the results from a clone detection tool are obtained, a programmer must evaluate the clones in clone groups by observing directly the actual sections of code that represent the clones. However, in some

instances, the clones can be scattered in several files and opening each file containing the clones can clutter the view of the code in an IDE. This suggests a similar situation with the display of dialog boxes in refactoring tools, as opening multiple files containing the clones and going back and forth between each clone instance separates the programmer from the main focus of program editing. This motivates the need to provide a localized representation of the clones that displays the properties of each clone and the relationships among them. This section describes a localized representation of a clone group where the information about each clone in the clone group can be viewed in one location. This representation is included as a feature in our CeDAR plug-in.

3.2.1 *Representing Clones in One Location*

In order to provide a localized representation of the clones in a clone group, the code associated with one of the clones is used to display information regarding all the clones in the group. That is, the code related to one clone is highlighted in its original location in the source editor and the differences and similarities among the clones in the group are displayed. Figure 3.9 provides a sample view of a localized clone group representation for a code fragment in a group of clones.

The parts of the code that differ are called the “parameterized” elements of the clones. These are highlighted in neon green in Figure 3.9 (i.e., the string `"Unable to delete file "` and variable `file` in the declaration of variable `message`). The clone group representation, as implemented in CeDAR, displays differing values above the highlighted section of code. In the figure, *Clone 4* contains a different string value, which also implies that *Clones 1-3* contain the same value as that presented in the code. Clones

containing parameterized elements represent clones that are syntactically identical, but differ in identifier values (i.e., *Type II* clones). Many clone detection tools provide the ability to detect such clones in addition to *Type I* clones. *Type III* clones can also be detected by some tools. The ability of tools to detect clones representing different levels of similarity offers developers information about the differences between various clones, whether it be slight differences or more complex differences (i.e., clones of *Type II* and higher).

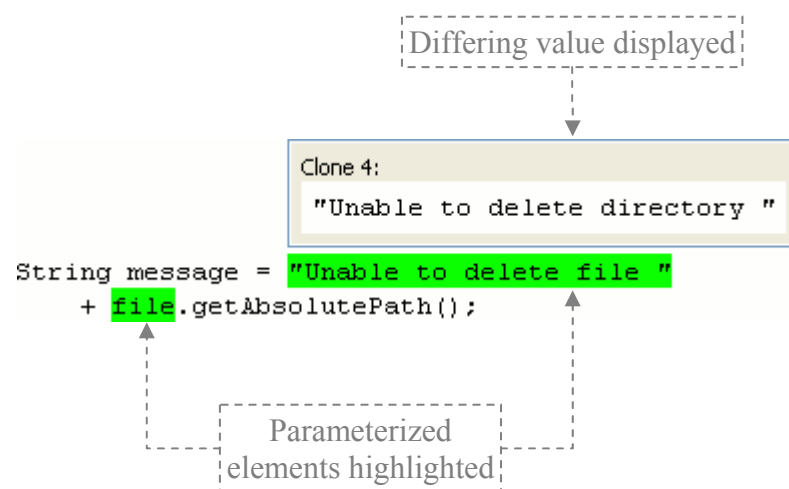


Figure 3.9 – Sample code fragment display.

3.2.2 The CeDAR Plug-in and Clone Visualization

The initial step of displaying localized clone group information in CeDAR is to obtain the location of clones from the results of clone detection tools. The process of obtaining this information is similar to the process described in Section 3.1 for the purposes of visualizing the locations of the clones. Unlike the plug-in described in Section 3.1, CeDAR does not rely on a specific clone detection tool; it can parse the

output of several tools to obtain the two main outputs of clone location and clone grouping. However, it should be noted that CeDAR does not automatically invoke the clone detection process as was done in the plug-in described in Section 3.1. Instead, a tool must be run independently first to obtain the detection results.

The actual clone detection process related to these tools does include the determination of differences among the clones. That is, in the process of performing the detection of clones, these tools include a mechanism to determine differences among the clones. However, in the majority of the tools such difference information is only part of an intermediary process and not made available in the final results that are reported to the user. Only CloneDR provides information about the parameterized elements in the clones, details of which and comparisons with CeDAR will be described in the related works section. The process described in the subsequent subsections performs analysis of the clones reported by a clone detection tool to determine the parameterized differences among the clones. In effect, CeDAR performs a “second pass” on the code to determine these properties. However, this independent process allows for CeDAR to utilize results from different clone detection tools.

Figure 3.10 provides a screenshot of the CeDAR plug-in. After the results of a clone detection tool have been obtained, the listing of clones is provided in the “Clone Detection Results” view (i.e., bottom part of the figure). When a user selects a clone group, information related to the clone group is presented in the “Clone Group” view (i.e., right side of the figure). This includes a listing of the clones in the group and the location of the clones in the class hierarchy. The first clone is initially selected as the default clone, which is identified by the blue circle in the “Clone Group” view (i.e.,

Clone 1 in the figure). The source file containing this default clone is opened in the source editor and the source code associated to the clone is properly highlighted with a light blue background and bordered by two horizontal lines. The parameterized elements of the clone group are appropriately highlighted in the source editor, which yields the localized representation as seen in the source editor view in Figure 3.10.

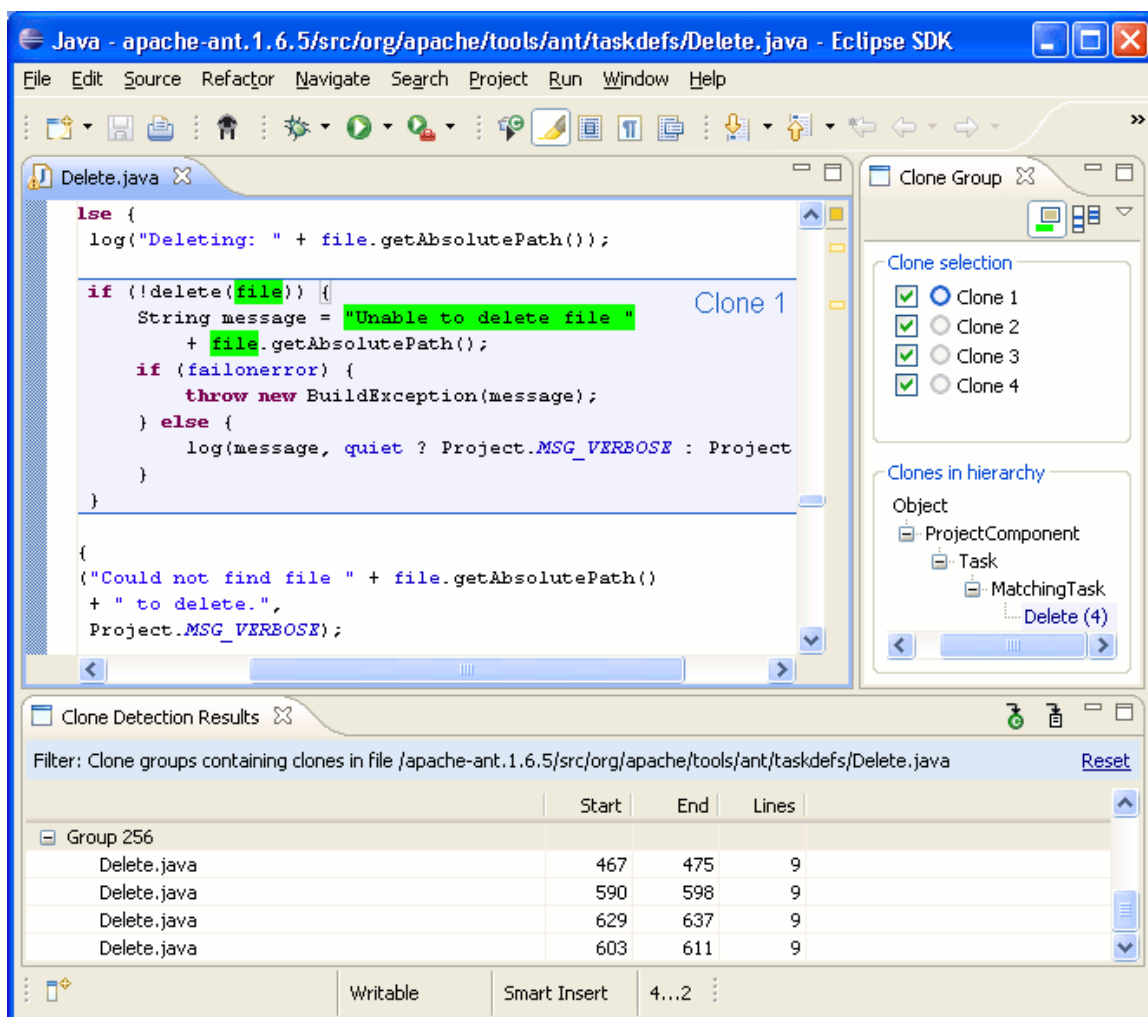


Figure 3.10 – Localized clone representation in CeDAR.

3.2.3 Detecting Clone Similarities and Differences

The process to display the localized representation includes an additional step that is not done in Section 3.1, where the grouping of clones and starting and ending lines from the tool results provide sufficient information to visualize the clones. In order to process the similarities and parameterized differences among the clones in a group, each clone in the group must be converted into their corresponding abstract syntax tree (AST) representations. Because CeDAR is an Eclipse plug-in, the Eclipse Java Development Tools (JDT) [Eclipse JDT, 2010] is used in the process. The JDT includes a Java Model with corresponding application programming interfaces (APIs) to parse and navigate the Java program's AST. The boundaries of a clone determine the AST node that represents the clone. In order to identify the AST node that represents a clone, the reported starting and ending lines of a clone is translated into character offsets, because the positions of AST nodes in JDT are stored as offsets with respect to the first character of a source file. After the offsets have been calculated for the clone, an AST node that covers the boundaries signified by the offsets is determined and associated to the clone.

After the AST of each clone in the group is obtained, comparisons can be made on these ASTs to determine the parameterized elements. It is already known that these clones are duplicates of each other from the information retrieved from the clone detection tool. The comparisons here are used to identify predefined parameterized differences among the clones. Comparisons are currently performed on the statement-level sub-trees within the ASTs. For example, for an *If-statement* clone, the first-level statements inside the block of the *If-statement* are the elements that are evaluated. The *If-statement* condition is also evaluated, but the evaluation is done separately.

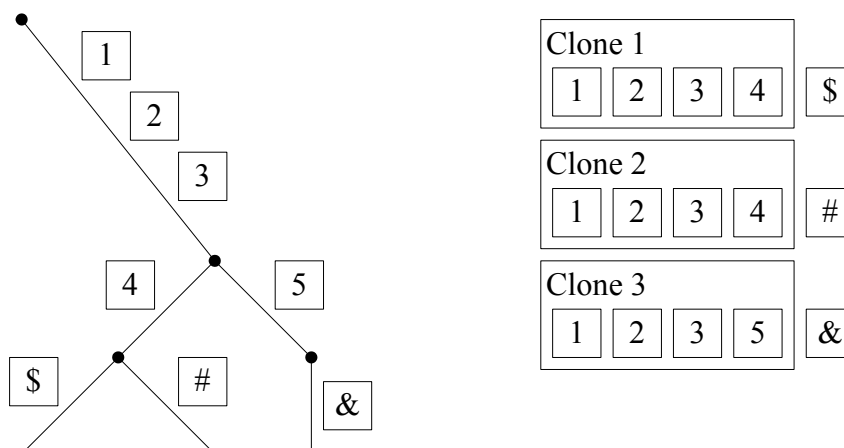


Figure 3.11 – Part of a suffix tree identifying clone similarities and differences.

The suffix tree technique [Gusfield, 1997] is used to determine the levels of similarities among the clones. We have previously used suffix trees in a separate work [Tairas and Gray, 2006] to identify function-level clones, details of which are available in Appendix A. In a suffix tree, a path from the root of the tree to a leaf is generated for each suffix of a given sequence. Sub-sequences that match each other are represented by the same branch or edge in the tree. Identifying such branches in the tree can be used to determine which parts of the sequence are matching sub-sequences. In this work, the sequence is a concatenation of all statements from each clone. The left part of Figure 3.11 illustrates a portion of the suffix tree generated for the statements associated with the three clones on the right part of the figure. The suffix tree is generated from the sequence of concatenated statements from the three clones: *1234\$1234#1235&*. The numbers represent statements in the clones and the special characters (i.e., \$, #, and &) represent terminating identifiers used by the suffix tree process.

By traversing the suffix tree that includes the edges in Figure 3.11, the similarities and differences among the statements can be determined. In this case, the first three statements are matched for all three clones, which are represented by a single edge. The final statement among the clones contains some differences, where the first two clones contain a matched statement and the last clone contains a differing statement, which are represented by the branching of two edges. This is a similar process performed by a clone detection tool that would use a suffix tree during the detection process. However, storing and visualizing the individual similarities and differences among clones is not performed in such clone detection tools.

The display of clone differences can be divided into two cases as can be seen in the example in Figure 3.11. The first case considers the differences of any parameterized elements in the first three statements of the clones. If either *Statements 1, 2, or 3* consist of parameterized elements, then the statement matching process considers the statements to “match,” but identifies the predefined or currently recognized parameterized elements to be displayed to the user. The second case considers the statements that cannot be matched based on the predefined parameterized elements. These differing statements are also identified and are visualized in a different way to the user. The presentation of these two types of differences will be discussed further in Section 3.2.5. The next section describes the different levels of similarity and the process of identifying these levels.

3.2.4 *Similarity Levels*

When two statements from two clones are compared and they match each other, these two statements share the same branch in the suffix tree. For *Type II* clones, the

matching process is relaxed to allow statements that do not exactly match each other to still be considered “matching.” In this work, the matching process compares the AST nodes representing the statements. The AST nodes are compared by calling the `subtreeMatch` API call in the JDT, which compares two sub-trees representing the two statements and their children. To identify the parameterized elements between two statements and the differences between two statements, several similarity levels are considered during the matching process. These similarity levels are described in the following paragraphs. Figure 3.12 outlines the filtering process. The initial matching looks at exact matching nodes. A non-match will compare the nodes for predefined parameterized elements, which is then followed by parameterized elements of non-identical nodes if the nodes still do not match.

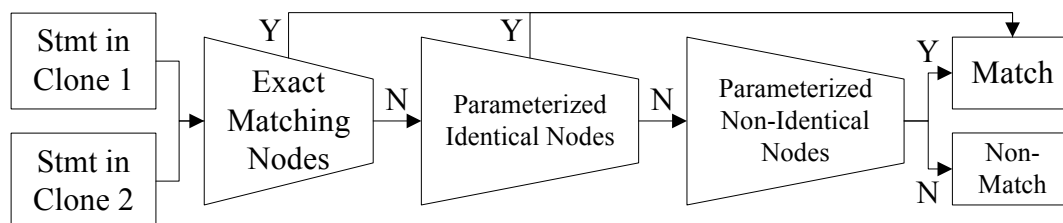


Figure 3.12 – Statement matching levels.

Exact matches. This represents a `subtreeMatch` that utilizes the default matcher as provided by the JDT. The sub-trees that are compared must exactly match each other. Otherwise, the sub-trees are considered not equal to each other. The comparison process performs this type of matching initially to determine whether the two sub-trees are exact matches.

Parameterized matches of identical nodes. At this point, the matcher is customized to allow certain nodes to differ corresponding to the parameterized elements of the clones. The matcher can identify differing values such as variable names and string literals. Currently, the customized `subtreeMatch` in CeDAR allows for differences between two `MethodInvocation`, `NumberLiteral`, `QualifiedName`, `SimpleName`, or `StringLiteral` AST nodes. This allows clones to have parameterized values for method calls, primitive integers, identifiers separated by ‘.’ tokens, simple identifiers, and strings.

In each case where a sub-tree is “matched,” but contains parameterized differences, these differences are stored by mapping the two differing values. The mapping is between the value of the “default” clone and the corresponding clone with the differing value. The mapping provides two properties regarding the parameterized elements. The first property is the different values between the clones, which will be displayed to the user. The second property is the code element in the default clone that needs to be highlighted to signify that the element represents differing values among the clones.

Figure 3.13 illustrates the process where the corresponding statements from four clones are compared using the customized `subtreeMatch`, which allows some parameterized differences. If the allowed parameterized differences are encountered, then `subtreeMatch` will still report the sub-trees as matching, but will also record the mapping between the parameterized elements. The `subtreeMatch` comparisons between *Clone 1* and *Clones 2, 3, and 4* yield three unique mappings: variable `file` in *Clone 1* to variable `f` in *Clones 2 and 3*; variable `file` in *Clone 1* to variable `dir` in *Clone 4*; string

"Unable to delete file " in *Clone 1* to string "Unable to delete directory " in *Clone 4*.



Figure 3.13 – Mappings of parameterized differences.

Parameterized matches of non-identical nodes. CeDAR also allows for differences to be considered as matches for combinations of non-identical nodes. The currently allowable combinations consist of nodes listed in the “Parameterized matches of identical nodes” subsection. For example, a `MethodInvocation` is matched with a `SimpleName` in two AST sub-trees of two clones.

Non-supported parameterized matches / statement differences. This case technically does not represent a similarity, but rather identifies situations when a comparison produces a non-match where two sub-trees are considered not equal to each other. If the matcher cannot identify two corresponding nodes in the sub-trees being compared as matching, then the statements represented by the sub-trees are considered not equal. In this case, the two nodes are not exactly equal and do not represent elements that are currently allowed to be different (i.e., parameterized). The non-matching statement will also be displayed to the user.

```

1: function DISPLAY( $T$ : suffix tree,  $S$ : statement): void
2:    $E \leftarrow \text{GetEdge}(T, S)$ ;
3:   if ( $E \neq \emptyset$ )
4:      $P \leftarrow \text{GetParameterizedPairs}(E)$ ;
5:      $PE \leftarrow PE \cup P$ 
6:     if (!RepresentsAllClones( $E$ ))
7:        $NM \leftarrow NM \cup S$ 
8:     end if
9:   else
10:     $NM \leftarrow NM \cup S$ 
11:  end if
12: end function

```

Listing 3.1 – Determining statement similarities.

3.2.5 Localized Representation in CeDAR

As described in Section 3.2.2, the localized display of a group of clones is generated when the user selects one of the clone groups from the “Clone Detection Results” view. At that point, the suffix tree process outlined in the previous subsection is performed. Because the code associated with the default clone will be used to display the localized representation of all the clones in the group, each statement in the default clone

needs to be determined whether it is an exact match, a parameterized match, or a non-match. This is done by evaluating the generated suffix tree based on the statements of the default clone. After the properties of each statement have been determined, the display of the clone group can be done. The following paragraphs describe the process of visualizing the localized representation in CeDAR.

Obtaining clone similarity properties. The method `DISPLAY` in Listing 3.1 outlines the process of determining how the default clone will be displayed in the source editor. For each statement in the default clone, an edge in the suffix tree that represents the statement is obtained using the `GetEdge` method (line 2). The edge must also represent statements from at least one other clone, which signifies that the statement is matched in at least one other clone in the clone group. If no edge is found, then the statement is considered to have no match with the other clones in the group and is added to the list of non-matched statements (*NM* in line 10). If an edge is found, the process looks for mappings or pairings (i.e., by method `GetParameterizedPairs` in line 4) that signify a parameterized element in the statement (i.e., at least two statements of two clones consist of differing values for a specific element). The mappings are stored in the list of parameterized elements (i.e., *PE* in line 5). It may be the case that not all clones match the default clone's statement. This is determined by whether the edge represents all clones in the clone group (i.e., by method `RepresentsAllClone`). If a statement does not match all clones, then it is added to *NM* (line 7).

The `GETEDGE` method in Listing 3.2 searches the suffix tree to find an edge containing the statement of the default clone. The process recursively evaluates each edge

of the tree starting from the top-most edges. The method `StmtInEdge` looks to see if the given statement is part of the edge that is being evaluated (line 3). If not, then the branches of that edge are recursively evaluated (i.e., by the method `RecursiveStmtInEdge` in line 9). If the edge contains the statement, then it is evaluated to determine whether it also represents two or more clones (i.e., by the method `RepresentsMultipleClones` in line 4). This is determined by looking at the branches of the edge to see if at least two branches represent sequences of statements of two separate clones. For example, in Figure 3.11, the top-most edge represents the first three statements of all three clones, because the edge has branches that represent the three clones, which in this case are the branches for the three special terminating characters (i.e., \$, #, and &). During the suffix tree generation process in this application, when a statement in an edge matches a new statement being compared, the new statement is also stored in the edge. The determination whether an edge represents at least two clones can be done by evaluating the matching statements that are stored in the edge.

```

1: function GETEDGE(T: suffix tree, S: statement): edge
2:   for each E in T do
3:     if (StmtInEdge(E, S))
4:       if (RepresentsMultipleClones(E))
5:         return E;
6:       end if
7:     else
8:       for each B in GetBranches(E) do
9:         return RecursiveStmtInEdge(B);
10:      end for
11:    end if
12:  end for
13: end function

```

Listing 3.2 – Determining matching statements.

Displaying clones in one location. The examples in this subsection represent clones found in Apache Ant 1.6.5 [Apache Ant, 2010]. In CeDAR, the localized representation is visualized directly in the source editor. An example can be seen in Figure 3.14 where the section of the default clone, which in this case is *Clone 1*, is highlighted in light blue and bordered by two horizontal lines in the figure. The sections of all other clones in the same file are highlighted in light grey, which is not shown in the figure.

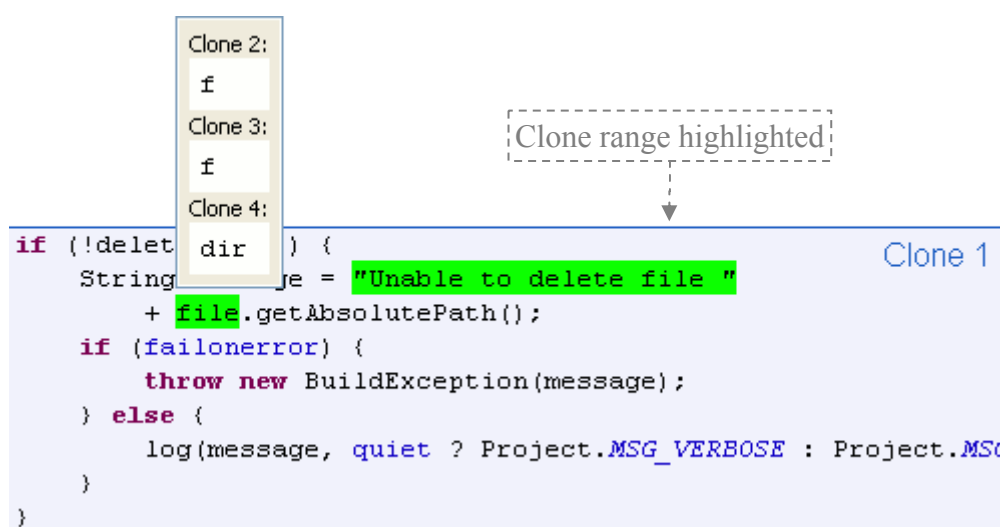


Figure 3.14 – Pop-up of simple variable differences.

For all parameterized elements in *PE*, the corresponding code element in the default clone is highlighted. These elements are highlighted in neon green. Hovering the mouse over one of these elements will invoke a pop-up containing the parameterized differences associated with that code element. In Figure 3.14, the parameterized differences of the `file` variable are visualized in the pop-up. In this case, *Clones 2 and 3* use a different variable name (i.e., `f`). *Clone 4* uses the variable name `dir`.

Non-matching statements highlighted!

```
String classname = (String) e.nextElement();
String location
    = classname.replace('.', File.separatorChar) + ".class";
File classFile = new File(config.srcDir, location);
if (classFile.exists()) {
    checkEntries.put(location, classFile);
    log("dependent class: " + classname + " - " + classFile,
        Project.MSG_VERBOSE);
}
```

Clone 1

(a) Clone 1

```
String classname = (String) e.nextElement();
String filename = classname.replace('.', File.separatorChar);
filename = filename + ".class";
File depFile = new File(basedir, filename);
if (depFile.exists() && parentSet.containsKey(filename)) {
    // This is included
    included.addElement(filename);
}
```

(b) Clone 2

Figure 3.15 – Highlighted statement differences.

For all non-matching statements in *NM*, the corresponding statement in the default clone is highlighted. In CeDAR, these statements are highlighted in dark grey. Figure 3.15 provides an example of the highlighting of statements that are not equal. In Figure 3.15(a), the second statement is highlighted (i.e., in dark grey). The differences can be seen as compared to the corresponding clone in Figure 3.15(b), where the second and third statements in *Clone 2* perform the task that was only done by one statement in *Clone 1* (i.e., the second statement). In addition, the differences between the fourth statement in *Clone 1* and the fifth or last statement in *Clone 2* are more profound. The only exact matching statement is the first statement. The third statement in *Clone 1* is a parameterized match with its corresponding statement in *Clone 2*, where it consists of

parameterized elements of identical node types (e.g., between variables `classFile` and `depFile`). A non-identical node type match is also evident (i.e., between `config.srcDir` and `baseDir`). This example demonstrates the various display properties for the clones, starting from exact matching statements to non-matching statements.

Figure 3.16 is an example scenario where the display of parameterized elements is within the display of a non-matching statement. The four clones in the clone group associated with this display consist of two separate sub-groups where each sub-group represents a tighter similarity. In this case, *Clone 1* (in Listing 3.3(a)) and *Clone 4* (in Listing 3.3(d)) are more similar to each other than with *Clone 2* (in Listing 3.3(b)) and *Clone 3* (in Listing 3.3(c)), and vice versa. The parameterized constant `COMMENTS_KEY` in *Clone 4* is visualized in the pop-up as the only clone with a parameterized difference (i.e., with the constant `CONTAINS_KEY` in *Clone 1*). The statement is not equal for the remaining clones and hence *Clones 2* and *3* are also listed in the pop-up as non-matches.

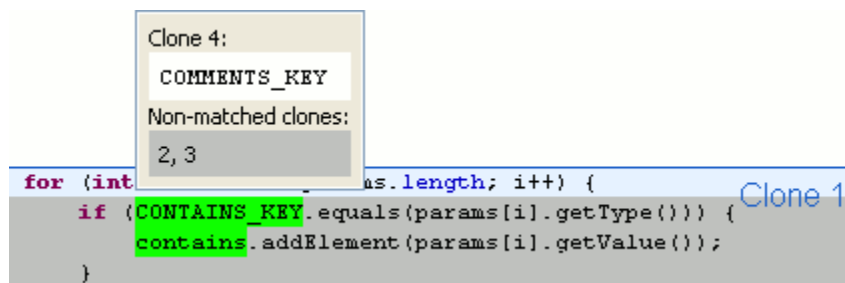


Figure 3.16 – Display of two sub-groups of parameterized clones.

```

for (int i = 0; i < params.length; i++) {
    if (CONTAINS_KEY.equals(params[i].getType())) {
        contains.addElement(params[i].getValue());
    }
}

```

(a) Clone 1

```

for (int i = 0; i < params.length; i++) {
    if (PREFIX_KEY.equals(params[i].getName())) {
        prefix = params[i].getValue();
        break;
    }
}

```

(b) Clone 2

```

for (int i = 0; i < params.length; i++) {
    if (LINE_BREAKS_KEY.equals(params[i].getName())) {
        userDefinedLineBreaks = params[i].getValue();
        break;
    }
}

```

(c) Clone 3

```

for (int i = 0; i < params.length; i++) {
    if (COMMENTS_KEY.equals(params[i].getType())) {
        comments.addElement(params[i].getValue());
    }
}

```

(d) Clone 4

Listing 3.3 – Clone sub-groups.

3.2.6 Clone Properties Based on Visualizations

This section illustrates scenarios where clone group properties can be obtained based on the visualizations provided by CeDAR.

Summary of differences of closely located clones. In Figure 3.17, three clones are located closely together, which does allow for evaluating the clones at the same time in the source editor. However, the representation of the clones in the default clone (i.e., section of code highlighted in blue) provides a centralized summary of the differences among the clones. This summary shows the code elements that differ and the remaining parts that

are identical without the need to determine such properties by evaluating each clone. As it relates to the task of removing the duplication associated with the clones, the clone group representation provides a quick summary of which parts of the clones differ. This knowledge is useful in the refactoring process of clone removal as the complexity of the clones can be observed in the single representation. For example, clones containing only renamed local variable names can be considered easier to refactor.

```

System.out.println("Extensions Supported By Library:");
for (int i = 0; i < available.length; i++) {
    final Extension extension = available[ i ];
    System.out.println(extension.toString());
}

(0 != required.length) {
System.out.println("Extensions Required By Library:");
for (int i = 0; i < required.length; i++) {
    final Extension extension = required[ i ];
    System.out.println(extension.toString());
}

(0 != options.length) {
System.out.println("Extensions that will be used by Librar");
for (int i = 0; i < options.length; i++) {
    final Extension extension = options[ i ];
    System.out.println(extension.toString());
}

```

Figure 3.17 – Summary of clone differences.

Small clone differences. Each clone in a group may represent many lines of code. These lines need to be evaluated to determine the extent of similarity and differences among the clones. However, in some cases the difference among the clones is very small. For

example, in Figure 3.18 the two class-level clones only differ in two string values. The representation can show the extent of how different the clones are in one location.

```

public class P4Delete extends P4Base {
    /**
     * number of the change list to work on
     */
    public String change = null;
    /**
     * An existing changelist number for the deletion; optional
     * but strongly recommended.
     * @param change the number of a change list
     */
    public void setChange(String change) {
        this.change = change;
    }
    /**
     * executes the p4 delete task
     * @throws BuildException if there is no view specified
     */
    public void execute() throws BuildException {
        if (change != null) {
            P4CmdOpts = "-c " + change;
        }
        if (P4View == null) {
            throw new BuildException("No view specified to delete");
        }
        execP4Command("-s delete" + P4CmdOpts + " " + P4View, n
    }
}

```

Figure 3.18 – Small clone class differences.

Excluding clones – CeDAR allows the display of only a select number of clones in a clone group. Checkboxes, as seen in the “Clone Group” view in Figure 3.10, allow a programmer to deselect one or more clones, which will remove the representation of the deselected clones. In Figure 3.19, almost the entire clone is highlighted as parameterized. Hovering over Enumeration displays a pop-up that shows the remaining five clones consist of Iterator instead. Reviewing the other parameterized elements suggests that the default clone (*Clone 1*) is different from the other clones in the group. Deselecting this clone from the list of clones and selecting *Clone 2* as the default clone yields a representation without any parameterized elements. This suggests that excluding *Clone 1*, all other clones are exact matches of each other.

```

for ba      atibility
recat      oache.tools.ant.util.JAXPUtils#getSystemId ins
ted St     mId(File file) {
urn J      ystemId(file);

fic co     for the TRaX liaison.
am xsl    _TProcess task instance from which this liaison
is t      ed.
void      _TProcess xsltTask) {
Proces    ctory = xsltTask.getFactory();
(factor   {
setFac    .getName ();
// con    y attributes
for (Enumeration attrs = factory.getAttributes();
      attrs.hasMoreElements();) {
  XSLTProcess.Factory.Attribute attr =
    (XSLTProcess.Factory.Attribute) attrs.nextElement ();
  setAttribute(attr.getName (), attr.getValue ());
}

```

Figure 3.19 – Sub-group with higher similarity.

3.2.7 Evaluation

The instances where the centralized representations can visualize parameterized elements or statement differences were evaluated on multiple open source software artifacts. Parameterized elements represent differences that are acknowledged by CeDAR, such as variable name differences. Statement differences represent statements that differ syntactically. The differences may also contain parameterized elements that are not currently supported by CeDAR (i.e., AST nodes not listed in Section 3.2.4). The clones were detected using Deckard, a tree-based clone detection tool that reports syntactically meaningful clones. In this case, the reported clones represent clearly separated statements. The similarity value was set to 0.95 to allow non-exact clones, including parameterized clones. The evaluation considers the number of clone groups that can be represented appropriately by the localized representation in CeDAR.

The evaluation considered several open source Java projects. These projects are also used in the evaluation of refactoring support in Chapter 5. The version selected for

each project represents a recent version with the source code available for download. Apache Ant is a Java library that assists in the building of Java applications [Apache Ant, 2010]. ArgoUML is a tool for developing Unified Modeling Language (UML) models [ArgoUML, 2010]. Apache JMeter is used to test the performance of a wide range of software applications [Apache JMeter, 2010]. JBoss AOP is a middleware framework that supports the Java 2 Platform, Enterprise Edition (J2EE) with aspect-oriented concepts [JBoss AOP, 2010]. JFreeChart is a Java library that allows the development and display of charts in applications [JFreeChart, 2010]. JRuby [JRuby, 2010] is a Java implementation of the Ruby programming language [Ruby, 2010]. The Eclipse Modeling Framework (EMF) allows for the development of tools based on structured data models and the generation of code based on these models [EMF, 2010]. jEdit provides an editing environment for programmers [jEdit, 2010]. Squirrel SQL is a client program that allows for the management of Java Database Connectivity (JDBC) compliant databases [Squirrel SQL, 2010].

Four different scenarios were considered. The first scenario is when a clone group consisted of exact matching clones (i.e., *Type I* clones). In this case, the localized representation will not show any differences among the clones. The second scenario is when a clone group consists only of recognized parameterized elements, which are listed in Section 3.2.4. The third scenario is when a clone group consists of statement differences. The final scenario is when a clone group consists of both parameterized elements and statement differences (e.g., as seen in Figure 3.15).

Table 3.1

Clone types identified by CeDAR in programs

Project	#CG	Exact (%)	Param (%)	StmtDiff (%)	Mixed (%)
Apache Ant 1.6.5	429	61 (14%)	152 (35%)	131 (31%)	85 (20%)
ArgoUML 0.26	650	61 (9%)	214 (33%)	124 (19%)	251 (39%)
EMF 2.4.1	285	54 (19%)	136 (48%)	52 (18%)	42 (15%)
Jakarta-JMeter 2.3.2	377	77 (20%)	158 (42%)	71 (19%)	71 (19%)
JBoss AOP 2.1.5	159	51 (32%)	81 (51%)	14 (9%)	13 (8%)
JEdit 4.2	345	91 (26%)	120 (35%)	88 (26%)	46 (13%)
JFreeChart 1.0.10	847	151 (18%)	415 (49%)	168 (20%)	113 (13%)
JRuby 1.4.0	318	113 (36%)	70 (22%)	63 (20%)	72 (23%)
Squirrel-SQL 3.0.3	428	78 (18%)	164 (38%)	70 (16%)	116 (27%)

#CG = Total clone groups

Exact = Clone groups with exactly matching clones

Param = Clone groups with parameterized clones

StmtDiff = Clone groups with non-supported parameterized clones or near-exact clones

Mixed = Clone groups consisting of both “Param” and “StmtDiff” type instances

Instances of the first two scenarios can be fully represented and visualized by CeDAR. When the clones are exactly the same, then no annotations are needed. When the clones only contain recognized parameterized elements, then the representation of the clones can be summarized accordingly in the visualization. The last two scenarios represent instances where CeDAR cannot fully identify the differences of the clones. This is especially the case for the non-matched statements, as these non-matches can signify several properties related to the differences of the clones. In these cases, CeDAR currently reports that the statements do not match, but does not provide more information about the reason for the non-match.

Table 3.1 documents the instances after running a batch process that looked at each clone group to determine what type of scenario is related to the clone group. The

first scenario (“Exact” column) occurred considerably. The second scenario (“Param” column) occurred in the majority of the groups except for ArgoUML and JRuby. In fact, in four of the artifacts (i.e., Jakarta-JMeter, JBoss-AOP, JFreeChart, and EMF) this scenario occurred in approximately half of all the instances. The third (“StmtDiff” column) and fourth (“Mixed” column) scenarios consist of statement differences. Only in ArgoUML did the number of instances containing statements that could not be matched by CeDAR (i.e., “Mixed” column) exceed the number of instances of parameterized elements that are currently recognized (i.e., “Param” column).

The evaluation from Table 3.1 suggests that in the majority of the cases, the clone groups consisted of parameterized elements that are currently recognized and supported by CeDAR. For the cases where statements could not be matched, there is a possibility of eliminating these non-matches by supporting more parameterized elements. Further evaluation of these instances can help determine the additional parameterized elements that can be included for the localized representation.

3.2.8 Discussion

This subsection describes some points for consideration related to the representation and visualization process described in the previous subsections.

Initial clone detection tool process. As stated in Section 3.2.2, our process is considered a “second pass” on the code associated with the clones that were initially reported by a clone detection tool. Several tools have utilized the suffix tree technique to find clones [Baker, 1995] [Falke *et al.*, 2008] [Kamiya *et al.*, 2002] [Tairas and Gray, 2006].

However, in the process used by these tools, the actual values of the identifiers are ignored as long as the identifier token or node matches. In some cases, the actual identifier values are only evaluated during a post-processing stage to determine the type of the clone (i.e., exact or parameterized). Incorporating the identification of the parameterized elements that can be used in the display of the clones within the clone detection process itself can remove the need for a second pass on the code. An alternative is to perform post-processing on the clone detection results directly within the tool rather than working with the textual output of the tool. However, this removes the flexibility of utilizing the technique on different clone detection tools.

Version clones. The existence of multiple versions of a class file in a version control system can give rise to what we call “version clones.” Visualizations of code changes based on version history are described in Section 3.4. Our focus is on “snapshot clones,” which are clones detected by a clone detection tool in a single snapshot of the source code. The question that arises is what are the characteristics of version clones as compared to snapshot clones? If we look at the techniques associated with version clones, they focus mainly on changes on a class as a whole and the differences between two versions of the class. In contrast, in our representation, clones can vary from statement-level, method-level, and also class-level clones. In addition, the mechanism allows for comparison of two or more clones. However, an interesting exercise would be to extend the techniques for version clones to support snapshot clones, and vice versa.

Limitations. A particular limitation of our technique is that it generates a suffix tree on the first-level statements associated with the detected clones. This presents a fixed granularity level during the visualization of the clones. The disadvantage in this case can be seen when the statement consists of multiple nested levels. When the first-level statements cannot be matched, then the entire statement is displayed as a non-match even though the difference may actually reside several levels below the top-level statement.

A further limitation is seen in the results of the evaluation in Section 3.2.7, specifically related to the display of non-matched statements. The reason for non-matches may be due to unsupported parameterized elements. Increasing the support for additional parameterized elements can potentially reduce the number of non-matched statements that are displayed.

3.3 CoCloRep: A DSL for Code Clone Representation

This section describes an investigation into the development of a DSL for code clones called CoCloRep [Tairas *et al.*, 2007]. This representation of clone groups is more textual in nature as compared to the visual representations described in Sections 3.1 and 3.2. This section provides an initial assessment to determine the benefits of MDE in the context of clone analysis. Instances of CoCloRep represent software clones and their explicit representation as model instances. Through the transformations of these models, information about the clones can be retrieved and analyzed. Two assumptions are made related to the input of CoCloRep. The first assumption is the existence of a clone detection tool that can detect the first three types of clones as described in Section 2.1 or

a combination of them. The second assumption is the ability to generate an instance that conforms to the grammar of the DSL from the results of a clone detection tool.

3.3.1 Overview of CoCloRep

CoCloRep is designed to represent *Types I-III* clones. The example of two clone instances in Listing 3.4 illustrates near exact matching clones with parameterized variables. This example will be used to describe the language elements of the CoCloRep DSL. The sequence of statements in *Clone 1* and *Clone 2* are the same with the exception of line 3 in *Clone 1* making the clones near exact matches. In addition, each clone uses two different sets of variables for the same code sequence. That is, variables f and g in lines 1, 2, and 4 for *Clone 1*, and variables p and q in lines 1-3 for *Clone 2*. These variables are the parameterized variables.

Clone 1:	Clone 2:
1: int g;	1: int q;
2: int f = g + 3;	2: int p = q + 3;
3: i = i + 1;	3: c = p + m;
4: c = f + m;	

Listing 3.4 – Example of two clone instances.

CoCloRep uses two elements to represent clones called *clone instances* and *clone groups*. Once the clones have been represented, an additional element called a *clone command* is used to perform actions on the clone instances and groups. The three elements are described further in the following paragraphs.

Clone instances. Each clone that is detected is represented by a clone instance. These instances in turn are associated with a clone group. In addition to the clone group association, parameterized variables are “passed” to the clone group using a function parameter-like syntax. Also, source code that is distinct to a specific clone instance is included in the instance body.

In Listing 3.5, instance *r* is declared for *Clone 1* in lines 1-5 and instance *s* is declared for *Clone 2* in line 7. These lines include the associations to the clone group *cg* and the two sets of actual variables that are “passed” to *cg*. The extra statement in *Clone 1* is represented in lines 2-4 inside the *t* element, which is associated with a box defined in *cg*.

```

1: instance r = cg(f, g) {
2:   t {
3:     i = i + 1;
4:   }
5: };
6:
7: instance s = cg(p, q);

```

Listing 3.5 – Clone instances in CoCloRep.

Clone groups. Clone groups represent the common properties shared among related clone instances. Properties that are distinct are handled through “placeholder” variables and “boxes” for additional code. To accommodate parameterized matching clones, a clone group can contain placeholder variables that can be swapped with the actual variables associated to a specific clone. In the clone group, these variables are prefixed with a ‘\$.’ To represent near exact matches, the clone group can contain boxes that can be filled by

code that is not found in all the clones of the clone group. Listing 3.6 provides an example of a clone group called `cg`.

```
1: clone cg($a, $b) {  
2:   int $b;  
3:   int $a = $b + 3;  
4:   {{ t }}  
5:   c = $a + m;  
6: }
```

Listing 3.6 – A clone group in CoCloRep.

In the declaration of the clone group `cg`, lines 2-5 contain the source code associated with `cg`. Two placeholder variables, `$a` and `$b`, are defined in line 1 and are used inside the source code of this clone group. These variables are associated with the actual variable names that are passed from the clone instance declarations. A box denoted by double curly brackets in line 4 represents additional sections of code that do not occur in all clone instances of the group.

Clone commands. After the clones have been represented through clone instances and groups, they can be “probed” for analysis purposes. In this context, clone commands are used to perform a specific activity on the clones. One command that has been implemented is the “variables” command. Executing this command on a clone group will provide information about the variables associated with the group. In the context of refactoring, information about the existence of variables that are both defined inside and outside a clone is needed to determine appropriate refactoring strategies [Higo *et al.*, 2004]. For example, it is easier to perform the *Extract Method* [Fowler, 1999] refactoring

task if all variables represented by a clone group are also declared within the clone group. The *variables* command separates the variables in a clone group into different categories. Variables can be declared inside the clone group, declared outside the group and assigned a new value or used only in an expression. The output of the command “variables cg” is given in Listing 3.7 (original layout simplified). The placeholder variables *a* and *b* are declared inside *cg*. The variables *c* and *i* are variables not declared in *cg*, but are assigned new values in *cg*. Note that variable *i* is only found in clone instance *r*. Variable *m* is an externally declared variable that does not change in *cg*.

```

1: Variable information for clone group cg
2: Declared variables:
3:   b
4:   a
5: Outside assigned variables:
6:   c
7:   i (in instance r)
8: Outside non-assigned variables:
9:   m

```

Listing 3.7 – Example output of *variables* command.

An *expand* command is also available, which “expands” a clone instance and generates the original code that a clone instance represents. This command was a command initially developed to test the process of obtaining information about clones in this context. It would not be used in the analysis of clones, per se. Additional commands can be considered that would provide analysis results on a specific clone group. For example, as an extension of the *variables* command, a *pre-conditions* command could determine whether a clone group satisfies the necessary pre-conditions for a specific

refactoring. Similarly, a *preview* command could output a preview of a modularized version of the duplicated code associated with the clone group. Such commands are typical in the refactoring process. In this case, the commands would be performed using model transformations, which can include declarative rules as opposed to fully imperative rules written in a general-purpose language. The transformation process in CoCloRep is described further in Section 3.3.4.

3.3.2 Implementation in the AMMA Platform

CoCloRep was developed using the AMMA platform, where the KM3 and TCS DSLs are respectively used for the specification of the abstract and concrete syntax of CoCloRep. ATL is used to generate output from the commands described in the previous subsection (i.e., *variables* and *expand*) through model transformations.

An overview of the process is displayed in Figure 3.20. The vertical dashed lines separate different “technical spaces” (TSs). In the graph, the technical spaces separate the elements between the EBNF technology and the modeling technology. Both the source (CoCloRep) and target (output of commands) are in the EBNF TS, because they use a textual DSL that is based on an EBNF grammar. The main process is done in the MDE TS, which requires transitions or projections from the EBNF TS to the MDE TS. As stated in Section 2.2.1, in AMMA, such projections are called *injections*. The right side of Figure 3.20 shows a projection from the MDE TS to the EBNF TS. This type of projection is called an *extraction*. The elements in the graph are also separated by a horizontal solid line into three levels: M1, M2, and M3. Following the MDE paradigm, the elements in M1 are called terminal models, which are the representation of an entity

in a particular system that is being observed. In this case, the entity being represented is a group of clones in the source code and commands performed on these clones. Terminal models conform to their respective metamodels in M2. Metamodels consist of the important abstractions of a domain as stated in Section 2.2.1. M3 consists of meta-metamodels (one per TS), which conform to themselves.

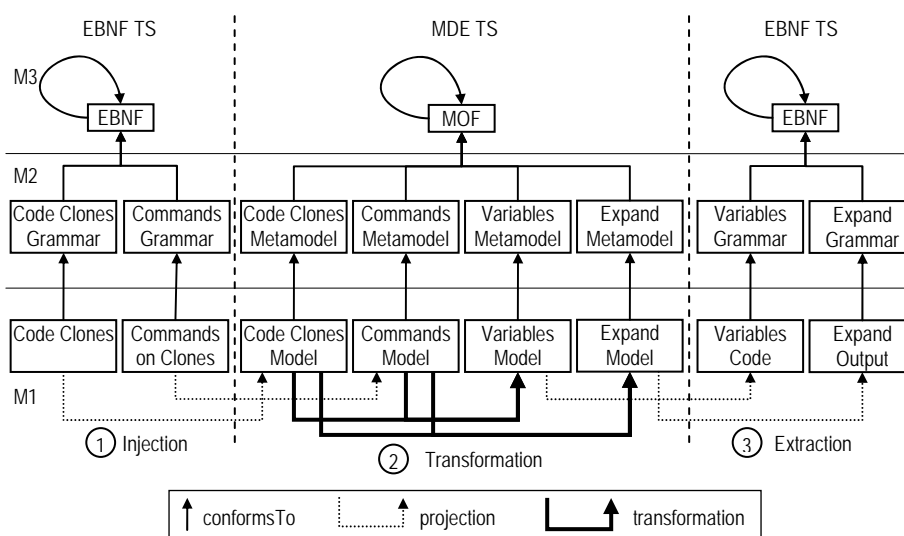


Figure 3.20 – Model transformation process applied to clone representation.

3.3.3 Metamodel Specification

The EBNF technical space on the left of Figure 3.20 consists of the code clones and the commands to be performed on them. These two are separated to allow different command sequences to be generated from one representation of clones. In order for processing to be performed in the modeling space, both the clone representation and the commands are injected into the MDE technical space (step 1 of Figure 3.20). The injection produces models that conform to the *Code Clones* and *Commands* metamodels.

In AMMA, these metamodels are specified in KM3 and correspond to the abstract syntax. The concrete syntax is specified using TCS. The code snippets below provide an example of these specifications.

```

1: class CloneGroup extends LocatedElement {
2:   attribute cloneName : String;
3:   reference parameters[1-*] container : Variable;
4:   reference statements[*] container : Statement;
5: }
```

Listing 3.8 – A snippet of CoCloRep’s abstract syntax.

Listing 3.8 provides a snippet of the abstract syntax of CoCloRep (written in KM3). The `CloneGroup` element consists of a clone group name (line 2), a set of parameters (line 3), and a set of statements (line 4). The clone group name is of primitive type `String`. The parameters consist of `Variable` elements that represent placeholder variables in the clone group. The statements consist of `Statement` elements that represent the statements associated to the clone. The specification of the `Variable` and `Statement` elements are not shown.

```

1: template CloneInstance context addToContext
2: : "instance" instanceName "=" cloneName{refersTo = cloneName}
3:   "(" arguments{separator = ","} ")"
4:   (isDefined (boxes) ? "{" boxes{separator = ","} "}" )
5:   ";"
```

Listing 3.9 – A snippet of CoCloRep’s concrete syntax.

Listing 3.9 provides a snippet of the concrete syntax of CoCloRep (written in TCS). The `CloneInstance` template defines the concrete syntax for a clone instance. The clone group associated to a clone instance is linked in line 2 through a reference to its clone group name. The existence of boxes in a clone instance is determined in line 4. This allows instances to contain zero or more boxes.

3.3.4 Model Transformations

The heart of the process is the model transformations that produce output based on the commands that are given on the clone representation. The output of the two types of commands described earlier (*variables* and *expand*) are also specified and implemented using KM3 and TCS. The transformations are between *Code Clones* and *Commands* models as the source models and *Variables* or *Expand* models as the target models (step 2 of Figure 3.20). The model transformations are defined using another DSL of AMMA called ATL. A set of rules that define the transformation between model elements of the source and target models provide the structure of the transformation as a whole. Code snippets of the model transformation of the *expand* command are given in the following paragraph.

```

1: lazy rule AssignStat {
2:   from s : Clones!AssignStat
3:     to t : Expand!AssignStat (
4:       variable <- thisModule.processVar(s.variable),
5:       initExp <- thisModule.processExp(s.initExp)
6:     )
7: }
```

Listing 3.10 – A transformation rule snippet for the *expand* command.

Listing 3.10 provides a snippet of a transformation rule for the *expand* command. The code snippet contains a rule for the transformation of `AssignStat` elements. This transformation rule is very simple, where the variable and expression associated to an assignment statement are transformed from the source *Code Clones* model to the target *Expand* model (lines 4-5). However, placeholder variables must be replaced by their corresponding actual variables. This is done by processing the variables and expression first through the helper functions called `processVar` and `processExp`. After the transformation is complete, the output of the commands is displayed by extracting the models back into the EBNF technical space (step 3 of Figure 3.20).

3.3.5 Observations

CoCloRep enables clones to be processed in the modeling space through specified model transformations. Processing only in the Grammarware space (staying solely in the EBNF technical space) would also require a representation of clones. One possible representation is the AST of the program. Sub-trees of the AST that represent the clones would be retrieved and processed for analysis. The representation in CoCloRep can be considered as a modified AST that can reduce the number of duplicate branches of the clones in the original AST. This requires the syntax of the original programming language to be part of CoCloRep, and thus must be specified in the metamodel of CoCloRep. In other words, clone representation of a group of clones in CoCloRep is unified into one representation, but this requires the grammar level specifications of a language to be included in the modeling level specifications as well. The boundaries

between the two levels are blurry and therefore it has to be ensured that complexities of the lower level do not hinder or slow down the process at the higher level.

The primary mechanisms of MDE are the transformations that occur between models. An ideal transformation considers each element from the source model and transforms it into an element in the target model. This is usually not the case as two models may differ greatly. Such a case can be seen in the transformations described in Section 3.3.4. If *expand* commands were given to five instances of a clone group, then the target model would contain five copies of the source model. The *variables* command is even more awkward, because it provides variable information, which is just a small part of the source model. Nevertheless, both commands have been implemented using transformations and they demonstrate the feasibility of processing clone representations to obtain specific information about the clones at the modeling level. Other developed commands related to the analysis of clones may not exhibit such characteristics and may feel more natural in these types of transformations. Furthermore, these commands are intermediary parts of an ultimate goal of the refactoring of clones, which fits appropriately in the MDE paradigm, because a source model can be viewed as the original source code that is then transformed to generate a target model representing the refactored source code.

3.4 Related Work

One of the earliest uses of visual representation of clones was actually part of the detection process. A table called a dot plot, where each line in the source code is represented by one column and one row, was utilized in [Church and Helfman, 1993].

Lines that are identical are marked with a dot in the cell representing those two lines. Clones are identified when a group of dots form a line parallel to the diagonal. Dot plots or scatter plots can also be seen in [Baker, 1992], [Kamiya *et al.*, 2002], and [Wettel, 2004]. However, in these cases the clones are detected in a different manner and the scatter plot is used only as a graphical representation. CCFinder provides this visualization as one of the features of the tool. Scatter plots can provide a useful overall view of sections of the source code that contain clones or even where clones are more concentrated. However, at the file level, scatter plots pose a challenge in the display of clones with specific properties (e.g., ubiquitous clones) due to their clone representation using diagonal sequences of dots and single color presentation. More recently, scatter plots were used to show cloning in multiple versions of a program by incorporating heat maps that incorporated coloring [Livieri *et al.*, 2007]. However, the observation was still focused on an overall view of cloning.

In addition to scatter plots, other visualization techniques have been proposed. Polymetric views representing nodes-and-edges graphs with semantic information, such as duplication web views and system model views, are introduced in [Rieger *et al.*, 2004]. Clone information is included in a visualization tool of software architectures and their dependencies [Kapsler and Godfrey, 2005]. More recently, a clone system hierarchy graph was proposed by [Jiang and Hassan, 2007]. Similar to scatter plots, duplication web views in [Rieger *et al.*, 2004] deal with the visualization of clones in general with a goal to understand the cloning situation in a program as a whole. The system model view in [Rieger *et al.*, 2004], and views in [Kapsler and Godfrey, 2005] and [Jiang and Hassan, 2007], provide lower level clone views. However, these views are not contained within a

programming environment such as an IDE, which allows connections and manipulation with the actual code associated with the visualizations.

The clone detection tool CloneDR provides what is called “Clone Abstraction,” which lists the parameterized elements associated to a group of clones. The Hypertext Markup Language (HTML) reports generated by CloneDR include a section that presents in textual format a single representation for a clone that identifies the parameterized elements. More recently, the COBOL edition of CloneDR displays parameterized elements within the Rational Development for System Z (RDz) Environment [(COBOL) CloneDR, 2010]. This work has been done independently from the work described in this section. The localized representations in CeDAR consist of additional capabilities, which include the display of the parameterized elements within the source editor and the display of statements among the clones that are not equal.

Various efforts have been made to visualize the differences between versions of code. The Eclipse Compare editor uses a tokenized version of the Unix *diff* command to show differences between two code segments. Version Editor (VE) [Atkins, 1998] extends the Emacs and vi editors to provide tighter integration with a version control system. To show the changes of the current file with the repository, it uses a text-based comparison algorithm. However, text-based algorithms have several disadvantages over AST-based approaches for comparing source code. For example, language-specific details such as changes in location of methods in the same class are not usually included in text-based comparisons. CSeR (Code Segment Reuse) is a tool to keep track of the copy-paste induced changes and visualizes the detailed differences within Eclipse [Jacob *et al.*, 2010]. The differences are calculated using an AST-based algorithm. Compared to

CeDAR, CSeR is limited to copy-paste induced clones. In addition, support for clone groups (more than two clones) in CSeR is also limited. ChangeDistiller [Fluri *et al.*, 2007] extracts changes from a repository and visualizes the changes using the Compare editor in Eclipse. The changes are extracted from hierarchically structured data, but are limited to class-level changes. Furthermore, comparisons are performed only on multiple versions of the same class.

A generic model for clones is introduced by Giesecke in [Giesecke, 2006]. This model is based on a different definition of clone sets where a set is based on all matching pairs between clones rather than one group that represents all similar clones. Giesecke's model currently only describes or represents clones. Analysis is delegated to external means, whereas in our work, analysis through the MDE paradigm is one of the main purposes in addition to representing the clones. It is worth noting that the model proposed by Giesecke does strive to separate clone description from clone detection to allow different detection algorithms to use one generic model.

3.5 Summary

This chapter has described representations of clones both visually and as models. These representations provide the display of information regarding the clones in the clone group level, which is the level of information that a programmer will consider when evaluating each set of duplicated code. The first visualization technique that extends the AJDT Visualiser provides a new way of visualizing clone detection results in a manner that is observably different from the popular visualization using scatter plots. This visualization in the CloViz plug-in complements the standard text listing and scatter plot

visualization and can help users to identify certain characteristics of the clones by graphically isolating clone groups across a list of source files. Video demonstrations of CloViz can be found at: <http://www.cis.uab.edu/softcom/visual>.

The visualization originally included in CloViz has been incorporated in the CeDAR plug-in as part of the display of clone group information when a clone group is selected by the user. Following the goal of providing clone group information in a more localized manner (as described in Section 3.2), the visualization is included within the view of the source editor. A screen shot of the visualization in CeDAR can be seen in Figure 3.21. In addition, this visualization has been adopted and included by a separate Eclipse plug-in associated with the clone detection feature available in ConQAT (Continuous Quality Assessment Toolkit) [ConQAT, 2010].

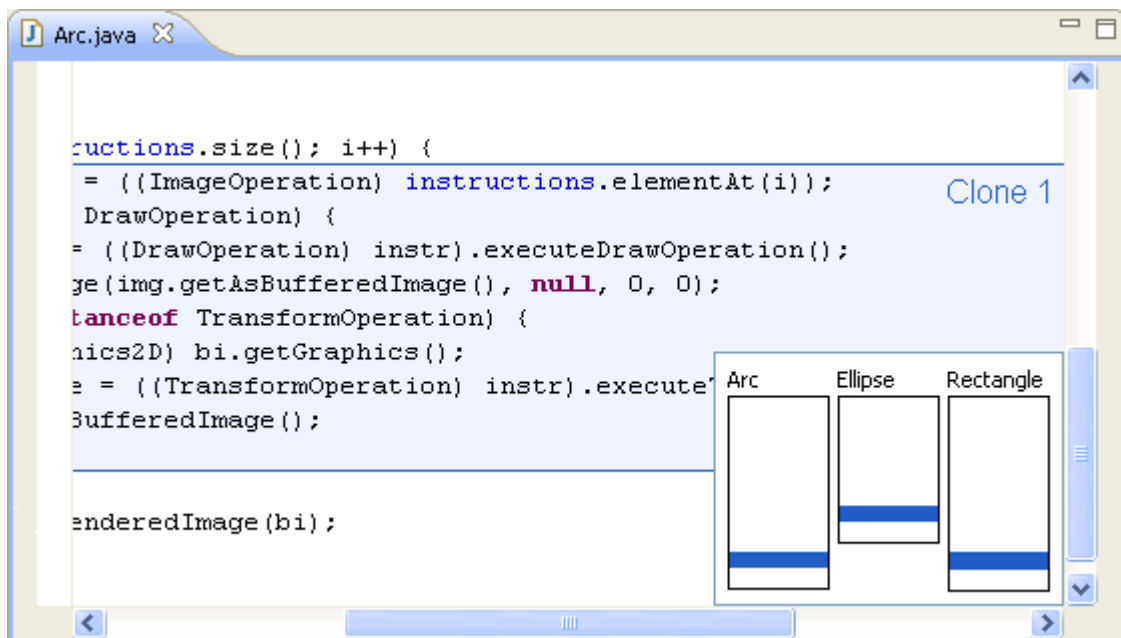


Figure 3.21 – Clone visualization inside source editor view in CeDAR.

The second visualization describes techniques to present and maintain clones in a more localized manner. This functionality is also incorporated in the CeDAR plug-in. The clone group representation allows the display of clone properties for all the clones in a clone group on just one clone instance. The representation of clones in a clone group in one location allows the programmer to learn about the clones without the need to open every occurrence of the clone in the file or application classes. In addition, as it relates to the task of removing the duplication associated with the clones, the clone group representation provides a quick summary of which parts of the clones differ. This knowledge is useful in the refactoring process of clone removal as the complexity of the clones can be observed in the single representation. For example, clones containing only renamed local variable names can be considered easier to refactor.

The final representation is realized through CoCloRep, a DSL that can represent code clones and perform commands on the representation to obtain information for the analysis of clones. Source code representing clones are elevated into models in an effort to utilize model transformations to assist in clone analysis tasks. A command is introduced on clone models as an initial demonstration of clone analysis at the modeling level. Future extensions will be developed to provide further evidence on how clones represented in CoCloRep can be manipulated. A project web site is available at: <http://www.cis.uab.edu/softcom/coclorep>. This initial investigation of MDE techniques through CoCloRep is not currently included in the unified process of clone maintenance that is described in Chapter 5.

The representations described in this chapter can be utilized by a programmer who is considering the comprehension of clones as they relate to a particular clone group.

However, the number of clones and clone groups in a program reported by a clone detection tool can be large. Attempting to understand cloning can be difficult with the potential amount of data. The next chapter describes our contributions to the understanding of clones in large software artifacts, both commercial and open source.

CHAPTER 4

CLONE ANALYSIS

The ability of clone detection tools to find clones in large software systems can potentially yield detection results that are considerable in size. As seen in Table 1.1, the number of clones that are detected will typically increase as the number of lines that are evaluated rise. The number of clones may become a hindering factor in the subsequent analysis of the clones. During analysis, the clones are analyzed to determine why they exist and what actions could be performed to reduce maintenance problems (i.e., what can be done to the clones to help improve the quality of the code and maintain the clones in the future). However, programmers may have difficulty to perform these tasks in terms of comprehending clones if the number of clones reported by a clone detection tool is large. From the examples in Table 1.1, the number of cloned lines discovered by Jiang *et al.* [2007a] in the JDK is about 204K LOC, which is 8% of the total LOC. The 122K copy-pasted segments found by Li *et al.* [2004] represent 15% of the total lines of the Linux kernel code. Both results illustrate the large amount of code identified as clones that can be returned by a clone detection tool that would consequently be part of the manual analysis process.

The existence of large numbers of clones in software systems as detected by a clone detection tool motivates the need to provide mechanisms in which these clones are analyzed with the potential for finding properties of the clones that would be useful for

their comprehension. This chapter presents our work in the analysis of large amounts of clone data. The work considers semi-automated techniques that process the results from a clone detection tool and performs analysis on these results. Section 4.1 introduces a technique that clusters clone groups using an information retrieval technique. The clustering reveals relationships among clone groups that were not originally identified by a clone detection tool and provides a higher level relationship among a collection of clone groups. Section 4.2 describes analysis of clones in multiple versions of the source code. The analysis focuses on how clones are actually refactored based on the changes in the code.

4.1 An Information Retrieval Process to Aid in the Analysis of Code Clones

Several techniques have been proposed to aid programmers in the comprehension of the large number of clones that may be detected in a program. These techniques can be divided into two groups: *classification* and *visualization*. Classification techniques partition clones into different categories based on properties of the clones. A set of clones can be classified based on where each clone is located with respect to the other clones in the hierarchy of files and directories for procedural languages [Kapsler and Godfrey, 2004] and in the type hierarchy for object-oriented languages [Koni-N'Sapu, 2001]. Clones can also be classified based on how different they are [Bellon *et al.*, 2007] or whether the types used are different [Balazinska *et al.*, 2000].

As described in Section 3.4, various techniques have been used to visually represent clones. These include scatter plots in Duploc [Rieger and Ducasse, 1998] and CCFinder [Kamiya *et al.*, 2002], duplication web and system model views in [Rieger *et*

al., 2004], clone system hierarchical graphs in [Jiang and Hassan, 2007], and an aspect browser-like view in [Tairas *et al.*, 2006].

The classification and visualization techniques just described can provide for a better understanding of the clones. Clones that are similar in syntax and structural properties are grouped together through these classification techniques. Subsequent understanding of the clones is based on the relationships among these classified clones and also the overall arrangement of the classification. However, these classification techniques consider only the syntactic and structural similarity properties of the clones. Other properties that are not necessarily syntax- or structure-based can potentially yield further relationships among the clones that can complement the classification techniques. As for the visualizations, they tend to give more of an overall understanding of the clones with respect to the source code as a whole or a section of the source code and are still based on the structural characteristics of the clones. An alternative method is needed to provide associations of clone groups to further aid in their comprehension.

4.1.1 An Information Retrieval Perspective on Code Clone Analysis

The field of information retrieval focuses on the extraction of useful information from a corpus of data. One specific technique in the area of information retrieval is Latent Semantic Indexing (introduced in Section 2.3.1), which is concerned with obtaining relationships among the documents that divide a corpus and the terms in these documents. The relationships are discovered through the analysis of the terms and documents using matrix manipulation techniques [Deerwester *et al.*, 1990]. This technique has already been used in relation to clone detection [Marcus and Maletic,

2001], where high-level concept clones were detected based on the identifiers and comments of the source code. However, the work does not incorporate structure-based clone detection (i.e., detection on lexical token or syntax tree representations), and thus the clones that are identified are more conceptual in nature. In this section, we consider addressing the following key questions: What if LSI is used on the clones that are initially detected by a structure-based clone detection tool? Could LSI be used to associate clone groups (detected by the clone detection tool) that it considers related to each other? Furthermore, could these groups aid in the comprehension of the clone groups, and in turn, the clones that are associated to them? The hypothesis is that LSI can reveal relationships among clone groups that are not based on their initial structure-based similarity properties and these relationships can further assist in the understanding of the initial clones that were identified in a first-pass by a traditional clone detection tool.

This section outlines an effort to determine the benefits of using LSI on the results of a clone detection tool to aid in the understanding and analysis of clones in the Microsoft Windows NT kernel source code [Tairas and Gray, 2009a]. The definition of terms and documents in LSI depends on the domain that it is being used. In this case, the terms are the identifier names of the source code that is represented by the clones and the documents are the clone groups that contain the clones. LSI provides the ability to determine term-term, term-document, and document-document similarities. The objective in this research is to determine the similarities between the documents (i.e., the clone groups). Clone groups that are similar are clustered together and these clusters are analyzed to determine whether they are useful towards the comprehension of the clones.

Figure 4.1 illustrates the hierarchy of the clones, clone groups, and clusters. The clones in a clone group represent duplicated code that is considered similar to each other based on a specific similarity property. These clone groups are in turn clustered to determine relationships among the clone groups, where each clone group contains different groups of duplicated code (or clones). Several relevant tools will be chained together to produce the necessary output for the analysis of the clones.

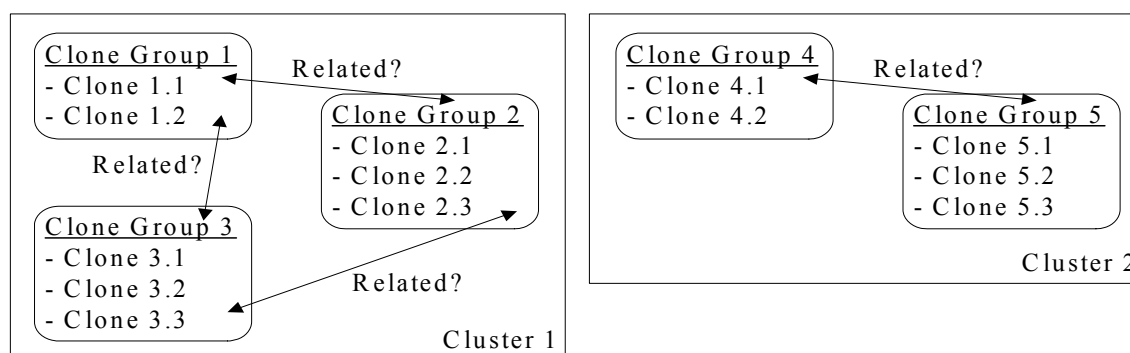


Figure 4.1 – Hierarchy of clones, clone groups, and clusters.

4.1.2 Corpus Description and Analysis Goals

The clones that comprise the corpus are obtained from the source code of the Windows Research Kernel 1.0, which is written primarily in the C language and has been made available by Microsoft to academia for teaching and research [Microsoft Windows Research Kernel, 2010]. Implementations of basic operating system functions (e.g., process and thread support, I/O, and memory management) for the Windows NT kernel are included in the distribution. Some parts that are not included in the research kernel are plug and play, power management, and a virtual DOS machine.

Our goal is to determine the benefits of clustering clone groups based on the LSI information retrieval technique for comprehension purposes. To achieve this, we forego the clustering of all the clone groups at once, but instead divide the clustering and analysis of the clone groups into specific directories. In the NT kernel, the source files representing major OS functionalities are grouped into separate directories. We assert a hypothesis that more interesting clones will be found in specific directories that represent focused OS functionalities. The top-five directories in the NT kernel source containing the most clone groups were selected for this study. To further constrain the clones that are specific to one directory, the clone groups that were selected contain the property that all the clones in the group are located in the same directory. An additional goal is aimed at combining the clone groups from all five directories and clustering them to determine any relationships among the clones from these separate directories. To summarize, we want to see if the clustering of clone groups using LSI aids in the comprehension of the clones in specific directories separately and also in combination. From the above description, the scope of this work encompasses the sections of the Windows NT kernel code that have been determined to be clones. More specifically, the clones that exist in five directories in the NT kernel will be considered as input to the process that represents the contribution of this section.

4.1.3 Clone Group Clustering Process

Figure 4.2 provides an overview of the process. The source code is first searched for clones using a traditional clone detection tool. Then, the identifier names contained in the source ranges of pre-selected clones are extracted and a term-document matrix is

generated. The singular value decomposition of this matrix as described in Section 2.3.1 is computed and the document-document similarity is used to cluster clone groups for further manual analysis. Several third-party tools (represented by rectangles in Figure 4.2) were used to provide the necessary computations. In addition to these tools, some helper programs (represented by rounded rectangles in the figure) were written in Java to enable the integration of each tool. These helper programs are described throughout the following steps.

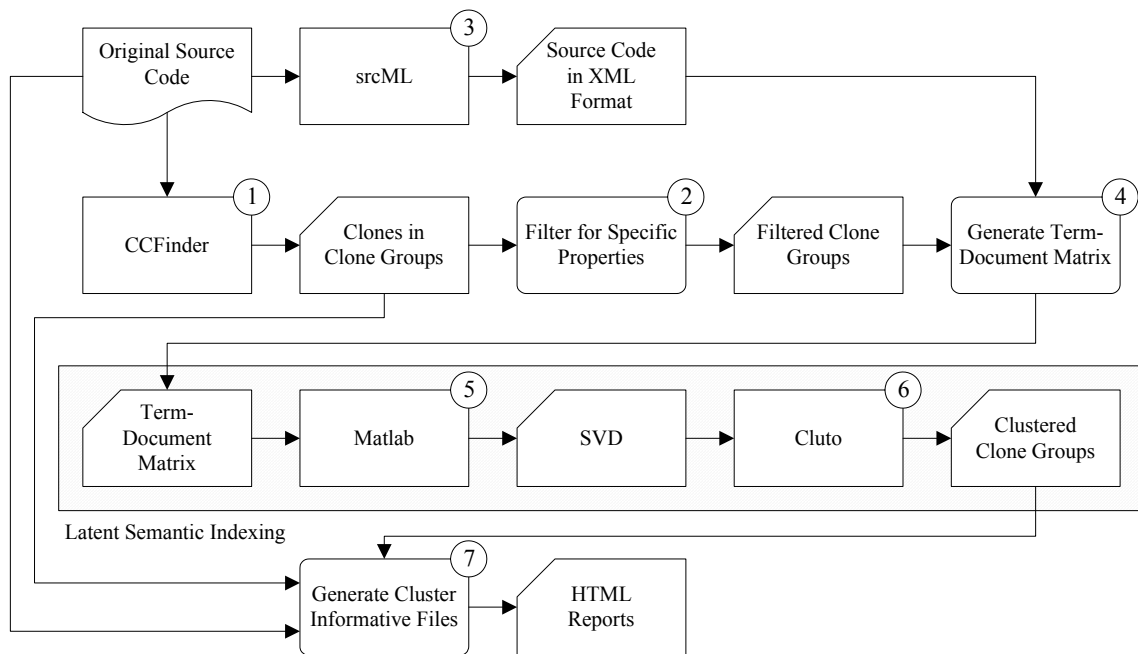


Figure 4.2 – Overview of the clone group clustering process.

Clone detection (step 1). The CCFinder clone detection tool was used to search for the initial clones in the NT kernel. CCFinder uses the token-based representation of the source code to find clones. It is freely available from the Software Engineering Laboratory at Osaka University [SEL, 2010]. CCFinder was selected because of its

availability (i.e., freely available) and ability (i.e., can detect clones for source code in the C language) and not for any specific output properties of the clone detection process. Our technique is open to results from other clone detection tools. The clone information is stored in a database to allow for use in multiple experiments. The database used is Apache's Derby [Apache Derby, 2010], which is a Java-based open source relational database. The information for each individual clone is stored in the database with its association to a clone group.

```

...
clone_pairs {
...
262  46.1806-1857      72.62-113
1471 47.1716-1768     13.9381-9433
...
}

```

Listing 4.1 – Sample CCFinder textual clone representation.

As can be seen in Section 2.1.2, CCFinder provides a textual representation of its results where clone groups are associated with the token ranges of clone pairs. Listing 4.1 is an example of the textual representation of clones given by CCFinder. The first line represents one clone pair of clone group 262. This pair consists of a clone in one file (ID: 46) with token ranges 1806 to 1857 and a clone in another file (ID: 72) with token ranges 62-113. Similarly, the second line represents a clone pair of clone group 1471. This pair consists of a clone in one file (ID: 47) with token ranges 1716 to 1768 and a clone in another file (ID: 13) with token ranges 9381 to 9433. The listing of this information is sorted by the file ID of the first clone in the clone pair (i.e., in Listing 4.1, a clone pair associated to file ID 46 is followed by a clone pair associated to file ID 47). These token

ranges are converted into line numbers, which will be used to extract the identifier names associated with the source code inside the range of the clones. The line numbers are obtained from a file that CCFinder generates for each source file that has been searched for clones. This file contains one line per token with the line containing that token's information. The project web site at <http://www.cis.uab.edu/softcom/ir4pc> contains the CCFinder textual representation output file for the NT kernel source code (with file names removed).

Table 4.1

Clone group totals of top-five directories in the NT kernel

Directory	All occurrences	Solely in directory	After subsets removed
Memory management	383	361	335
Registry configuration	213	209	196
Process/thread support	175	157	132
Security functions	155	148	121
I/O management	154	145	138

Clone group filtering (step 2). The analysis of the clones using LSI is focused on clones that are found in specific NT kernel directories. In particular, we looked at the clones in five directories containing the most clone groups. LSI was performed separately on the clone groups of each of these five directories. Table 4.1 displays the top-five directories with their clone group totals. The decision to focus separately on each of these five directories was made due to the fact that the clones that span multiple directories were observed to be few among the clone groups in these directories. That is, the majority of clone groups in the five directories contained clones that reside in the same directory.

This can be seen in the difference of the total clone groups in the “All occurrences” column and the “Solely in directory” column in Table 4.1. A total of only 60 clone groups contained clones spanning multiple directories.

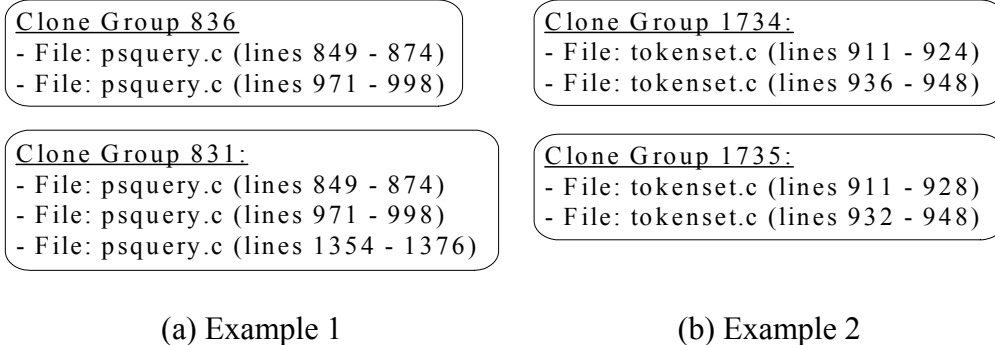


Figure 4.3 – Examples of eliminated clone group subsets.

The totals under “All occurrences” represent the number of clone groups that contain at least one clone in the directory. Because the clustering process calls for clone groups with clones that are all contained in one directory, these clone groups are determined and the totals under “Solely in directory” represent these clone groups. Finally, it was observed that some clone groups were contained in other clone groups. That is, if two clone groups are defined as $CG_1 = \{c_1, c_3, c_5\}$ and $CG_2 = \{c_1, c_3\}$, then $CG_2 \subseteq CG_1$ and the clone group subset CG_2 is removed from further consideration. A clone group is also removed if the clones in that group are contained in the clones of another group. In this case, the clones are not exactly equal to each other, but the section of code representing the clones in the group that is removed is covered fully by larger clones in another group. Figure 4.3 provides examples of such eliminations. In Figure 4.3(a), clone group 836 is removed, because the exact same clones are located in clone

group 831. In Figure 4.3(b), clone group 1734 is removed, because its clones are contained within the clones in clone group 1735. The totals under “After subsets removed” represent clone groups with clones that are all contained in one directory and are not subsets of any other clone group. Thus, the clone groups representing the totals from the last column are used in the clustering process.

Source code XML (Extensible Markup Language) representation (step 3). In conjunction with the detection of clones, a tool called srcML [srcML, 2010], which is a freely available tool developed by the Software Development Laboratory at Kent State University, was used to convert the NT kernel source code into an XML representation containing selected AST tags. This representation is used to extract the identifier names that are needed in order to generate the term-document matrix. The comments are not considered because we would like to see how well the relationships of the clones are through the clustering without comment information.

Table 4.2 shows an example of the XML representation of the source code. The values of the elements of type *name* are of interest. A srcML XML representation of each source file in the directories that are analyzed is generated. Given the source code ranges (starting and ending line numbers) of the clones in the clone groups that were determined in Step 2, the identifier names that are contained in these ranges are extracted from the XML representations. This is possible because the original source code layout is preserved in the XML representation; as such, a line number in the original code corresponds to the XML representation for that same line in the XML file. No special filtering of names is performed. All values from elements of type *name* are included;

hence, no stop words are removed. Furthermore, stemming of the identifier names is not performed.

Table 4.2

srcML representation of simple declaration

Original code	XML Representation
<code>int x = 3;</code>	<code><decl_stmt><decl><type><name>int</name></type> <name>x</name> =<init> <expr>3</expr></init></decl>;</decl_stmt></code>

Generating term-document matrix (step 4). In our work, the definition of *term* is the name of an identifier. The definition of *document* is a clone group. A term-document matrix is generated for each of the five directories. In addition, a final matrix is generated containing the clone groups and terms from all five directories. Table 4.3 provides totals of terms and documents for each matrix yielding the dimensions of the matrices. Please note that the number of terms in the last row is not the same as the sum of the previous five rows because an identifier name can exist and be counted in each directory, but when all directories are considered it is counted as a term only once. The number of documents in the last row equals the sum of the previous five rows, because the clone groups in each directory are not duplicated in other directories. Moreover, the document totals are identical to the clone group totals (last column) of Table 4.1.

Table 4.3

Term and document totals

Directory	# of terms	# of documents
Memory management	1506	335
Registry configuration	1035	196
Process/thread support	597	132
Security functions	670	121
I/O management	908	138
All five directories	4153	922

Computing the SVD of matrices (step 5). An important part of the LSI process is obtaining the SVD of the term-document matrix, the calculations of which have been described in Section 2.3.1. The commercial tool Matlab [Matlab, 2010] was used to obtain the SVD of the term-document matrix and from this the generation of the lower dimensionality matrix that will be used to perform document-document similarity calculations, which is done in Step 6. The term-document matrix from Step 4 can be imported into Matlab and the SVD is generated using Matlab's standard `svd` function. Before this is done, the values in the matrix are normalized using the `zscore` function. Regarding the determination of the value for k , representing how many of the first largest singular values to include, we use the formula in [Kuhn *et al.*, 2007], where $k = (m \times n)^{0.2}$. In this case, m represents the number of rows and n represents the number of columns in the term-document matrix.

Clustering the clone groups (step 6). The clustering of the clone groups based on their similarities represented by the column vectors from the matrix \hat{X} is obtained using Cluto [Cluto, 2010], which is a clustering tool available from the Karypis Laboratory at the

University of Minnesota. Cluto provides several clustering techniques that can be divided into *partitional* and *agglomerative*. Partitional clustering starts with one cluster containing all elements and iteratively partitions or divides the original cluster into smaller clusters. Agglomerative clustering is the opposite of partitional [Han and Kamber, 2006]. Cluto also provides several techniques for calculating vector similarities including calculating the cosine of two vectors, which is the similarity calculation that is used in this case to generate the clusters.

The default clustering approach in Cluto is called *repeated bisections clustering*, which is a partitional clustering technique. In addition to the input matrix containing the vectors representing the clone groups, the only other argument requested by Cluto is the final number of clusters to be generated containing the clone groups. Several values were tested for each of the directories. Two properties of the clusters were observed: the number of members in each of the clusters and the average similarity of the members of the clusters. The average similarity is the average of all the cosine similarity values that are calculated between the objects of a cluster. We look for a fairly uniform distribution of clone groups in each cluster, thus trying to avoid clusters that dominate other clusters or clusters that contain only one clone group. We observe the collection of the average similarity values from each clustering size that is generated and look for a considerable number of high average similarity values. This resulted in a cluster size of 30 for the first two directories (memory management and registry configuration) and a size of 20 for the last three directories (process/thread support, security function, and I/O management).

Generating cluster reports for analysis (step 7). The result from Step 6 is a mapping of each clone group as a member of one of the generated clusters. At this point, the clusters need to be analyzed manually for trends and associations among the members of each cluster and in turn the clones of each clone group in the clusters. Informative HTML files containing all the source code in the clone groups that are contained in a cluster are generated from the results of Cluto, the results of CCFinder, and the original source code. The results of the analysis of the clusters are described in Section 4.1.4.

Scalability. With the exception of Matlab, the third-party tools used in the process outlined in this subsection are freely available. All of these non-commercial tools are scalable and can be used on the source code of large-scale software systems. As can be seen in [Li *et al.*, 2004], CCFinder has been used to detect clones in the source code of the Linux kernel. The srcML translator can translate code into XML at a rate of 7500 LOC per second [Collard and Maletic, 2004]. Cluto has been used to cluster large object sets including topic clustering of thousands of document datasets [Zhao and Karypis, 2005]. The main bottleneck is the final step of the process related to the manual evaluation of the generated clusters. This is a limitation in terms of scalability, which is described further in Section 4.1.6.

4.1.4 Clone Group Clustering Analysis within the Windows NT Kernel

Table 4.4 provides a statistical overview of the clusters that were separately generated from the clone groups of the top-five directories in the Windows NT kernel using LSI and a combination of all clone groups from the directories. The first column

contains the number of clusters that were generated. The second column represents the number of clone groups for each clustering process of each directory after filtering is done to remove some clone groups as described in Section 4.1.3. The third column indicates the average number of clone groups in each generated cluster. The rest of the columns contain information about the average similarities (grouped by percentile ranges) among the members (i.e., clone groups) of the clusters that were generated. These values were reported by Cluto after the clustering was performed. For example, in the memory management directory, six clusters contained clone groups with average similarity 90% and above, 13 clusters with average similarity between 80% and 89%, and 11 clusters with similarity between 70% and 79%.

Table 4.4

Statistical information on the generated clusters

Directory	#oC	#oCG	AvgCG	Percentile Range							
				90		80		70		60	
				#oC	%oC	#oC	%oC	#oC	%oC	#oC	%oC
Memory management	30	335	11.1	6	20%	13	43%	11	37%	0	0%
Registry configuration	30	196	6.5	11	37%	9	30%	9	30%	1	3%
Process/thread support	20	132	6.6	16	80%	4	20%	0	0%	0	0%
Security functions	20	121	6.0	10	50%	9	45%	1	5%	0	0%
I/O management	20	138	6.9	9	45%	10	50%	1	5%	0	0%
All five directories	50	922	18.4	10	20%	25	50%	13	26%	2	4%

#oC = Number of clusters

#oCG = Number of clone groups in directory

AvgCG = Average number of clone groups in each cluster

%oC = Percentage of clusters in each percentile range

A large number of the similarity values in Table 4.4 were 70% and above. The possible explanation for this is the nature of the clone groups reported by CCFinder. Two types of clone group combinations may have contributed to this observation:

- *Linked clone groups*: Two clone groups may be linked to each other by one or more clones that are present in both groups. For example, if two clone groups are defined as $CG_1 = \{c_1, c_2, c_3\}$ and $CG_2 = \{c_3, c_4, c_5\}$, then CG_1 and CG_2 will produce a very high similarity value.
- *Sliding clones*: The only difference between the clones in two clone groups is the slightly different starting and/or ending lines of the clones, which may amount to just a few lines.

The removal of clone group subsets described in Section 4.1.3 does not capture the two cases above. However, for both cases it was decided not to try to combine these clone groups together before generating the term-document matrix, because keeping these clone groups separated may yield interesting observations during analysis. Nevertheless, these two types of clone groups were still detected and reported in the cluster reports, so during the analysis the existence of such clone groups in the clusters can be noted. For sliding clones, clone groups with clones that differ with a maximum of five lines between the values of their starting and/or ending lines are associated with each other and noted in the report. This will allow the clones in the clone groups to differ by a few statements, but still be considered as representatives of the same clones.

The results of the analysis of the clustering of NT kernel clone groups are given in the following paragraphs, where the individual results from selected clusters are discussed. Each set of results is preceded with some statistical information about the

cluster, such as the number of unique clone groups and the average similarity values among members of the clusters. Unique clone groups are those individual clone groups that cannot be grouped based on the two types that were previously described (i.e., linked clone groups and sliding clones). Clone groups that can be grouped based on these two types are counted as one unique clone group. The *Microsoft Windows Internals* book [Rusinovich and Solomon, 2005] provided a high-level understanding of how the code fits within the kernel operations. The results from each cluster data set are discussed separately. An overview and discussion of the entire analysis is given in Section 4.1.5. It is worth noting that the observations given in the following paragraphs and the overview reported in Section 4.1.5 can be considered subjective based on our observations, because the final step of the process requires a manual evaluation of the clusters. However, these observations are based on detailed evaluation of each section of code associated with the clones in the clustered clone groups.

```
Free pools = Maximum (non) paged pools - Allocated (non) paged pools
if (Free pools (<|>) (High|Low) (non) paged pool threshold)
  if ((High|Low) (non) paged pool event is in signaled state)
    Set (High|Low) (non) paged pool event to nonsignaled state
  if (Free pools (<|>) = (Low|High) (non) paged pool threshold)
    if ((Low|High) (non) paged pool event is in nonsignaled state)
      Set (Low|High) (non) paged pool event to signaled state
```

Listing 4.2 – Pseudo-code of setting paged events.

Memory management directory. This subsection presents four of the clusters observed in the memory management directory. The first cluster contains clone groups where the clones are logically opposite of each other. The clones in the second cluster utilize a similar structure of conditional statements. The third cluster reveals relationships based

on the assignment of two variables, and the fourth cluster contains clones that revolve around a sequence of statements for initialization purposes.

Cluster MM-1 (Unique clone groups: 3 (out of 9 clone groups); Average similarity: 0.997)

All clone groups in this cluster reside in one file containing the implementation of memory page allocation. The clones in these clone groups account for ~230 cloned lines of code (CLOC) and represent a task that checks the availability of paged and non-paged memory pools and whether a low or high threshold has been reached. The signal state of a paged event is set when values surpass a threshold. Listing 4.2 provides pseudo-code of the two types of clones that are clustered together with differences highlighted. The operator in the conditional statement could be either a '<' or '>.' Based on this operator, the "High" or "Low" thresholds are checked. This sequence of statements is used in four different functions ranging from one to four occurrences in each function.

Cluster MM-11 (Unique clone groups: 8 (out of 12 clone groups); Average similarity: 0.854)

This cluster contains clone groups that in turn contain clones (~570 CLOC) with conditional statements (i.e., `if` and `switch` statements). These statements are primarily responsible for testing the property of a page mapping (e.g., `cached`, `not cached`, or `write-combined`) before executing. The clones are scattered among seven files in the directory. It is worth noting that two clone groups that should be included in this cluster are located in another cluster.

Implementation 1:

```

try {
    if (PreviousMode != KernelMode) {
        ProbeForWritePointer (BaseAddress);
        ProbeForWriteUlong_ptr (RegionSize);
    }

    CapturedBase = *BaseAddress;
    CapturedRegionSize = *RegionSize;
} except (ExSystemExceptionFilter ()) {
    return GetExceptionCode ();
}

```

Implementation 2:

```

try {
    *RegionSize = ((PCHAR)EndingAddress -
(PCHAR)PAGE_ALIGN(CapturedBase)) + PAGE_SIZE;
    *BaseAddress = PAGE_ALIGN(CapturedBase);
} except (EXCEPTION_EXECUTE_HANDLER) {
    return GetExceptionCode ();
}

```

Implementation 3:

```

try {
    *RegionSize = CapturedRegionSize;
    *BaseAddress = StartingAddress;
} except (EXCEPTION_EXECUTE_HANDLER) {
    NOTHING;
}

```

Listing 4.3 – Three different implementations of obtaining base address and region size.

Cluster MM-13 (Unique clone groups: 9 (out of 14 clone groups); Average similarity: 0.758)

Except for five clone groups, this cluster includes clones (~910 CLOC) containing sequences of statements that obtain the base address of the allocated region of pages and size of the page region inside an exception handler. The two variable assignment statements inside an exception handler represent the duplication among the clones. Three different implementations are used to obtain the two values as displayed in Listing 4.3. The clones are located in files related to the management of virtual memory (e.g., allocation, flushing, freeing, locking, and protection). The first implementation is used in

six functions in five different files. The second implementation is used in two functions in one file. The third implementation is used in two functions in two files.

The first implementation is used at the beginning of a function to determine write access on the address that is passed. The last two implementations are located at the end of a function and are used to update the value of the pointers. These clones were separated into the three clone groups, because of the difference in the structure of the assignments.

Cluster MM-22 (Unique clone groups: 8 (out of 9 clone groups); Average similarity: 0.798)

Six out of the eight unique clone groups contain a sequence of statements that initializes a page and maps it with a page table entry. Most of the clones (~470 CLOC) in the cluster are scattered in four functions in three files and are called during system initialization. Different clone groups were detected possibly because of several reasons. The statements are used both in a one-time execution and in an iterative process inside a loop. Also, the code preceding this operation is not similar among the different clone groups. Finally, some clones utilize a function that performs both the initialization and setting the page table entry to a valid state. Other clones utilize a different function for initialization, while the page table entry is set to the valid state after the call to this function. Listing 4.4 displays three statements that are present in most of the clones, where the index of a page that contains all zeroes is retrieved based on a page coloring attribute (which is used to optimize cache behavior). The index is then assigned and initialized.


```

DirectoryFrameIndex = MiRemoveZeroPage (
    MI_GET_PAGE_COLOR_FROM_PTE (StartPpe));
TempPte.u.Hard.PageFrameNumber = DirectoryFrameIndex;
MiInitializePfnAndMakePteValid (DirectoryFrameIndex,
    StartPpe, TempPte);

```

Listing 4.4 – Example code for page initialization.

Registry configuration directory. The clusters generated from this directory are dominated by clone groups containing short sequences of code that reside mostly in one file. This is unlike the clusters observed in the other directories, where the clones in a clone group can be scattered around several files and contain slight variations. Because this directory is related to the Windows registry configuration, short sequences of duplicated code can be observed that are used to perform key look-up, setting of key values, and retrieving key values. Table 4.5 lists some examples of the clusters in this directory. A description of the code represented by the clones is given with the number of unique clone groups containing the clones. In addition, the number of files where these clone groups reside are included. It can be seen that the clones are uniquely contained in just one or two files.

Process/thread support directory. Two of the clusters observed in the process/thread support directory are introduced in this subsection. Both clusters are associated by similar sequences of statements that are needed as part of a large sequence to perform different tasks.

Table 4.5

Clusters of registry functions and file totals

Code description of cluster	#oCG	#oFi
Comparison of values in a binary search	4 (4)	1
Checking the consistency of the values in the registry list	3 (4)	1
Setting value of a key	2 (3)	2
Getting value of a node	2 (8)	1
Getting value of a node	9 (21)	1
Getting value of a node	11 (14)	1

#oCG = Number of unique clone groups (total number of clone groups)
#oFi = Number of files containing clones

Cluster PS-0 (Unique clone groups: 4 (out of 8 clone groups); Average similarity: 0.985)

Several tasks are performed that are based around a loop and initial statements inside the loop that iterate through an array of pointers to callout routines. The tasks include notifying the registered callout routines when an image, process, or thread is created and when a process or thread is deleted. Another task utilizes the same loop structure and statements to remove a single callout routine from the list.

Cluster PS-9 (Unique clone groups: 7 (out of 7 clone groups); Average similarity: 0.827)

Two of the clone groups in this cluster represent two pairs of function clones. The first pair consists of accessor functions to get and set the context of a thread. The second pair consists of functions to resume and suspend threads in a process. In both cases a similar sequence of statements is used to reference the thread or process object, perform an operation on the objects (e.g., get/set context, resume/suspend process), and generate a status value of the execution that is the return value of the function.

Security functions directory. The clusters related to the security functions directory contain similar properties to the clusters in the memory management directory. This can be observed from the first cluster being associated with a sequence of conditional statements where the clones differ in what is done inside the statements. Also, the clones in the third cluster are related by a variable assignment. The second cluster is based on clones where the statements are in different orderings.

Cluster SE-0 (Unique clone groups: 2 (out of 2 clone groups); Average similarity: 0.985)

This cluster represents two clone groups in two files that contain a sequence of conditional statements in which one sequence acts as a counter and the other copies information. Both files are related to the audit policy elements for security auditing purposes. The audit policy determines which properties should be recorded for future auditing. In the clone groups, each property is represented by a conditional statement. In the first clone group, the conditional statements are used to update a global counter of all tokens that have certain properties enabled. The second clone group uses the conditional statements to copy properties of a specific token to a token information variable. Further investigation revealed an additional sequence of similar conditional statements outside, but near, the clone ranges of one of the clone groups.

Cluster SE-7 (Unique clone groups: 2 (out of 4 clone groups); Average similarity: 0.992)

This cluster contains six unique clones located in six different functions in a file implementing security audit and alarm procedures. The four clone groups that comprise these clones can be divided into two unique groups, because of a slight difference in the

syntax of the clones. All clones perform an initialization of the array that will contain audit parameters, but the location of this initialization is different for the two groups. Moreover, one group does not include the initialization code as part of the clones. Figure 4.4 illustrates the structure of the two groups.

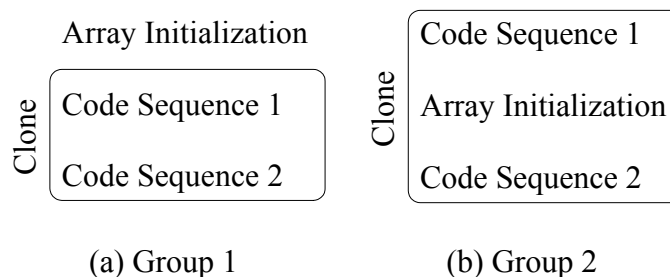


Figure 4.4 – Differing location of array initialization.

Cluster SE-11 (Unique clone groups: 11 (out of 13 clone groups); Average similarity: 0.917)

In this cluster, we see a pattern of code similar to cluster MM-13 where assignments are enclosed inside a *Try-block* for exception handling. However, in this case the code inside the *Try-block* is identical. The difference is the statement immediately preceding the block. Identical *Try-blocks* are separated into different clone groups based on the assignment statement preceding the *Try-block*. The clone groups all reside in the same file that returns information about a security token that is queried. The value that is assigned in the *Try-block* is the length of the token information that will be returned. The length is determined in three different ways in statements preceding the block, as seen in Listing 4.5. Not only are simple assignment statements used, but also one assignment statement is contained in a loop.

Example 1:

```
RequiredLength = (ULONG)sizeof( TOKEN_STATISTICS );
```

Example 2:

```
RequiredLength = (ULONG)sizeof(TOKEN_GROUPS_AND_PRIVILEGES) +
    PrivilegesLength + RestrictedSidsLength + GroupsLength;
```

Example 3:

```
while (Index < Token->RestrictedSidCount) {
    RequiredLength += SeLengthSid( Token->RestrictedSids[Index].Sid );
    Index += 1;
}
```

Listing 4.5 – Three examples of obtaining token information length.

I/O management directory. From the observed clusters in the I/O management directory, one cluster shows a similar finding to the related work of [Marcus and Maletic, 2001]. Four other clusters that are described in this subsection contain clone associations based on specific variations in the code represented by the clones.

```
// Allocate and initialize the I/O Request Packet (IRP) for this
// operation.
// The allocation is performed with an exception handler in case the
// caller does not have enough quota to allocate the packet.
irp = IoAllocateIrp( deviceObject->StackSize, (...) );
if (!irp) {
    //
    // An IRP could not be allocated. Cleanup and return an appropriate
    // error status code.
    //
    if (...) {
        ExFreePool( event );
    }
    IopAllocateIrpCleanup( fileObject, (...) );
    return STATUS_INSUFFICIENT_RESOURCES;
}
irp->Tail.Overlay.OriginalFileObject = fileObject;
irp->Tail.Overlay.Thread = CurrentThread;
```

Listing 4.6 – I/O Request Packet (IRP) allocation and initialization general code.

Cluster IO-0 (Unique clone groups: 5 (out of 5 clone groups); Average similarity: 0.961)

Similar to the results found in [Marcus and Maletic, 2001], which detected implementations of linked lists, this cluster contains clone groups (except for two groups) related to code that implements and utilizes linked lists. One clone group represents function clones to add items to the head and tail of a list. Another clone group represents function clones that call list insertion functions. The final clone group contains a loop that calls the removal functions.

Cluster IO-5 (Unique clone groups: 16 (out of 24 clone groups); Average similarity: 0.959)

This cluster contains 20 clones (~990 CLOC) in 20 functions in 11 files. The clones represent a sequence of code related to allocating and initializing an IRP as seen in Listing 4.6. The highlighted parts of this code represent slight variations even in the comments area. Variations also occur in the parameters of two functions and the Boolean expression of a conditional statement, which are represented by “(…).”

Three other clusters displayed similar content where the clone groups represented code that contained slight variations. These clusters are listed in Table 4.6. Similar to the clones in the cluster of Listing 4.6, the clones in these clusters also reside in separate functions and in multiple files.

Table 4.6

Additional clusters containing clones with slight variations

Code description of cluster	#oCG	#oClo	#oFu	#oFi
Code to determine if an I/O operation on a file is synchronous and to wait until current thread owns the file	8 (11)	16	16	10
Allocation of a memory description list, which is a specification of the physical memory occupied by a user's buffer when an IRP is created	4 (4)	7	7	4
Code related to waiting for an I/O operation to complete	5 (6)	8	8	6

#oCG = Number of unique clone groups (total number of clone groups)
 #oClo = Number of unique clones
 #oFu = Number of functions containing clones
 #oFi = Number of files containing clones

All five directories. The clone groups of all five directories were clustered to determine any relationships among these clone groups as a whole. Almost half of the clusters that were generated contained clone groups from the same directories. Table 4.7 provides the number of clusters containing clone groups from exclusively one directory. This is not an unreasonable result as the clone groups that were clustered were chosen to be highly unique to the specific directories.

Table 4.7

Clusters where all clone groups are from one directory

Directory	30 clusters	50 clusters
Memory management	6	13
Registry configuration	2	4
Process/thread support	1	2
Security functions	2	3
I/O management	3	5
Total	14	27

Considering these clusters that contained clone groups from multiple directories, similarities of the code were mainly based on generic functions that appear across the directories. One cluster contained exception handling code covering a single variable assignment. Another cluster contained the frequently used `Status` variable that is the return value of many functions and used to determine the successful completion of a function execution. It was also observed that clusters were generated for code containing a `Process` variable that performs tasks on a process. The observations from the clusters generated for each specific directory produced more interesting results in terms of clone location and functionality compared to the clusters generated from all five directories together.

4.1.5 Discussion of Analysis Results and Clustering Process

This subsection provides several discussion points. The analysis outlined in the previous subsection is evaluated and we discuss the benefits of this information toward the comprehension of the clones. The influence of utilizing the results from a clone detection tool and using a specific clone detection tool are also evaluated.

Evaluation of cluster analysis. A recurring theme from the analysis results is the discovery of sections of code that could be considered clones of each other, but were not identified by the clone detection tool because of some statement variations. In cluster MM-1, slight changes were observed in the conditional operators that produced multiple groups of clones. In cluster MM-11, duplicate code was detected inside different conditional structures. In clusters MM-13 and SE-11, depending on the method of

variable assignment a clone can be grouped with other clones. The clones in cluster MM-22 contain three statements that are present in most of the clones. The code surrounding these three statements differ, so in this case the clone groups in the cluster represent clones that are grouped based on the similar code that surrounds these three statements.

Variations are also discovered inside sequences of conditionals, loops, and functions in clusters SE-0, PS-0, and PS-9, respectively. In each case, the statements inside the blocks are varied to perform specific tasks, but the target elements in which the tasks are performed are the same. In cluster IO-5, the grouping of clones is based on whether a conditional statement is present. Moreover, clones that are semantically the same can also be clustered as evident in cluster SE-7, where the only difference is the statements not being in the same order.

The observations of these noted clusters show associations among clones in the different clone groups. These associations are mainly based on similar code or functionality that include some variations, which resulted in the clones being assigned different clone groups by the initial clone detection tool. With the analysis obtained from these clusters, the goal is to determine whether the results of this analysis provide a better understanding of the clones. The knowledge from the clustering observations can give the programmer an improved understanding of the clones, because he or she is exposed to the relationships of clones from multiple clone groups where the clone groups were originally separated by the clone detection tool. Comprehension in this case is related to the additional knowledge of how the code in clustered clone groups that differ by only slight changes in structure are used, such as varying assignment statements in clusters MM-13 and SE-11, different orderings of statements in cluster SE-7, and the existence of

statements and differing function parameters in cluster IO-5. In addition, comprehension can be related to the knowledge of how sections of code that perform different tasks on an element or object in the program are associated to each other through the clusters, such as the various uses of code for page initialization in cluster MM-22, different tasks performed on an array of pointers in cluster PS-0, and two uses of a sequence of conditional statements in cluster SE-0. Furthermore, knowing the location of related clone groups based on the clusters identified in Section 4.1.4 can help in the maintenance process, because clones that are related but are not exactly identical may still need to be updated at the same time or also possibly be refactored. For example, when the addition of a conditional statement is needed in one of the clone groups in either cluster MM-11 or SE-0, the knowledge of associated clone groups containing similar conditional statements becomes useful to determine whether or not the clones in these other clone groups should be updated as well.

Clones that dominate sections of code can also be identified, as seen in some of the clusters of generic functions in the clustering of all five directories and in some of the clusters in the registry configuration directory where the clones are uniquely located in one or two files. It may be argued that this information could be found in an IDE by searching for references to a function that surrounds a sequence of statements, but clone detection is needed to first determine which collection of statements are duplicated. Clustering provides a method to group clones further to discover the functions that are used the most.

Consider the effort by the programmer to understand the clones of a program by utilizing the process that we have outlined as compared to having only the results of the

clone detection tool. Without the additional associations provided through the clustering of the clone groups, the relationships of groups that contained some variations would not have been known initially, because they were reported separately by the clone detection tool. This is in addition to the possibility of clone groups containing clones that are used in different contexts based on the code surrounding the clones. Without the clustering, these relationships would have to be determined manually by evaluating each of the clone groups. The process described in this research also requires manual evaluation at the end, but the clustering of the groups provides some assistance during this evaluation by associating clone groups based on the information retrieval technique of LSI.

Influence of clone detection results. The novelty of the process outlined in this dissertation compared to existing works that will be described in Section 4.3 (i.e., [Marcus and Maletic, 2001] and [Kuhn *et al.*, 2007]) is the narrowing of the corpus from the entire source code to only sections of source code that have been identified by a clone detection tool. Our purpose of narrowing the corpus is to determine associations among clone groups from the clone detection results. If the information from the clone detection results is removed and LSI is performed solely on the entire source code as the corpus, the determination of the definition of *documents* for the LSI process must be specified differently. The definition of *terms* for the LSI process would remain the same, namely the identifier names found in the code. Documents, however, would need to be specified as being an entire source file, a function block, or a block of statements. Based on the definition of documents, the clustering of these documents can reveal similar results as the clusters produced by our technique. For example, if the documents were set as

function blocks, a cluster such as cluster IO-0 that represents function-level clones could be generated. However, in this case the clusters would only consist of function blocks. Statement-level clusters could only be generated if the documents were defined as statement blocks.

Focusing on sections of code that represent clones identified by a clone detection tool and specifying documents to be the clone groups containing these clones provides a clustering process where the clusters can ultimately consist of code at different levels. The clusters will consist of clone groups, which contain clones that can be statement-level clones, function-level clones or even file-level clones. The limitation of having to determine various definitions of documents in the LSI process is eliminated as the structure of clone groups is based on the syntactic similarity of different levels of code and in turn can consist of these different levels.

Influence of chosen clone detection tool. CCFinder is used as the source for information of clones in the NT kernel source code. As seen in Section 4.1.3, CCFinder was chosen because of its availability and ability to detect clones in the NT kernel source code. The influence of this clone detection tool is considerable, because the terms and documents used in the LSI process is dependent upon the sections of code that have been identified as the initial clones by the clone detection tool. However, the sections of code representing the clones are exactly the part of the source code that is of interest rather than the entire source code altogether. As the clustering is dependent on the results of the clone detection tool that is used, the clustering will be different if another clone detection tool is used, because the results from various clone detection tools generally differ, as

evident in [Bellon *et al.*, 2007]. Performing the process with another clone detection tool is considered future work.

Regarding the characteristics of a clone detection tool, the beginning of Section 4.1.5 has provided an evaluation of cluster analysis. The recurring theme that we observed was that the clones in the clone groups inside a cluster could have been lumped into one clone group, but because of some statement variations, the clone detection tool separated the clones into several clone groups. Clone detection tools try to return the maximal size of clones. For example, in clusters MM-13 and SE-11, because the different assignment methods are duplicated in several locations, the clones are collected into groups based on the assignment method to produce the maximal size clones.

Another characteristic of a clone detection tool is the configuration setting that sets the minimum allowable clone size. The three statements evident in the clones in cluster MM-22 are considered lower than the minimum clone size and as several duplicate patterns exist around them, multiple groups of clones are generated. Determining the minimal size of a clone that can still be of interest is not an easy decision. Keeping the minimum size of clones to at least five lines will reduce the number of generally uninteresting clones (i.e., one line accessor methods). However, this will also separate the small clones with duplicate functionality such as those in cluster MM-22.

At the beginning of Section 4.1.4, two types of properties of the clusters (i.e., linked clone groups and sliding clones) were acknowledged. These properties are related to the resulting clone groups that were produced by CCFinder. It was decided that the groups containing these properties would be kept and not removed, but noted during the

evaluation of the clusters. These two properties did not strongly influence the observations of the clusters. However, the knowledge of the linked clone groups and sliding clones provided information about the uniqueness of the clones in the clusters. Knowing which clones are linking multiple clone groups and which clones are being represented by slightly varying line numbers provides a count of how many unique clones are in the cluster. The more individually unique clones contained in a cluster, the more sections of code that cluster is related to and the more interesting the cluster is to evaluate.

4.1.6 Limitations

It should be emphasized that the clustering results are solely dependent on the clones that are initially detected. As seen in cluster SE-0, a separate sequence of conditionals was not clustered, because it was not detected as a clone. During the use of this process it must be emphasized that only clones are being considered and there is a possibility that the clustering will not contain sections of code that were not detected as clones. Also, depending on the total number of clusters generated, there may be situations where a few related clones are located in a different cluster, as seen in cluster MM-11.

Other observed shortcomings associated with the process include the fact that the information retrieval process has been able to provide further grouping of clone groups, but the amount of data that must be analyzed is still the same. However, the groupings or clusters provide for the collective analysis of the related clone groups and the evaluation of the clusters can be prioritized to start with those clusters containing the higher similarity values. Without this proposed process, the additional relationships found from

the clusters would not be available, in which case the programmer must manually determine whether certain clone groups are associated with each other. In addition, the observations of the clusters in Section 4.1.4 represent a small portion of the total clusters that were generated. The distinctive relationships of the clone groups in some of the clusters could not be defined clearly.

4.2 Sub-clone Refactoring

The analysis of clones in Section 4.1 considered clones in a single snapshot of the source code. In this section, the analysis of clones is based on the changes observed on the clones between multiple consecutive versions of the source code. This type of analysis considers how clones evolve, and more specifically, how they are refactored.

Research into the automatic detection of clones in code have produced various techniques and tools, but knowledge regarding the utilization of these tools in the practice of software maintenance, and specifically within the context of clone refactoring, is still limited. In this case, clone refactoring is concerned with the removal of the duplication represented by the clones. This section describes our investigation into how clones reported by an automated clone detection tool are refactored [Tairas and Gray, 2010a] [Tairas and Gray, 2010b]. We focus our interest on observing characteristics of the clones in a scenario where a programmer uses a clone detection tool to find the clones before performing clone refactoring. Our observations are based on mining the changes identified between consecutive versions of open source software artifacts. We found that in some cases the range of the clone and the range of code that was actually refactored differed in that refactoring was performed on only part of the clone (i.e., a sub-clone). By

evaluating the instances of the partially refactored clones, we discovered characteristics influencing sub-clone refactoring. Such characteristics can inform a process that utilizes clone detection in an effort to remove duplicate code where sub-clones are also considered, thus augmenting the choices of refactoring given to the maintainer of the code.

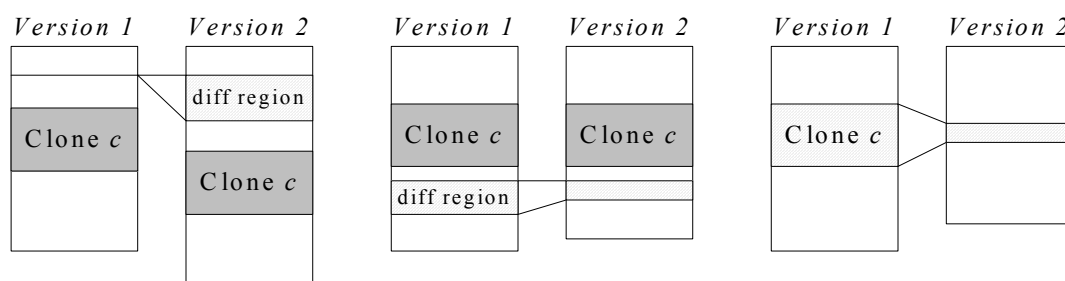


Figure 4.5 – File changes based on *diff* information.

4.2.1 Observing Clone Refactoring

We focus our interest on observing the refactoring of clones that occur between consecutive versions of a software release. A programmer may perform some activity of refactoring on the original code of one version and the result of this refactoring is subsequently evident in the next version of the code. In this case, the original code and refactored code is stored within two consecutive versions. By comparing two consecutive versions of the code, we are able to find actual refactorings that were performed between the two versions. More specifically, we concentrate on finding refactorings related to clones.

Our method of finding clone refactorings between code versions starts with the use of a clone detection tool. The tool is executed on the source code of one version. This

step identifies the sections of code that are reported as clones by the tool. The next step is to compare these sections of code with the corresponding sections of code in the next version of the source code. The comparison utilizes the Unix *diff* command. For each section of code reported as a clone, the file where it resides is compared using *diff* with the next version of that same file. A batch process automates this step for all clones that were reported by the clone detection tool. We are interested in the changes reported by *diff* that consist of deletions and additions of lines in the approximate location of the reported clone range. Figure 4.5 illustrates three different scenarios of changes in two versions of a file. Our focus is on instances of Figure 4.5(c), because the changes occur within the range of a clone. For example, a change that consists of the deletion of lines followed by the addition of one line within the clone range can signal the possibility of an *Extract Method* refactoring activity. Figure 4.5(a) and Figure 4.5(b) represent changes outside the clone range, which are ignored in our observations.

Table 4.8

Extract Method-related refactorings in JBoss

Extract Method	14
Extract Method with Pull-up Method	1
Extract Method to utility class	6
Total	21

4.2.2 *Observing Sub-clone Refactoring*

We conducted a study to observe the refactorings that were performed on clones among versions of the JBoss Application Server, which is an open source implementation of J2EE [JBoss, 2010]. Specifically, these refactorings were identified by observing

changes between two consecutive versions from version 2.2.0 to 4.2.3 (44 versions). It is worth noting that we do not know if the developers of JBoss used a clone detection tool to identify clones, which quite possibly was not the case. However, the main purpose of the study is to see how a section of code that would have been reported as a clone was maintained in actual practice. The Simian clone detection tool was initially chosen to detect the clones in the first part of our study. Simian was selected because of its relatively fast speed in detecting and reporting clones.

The process outlined in the previous subsection was performed and changes reported by *diff* within the clone ranges were further evaluated. Some common refactoring activities within the range of the clones were discovered in the JBoss version sequences, such as *Extract Method*, *Pull-up Method*, and *Extract Class* [Fowler, 1999]. In many cases, the refactorings removed the duplication of code by extracting the duplicate functionality into a new method or pulling up duplicated code to a super class. Table 4.8 summarizes the 21 instances of *Extract Method*-related refactorings associated with clones that were found across the versions of JBoss. In addition to *Extract Method*, a *Pull-up Method* refactoring also contains *Extract Method* if only a part of a method is pulled up. Also, some statements were extracted to a separate class, generating more of a utility-type method. It was observed that only in two of the refactorings listed in Table 4.8 the range that was reported by Simian as a clone was exactly the range that was refactored. In the remaining cases, refactoring was not performed on the exact range of the reported clone.

To provide a wider spectrum of observation, we evaluated additional clone detection tools. Four other clone detection tools were selected: CCFinder, which is a

token-based tool, and CloneDR, Deckard, and SimScan, which are tree-based tools. The detection configuration settings for each tool are listed in Table 4.9, which summarizes the non-default configuration settings chosen for the clone detection tools. In most cases, the value set was the default value given by a tool. In the evaluation, these tools were not requested to find clones on all of the source files of JBoss, but rather only on the source files containing the Simian clones associated with the identified refactorings in Table 4.8. The purpose of this activity was to see how other tools presented (or not) the code ranges of the same clones that were originally detected by Simian.

Table 4.9

Clone detection tool configuration settings

Tool	Setting	Value	Description
CCFinder	Shaper level	Hard	Generate clones that are enclosed in blocks as much as possible
CloneDR	All default settings		
Deckard	Similarity	0.95	Allow for small differences in clones
	Stride	0	Limit the size of the clones
Simian	All default settings		
SimScan	Volume	Small	Include smaller matches
	Similarity	Loosely similar	Allow for small differences in clones
	Speed / Quality	Exhaustive search	Provide more detailed results

Table 4.10 provides the detection results from Simian and the four additional tools for the refactorings observed in the clone ranges initially detected by Simian. The “Exact Coverage” column records the number of times each clone detection tool reported a clone group consisting of clones that represented exactly the code range that was refactored. As can be seen in the table, the number of times this is the case for each tool

were all less than half of the 21 total instances. For example, Deckard exactly matched only eight of the 21 instances. The “Larger Coverage” column represents the number of times a clone group contains clones that represent a larger coverage of code that included the statements that were refactored. In these instances, refactoring was only performed on part of the code range associated to the clones, an example of which can be seen in Listing 4.7 and is explained in the next paragraph.

Table 4.10

Coverage of Extract Method-related refactorings

No.	Clone Detection Tool	Exact Coverage	Larger Coverage	Total
1.	CCFinder	4	8	12
2.	CloneDR	6	9	15
3.	Deckard	8	3	11
4.	Simian	2	0	2
5.	SimScan	6	12	18

Listing 4.7 shows an example of the coverage of the tools for a sequence of refactored statements between versions 4.0.5 and 4.2.0 of JBoss in the `EjbJarDDObjectFactory` class. Lines containing “...” were truncated for brevity. The range of the clone detected by each tool is marked with the corresponding number as assigned in Table 4.10 (i.e., CCFinder is 1 and SimScan is 5). The refactoring that occurs is represented by the sequence of deleted lines (i.e., identified with ‘-’) being replaced by the sequence of added lines (i.e., identified with ‘+’). The refactored code in the method `getValue` is only the *Try-statement* inside the *If-statement*. Most tools (i.e., CCFinder, CloneDR, and SimScan) contain the entire method, which is more than the refactored code range and was counted as an instance of a “Larger Coverage” in Table 4.10.

Simian’s clone region contains only the method header and part of the code. Only the clone reported by Deckard is exactly the same as the range of the refactoring.

```

1 2   4 5   protected String getValue(String name, String value) {
1 2   4 5       if (value.startsWith("${") && value.endsWith("}")) {
1 2 3 4 5 -     try {
1 2 3 4 5 -         String propertyName = value.substring(...);
1 2 3 4 5 -         ObjectName propertyServiceON = new ObjectName(...);
1 2 3 4 5 -         KernelAbstraction kernelAbstraction = ...;
1 2 3 4 5 -         String propertyValue = ...;
1 2 3   5 -         log.debug(...);
1 2 3   5 -         return propertyValue;
1 2 3   5 -     } catch (Exception e) {
1 2 3   5 -         log.warn(...);
1 2 3   5 -     }
          +     String replacement = ...;
          +     if (replacement != null)
          +         value = replacement;
1 2     5     }
1 2     5     return value;
1 2     5     }

```

Listing 4.7 – Clone ranges of clone detection tools and associated refactoring.

Simian looks for similarities in the textual representation of the code. This technique may not detect well-formed clones (i.e., clones representing proper syntactic blocks of code) or clones with subtle differences not related to the syntax of the code. Hence, Simian may not report the appropriate code range associated to code that was refactored. This is mainly why Simian did not have any instances designated as “Larger Coverage” in Table 4.10. In many cases the clone range reported by Simian did not consist of a syntactically meaningful block of code. The other tools utilize more structured representations of the code such as token or tree representations and thus can report more structured clones.

4.2.3 *Sub-clone Refactoring of Deckard Clones*

In this subsection, we describe evaluations of observed refactorings related to clones in JBoss, and two additional open source projects: ArgoUML [ArgoUML, 2010] and Apache Derby [Apache Derby, 2010]. ArgoUML is an open source tool to develop UML diagrams written in Java. Apache Derby is an open source implementation of a relational database also written in Java. We used Deckard on these projects to further discover and evaluate instances of sub-clone refactoring.

Deckard characteristics. In Table 4.10, Deckard provided the most clones with ranges that are exactly the ranges of the observed refactorings. Clone detection tools typically combine smaller clones into larger clones that encompass the smaller clones to report maximal sized clones. Deckard reports both the maximal sized clone groups and groups representing smaller related clones. Because of this, Deckard not only reported a clone group of the exact code range that is refactored in Listing 4.7, but also reported a separate clone group containing a larger clone range consisting of the *If-statement* above the refactored *Try-statement*.

As a tree-based tool, Deckard can provide more syntactically meaningful clones compared to the text-based Simian tool. CCFinder, CloneDR, and SimScan also report syntactically meaningful clones, but Deckard's output consists of both the smaller clone groups and larger clone groups in which the smaller clone groups are contained. With its ability to report syntactically meaningful clones and the reporting of multiple sized clone groups, Deckard was used in a further study to observe relationships between reported clones and actual refactorings associated with these clones. JBoss was re-evaluated in

addition to ArgoUML and Apache Derby. The evaluation of Deckard in the previous subsection differs in that it only detected source files associated with refactored clones initially detected by Simian, whereas in the study described in this subsection Deckard was run independently on all source files. For ArgoUML, nine versions from 0.10.1 to 0.26 were observed. For Apache Derby, ten versions from 10.1.1.0 to 10.5.3.0 were observed. From the studies of these software artifacts, we determined properties of situations when a sub-clone, rather than the whole clone, was refactored.

Evaluation results. Table 4.11 (Refactoring Coverage) provides a summary of clones reported by Deckard that are related to refactorings observed in the three projects. The first general observation is that the number of clones detected by Deckard in JBoss that are associated with *Extract Method*-related refactorings is more than the results from Simian (i.e., 36 with Deckard compared to 21 with Simian). This larger number is mainly because Deckard can detect more structured clones that can contain superficial differences, which Simian does not identify as clones. Although Simian provided fast processing, its results were less desirable. In addition, refactoring performed on a selection of clones in a clone group is also included. The second general observation is that although Deckard provides smaller sized clone groups representing syntactic blocks below the maximal sized clone group, the observed refactorings related to some of these clones still only account for part of the code associated to the clone ranges. In Table 4.11, 14 instances in JBoss, 9 instances in ArgoUML, and 15 instances in Apache Derby represent refactorings that were not performed on the entire code range of clones detected by Deckard.

Table 4.11

Refactoring coverage and code properties

Property		JBoss	ArgoUML	Derby
Refactoring	Exact clone coverage	19	17	12
Coverage	Sub-clone coverage	14	9	15
Coverage Levels	Same level	4	4	6
	1 level above	9	2	8
	> 1 level above	1	3	1
Clone Differences	Refactorable	7	4	8
	Not refactorable	7	5	7

Sub-clone refactoring properties. We further evaluated the refactorings related to the “Sub-clone coverage” instances in Table 4.11 to determine characteristics that may have influenced the refactoring to be performed on only part of the duplicated code.

Deckard results

As can be seen in Table 4.10, Deckard provided the most exact matches. Its results included smaller clone groups of the maximal sized groups allowing more exact ranges to be found. Even with these reported smaller clone groups, Table 4.11 (Coverage Levels) shows that in some cases refactoring was still performed under the syntactic level of the reported clone. For JBoss and Apache Derby, this is mostly the case (i.e., nine and eight instances, respectively) as seen in the “1 level above” row. These instances suggest a practice that keeps some logic of the code at the original location for better program comprehension. For example, in Listing 4.7, the *If-Statement* is not refactored; only the

statements inside the block are refactored. This eliminates most of the duplicated code, but keeps some logic in the original location.

Excluded statements

In Table 4.11 (Coverage Levels), JBoss and ArgoUML consisted of four instances of clones being at the same level as the refactored code. Apache Derby consisted of six instances. However, some statements in the same level were not included as part of the refactoring. An example can be seen in Listing 4.8, where the first and last statements in the *If-block* were not refactored although they were part of the clone. This sequence of refactored statements occurred between versions 0.10.1 and 0.12 of ArgoUML in the `StateDiagramRenderer` class.

```

    if (edge instanceof MTransition) {
        MTransition tr = (MTransition) edge;
-       FigTrans trFig = new FigTrans(tr);
-       // set source and dest
-       // set any arrowheads, labels, or colors
-       MStateVertex sourceSV = tr.getSource();
-       MStateVertex destSV = tr.getTarget();
-       FigNode sourceFN = (FigNode) lay...
-       FigNode destFN = (FigNode) lay...
-       trFig.setSourcePortFig(sourceFN);
-       trFig.setSourceFigNode(sourceFN);
-       trFig.setDestPortFig(destFN);
-       trFig.setDestFigNode(destFN);
+       FigTrans trFig = new FigTrans(tr, lay);
        return trFig;
    }

```

Listing 4.8 – Incomplete block of refactored code in ArgoUML.

Clone Differences

Differences between clones can include variable names, literal values, and object types. The extent of a group of clones' similarity influences their possibility to be refactored. In Table 4.11 (Clone Differences), we consider the hypothetical situation in which the entire clone was refactored rather than just the sub-clone. In this case, a section of code is refactorable if it meets the pre-conditions for the *Extract Method* refactoring activity. Specifically for clone refactoring, differences in the clones should be able to be passed to the new method to allow for a generalized version of the duplicated code. For example, clones with variable name differences can be refactored by including a formal parameter for those variables in the new method signature. In all three software artifacts, the instances in which an entire clone could have been refactored and the times it could not be refactored did not differ much. This implies that in some cases the programmer could have refactored the entire clone, but did not.

Instances counted as “not refactorable” include situations where the entire clone ranges contained object type differences where a variable in one clone is declared as one type and the same variable in another clone is declared as a different type. Such a situation is more difficult to refactor. A possible way of refactoring situations with more complex differences is to use the control coupling [Lawrence and Atlee, 2006] mechanism that includes a flag to determine which extracted code to execute (i.e., for one clone execute one sequence of statements and for another clone execute a different sequence of statements in the extracted method). It was observed that the programmer did not use such flags to refactor these instances. In most cases, the programmer refactored

the more exact parts of the clones with simpler differences, such as variable names and literal values, and did not include differences such as variable types.

4.2.4 Discussion

This subsection provides some points for consideration related to the studies of clone refactoring that we have described.

Differences of clone detection results. Table 4.10 shows the variations in the results of clone detection tools that are run on the same set of source files. Even changing the configuration settings of one tool can result in a separate listing of clones. Sub-clones can be considered relative to the tool that is used such that one tool may report a section of code as a clone, while another tool reports it as a sub-clone of a larger clone. However, if we look in terms of the use of these tools to assist in finding clones for refactoring, running multiple tools to search for clones could potentially increase the effort during the maintenance process in an unnecessary way. Selecting a single tool based on the maintainer's decision provides a more straightforward process. In this case, each tool, whichever is selected, will have its own set of clones with related instances of sub-clones that would be considered for refactoring.

Exact matching clones are easier to refactor and clone detection tools can be set to report only these types of clones. However, limiting the detection to only exact matching clones reduces the ability to observe the overall cloning of the system. If clones that contain specified differences were initially detected, the results from this detection can then be followed by evaluating whether the clone group with differences should be

refactored rather than sub-clones that have more limited differences or exactly match each other. Sub-clones then can allow for more refactoring options.

Incorporating sub-clone refactoring into the clone maintenance process. Currently, if a clone reported by a clone detection tool is selected for refactoring, the process of refactoring requires many manual steps. For example, to replace several clones with a call to a new method, the method must first be extracted from one of the clone instances, which may include using a refactoring engine in an IDE. However, once the method is created each clone must be replaced with a call to that method. These are the same steps that need to be done when a sub-clone is selected for refactoring. A mechanism that can keep track of the clones and forward the necessary information to a refactoring engine upon approval of the programmer can reduce the amount of manual steps needed during clone maintenance. Based on the evaluation of sub-clones and their related refactorings in the previous subsection, a mechanism that can select sub-clones for refactoring should focus on allowing a programmer to select a sub-clone that is one or more syntactic levels below the main clone and the ability to include/exclude bordering statements.

4.3 Related Work

Related work on the analysis of clones is summarized in this section. These include analysis using IR techniques, classification techniques, and clone analysis on the operating system kernel. In addition, work related to the observation of clone evolution, refactoring between versions, and identifying crosscutting concerns with clones are also described.

IR-based program analysis. The only known work that combines the fields of IR and clone detection was proposed in [Marcus and Maletic, 2001], who used LSI on the comments and identifiers of the *whole* program to determine high-level concept clones, such as different implementations of a linked list. The detection is done by evaluating files that are strongly related to each other based on the clustering of documents (or sequence of lines of source code) that are associated to these files. The detection is based on the semantic relationships of the comments and identifiers through LSI. A related approach called *semantic clustering* uses LSI to find topics instead of clones [Kuhn *et al.*, 2007]. The corpus of this approach is also the comments and identifiers of the whole program. In both cases the entire source code must be evaluated after being divided into documents. Furthermore, no structural information is considered in the detection of the clones (before the manual evaluation by the user). Our approach incorporates a type of *filtering* by first performing clone detection on the source code and thus only sections of code that are known to be clones based on their token representations are considered in the LSI process.

Clone classification techniques. Similar to the goal in this research to group clones in order to provide better understanding of them, several classification methods have been proposed. Kapser and Godfrey provide a categorization based on the locations of clones in a clone pair [Kapsler and Godfrey, 2004]. These clones can be in the same or different blocks, functions, files, and directories. As part of an effort to perform refactoring on clones, Balazinska *et al.* provide a classification based on differences between clones

[Balazinska *et al.*, 2000]. Single token differences include the types of global and local variables, parameters, and return value. Also, differences in a sequence of tokens corresponding to an expression or statement are considered. These differences are used to determine the potential opportunity for refactoring. Koni-N'Sapu classifies clone groups in an object-oriented program based on their respective locations in the class hierarchy [Koni-N'Sapu, 2001]. The clones can be in sibling groups, parent-child groups, groups in a common hierarchy, and groups in different hierarchies. These approaches utilize the syntax and structure of the clones to perform the classification. The relationships among the clones from these classification techniques are concrete, because the classifications are based on well-defined similarity properties. However, the classifying of clones is based upon stringent requirements of the predetermined syntax and structural similarity properties. Classifying the clones through a process that is less stringent and can yield additional relationships will complement the syntax-and structure-based classification techniques. Instead of focusing on the syntax or structure of the clones, our approach utilizes the identifiers of the clones to cluster the clones. Whereas current classifications provide concrete relationships among the clones, the clusters in our approach are not dependent on the well-defined structure of the clones and more dependent on the relationships of the identifier names between the clones.

Analysis of kernel clones. The analysis of clones contained in the source code of a kernel is provided in [Antoniol *et al.*, 2002]. The clones of the Linux kernel were observed in multiple release versions to determine the evolution of clones. Livieri *et al.* also performed analysis of clones in multiple Linux kernel releases using a distributed version

of CCFinder [Livieri *et al.*, 2007]. A heat map of the coverage of clones among the different releases was produced. Our approach is the first to focus on the clones in one release of a different kernel (e.g., the Windows NT kernel) and performs analysis of the clones after clustering rather than between release versions.

Clone evolution analysis. The evolution of clones in multiple release versions has been studied for various purposes, such as how the clones are maintained. Kim *et al.* generated genealogies of clones and provided several categories related to how clones evolved (e.g., a new clone was added or subtracted, clones were consistently or inconsistently changed) [Kim *et al.*, 2005]. The studies described in Sections 4.2.2 and 4.2.3 differ from [Kim *et al.*, 2005] in that they are specifically looking at the properties of the code where refactoring occurred, whereas Kim *et al.* focused on characteristics of clones that made refactoring unsuitable.

The work in both [Aversano *et al.*, 2007] and [Krinke, 2007] looked at how consistently clones were maintained in terms of keeping the code associated with clones of the same group consistent with each other when an update is required. They provided overall trends of how clones were consistently or inconsistently changed during a specific time frame, whereas our evolution analysis mainly considered refactoring of clones between two versions of the same source code.

Refactoring identification. Some works have looked at the instances of refactoring between versions in general without a specific focus on refactoring of clones [Counsell *et al.*, 2006] [Schofield *et al.*, 2006]. However, these focused on giving an overall view of

the activity of refactoring. We focus specifically on refactoring instances related to clones resulting in the observation of the relationship between the ranges of refactored code and the actual reported clone. Weißgerber and Diehl proposed a technique to detect refactorings, where a clone detection tool was used [Weißgerber and Diehl, 2006]. However, the detection tool was not used to determine clone-related refactorings, but rather to improve the results of the technique. Clone-related refactorings could be part of the results of the technique, but a post-processing step must be done to identify them.

Clone range analysis. A comparison of the line ranges of clones reported by clone detection tools and the lines annotated as crosscutting concerns was studied in [Bruntink *et al.*, 2005]. Our evaluation of multiple clone detection tools looks at instances of refactoring that had already occurred between two versions, whereas in [Bruntink *et al.*, 2005] the crosscutting concerns were not already changed into aspects, but rather were determined at the beginning by a human observer.

4.4 Summary

This chapter has described two approaches that evaluated clones obtained from one or more clone detection tools. In both cases, a large amount of clone data was processed and analyzed to determine additional clone properties. In the first case, an information retrieval process was introduced in which a large number of clone groups (46K CLOC) in the Windows NT kernel source code are clustered using LSI to help determine relationships among the clone groups. The results of the process yield connections between clones in the clone groups that would not be detected by a clone

detection tool alone. These connections range from variations in the syntax of the clones to the use of the clones in different contexts based on the code surrounding the clones. This information could assist a programmer to both understand how the clones are used and to assist when maintenance of the clones is required.

In the second case, we described the analysis of clones and their actual refactorings obtained from source code changes between consecutive versions of the source code. According to our evaluation of running different clone detection tools on open source software artifacts, the refactoring of parts of clones or sub-clones is evident in several cases. We conclude that sub-clone refactoring should be included in the clone maintenance process. Such support should allow programmers to selectively determine partial ranges in a clone for refactoring within its syntactic hierarchy in addition to the exclusion of edge statements.

The analysis of clones described in this chapter has provided a means to understand properties of clones such as higher level relationships between respective clone groups and how they are actually refactored based on historic source code changes between consecutive versions. These properties inform the process of clone maintenance by identifying related clone groups that may need to be updated in addition to the single clone group being considered for maintenance. Furthermore, the opportunity of sub-clone refactoring provides an additional option in the refactoring of clones. However, the process of refactoring clones still contains gaps between the detection and analysis for maintenance purposes and the actual refactoring of the clones. The next chapter describes our effort to unify the clone maintenance process of detection, analysis, and refactoring through a coordinated process within an IDE. This work is realized through the CeDAR

Eclipse plug-in. It should be noted that the analysis techniques introduced in this chapter are currently not included in CeDAR. However, Chapter 5 will include a different type of analysis related to determining clones that can actually be refactored.

CHAPTER 5

CLONE REFACTORING

Clone maintenance relates to the activity of updating the sections of code that are duplicates of each other. One specific activity is removing the duplication associated with the clones by improving the modularity of the code. This results in a single copy of the originally duplicated sections of code, which simplifies future maintenance in one location and limits the possibility of errors to propagate in the future. As has been described in previous chapters, the activity of refactoring can be used to eliminate the duplication associated with clones in a clone group.

A collection of tools for clone detection and analysis are available to assist the programmer with the refactoring of clones. However, connections between the tools are still limited and require several manual steps to be performed by the programmer. One important step towards the end of the process delegates the actual refactoring of the clones to the programmer. In this chapter, we introduce an effort to reduce the manual steps needed to refactor clones. Extensions to Eclipse's refactoring engine are made to allow clones with additional properties to be refactored in a more automated manner, which in some cases eliminates the need for the programmer to be delegated the task of manually refactoring the clones. This process also includes obtaining and forwarding the results from a clone detection tool to the refactoring engine. This approach is realized in the CeDAR Eclipse plug-in [Tairas and Gray, 2009b] [CeDAR, 2010].

5.1 Clone Maintenance Process

In general, the process of removing the duplication associated with clones can be considered in three phases: detection, analysis, and refactoring, as seen in Figure 5.1. The figure also lists current state-of-the-art tools that can be utilized in each phase, which can assist the programmer in that specific phase. The following paragraphs provide further details concerning these supporting mechanisms and the connections between the phases that are either automatic or manual. Limitations with current capabilities are given that motivate the work described in this chapter.

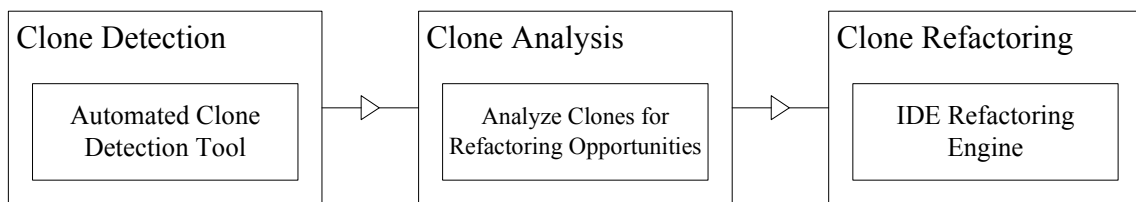


Figure 5.1 – Clone maintenance process support.

Clone detection. Consider a scenario where a programmer attempts to eliminate clones through refactoring. The programmer can utilize an automated clone detection tool to look for clones in the source code that he or she maintains. These tools provide varying types of reports to the programmer. Most provide a textual file containing the detected clones (e.g., Deckard and Simian), while others incorporate their results within an IDE (e.g., SimScan). As described in Chapter 3, some provide a graphical representation of the clones (e.g., CCFinder). However, information about clones from these tools, such as

the location of the clones, currently must be forwarded to the clone refactoring step in a manual way.

Clone analysis. The reports from clone detection tools can be used by the programmer to analyze and determine clone candidates for refactoring with the purpose of removing their duplication. As described in Chapter 4, depending on the size of the software and level of cloning in the source code, the number of clones returned by a clone detection tool can be large (e.g., evaluating over several thousand clones). Several automated techniques can assist maintainers of code in deciding potential refactoring opportunities for large amounts of clones. Such techniques evaluate properties such as the location of clones in the class hierarchy of a software system (SUPREMO) [Koni-N'Sapu, 2001] and offer metrics related to the number of variable references and assignments (ARIES) [Higo *et al.*, 2004]. A limitation found in these techniques is that they only provide clones that have potential to be refactored, which may include clones that cannot be refactored. In addition, the actual refactoring of the clones is delegated to the maintainer of the code. That is, after suggesting specific clones for refactoring, these techniques pass the responsibility of refactoring to the maintainer, who will need to perform the refactoring task based on the informed analysis.

Clone refactoring. After deciding the clones for refactoring, the programmer must actually perform the activity of refactoring. IDEs such as Eclipse provide refactoring support through its refactoring engine. This support allows programmers to delegate the code structure changes to the refactoring engine and reduces errors that may occur if

changing the code was done manually. However, the clone refactoring step of feeding the clone information to the refactoring engine is still required as a manual process. After obtaining information from the clone detection tools from the detection phase or even information about clones that could potentially be refactored from the analysis phase, this information must still be passed manually to the refactoring engine. This is due to the input mechanism of the engine which requires the programmer to select the individual code section that needs to be refactored.

The limitations described in the detection, analysis, and refactoring phases provide the motivation for our work. There is a need to introduce a mechanism that incorporates the phases in such a way that information from a preceding phase is utilized in the subsequent phase. Thus, the goal is to streamline the clone maintenance process as it relates to clone elimination. The contributions of the work described in this chapter, which are realized through the CeDAR prototype Eclipse plug-in, are given below:

- Incorporating clone detection tool results in the clone maintenance process by communicating these results to the refactoring engine of an IDE to allow more steps to be automated in the clone maintenance process.
- Utilizing the checking of pre-conditions for clone refactoring as extended in the refactoring engine to filter clone groups that can be refactored from clone groups that have potential to be refactored.
- Extending the refactoring capabilities of an IDE to allow for more types of clones that can be refactored.

5.2 Combining Clone Detection, Analysis, and Refactoring

In this section, we provide details of our contributions to the process of clone maintenance as it relates to clone refactoring to eliminate the duplication. The phases of detection and analysis can be considered as the preprocessing steps before the refactoring of the clones. That is, the ultimate goal is the refactoring of the clones, but initial steps that need to take place before this include the detection and analysis of the clones. This process is realized through the CeDAR prototype plug-in for Eclipse.

Table 5.1

Clone detection tool results availability

Parsable text file results	ConQAT, CloneDR, CPD [CPD, 2010], Deckard, Duplo [Duplo, 2010], Scorpio [Scorpio, 2010], Simian, SimScan
Text files requiring additional processing	CCFinder, CloneDigger [CloneDigger, 2010]
No text file output	DMF [DMF, 2010], SDD [SDD, 2010]

5.2.1 Clone Detection Tool Results as Input

Table 5.1 summarizes the availability of textual output from various clone detection tools that are freely available for download. CCFinder, CloneDR, ConQAT, Deckard, Simian, and SimScan have been introduced in previous chapters. Similar to SimScan, DMF and SDD are both implemented as plug-ins for Eclipse. CloneDigger, CPD, Duplo, and Scorpio are standalone tools, each of which works on a specific representation of the source code (i.e., tokens, ASTs, or program dependence graphs).

Table 5.1 also identifies whether additional processing of the output is needed in order to obtain the reported clones. As described in Section 2.1.2, CCFinder requires

additional processing, because the reported clones are identified with their token ranges rather than line ranges. The majority of the tools listed in the table provide some type of textual output that can be parsed for their results. As with the work of representation in Chapter 3 and analysis in Chapter 4, the initial step performed by CeDAR is to parse the results from a clone detection tool. From the tools listed in Table 5.1, CeDAR can currently parse CCFinder, CloneDR, Deckard, Simian, and SimScan output results. The results are utilized within Eclipse for further analysis and maintenance purposes. In a separate work, we have investigated standardizing the results of clone detection tools using MDE techniques [Sun *et al.*, 2008].

The capability to obtain clone information from different detection tools allows for the use of the results in subsequent clone maintenance activities. With the availability of more than one clone detection tool, a question that may arise is which tool provides the best results. This can be interpreted as a tool providing the most complete result set or a tool providing the least false positives or negatives. Comparisons of clone detection tools have been performed (i.e., [Bellon *et al.*, 2007]), but a consensus on the best tool has not been reached. When considering clone refactoring, appropriate clone regions for refactoring should consist of well-formed blocks of code or syntactically meaningful blocks of code. Tree-based detection tools such as CloneDR, Deckard, and Simscan can provide such results because the results are represented by a sub-tree within the parse tree or AST of the code. As a tree-based clone detection tool, Deckard is used as the tool of choice in the subsequent sections of this chapter. Text-based or even token-based detection tools can sometimes include clone regions that are in between blocks or statements. However, some of these tools try to alleviate this problem. For example, the

detection settings for CCFinder include a *shaper level* setting, which when set to “hard” will only allow clone candidates whose token sequences are enclosed by a block.

5.2.2 *Analysis for Refactoring Opportunities*

Tools such as ARIES [Higo *et al.*, 2004] and SUPREMO [Koni-N’Sapu, 2001] provide a categorization of clone groups to filter the clone groups that have potential to be refactored. However, these filtered clone groups may include groups that cannot be refactored. The determination of the groups that can actually be refactored from the listing of potentially refactorable groups must be done manually. For example, the results of ARIES were manually evaluated to determine which groups can actually be refactored [Higo *et al.*, 2004].

By extending the refactoring engine to support more instances of clone refactorings, which is explained in the next section, CeDAR can determine whether a clone group that has potential to be refactored, as informed by tools such as ARIES and SUPREMO, can actually be refactored. This is done by utilizing the feature of the Eclipse refactoring engine that analyzes a section of code to determine whether it meets the pre-conditions for a specific refactoring activity. With our incorporation of extensions for clone refactoring, pre-conditions can be determined for a clone group to identify whether it can be refactored. This reduces the number of clone groups that a programmer needs to evaluate and confirm for refactoring. In Section 5.4, such an activity is performed albeit with a different purpose of determining how many more refactorings can be supported with the extensions available in CeDAR. For example, when evaluating *Extract Method*,

each clone group with clones in the same file was run through the pre-condition checking step to determine whether the group can actually be refactored.

5.2.3 Clone Refactoring in an IDE

The Eclipse IDE refactoring engine provides support for clone refactoring through *Extract Method* refactoring for clones with parameterized local variables. Figure 5.2(a) illustrates the process of such refactoring within Eclipse. The user determines a code selection and activates the *Extract Method* process. The AST of the selected code is determined and, using sub-tree matching on the AST of the class where the code is located, matching sub-trees are identified. Differences of the matches are limited to local variable names (i.e., different values of the `SimpleName` AST node). In addition, in this scenario the programmer only knows the existence of clones after confirming and invoking the refactoring of one section of code. That is, the programmer does not know before selecting the section of code whether the code has duplicates in another location.

Figure 5.2(b) proposes changes to the process that incorporates the results from a clone detection tool. The results (i.e., detected clones in clone groups) are displayed and accessible within the IDE. It should be noted that ultimately it is the decision of the programmer to actually refactor any group of clones. Confirmation is given before automated refactoring is performed in the process depicted in Figure 5.2(b). Incorporating the results from the tools provides additional capabilities for the clone maintenance process, which are outlined below:

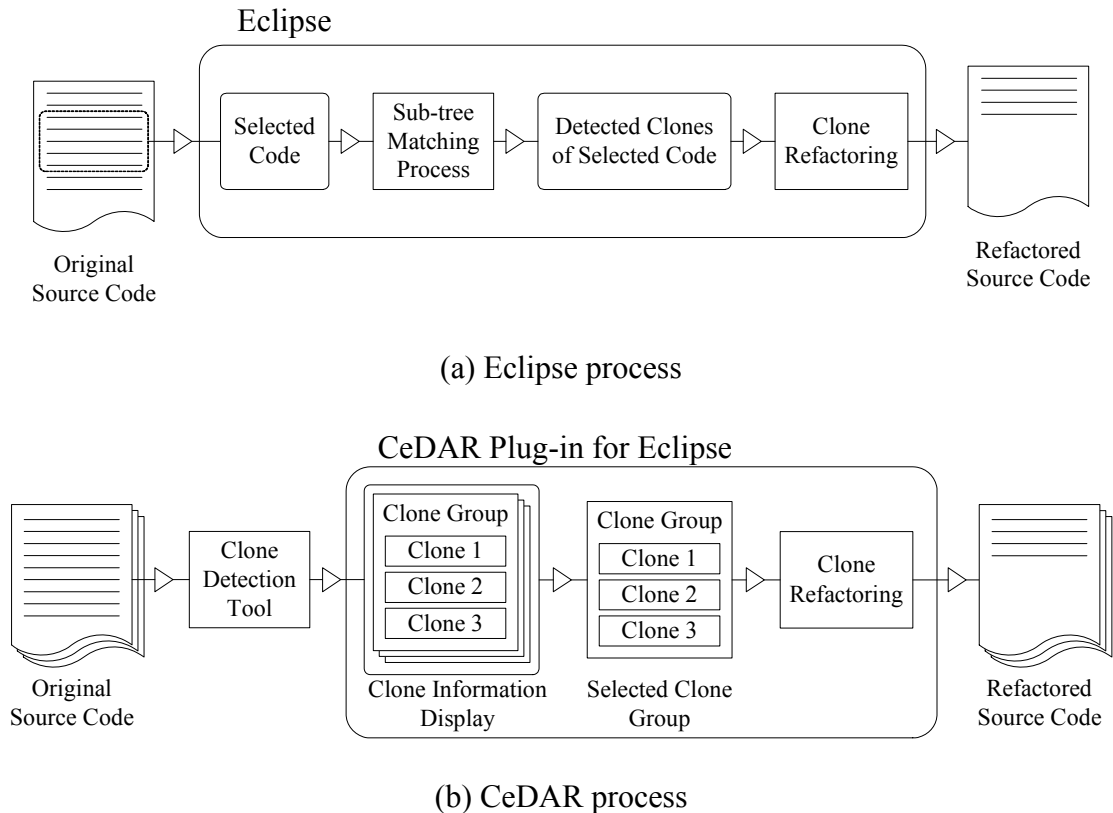


Figure 5.2 – Clone maintenance processes.

- Forwarding of clone information* – The results taken from a clone detection tool are stored in CeDAR. This allows the information of a specific clone group to be forwarded to the Eclipse refactoring engine. The need for the programmer to determine the selection of code related to a group of clones is replaced by the clone information of a selected clone group that is automatically forwarded to the refactoring engine. In this case, the delegation of the actual refactoring of the clones to the programmer is removed.
- Additional clone types detected* – As seen in Figure 5.2(b), the internal sub-tree matching process is replaced by the results of a clone detection tool. This provides information about clones before any refactoring is activated. In effect, the

programmer has prior knowledge of the clones before actually confirming the refactoring. In addition, this expands the types of clones that are reported to include clones in multiple classes, which in turn allows utilizing the clone information for refactoring options such as extracting to a utility-type class.

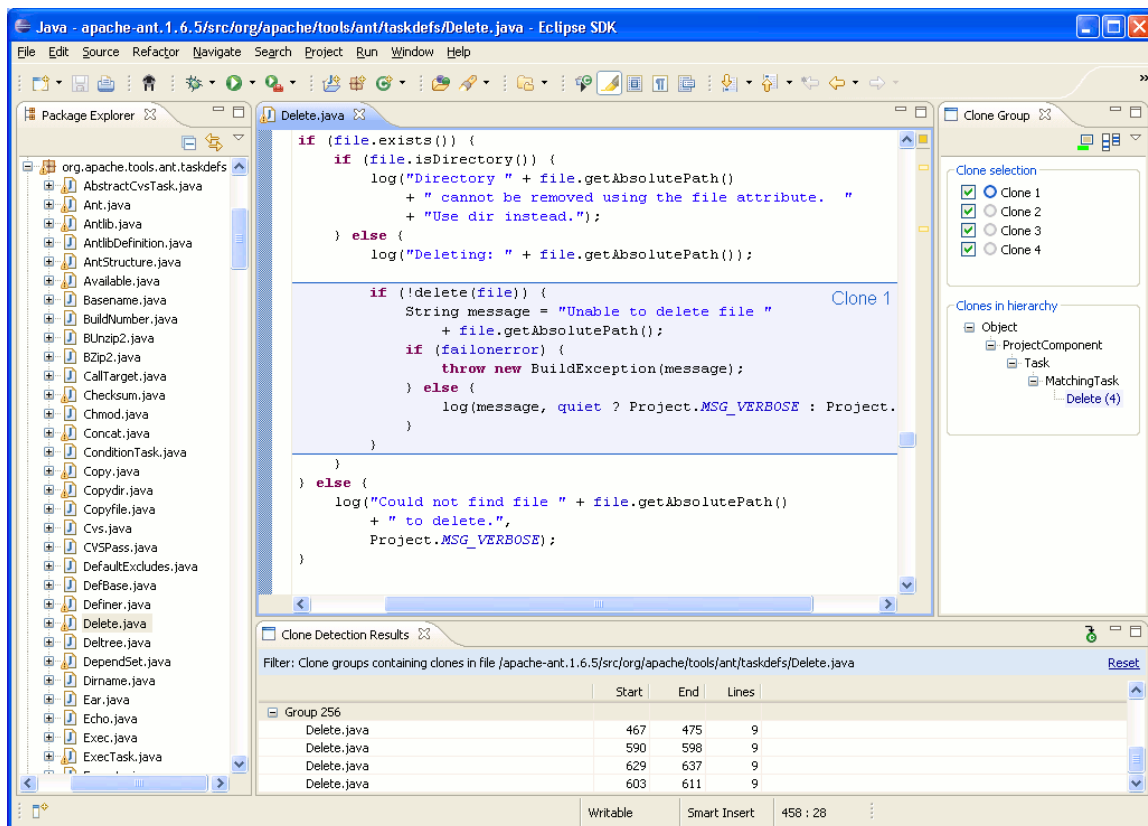


Figure 5.3 – Screenshot of CeDAR.

The next section provides further details of our extensions to the Eclipse refactoring process that enables additional opportunities for refactoring clones in a single class and in multiple classes. In both cases, the process starts with a clone group being selected by the programmer for refactoring. When a clone group is selected in CeDAR, at least one of the clones in the group is set as the *default* clone. In Figure 5.3, *Clone 1* is the

default clone and is denoted with a dark circle marker (i.e., in blue) in the “Clone Group” view. The section of code associated with *Clone 1* is also highlighted (i.e., in blue) in the source editor and bordered by horizontal lines. The remaining clones of the group are marked with a light circle marker (i.e., in grey) and are highlighted in a different color (i.e., also in grey) in the source editor (not viewable in the figure). Similar to the selection of the default clone in Section 3.2 to show a localized representation of a clone group, the purpose of specifying a default clone in this section is to determine which clone will be kept as the single remaining copy of the duplicated code being refactored. For example, the code associated with the default clone is used in the newly extracted method containing the duplicated code. This process replaces the task of selecting a snippet of code when activating a refactoring in the current mechanism in Eclipse.

5.3 Clone Refactoring Extensions

We propose extending the refactoring capabilities of Eclipse to enable additional refactorings of clones both in a single class and in multiple classes. The frontend extension of incorporating the results from clone detection tools has been described in Chapter 3. In this section, we provide details of enabling further clone refactoring capabilities within Eclipse. This is done by updating the various steps in the Eclipse refactoring framework to include consideration of new parameterized differences in the clones.

5.3.1 Refactoring Clones in a Single Class

As described in Section 2.1, clones can be categorized into different types based on their similarity properties. *Type I* clones are clones that exactly match each other. *Type II* clones provide for variations in the variable, type, or function identifiers between the clones. The Eclipse refactoring engine accepts parameterized local variable names among detected clones and can refactor these clones with *Extract Method*. However, *Type II* clones can consist of other parameterized differences. The remainder of this subsection describes our work of incorporating additional parameterized differences for *Extract Method* refactoring related to clones.

Parameterized non-local variables. We consider incorporating other variable accesses in addition to the local variable currently allowed by the Eclipse refactoring engine. These include incorporating fields in the same class of the clones and from external classes. Fields in the same class of the clones are identified by extending the local variable identification process currently available in the Eclipse refactoring engine. For fields from external classes, an additional process is used to compare `QualifiedName` nodes between the clones.

Parameterized method calls. Modularizing code clones with parameterized method calls (i.e., `MethodInvocation`) considers similar requirements as those of parameterized variables. For example, the parameterized method calls within the clones should return the same type. This will allow the calls to be passed to the newly extracted method with the same argument type in the newly extracted method's signature. However, additional

requirements that are more specific to parameterized methods must be considered. This includes instances where methods do not return a type (i.e., void return type). These methods cannot be included as a formal parameter in the call to the newly extracted method. A possible solution for this is to pass a flag that determines which part of an *If-Statement* is executed where the *If-Statement* contains separate calls for each parameterized methods. Such a situation can be considered “Control Coupling” [Lawrence and Atlee, 2006], as one method is determining the execution of another method. However, such instances reduce the independence of the methods and introduce coupling between the methods, which can be less desirable in practice.

Based on refactoring pre-conditions, not all clone groups with parameterized methods that are reported by a clone detection tool can be refactored. Figure 5.4 details the filtering done among the candidate clones with parameterized methods. It is worth noting that the filtering of methods with no arguments was done to limit the scope of the investigation of the technique for clone refactoring.

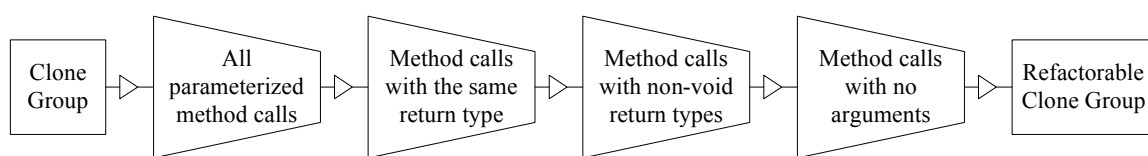


Figure 5.4 – Filtering of clones with parameterized methods.

Figure 5.5 outlines the class responsible for performing *Extract Method* refactoring (i.e., `ExtractMethodRefactoring`). Eclipse’s refactoring framework consists of several steps that perform tasks such as initially checking the section of code being prepped for refactoring to determine whether pre-conditions are met and actually

changing the code associated with the refactoring. This allows the framework to provide features such as user input, preview of the refactoring, and an undo mechanism if an executed refactoring must be reversed. In Figure 5.5, method `checkInitialConditions` performs tasks such as finding sections of code that are duplicates of the initially selected code, in addition to checking the pre-conditions for refactoring. These duplicates can consist of parameterized local variables. Method `createChange` performs the refactoring, which includes the generation of the newly extracted method and replacement of all identified duplicates with a call to the newly extracted method.

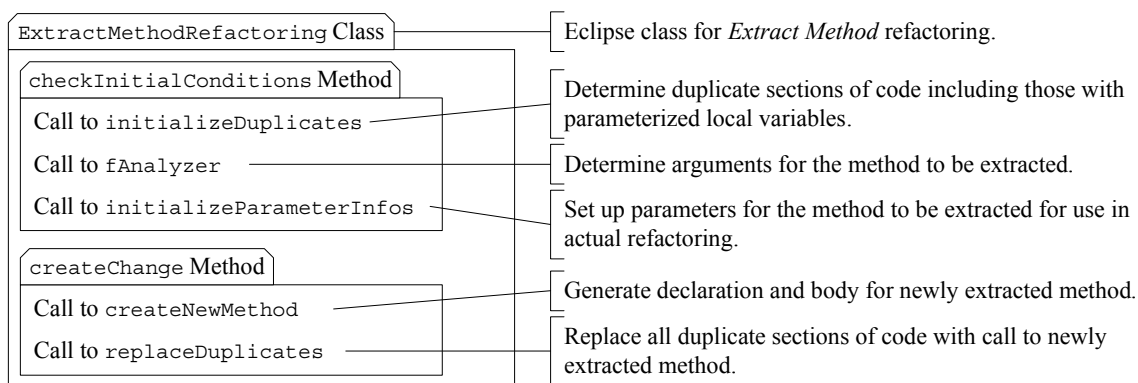


Figure 5.5 – Outline of `ExtractMethodRefactoring` class in Eclipse.

As can be seen in Figure 5.5, a framework to eliminate duplicate sections of code through *Extract Method* refactoring is available in Eclipse. However, the refactoring of clones to include additional parameterized differences requires extensions on the current process. These extensions within the associated tasks in Figure 5.5 where changes were made are described in the following paragraphs.

Call to initializeDuplicates. Section 5.2.1 and Figure 5.2(b) describe the incorporation of an external clone detection tool as the input to identify the duplicated sections of code. In this case, a clone group reported from the tool and observed by the programmer can be selected for refactoring. This process also includes finding the parameterized elements of the clones, which in addition to local variables will also include non-local variables and method calls. Both local variables and fields in the same class are identified by comparisons of `SimpleName` nodes. Because initially only local variables are included, the comparison of `SimpleName` rejects those nodes that are associated with fields. This rule is removed and additional tasks are included in subsequent steps (i.e., in “Call to `fAnalyzer`”) to incorporate fields. New comparisons are also included for `MethodInvocation` and `QualifiedName` nodes to find parameterized method calls and fields from external classes. Related to method calls, the filtering of method calls as illustrated in Figure 5.4 is done at this point.

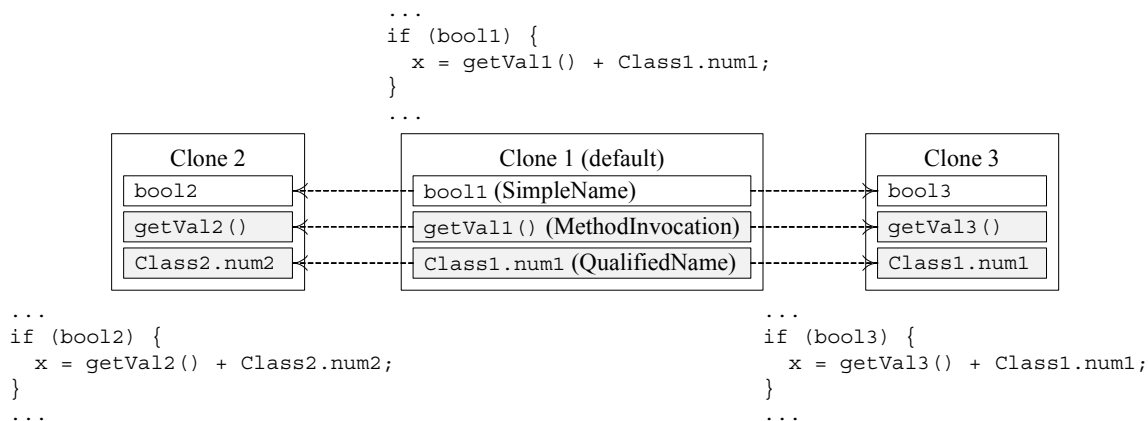


Figure 5.6 – Mapping of parameterized elements.

All parameterized elements (i.e., `SimpleName`, `MethodInvocation`, and `QualifiedName`) are associated among the clones in the group by mapping the elements in the default clone with the elements of the other clones. Figure 5.6 illustrates this mapping where, for example, the variable `bool1` in the default clone (i.e., *Clone 1*) is mapped with the variables `bool2` in *Clone 2* and `bool3` in *Clone 3*. We extended the mechanism that originally mapped only local variables to also map method calls and fields from external classes.

```

public class A {
    int field1;
    int field2;

    public void method() {
        {cloned statements}
        {reference to field1}
        {cloned statements}
        ...
        {cloned statements}
        {reference to field2}
        {cloned statements}
    }
}

public class A {
    int field1;
    int field2;

    public void method() {
        newMethod(field1);
        ...
        newMethod(field2);
    }

    public void newMethod(int field) {
        {cloned statements}
        {reference to field}
        {cloned statements}
    }
}

```

Figure 5.7 – Including parameterized fields.

Call to fAnalyzer. This is a call to an object of type `ExtractMethodAnalyzer`, which among other things determines the arguments that will need to be passed to the newly extracted method. Initially, such arguments only include local variables declared outside the scope of the selected snippet of code, but still within the block of the method body. Field variables were not included, because they are globally accessible in the class and thus do not need to be passed. However, clones may use different fields in their

corresponding code. For example, in Figure 5.7, two sections of code contain cloned statements that include references to `field1` and `field2`, respectively. This leads to the existence of parameterized fields within the clones. In order to refactor the clones, the fields must be passed to the extracted method (as seen in the right side of Figure 5.7).

To incorporate parameterized fields, the algorithm in Listing 5.1 extends the argument list to include any fields that were in the list of parameterized variables as identified in previous steps. Each parameterized variable is evaluated to determine if it is already included in the original list of arguments (i.e., `ContainsVariable`). In addition, `IsSingleField` checks whether the variable is a field and if the field binding differs between at least two of the clones in the group. This latter task is done by observing the mappings of elements, as seen in Figure 5.6. If the variable is both not already in the original list of arguments and the variable is a field that represents more than one variable binding, then the variable is included in the list of arguments. The output of the function in Listing 5.1 is a new list of arguments that take into account any parameterized fields.

```

1: function ADDFIELDS(L: orig_args_list, PV: parameterized vars): new_args_list
2:   for each parameterized variable V in PV do
3:     if !ContainsVariable(L, V)  $\wedge$  !IsSingleField(V) then
4:        $L \leftarrow L \cup V$ 
5:     end if
6:   end for
7:   return L
8: end function

```

Listing 5.1 – Including parameterized fields in argument list.

Call to initializeParameterInfos. Information related to identified parameters for the method to be extracted are stored for usage during the actual refactoring of the code. The

identified parameters relate to the local variables and newly included fields obtained from the previous step. These parameters are supplemented with additional parameters representing fields from external classes and method calls that are identified in the first step. With the identification of parameterized variables and method calls, a scenario may arise where both elements are passed separately. For example, in Figure 5.8, the variable `p` and the call `p.call()` could potentially be passed separately in the call to the new extracted method (middle column of Figure 5.8). To avoid this situation, the list of parameters must be evaluated to remove unnecessary passing of duplicate variables (right column of Figure 5.8).

<pre>public void method() { ... {cloned statements} {reference to p} {reference to p.call()} {cloned statements} ... }</pre>	→	<pre>public void method() { ... newMethod(p, p.call()) ... }</pre>	<pre>public void method() { ... newMethod(p) ... }</pre>
--	---	--	--

Figure 5.8 – Removing duplicate variables.

The algorithm in Listing 5.2 outlines the steps to remove the duplicate variables in the parameters list. The first step is to obtain all instance variables that are associated to a method call within the code range of the default clone (line 2). For each of these variables, we determine if their corresponding method call parameters can be removed from the parameters list (lines 7-20). In line 5, the method calls associated to the instance variable are retrieved. `IsUniformCall` looks at all of the corresponding method calls in the other clones to determine if the calls are the same. If all calls are the same, then these method calls can be removed from the parameters list as the associated variable can be

used instead. This is done by including each method call (line 11) in the original parameters list in the new parameters list (i.e., NPL) (line 14), if the method call is not associated to the variable V (line 13). All other parameters that are not method calls are automatically added in the new parameters list (line 17), which includes the variable V .

```

1: function REMOVEDUPS( $PL$  : original_parameters_list): new_parameters_list
2:    $V \leftarrow$  variables of method calls in  $PL$ 
3:   for each variable in  $V$  do
4:      $UNIFORM \leftarrow$  true
5:      $MCL \leftarrow$  method calls of variable  $V$  in default clone code range
6:     if !IsUniformCall( $MCL$ ) then
7:        $UNIFORM \leftarrow$  false
8:     end if
9:     if ( $UNIFORM$ ) then
10:       $NPL \leftarrow \emptyset$ 
11:      for each parameters  $P$  in  $PL$  do
12:        if IsMethodCall( $P$ ) then
13:          if !IsRelated( $P$ ,  $V$ ) then
14:             $NPL \leftarrow NPL \cup P$ 
15:          end if
16:        else
17:           $NPL \leftarrow NPL \cup P$ 
18:        end if
19:      end for
20:       $PL \leftarrow NPL$ 
21:    end if
22:  end for
23:  return  $PL$ 
24: end function

```

Listing 5.2 – Removing duplicate variables in parameter list.

Call to createNewMethod and replaceDuplicates. The parameter information from the previous steps is used to create the newly extracted method and replace the clones with a call to the extracted method. In both tasks, the additional parameterized differences were incorporated in the process. For example, within the newly extracted method body, the

method calls and fields are replaced with a new name as determined by the programmer during the refactoring confirmation process. Similarly, these same elements are included in the new method calls that replace each clone instance.

5.3.2 Refactoring of Clones in Multiple Classes

The previous subsection deals with clones contained in a single class where the associated duplicate code is extracted into a new method. Clones that reside in multiple classes require a different refactoring approach. In this subsection, we describe current capabilities of CeDAR that can assist in the refactoring of clones across multiple classes. The descriptions are grouped into two types of refactoring approaches: *Pull-up Method* and *Extract to Utility-type Class*. In both cases, the results from clone detection tools provide information about the clones, which will include clones in multiple classes.

Pull-up Method. Method-level clones in classes that extend the same super class could be pulled up in order to remove the duplication. The Eclipse refactoring engine can identify methods in sibling classes that have the same signature if a method was selected for *Pull-up Method* refactoring. However, if the clones in the group represented code below the method block, then *Extract Method* refactoring must be performed first before the newly extracted method can be pulled up. In CeDAR, clones can be selected/de-selected for refactoring. In the case of a *Pull-up Method* refactoring that requires an initial *Extract Method* refactoring, the inclusion of all clones in the group (i.e., Figure 5.9(a)) can be updated to include only a selection of the clones (i.e., Figure 5.9(b)). These selected clones in the same class can then be refactored (i.e., *Extract Method*) first, followed by

the refactoring of the remaining two clones. Then, the *Pull-up Method* refactoring can be activated on the newly extracted methods in the two classes. At this point the refactoring cannot be done at once, but the programmer does not need to select any section of code because the clone information is available and can be forwarded to the refactoring engine.

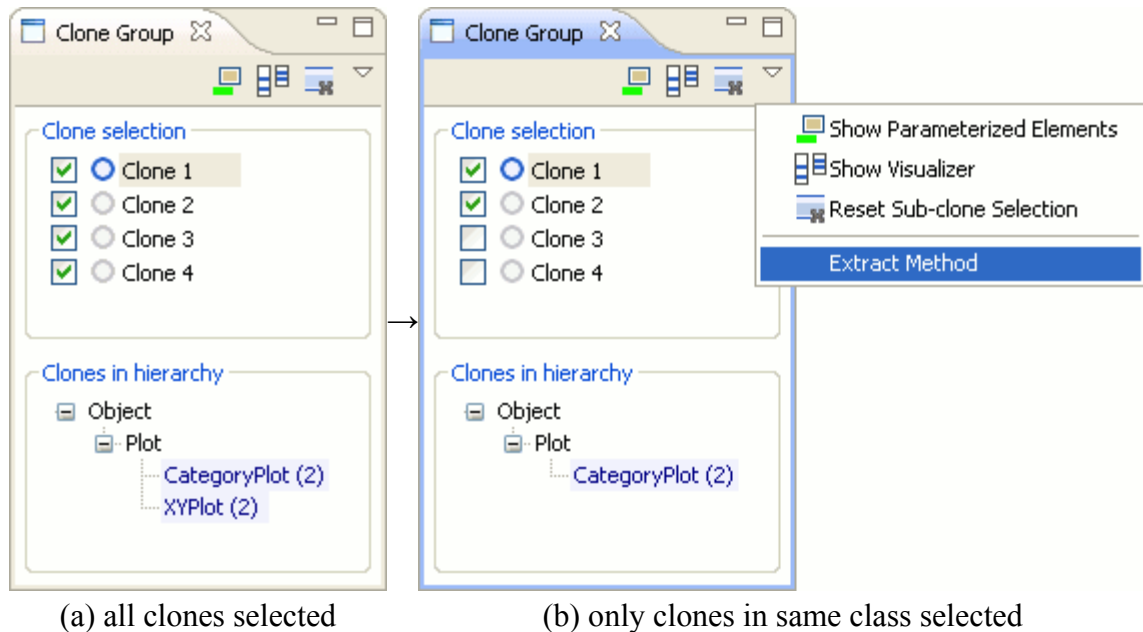


Figure 5.9 – Filtering clone selection.

Extract to Utility-type class. For clones that are scattered in unassociated classes, a possible solution is to extract a method containing the duplicated code of the clones into a separate class. More specifically, a method is extracted into a static utility-type class, where the functionality that is extracted performs some logic that is not as tightly coupled to the inner workings of a class. Such methods resemble standard Java library APIs such as those found in the predefined `Math` class.

Our effort includes an initial framework to provide a more automated mechanism of extracting the duplicate code of clones into a method in a utility-type class. In CeDAR (as seen in Figure 5.2(b)), the clone detection results provide the information of the clones located in multiple classes, which a programmer can select for refactoring. Extracting a method into a utility-type class can be considered a combination of several atomic refactoring activities. A snippet of code is initially extracted into a new method and a call to the newly extracted method replaces the code in the original location (i.e., *Extract Method* refactoring). This new method is then extracted into a new class and the call to the original method is replaced by a call to the method in the new class (i.e., *Extract Class* refactoring). When clones are involved, a programmer would need to select one of the clone instances and perform the steps described above. The programmer would then replace the remaining sections of duplicated code with the call to the method in the new class.

5.4 Evaluation

In this section, we evaluate the inclusion of refactoring clones with additional parameterized differences in several open source software artifacts. Most of the artifacts used in the evaluation in Section 3.2 were used in this section with the exception of ArgoUML and JBoss-AOP. The exclusion of these artifacts were because they had already been used in refactoring-related analysis in Section 4.2. Two artifacts unique to this evaluation include Columba, an email client application [Columba, 2010], and Hibernate, which provides mapping of Java classes to database tables [Hibernate, 2010].

We want to observe the potential increase of refactoring capabilities as compared to the current capabilities in Eclipse. As stated in Section 5.2.3, Eclipse supports *Extract Method* refactoring of clones with local variable name differences. Section 5.3 has outlined techniques to incorporate additional parameterized differences to add to the opportunities for *Extract Method*. We would like to know to what extent is the increase in clones being supported for refactoring with the incorporation of additional parameterized differences. Table 5.2 summarizes experiments on open source Java projects where clone detection was performed using Deckard. For the detection settings in Deckard, the “minimum tokens,” which sets the minimum size of the clones was set to 50. The “similarity” was set to 0.95. “Stride,” which determines how large code fragments are encoded together was set to 0. The purpose of Table 5.2 is to identify the instances of clone groups that can be refactored by Eclipse and CeDAR. It should be noted that a programmer may not refactor all instances that were identified, but the instances represent cases where Eclipse and/or CeDAR can assist the programmer in refactoring.

The column “#Cand. CG” reports the number of clone groups that satisfy the general property of *Extract Method* (i.e., all clones in one file). The table shows that the varying totals of candidate groups are not based on the program size. For example, JFreeChart consisted of the largest number of groups (i.e., 291 groups), but did not have the most lines of code. For each of these candidate groups, the *Extract Method* option in Eclipse was executed to determine how many of these groups satisfied the requirements for refactoring and thus can be refactored. To satisfy the requirements, activating refactoring on a clone group will not return any error messages from the refactoring engine (i.e., “OK” status returned). This is the step that can determine which clone groups

can actually be refactored from the results of clone detection or analysis tools as mentioned in Section 5.2.2.

The column “Eclipse” reports the number of groups that are refactorable using *Extract Method* in Eclipse. This number filters out non-refactorable clone groups such as those with multiple assigned variables within the cloned code. The column “CeDAR” reports the number of groups that are refactorable with the addition of other parameterized differences as outlined in Section 5.3. These totals include the instances that were refactorable by the original Eclipse technique. It can be seen that in half of the artifacts evaluated, the number of refactorings doubled. This demonstrates a considerable increase in instances that CeDAR provides in terms of assistance for programmers during clone maintenance.

Table 5.2

Additional Extract Method refactorings by CeDAR

Project	LOC	#Cand. CG	Eclipse	CeDAR	Δ
Apache Ant 1.7.0	67K	120	14 (12%)	28 (23%)	+14
Columba 1.4	75 K	88	13 (15%)	30 (34%)	+17
EMF 2.4.1	118 K	149	8 (5%)	14 (9%)	+6
Hibernate 3.3.2	209K	177	15 (8%)	18 (10%)	+3
Jakarta-JMeter 2.3.2	54K	68	3 (4%)	11 (16%)	+8
JEdit 4.2	51K	157	15 (10%)	20 (13%)	+5
JFreeChart 1.0.10	76K	291	29 (10%)	62 (21%)	+33
JRuby 1.4.0	101K	81	23 (28%)	23 (28%)	0
Squirrel-SQL 3.0.3	141K	75	8 (11%)	20 (27%)	+12

#Cand. CG = Number of candidate clone groups

Δ = Difference between Eclipse and CeDAR refactorings

The clone groups found only by CeDAR (i.e., representing the difference in the last column in Table 5.2) were evaluated to determine the occurrence of the types of

parameterized differences that were included in the arguments list of the newly extracted method, which is given in Table 5.3. The table includes occurrences of parameterized strings that are also included in the refactoring extensions. For example, in Apache Ant, the number of clone groups that contained at least one field (global variable) parameter in the arguments of the newly extracted method was eight. It should be noted that one clone group can be represented multiple times, because an arguments list can include one or more types of parameterized differences. As can be seen from the table, local variable parameters still comprise the majority of the clone groups as for each artifact it represents the most instances. For the remaining parameterized differences, which were added to the clone refactoring process, the next largest number of instances varies. From the artifacts evaluated, fields from external classes tend to represent the lowest number of instances compared to the other three elements. However, the evaluation of the artifacts demonstrates that each parameterized difference was utilized during varying instances of the *Extract Method* observed refactoring activities.

Table 5.3

Parameterized differences in arguments list of extracted method

Project	LV	IF	EF	MC	S
Apache Ant 1.7.0	10	8	2	8	6
Columba 1.4	14	7	7	7	5
EMF 2.4.1	6	2	0	2	4
Hibernate 3.3.2	3	0	0	2	2
Jakarta-JMeter 2.3.2	8	1	1	2	7
JEdit 4.2	4	1	1	1	2
JFreeChart 1.0.10	34	19	11	13	5
Squirrel-SQL 3.0.3	12	6	3	9	4

LV = local variable; IF = internal field; EF = external field; MC = method call; S = string

5.5 Related Work

Fanta and Rajlich perform clone removal through processes that include *function insertion*, *function encapsulation*, *variable renaming*, and *argument reordering* [Fanta and Rajlich, 1999]. The candidate clones for removal must be selected and determined manually, while the removal of clones is automated. In CeDAR, candidate clones for refactoring can be determined by filtering properties of the clones. Further filtering can be done to determine clones that can be refactored from the clones that can potentially be refactored. Balazinska *et al.* also propose an automated refactoring technique in which the Strategy design pattern [Gamma *et al.* 1995] is used to transform the clones into a more manageable format [Balazinska *et al.*, 1999]. The refactoring was highly automated, and the authors concluded in a subsequent paper that a fully automated refactoring approach is less effective compared to one with some user interaction [Balazinska *et al.*, 2000]. Juillerat and Hirshbrunner evaluate clones based on predefined constraints [Juillerat and Hirsbrunner, 2006]. However, only parameterized local variables are considered. The refactoring of clones in the C language was considered by [Baxter *et al.*, 1998], [Komondoor and Horwitz, 2003], and Liu [Liu, 2004]. Baxter *et al.* replace detected clones with macros in the C language. Komondor and Liu suggest approaches to extract a function from the clones, which is similar to *Extract Method*. These works provide extensive mechanisms for function extraction in procedural languages, but are currently not incorporated within an IDE that can provide a centralized location for source code maintenance. Li and Thompson propose code clone removal for the functional language Erlang [Li and Thompson, 2009]. This technique is incorporated within a refactoring environment for the language. The detection process is limited to the

associated detection tool for Erlang. The refactoring is performed one at a time on each duplicate code that is detected, rather than as a combined process as given in CeDAR.

It is worth noting that a separate approach to clone maintenance keeps the clones in place and performs maintenance where the clones are located [Duala-Ekoko and Robillard, 2007] [Toomim *et al.*, 2004]. In this case, the duplication of the clones is not removed. This approach links the sections of code that are duplicates of each other, allowing the programmer to just edit one instance and all other instances will be appropriately edited. A motivation for this approach is the observation that some clones are short-lived and over time become harder to refactor [Kim *et al.*, 2005]. It should be noted that the refactoring of all clones is not necessarily the ideal solution. However, when instances arise in which a clone group can be refactored, the programmer can decide to refactor the group with the aid of CeDAR.

5.6 Summary

This chapter has described extensions to the Eclipse refactoring engine with the goal of increasing the assistance to programmers during clone maintenance activities. The extensions include the incorporation of more parameterized differences among the clones to enable additional accepted refactorings. Processes before refactoring are performed that include the ability of obtaining results from a selection of clone detection tools and the ability to determine actual refactoring opportunities from potential ones through the extensions of the refactoring engine. The increase in instances of refactoring on clones was seen to double in many of the software artifacts that were evaluated.

CHAPTER 6

FUTURE WORK

Future research directions are outlined in this chapter. Section 6.1 will consider future work in which clone maintenance remains the focus. This direction will consider the expansion of current techniques and the investigation of novel techniques as they relate to improving the maintainability of clones. A separate direction will consider the techniques learned from the research of this dissertation for application in newer venues of work.

6.1 Continued Focus on Clone Maintenance

This section describes extensions to the capabilities of CeDAR through additional clone refactoring capabilities and incorporating visualizations in the refactoring process. In addition, extending the initial work of utilizing techniques from IR and MDE are described.

6.1.1 Increasing Refactoring Capabilities

Related to the clone refactoring work described in Chapter 5, we will consider additional parameterized differences for refactoring in an effort to increase the types of clones that can be refactored by CeDAR. Such differences include node types in the same syntactic location that are different. For example, where one clone uses a local variable,

another clone uses a method call in the same syntactic location. Future work on the refactorings described in Section 5.3.2 includes adding more automation to the process of pulling up a method that represents a section of code that needs to be extracted first. However, when considering any extension, an evaluation must be performed to determine the prevalence of instances that can use such a refactoring. The evaluation in Chapter 5 showed mostly a doubling of refactoring instances for detected clone groups. The inclusion of further parameterized differences should be evaluated to determine to what extent the increase in refactoring support is provided.

Related to the studies of refactorings in Section 4.2 revealing sub-clone refactoring, we are incorporating support for sub-clone refactoring in CeDAR. In this case, the localized representation of clones described in Section 3.2 will include an option for the programmer to select a sub-clone of the default clone. The corresponding statements in the other clones in the group are automatically selected, and refactoring activities that were available for the entire clone are made available for the sub-clones.

We would also like to perform additional studies to evaluate refactorings related to clones. We will consider independent evaluations of different clone detection tools through the observations of clones detected directly by the tools, as was done with Deckard in the second study in Section 4.2. In addition, we will consider a more robust differencing technique other than *diff* to identify refactorings that occur in more complex source code changes.

6.1.2 *Incorporating Visualizations in the Refactoring Task*

Some of the non-matching statements in the localized representation of clones that is described in Section 3.2 are related to parameterized differences that are currently not recognized by CeDAR. Future work on increasing the parameterized differences that are recognized by CeDAR can potentially provide a more complete visualization of the clones. As it relates to maintenance, future work will also consider the further utilization of the localized display to provide a summary of refactoring opportunities to the programmer. In this case, the refactoring options and changes for duplication removal can be visualized in the representation to provide the programmer with a localized display of the actual refactoring suggestion. A further consideration is the inclusion of the visualization technique described in Section 3.1 in the refactoring process. For example, the classes could be displayed with respect to their relationships in a manner similar to a UML class diagram. This display could reveal clones such as those that are duplicates in subclasses, which could be addressed through the *Pull-Up Method* refactoring.

6.1.3 *Clone Analysis with MDE*

The commands on clones described in Section 3.3 are initial implementations. These commands provide an evaluation of the feasibility of processing clones at the modeling level. Future work will include the investigation of additional commands on the clone representation, such as the analysis of clones with respect to determining refactoring opportunities and comparisons with techniques indigenous to Grammarware. Currently, the representation of clones is limited to a small subset of Java. To improve the representation, future work will also include the use of model weaving. The Atlas

Model Weaver (AMW) [AMW, 2010] provides the framework necessary to weave two metamodels through the linking of element relationships between the two metamodels. In this case, a metamodel representing the full Java language is weaved with a metamodel representing clones in the code.

6.1.4 Clone Analysis with IR

Clone detection tools typically offer various control settings that can influence the detection process and types of clones that are reported to the user. Some settings are related to the strictness criteria for detection. It would be interesting to perform a comparison of the clone group clustering technique described in Section 4.1 with the results of a tool in which the most liberal detection configuration was selected (i.e., the least strict setting). In terms of the corpus that is used, including the words in the comments as possible terms may be considered in future clone clustering. In addition, the observation of the most influential terms (or identifier names) in the generated clusters was done manually, which was part of the whole process of understanding the clones. Future work will consider statistically determining which identifier names are the most influential for each of the generated clusters. This should give greater insight into the relationships of the clones in the clusters.

Additional considerations will include efforts to improve the quality of the results of the process. For example, stemming the identifiers to obtain the root terms and separating terms hidden within the identifiers will be done. A separate process will consider an alternative way of generating the term-document matrix where only the count

of unique identifiers in each document or clone group is recorded rather than every instance of the identifier in the clones of the groups.

Further future directions will include the use of other techniques to determine associations among the clone groups. The Latent Dirichlet Allocation technique, which has been compared with LSI [Lukins *et al.*, 2008], will be considered. The n-gram [Manning and Schütze, 1999] natural language processing technique will also be considered although this technique may require observing the structural properties of the clones.

6.2 Broader Application of Work

This section describes utilizing the techniques from the research described in the dissertation in novel directions. The topics considered for broader applications of the research include the analysis of clones for additional properties such as how clones are scattered within the source files. In addition, we will consider the use of information retrieval techniques to understand properties of models. We will also investigate a related topic on the use of ontologies to generate metamodels.

6.2.1 *Additional Clone Property Analysis*

The evaluations and case studies in the research described in this dissertation have considered both commercial and open source software. For example, the Microsoft NT kernel was used in Section 4.1 and several Java open source software artifacts were used in Section 4.2. Commercial software are typically closed in the sense that only a select few programmers have access to the source code and provide the development expertise

for the product. This is in contrast with open source projects where contributions from the public are generally welcome, which can swell the number of programmers involved in the project. A comparison of the level of cloning between open source and proprietary projects would be interesting to determine how much cloning exists and if unique characteristics of clones are evident. For example, a better understanding of the degree of clones and the clone types might provide some insight into how different development approaches lead to duplicated code. This is a future area of work that we plan to investigate.

A separate analysis will focus on outlier clones. These clones exhibit the property of being separated from the main group of clones within the source files of a program. For example, an outlier clone or clones may be distanced from the remaining clones in a group that are close to each (i.e., the same class hierarchy). This future work will investigate the properties of such clones by considering reasons that include crosscutting concerns, by analyzing the surrounding code to determine if there is a dominating concern; independent code that performs a specific task, but is not necessarily part of the main function of the class; and decaying design that may require moving the outlier clones closer to the main group of clones. A further outcome is the use of the results of this study to inform instances of clones exhibiting crosscutting properties that can potentially be replaced with AOP-based aspects.

6.2.2 Information Retrieval and Model Analysis

As MDE becomes more popular and more MDE-related artifacts become available, we would like to investigate the use of IR techniques to consider relationships

among elements in models and metamodels. This investigation can potentially yield a type of semantic connection among the elements similar to the relationships shown among clone groups as described in Section 4.1. In the MDE space, the lexical relationships within the models and/or metamodels are considered and evaluated rather than the graph-based or grammar-based properties of the associated DSL.

In addition, related to the topic of DSLs, continuing our work on the utilization of ontologies in the domain analysis of DSLs [Tairas *et al.*, 2008], we would like to investigate the generation of a metamodel using the information obtained from an ontology that would inform a domain-specific modeling language. This previous work has considered ontologies in the generation of a class diagram for a DSL.

CHAPTER 7

CONCLUSION

The availability of clone detection tools offers developers an automated means of finding clones in software. However, the knowledge of the existence of clones through these tools raises the need to perform the appropriate maintenance of associated clones as part of the overall software maintenance process. This maintenance activity can consist of a program comprehension effort to further understand the clones and the consideration of removing the duplication associated with the clones. In Chapter 1, challenges related to such software development activities were given, specifically related to the representation, analysis, and refactoring of clones. These challenges motivated the research described in this dissertation focusing on objectives in the areas of clone comprehension through representation and analysis, and clone maintenance associated with the refactoring of clones to remove the duplication. The following sections describe the contributions of the research in these areas. These contributions can be summarized as follows:

- Visualization and representation of clones at the clone group level and a transformation-based clone analysis approach.
- The discovery of additional clone properties related to the semantic relationships of clone groups, and refactoring of partial clones.

- A unified clone maintenance process that reduces the manual steps required for refactoring and increases support for refactoring of different clone types.

7.1 Clone Representation

In Chapter 3, we advocated the use of novel clone visualization techniques: aspect browser-like and localized clone representations. These representations supplement evaluating clones through the original textual reports from clone detection tools and the actual source files associated with the clones. The visualization extension of the AJDT Visualiser plug-in offers a graphical representation of the clone locations in the source files and can be an alternative to the scatter plots for purposes of viewing clone group information. The localized representation of clones offers a visualization that is localized in one location providing a quick summary of the clones to the programmer. Both visualization techniques are incorporated into the CeDAR Eclipse plug-in.

In addition to the visualization techniques, we have also investigated the use of MDE to represent and analyze clones. The representation through CoCloRep is textual, but the instances of CoCloRep can be projected to the MDE technical space and analysis of the clones can be done through model transformations. This work considered the use of transformations that consist of both declarative and imperative rules to transform the source model (i.e., CoCloRep) into a target model revealing analysis results for the clones.

7.2 Clone Analysis

Our contributions in the area of clone analysis provide a novel understanding of clones through the discovery of additional clone properties. These properties were revealed through two separate works on the analysis of clones in both commercial and open source software artifacts as described in Chapter 4. Although these works have not been fully incorporated in the unified process of clone maintenance in CeDAR, they still provide relevant maintenance-related information that is described in the following paragraphs.

The first work in Chapter 4 revealed higher level relationships among clone groups through the clustering of these groups based on the LSI technique. The relationships are based on semantic properties (i.e., identifier names) rather than structural properties used by most clone detection tools. While the use of LSI to detect higher level clones has been proposed before, our approach offers a novel combination of both structure-based clone detection and semantic-based clustering. From a maintenance perspective, the knowledge of relationships among clone groups allows programmers to consider multiple clone groups that form a semantic-based relationship that was not originally reported by a structure-based clone detection tool.

The second work in Chapter 4 revealed instances of sub-clone refactoring occurring on clones reported by clone detection tools when the changes associated with these clones were evaluated in consecutive versions of the source code. In some cases, programmers refactored only part of the originally reported clone range (i.e., a sub-clone). The range of a reported clone can vary among clone detection tools, where a clone in one tool is a sub-clone in another tool. However, the use of a single tool in the

maintenance process still forces a fixed reported range for each clone, where in some cases only the sub-clone is considered for refactoring. Based on this work, we advocate incorporating a mechanism to support sub-clone refactoring within the general clone refactoring process.

7.3 Clone Refactoring

Concerning the activity of clone removal through refactoring, we have contributed a more unified process within an IDE that is realized through CeDAR. The manual steps that must be performed by the programmer in order to refactor a group of clones are reduced in CeDAR. The results from a clone detection tool are obtained and incorporated within the IDE (i.e., Eclipse). Integrating the results of clone detection tools in the IDE through CeDAR removes the need for the programmer to forward the necessary information about the clones to processes inside the IDE. Utilizing techniques introduced in Chapter 3, CeDAR also allows programmers to evaluate clones through visualizations.

CeDAR also includes a mechanism that can determine whether a clone group can actually be refactored (i.e., they meet specific pre-conditions for a refactoring) from a list of clone groups that have potential to be refactored. This provides a further filtering that reduces the number of clone groups to be presented to the programmer and removes the need for the programmer to manually evaluate potentially refactorable clones. Finally, the information about a selected clone group can be forwarded to the Eclipse refactoring engine for the purpose of refactoring. This process eliminates the need for a programmer to select each section of code associated with a clone group, which was a main

disadvantage of current approaches related to the analysis of clone refactoring opportunities. The process also removes a current limitation of reporting clones only after a section of code is selected for refactoring.

In addition to the reduction of manual steps in the clone refactoring process, CeDAR demonstrates increases in the types of clones for refactoring. The additional parameterized differences considered in Chapter 5 through extension of the refactoring engine allow CeDAR to refactor more instances of clones. This can be seen in the evaluation of several software artifacts where the number of clones supported for refactoring doubled in many of the evaluated artifacts. This can serve as an alternative to the current internal clone detection process in the refactoring engine, which is limited in terms of the parameterized differences that it supports.

LIST OF REFERENCES

- [Abyss, 2010] Abyss Web Server, 2010, <http://abyss.sourceforge.net>.
- [AJDT Visualiser, 2010] AspectJ Development Tools Visualiser, 2010, <http://www.eclipse.org/ajdt/visualiser>.
- [AMW, 2010] Atlas Model Weaver, 2010, <http://www.eclipse.org/gmt/amw>.
- [Antoniol *et al.*, 2002] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta, "Analyzing Cloning Evolution in the Linux Kernel," *Information and Software Technology*, Volume 44, Number 13, October 2002, pages 755 - 765.
- [Apache Ant, 2010] Apache Ant, 2010, <http://ant.apache.org>.
- [Apache Derby, 2010] Apache Derby, 2010, <http://db.apache.org/derby>.
- [Apache JMeter, 2010] Apache JMeter, 2010, <http://jakarta.apache.org/jmeter>.
- [ArgoUML, 2010] ArgoUML, 2010, <http://tigris.argouml.org>.
- [Atkins, 1998] David Atkins, "Version Sensitive Editing: Change History as a Programming Tool," *Symposium on System Configuration Management*, LNCS 1439, Brussels, Belgium, July 1998, pages 146 - 157.
- [Aversano *et al.*, 2007] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta, "How Clones are Maintained: An Empirical Study," *European Conference on Software Maintenance and Reengineering*, Amsterdam, The Netherlands, March 2007, pages 81 - 90.
- [Baeza-Yates and Ribeiro-Neto, 1999] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*, Addison Wesley-Longman, 1999.
- [Baker, 1992] Brenda Baker, "A Program for Identifying Duplicated Code," *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, College Station, Texas, March 1992, pages 49 - 57.
- [Baker, 1995] Brenda Baker, "On Finding Duplication and Near-Duplication in Large Software Systems," *Working Conference on Reverse Engineering*, Toronto, Canada, July 1995, pages 86 - 95.

[Balazinska *et al.*, 1999] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis, "Partial Redesign of Java Software Systems Based on Clone Analysis," *Working Conference on Reverse Engineering*, Atlanta, Georgia, October 1999, pages 326 - 336.

[Balazinska *et al.*, 2000] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis, "Advanced Clone-analysis to Support Object-oriented System Refactoring," *Working Conference on Reverse Engineering*, Brisbane, Australia, November 2000, pages 98 - 107.

[Baxter *et al.*, 1998] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier, "Clone Detection using Abstract Syntax Trees," *International Conference on Software Maintenance*, Bethesda, Maryland, November 1998, pages 368 - 377.

[Bellon *et al.*, 2007] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, Volume 33, Number 9, September 2007, pages 577 - 591.

[Bézivin, 2005] Jean Bézivin, "On the Unification Power of Models," *Software and Systems Modeling*, Volume 4, Number 2, May 2005, pages 171 - 188.

[Bruntink *et al.*, 2005] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé, "On the Use of Clone Detection for Identifying Crosscutting Concern Code," *IEEE Transactions on Software Engineering*, Volume 31, Number 10, October 2005, pages 804 - 818.

[CeDAR, 2010] Clone Detection, Analysis, and Refactoring, 2010, <http://www.cis.uab.edu/softcom/cedar>.

[Church and Helfman, 1993] Kenneth Church and Jonathan Helfman, "Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code," *Journal of Computational and Graphical Statistics*, Volume 2, Number 2, June 1993, pages 153 - 174.

[CloneDigger, 2010] CloneDigger, 2010, <http://clonedigger.sourceforge.net>.

[CloneDR, 2010] CloneDR, 2010, <http://www.semdesigns.com/Products/Clone>.

[Cluto, 2010] Cluto, 2010, <http://glaros.dtc.umn.edu/gkhome/cluto/cluto/overview>.

[(COBOL) CloneDR, 2010] (COBOL) CloneDR, 2010, <http://www.semanticdesigns.com/Products/Clone/COBOLCloneDR.html>.

[Collard and Maletic, 2004] Michael Collard and Jonathan Maletic, "Document-Oriented Source Code Transformation using XML," *International Workshop on Software Evolution Transformation*, Delft, The Netherlands, November 2004, pages 11 - 14.

[Columba, 2010], Columba, 2010, <http://sourceforge.net/projects/columba>.

[ConQAT, 2010] Continuous Quality Assessment Toolkit, 2010, <http://conqat.cs.tum.edu/index.php/ConQAT>.

[Counsell *et al.*, 2006] Steve Counsell, Youssef Hassoun, George Loizou, and Rajaa Najjar, "Common Refactorings, a Dependency Graph and Some Code Smells: an Empirical Study of Java OSS," *International Symposium on Empirical Software Engineering*, Rio de Janeiro, Brazil, September 2006, pages 288 - 296.

[CPD, 2010] Copy/Paste Detector, 2010, <http://pmd.sourceforge.net/cpd.html>.

[Deerwester *et al.*, 1990] Scott Deerwester, Susan Dumais, George Furnas, Thomas Landauer, and Richard Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, Volume 41, Number 6, September 1990, pages 391 - 407.

[DMF, 2010] Duplication Management Framework, 2010, <http://sourceforge.net/projects/dupman>.

[Duala-Ekoko and Robillard, 2007] Ekwa Duala-Ekoko and Martin Robillard, "Tracking Code Clones in Evolving Software," *International Conference on Software Engineering*, Minneapolis, Minnesota, May 2007, pages 158 - 167.

[Ducasse *et al.*, 1999] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer, "A Language Independent Approach for Detecting Duplicated Code," *International Conference on Software Maintenance*, Oxford, United Kingdom, August 1999, pages 109 - 118.

[Duplo, 2010] Duplo, 2010, <http://sourceforge.net/projects/duplo>.

[Eclipse JDT, 2010] Eclipse Java Development Tools, 2010, <http://www.eclipse.org/jdt>.

[EMF, 2010] Eclipse Modeling Framework, 2010, <http://www.eclipse.org/modeling/emf>.

[Erlikh, 2000] Len Erlikh, "Leveraging Legacy System Dollars for E-Business," *IT Professional*, Volume 2, Number 3, May/June 2000, pages 17 - 23.

[Evans and Fraser, 2005] William Evans and Christopher Fraser, "Clone Detection via Structural Abstraction," *Technical Report*, MSR-TR-2005-104, Microsoft Research, 2005.

[Falke *et al.*, 2008] Raimar Falke, Pierre Frenzel, and Rainer Koschke, “Empirical Evaluation of Clone Detection using Syntax Suffix Trees,” *Empirical Software Engineering*, Volume 13, Number 6, December 2008, pages 601 - 643.

[Fanta and Rajilich, 1999] Richard Fanta and Vaclav Rajilich, “Removing Clones from the Code,” *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 11, Number 4, August 1999, pages 223 - 243.

[Fluri *et al.*, 2007] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall, “Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction,” *IEEE Transactions of Software Engineering*, Volume 33, Number 11, November 2007, pages 725 - 743.

[Fowler, 1999] Martin Fowler. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*, Addison-Wesley, 1995.

[Giesecke, 2006] Simon Giesecke, “Generic Modeling of Code Clones,” *Duplication, Redundancy, and Similarity in Software*, Internationales Begegnungs- und Forschungszentrum für Informatik Schloss Dagstuhl, D-06301, Saarbrücken, Germany, July 2006.

[Gusfield, 1997] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[Han and Kamber, 2006] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*, Morgan Kaufman, 2006.

[Hibernate, 2010] Hibernate, 2010, <http://www.hibernate.org>.

[Higo *et al.*, 2004] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, “ARIES: Refactoring Support Environment Based on Code Clone Analysis,” *International Conference on Software Engineering and Applications*, Cambridge, Massachusetts, November 2004, pages 222 - 229.

[Hunt and Thomas, 1999] Andrew Hunt and Dave Thomas. *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley Professional, 1999.

[IWDSC, 2002] First International Workshop on the Detection of Software Clones, 2002, <http://www.bauhaus-stuttgart.de/clones/workshop.html>.

[Jacob *et al.*, 2010] Ferosh Jacob, Daiqing Hou, and Patricia Jablonski, “Actively Comparing Clones Inside The Code Editor,” *International Workshop on Software Clones*, Cape Town, South Africa, May 2010, pages 9 - 16.

[JavaGenes, 2010] JavaGenes, 2010, <http://opensource.arc.nasa.gov/project/javagenes>.

[JBoss, 2010] JBoss, 2010, <http://www.jboss.org>.

[JBoss AOP, 2010] JBoss AOP, 2010, <http://www.jboss.org/jbossaop>.

[jEdit, 2010] jEdit, 2010, <http://www.jedit.org>.

[JFreeChart, 2010] JFreeChart, 2010, <http://www.jfree.org/jfreechart>.

[JHotDraw, 2010] JHotDraw, 2010, <http://www.jhotdraw.org>.

[Jiang *et al.*, 2007a] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondou, “DECKARD: Scalable and Accurate Tree-based Detection of Code Clones,” *International Conference on Software Engineering*, Minneapolis, Minnesota, May 2007, pages 96 - 105.

[Jiang *et al.*, 2007b] Lingxiao Jiang, Zhendong Su, and Edwin Chiu, “Context-Based Detection of Clone-Related Bugs,” *Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, September 2007, pages 55 - 64.

[Jiang and Hassan, 2007] Zhenming Jiang and Ahmed Hassan, “A Framework for Studying Clones in Large Software Systems,” *International Working Conference on Source Code Analysis and Manipulation*, Paris, France, October 2007, pages 203 - 212.

[Jouault and Bézivin, 2006] Frédéric Jouault and Jean Bézivin, “KM3: a DSL for Metamodel Specification,” *International Conference on Formal Methods for Open Object-Based Distributed Systems*, LNCS 4037, Bologna, Italy, June 2006, pages 171 - 185.

[Jouault *et al.*, 2006] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev, “TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering,” *International Conference on Generative Programming and Component Engineering*, Portland, Oregon, October 2006, pages 249 - 254.

[Jouault *et al.*, 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev, “ATL: A Model Transformation Tool,” *Science of Computer Programming*, Volume 72, Numbers 1-2, June 2008, pages 31 - 39.

[JRuby, 2010] JRuby, 2010, <http://www.jruby.org>.

[Juergens *et al.*, 2009] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner, “Do Code Clones Matter?,” *International Conference on Software Engineering*, Vancouver, Canada, May 2009, pages 485 - 495.

[Juergens *et al.*, 2010] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel, "Code Similarities Beyond Copy & Paste," *European Conference on Software Maintenance and Reengineering*, Madrid, Spain, March 2010.

[Juillerat and Hirsbrunner, 2006] Nicolas Juillerat and Beat Hirsbrunner, "An Algorithm for Detecting and Removing Clones in Java Code," *Workshop on Software Evolution through Transformations*, Natal, Brazil, September 2006, pages 63 - 74.

[Kamiya *et al.*, 2002] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Transactions on Software Engineering*, Volume 28, Number 2, July 2002, pages 654 - 670.

[Kasper and Godfrey, 2004] Cory Kasper and Michael Godfrey, "Aiding Comprehension of Cloning Through Categorization," *International Workshop on Principles of Software Evolution*, Kyoto, Japan, September 2004, pages 85 - 94.

[Kasper and Godfrey, 2005] Cory Kasper and Michael Godfrey, "Improved Tool Support for the Investigation of Duplication in Software," *International Conference on Software Maintenance*, Budapest, Hungary, September 2005, pages 305 - 314.

[Kiczales *et al.*, 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming," *European Conference on Object-Oriented Programming*, LNCS 1241, Jyväskylä, Finland, June 1997, pages 220 - 242.

[Kim *et al.*, 2004] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin, "An Ethnographic Study of Copy and Past Programming Practices in OOPL," *Symposium on Empirical Software Engineering*, Redondo Beach, California, August 2004, pages 83 - 92.

[Kim *et al.*, 2005] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy, "An Empirical Study of Code Clone Genealogies," *Joint Meeting of the European Software Engineering Conference and Foundations of Software Engineering*, Lisbon, Portugal, September 2005, pages 187 - 196.

[Klint *et al.*, 2005] Paul Klint, Ralf Lämmel, and Chris Verhoef, "Toward an Engineering Discipline for Grammarware," *ACM Transactions on Software Engineering Methodology*, Volume 14, Number 3, July 2005, pages 331 - 380.

[Komondoor and Horwitz, 2003] Raghavan Komondoor and Susan Horwitz, "Effective, Automatic Procedure Extraction," *International Workshop on Program Comprehension*, Portland, Oregon, May 2003, pages 33 - 42.

[Koni-N'Sapu, 2001] Georges Koni-N'Sapu, "A Scenario-Based Approach for Refactoring Duplicated Code in Object-Oriented Systems," *Diploma Thesis*, University of Bern, Switzerland, 2001.

[Kontogiannis, 1996] Kostas Kontogiannis, "Pattern Matching for Clone and Concept Detection," *Automated Software Engineering*, Volume 3, Numbers 1-2, July 1996, pages 77 - 180.

[Koschke *et al.*, 2006] Rainer Koschke, Raimar Falke, and Pierre Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," *Working Conference on Reverse Engineering*, Benevento, Italy, October 2006, pages 253 - 262.

[Kraft *et al.*, 2008] Nicholas Kraft, Brandon Bonds, and Randy Smith, "Cross-language Clone Detection," *International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, July 2008, pages 54 - 59.

[Krinke, 2007] Jens Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones," *Working Conference on Reverse Engineering*, Vancouver, Canada, October 2007, pages 170 - 178.

[Kuhn *et al.*, 2007] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba, "Semantic Clustering: Identifying Topics in Source Code," *Information and Software Technology*, Volume 49, Number 3, March 2007, pages 230 - 243.

[Kurtev *et al.*, 2006] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez, "Model-based DSL Frameworks," *International Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, October 2006, pages 602 - 615.

[Lacey, 1986] Robert Lacey. *Ford: The Men and the Machine*, Little Brown, 1986.

[Lawrence and Atlee, 2006] Shari Lawrence and Joanne Atlee. *Software Engineering: Theory and Practice*, Prentice Hall, 2006.

[Li and Thompson, 2009] Huiqing Li and Simon Thompson, "Clone Detection and Removal for Erlang/OTP within a Refactoring Environment," *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Savannah, Georgia, January 2009, pages 169 - 178.

[Li *et al.*, 2004] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," *Symposium on Operating System Design and Implementation*, San Francisco, California, December 2004, pages 289 - 302.

[Lientz and Swanson, 1980] Bennet Lientz and Burton Swanson. *Software Maintenance Management*, Addison-Wesley, 1980.

[Liu, 2004] Yidong Liu, "Semi Automatic Removal of Duplicated Code," *Diploma Thesis*, University of Stuttgart, Germany, 2004.

[Livieri *et al.*, 2007] Simone Livieri, Yoshiki Higo, Makoto Matsushita, and Katsuro Inoue, "Analysis of the Linux Kernel Evolution Using Code Clone Coverage," *International Workshop on Mining Software Repositories*, Minneapolis, Minnesota, May 2007.

[Lukin *et al.*, 2008] Stacy Lukin, Nicholas Kraft, and Letha Etzkorn, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation," *Working Conference on Reverse Engineering*, Antwerp, Belgium, October 2008, pages 155 - 164.

[Manning and Schütze, 1999] Christopher Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.

[Manning *et al.*, 2008] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[Marcus and Maletic, 2001] Andrian Marcus and Jonathan Maletic, "Identification of High-Level Concept Clones in Source Code," *International Conference on Automated Software Engineering*, San Diego, California, November 2001, pages 107 - 114.

[Matlab, 2010] Matlab, 2010, <http://www.mathworks.com/products/matlab>.

[Mernik *et al.*, 2005] Marjan Mernik, Jan Heering, and Anthony Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, Volume 37, Number 4, December 2005, pages 316 - 344.

[Microsoft Phoenix, 2010] Phoenix Compiler and Shared Source Common Language Infrastructure, 2010, <http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx>

[Microsoft Windows Research Kernel, 2010] Microsoft Windows Research Kernel, 2010, <http://www.microsoft.com/resources/sharedsource/Licensing/researchkernel.msp>.

[Murphy-Hill and Black, 2008] Emerson Murphy-Hill and Andrew Black, "Refactoring Tools: Fitness for Purpose," *IEEE Computer*, Volume 25, Number 5, September/October 2008, pages 38 - 44.

[OMG, 2000] Object Management Group, "Model Driven Architecture," 2000.

[OMG, 2001] Object Management Group, "Model Driven Architecture - A Technical Perspective," 2001.

[OMG, 2002] Object Management Group, “MOF 2.0 Query/Views/Transformations RFP,” 2002.

[OMG, 2003] Object Management Group, “MDA Guide V1.0.1,” 2003.

[Rajapakse and Jarzabek, 2007] Damith Rajapakse and Stan Jarzabek, “Using Server Pages to Unify Clones in Web Applications: A Trade-Off Analysis,” *International Conference on Software Engineering*, Minneapolis, Minnesota, May 2007, pages 116 - 126.

[Refactor! Pro, 2010] Refactor! Pro for Visual Studio, 2010, http://www.devexpress.com/Products/Visual_Studio_Add-in/Refactoring.

[Rieger and Ducasse, 1998] Mathias Rieger and Stéphane Ducasse, “Visual Detection of Duplicated Code,” *ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*, LNCS 1543, Brussels, Belgium, July 1998, pages 75 - 76.

[Rieger *et al.*, 2004] Matthias Rieger, Stéphane Ducasse, and Michele Lanza, “Insights into System-Wide Code Duplication,” *Working Conference on Reverse Engineering*, Delft, The Netherlands, November 2004, pages 100 - 109.

[Rijsbergen, 1979] Keith van Rijsbergen. *Information Retrieval*, Butterworths, 1979.

[Roy *et al.*, 2009] Chanchal Roy, James Cordy, and Rainer Koschke, “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach,” *Science of Computer Programming*, Volume 74, Number 7, May 2009, pages 470 - 495.

[Ruby, 2010] Ruby Programming Language, 2010, <http://www.ruby-lang.org/en>.

[Rusinovich and Solomon, 2005] Mark Rusinovich and David Solomon. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, Microsoft, 2005.

[Schmidt, 2006] Doug Schmidt, “Model-Driven Engineering,” *IEEE Computer*, Volume 39, Number 2, February 2006, pages 25 - 31.

[Schofield *et al.*, 2006] Curtis Schofield, Brendan Tansey, Zhenchang Xing, and Eleni Stroulia, “Digging the Development Dust for Refactorings,” *International Conference on Program Comprehension*, Athens, Greece, June 2006, pages 23 - 34.

[Scorpio, 2010] Scorpio, 2010, <http://www-sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio-e>.

[SDD, 2010] SDD, 2010, http://wiki.eclipse.org/index.php/Duplicated_code_detection_tool_%28SDD%29.

[SEL, 2010] Software Engineering Laboratory, Osaka University, 2010, <http://sel.ist.osaka-u.ac.jp>.

[Simian, 2010] Simian, 2010, <http://www.redhillconsulting.com.au/products/simian>.

[SimScan, 2010] SimScan, 2010, <http://blue-edge.bg/simscan>.

[Squirrel SQL, 2010] Squirrel SQL, 2010, <http://www.squirreysql.org>.

[srcML, 2010] srcML, 2010, <http://www.sdml.info/projects/srcml>.

[Standish, 1984] Thomas Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, Volume SE-10, Number 5, September 1984, pages 494 - 497.

[Strang, 1993] Gilbert Strang. *Introduction to Linear Algebra*, Wellesley-Cambridge, 1993.

[Sun *et al.*, 2008] Yu Sun, Zekai Demirezen, Frédéric Jouault, Robert Tairas, and Jeff Gray, "Tool Interoperability through Model Transformations," *International Conference on Software Language Engineering*, LNCS 5452, Toulouse, France, September 2008, pages 178 - 187.

[Tairas and Gray, 2006] Robert Tairas and Jeff Gray, "Phoenix-Based Clone Detection Using Suffix Trees," *Association for Computing Machinery Southeast Conference*, Melbourne, Florida, March 2006, pages 679 - 684.

[Tairas *et al.*, 2006] Robert Tairas, Jeff Gray, and Ira Baxter, "Visualization of Clone Detection Results," *Eclipse Technology Exchange Workshop*, Portland, Oregon, October 2006, pages 50 - 54.

[Tairas *et al.*, 2007] Robert Tairas, Shi-Hsi Liu, Frédéric Jouault, and Jeff Gray, "CoCloRep: A DSL for Code Clones," *International Workshop on Software Language Engineering*, Nashville, Tennessee, October 2007, pages 91 - 99.

[Tairas *et al.*, 2008] Robert Tairas, Marjan Mernik, and Jeff Gray, "Using Ontologies in the Domain Analysis of Domain-Specific Languages," *Workshop on Transformation and Weaving Ontologies in Model-Driven Engineering*, LNCS 5421, Toulouse, France, September 2008, pages 332 - 342.

[Tairas, 2009] Robert Tairas, "Centralizing Clone Group Representation and Maintenance," *Student Research Competition, International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, Florida, October 2009, pages 781 - 782.

[Tairas and Gray, 2009a] Robert Tairas and Jeff Gray, “An Information Retrieval Process to Aid in the Analysis of Code Clones,” *Empirical Software Engineering*, Volume 14, Number 1, February 2009, pages 33 - 56.

[Tairas and Gray, 2009b] Robert Tairas and Jeff Gray, “Get to Know Your Clones with CeDAR,” *Tool Demonstration, International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, Florida, October 2009, pages 817 - 818.

[Tairas and Gray, 2010a] Robert Tairas and Jeff Gray, “Sub-clone Refactoring in Open Source Software Artifacts,” *Symposium on Applied Computing*, Sierre, Switzerland, March 2010, pages 2364 - 2365.

[Tairas and Gray, 2010b] Robert Tairas and Jeff Gray, “Sub-clones: Considering the Part Rather than the Whole,” *International Conference on Software Engineering, Research, and Practice*, Las Vegas, Nevada, July 2010, *to appear*.

[Tarr *et al.*, 1999] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, Jr., “N Degrees of Separation: Multi-Dimensional Separation of Concerns,” *International Conference on Software Engineering*, Los Angeles, California, May 1999, pages 107 - 119.

[Toomim *et al.*, 2004] Michael Toomim, Andrew Begel, and Susan Graham, “Managing Duplicated Code with Linked Editing,” *Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September 2004, pages 173 - 180.

[Travis, 1997] John Travis, “Dolly, Polly, Gene – Send in the Clones,” *Science News*, Volume 152, Number 8, August 1997, page 127.

[Wahler *et al.*, 2004] Vera Wahler, Dietmar Seipel, Jurgen Wolff von Gudenberg, and Gregor Fischer, “Clone Detection in Source Code by Frequent Itemset Techniques,” *International Workshop on Source Code Analysis and Manipulation*, Chicago, Illinois, September 2004, pages 128 - 135.

[Walenstein, 2006] Andrew Walenstein, “Code Clones: Reconsidering Terminology,” *Duplication, Redundancy, and Similarity in Software, Internationales Begegnungs- und Forschungszentrum für Informatik Schloss Dagstuhl*, D-06301, Saarbrücken, Germany, July 2006.

[Weckerle, 2008] Valentin Weckerle, “CPC: An Eclipse Framework for Automated Clone Life Cycle Tracking and Update Anomaly Detection,” *Diploma Thesis*, Freie Universität Berlin, Germany, 2008.

[Weißgerber and Diehl, 2006] Peter Weißgerber and Stephan Diehl, “Identifying Refactorings from Source-Code Changes,” *International Conference on Automated Software Engineering*, Tokyo, Japan, September 2006, pages 231 - 240.

[Weltab, 2010] Weltab, 2010, <http://www.bauhaus-stuttgart.de/clones>.

[Wettel, 2004] Richard Wettel, "Automated Detection of Code Duplication Clusters," *Diploma Thesis*, "Politehnica" University of Timisoara, Romania, 2004.

[Wilmot *et al.*, 1997] Ian Wilmot, Angelika Schnieke, Jim McWhir, A. Kind, and Keith Campbell, "Viable Offspring Derived from Fetal and Adult Mammalian Cells," *Nature*, Volume 385, Number 6619, February 1997, pages 810 - 813.

[Zhao and Karypis, 2005] Ying Zhao and George Karypis, "Topic-Driven Clustering for Document Datasets," *SIAM International Conference on Data Mining*, Newport Beach, California, April 2005, pages 358 - 369.

[Zukas and Price, 2003] Anthony Zukas and Robert Price, "Document Categorization Using Latent Semantic Indexing," *Symposium on Document Image Understanding Technology*, Greenbelt, Maryland, April 2003, pages 87 - 91.

APPENDIX A

PHOENIX-BASED CLONE DETECTION USING SUFFIX TREES

This appendix describes an investigation into an automatic clone detection technique developed as a plug-in for Microsoft's Phoenix framework [Microsoft Phoenix, 2010]. The investigation finds function-level clones in a program using ASTs and suffix trees [Tairas and Gray, 2006]. An AST provides the structural representation of the code after the lexical analysis process. The AST nodes are used to generate a suffix tree, which allows analysis on the nodes to be performed rapidly. The same methods that have been successfully applied to find duplicate sections in biological sequences are used to search for matches on the suffix tree that is generated, which in turn reveal matches in the code.

A.1 Exact Matching Algorithm

A.1.1 *The Original Suffix Tree*

As its name suggests, a suffix tree is a tree of suffixes. A suffix tree of a string is generated from the suffixes of that string. For each suffix of a string, a path is made from the root to a leaf. This is done by evaluating each character in the suffix and generating new edges when no existing edges that represent the character in the suffix tree exists [Gusfield, 1997]. This is the characteristic of the suffix tree that is useful in string matching, because duplicate patterns in the suffixes will be represented by a single edge in the tree. A string *abcdabe\$* is represented by the suffix tree in Figure A.1.

The pattern *ab* is represented by a single edge. Two suffixes pass through this edge (i.e., they both start with the substring *ab*). These two suffixes are *abcdabe\$* and *abe\$*. The split at the end of this edge continues the two suffixes where the next character differs between the two suffixes. The last character, *\$*, is a special terminating character

that identifies the end of the string. By looking at the suffixes that pass through the edge that represents the pattern *ab*, the location of this string pattern can be determined.

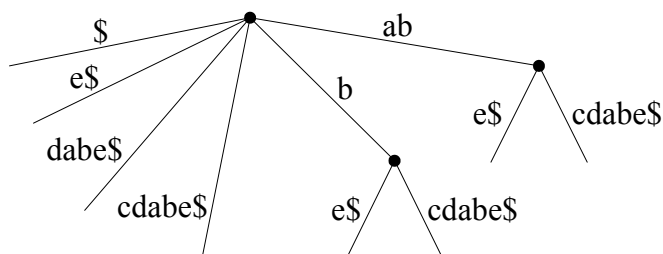


Figure A.1 – Suffix tree of *abcdeabe\$*.

The use of suffix trees to search for duplicate patterns is not limited to just one string. Searching for duplicate patterns in multiple strings is also possible. The suffix tree used to search for duplicate patterns in multiple strings is generated from the concatenation of the strings. For example, to generate the suffix tree of two identical strings *abgf* and *abgf*, the two strings are concatenated into one string *abgf\$abgf#*, with *\$* and *#* as the special characters that determine where each string terminates. This new string is evaluated as a single string and the same process used in the previous example is used on this string to generate the suffix tree. The result is the suffix tree in Figure A.2. Duplicate patterns can be identified in this suffix tree and with some additional processing, the individual strings that contain these patterns can be determined.

The edge labeled “*abgf*” represents two suffixes of the concatenated string that start at the beginning positions of each individual string. That is, the first suffix is the whole string *abgf\$abgf#*, which starts at the beginning position of the first individual string. The second suffix is the substring *abgf#*, which starts at the beginning position of

nodes are inserted between each function representation in the sequence of nodes. A suffix tree is generated from this sequence and by using the method of searching for certain edges in the suffix tree described earlier, functions that are duplicates of each other can be determined.

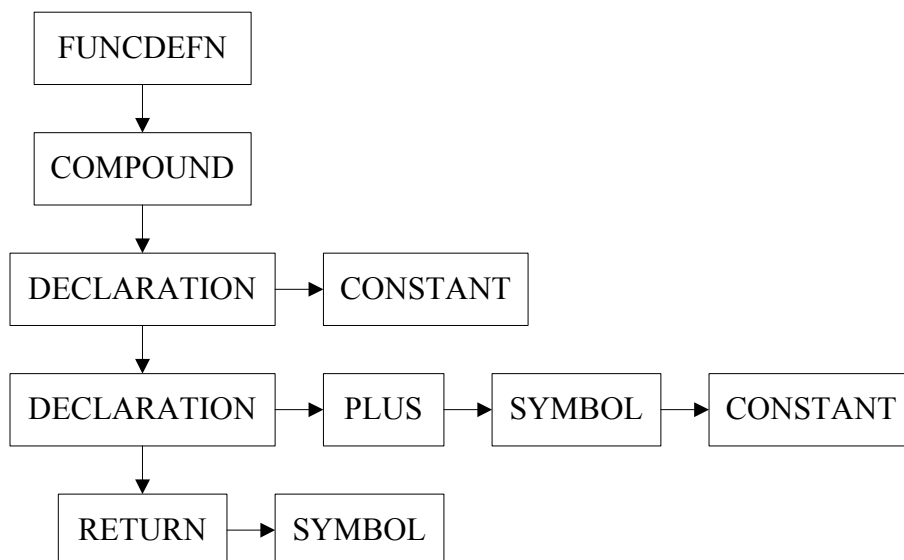


Figure A.3 – Abstract syntax tree nodes.

A.1.3 Potential False Positives

It is not sufficient to determine if two functions exactly match based only on the suffix trees of AST node names. Several situations can lead to false positives (i.e., clones reported as exact matches, but upon further observation are not). It is possible that the same sequence of AST node names can represent a function that is not exactly the same. For example, observe Function #1 in Figure A.4. The AST node sequence, which consists of the node names, will be identical to the sequence for Function #2. The two functions

would be considered exact duplicates. However, the constant values that are assigned to variable x in the two functions are different (i.e., one is set to 1 and the other is set to 3).

```

Function #1: int main() {
              int x = 1;
              int y = x + 5;
              return y;
            }

Function #2: int main() {
              int x = 3;
              int y = x + 5;
              return y;
            }

Function #3: int main() {
              int x = 1;
              int y = y + 5;
              return y;
            }

```

Figure A.4 – Example functions.

Another problem that can arise occurs when the variable locations differ from one function to another even if their node sequences are the same. Function #3 in Figure A.4 demonstrates this situation. The statement $y = x + 5$ in Function #1 will have the same node sequence as the statement $y = y + 5$ in Function #3 (i.e., the similar sequence is [DECLARATION] [PLUS] [SYMBOL] [CONSTANT]). However, the lines are not exact matches, because the line in Function #3 does not contain an x variable.

If the suffix tree method of finding duplicates is used, all three functions would be reported incorrectly as duplicates. In order to account for these situations, an additional step is added after duplicates are reported from the suffix tree. This step will observe the *types* and *values* of the nodes of the AST and the positions of the variables in the function. These additional steps are discussed further in A.2.2.

A.1.4 Sketch of Clone Detection Algorithm

The following algorithm represents the approach used in our clone detection tool.

```

/* Generate node sequence */
For each node in the AST:
    Add the node to the sequence of nodes
    If node is the end of a function definition:
        Add a terminating node to the sequence of nodes
/* Generate suffix tree */
For each suffix of the sequence of nodes:
    Generate a path from the root to a leaf combining the path with
    existing edges when the edges represent the same sequence of
    nodes
/* Look for duplicate functions */
For each leaf that represents a suffix that starts with a function
definition node:
    Traverse the path up to the root
    If the last edge before the root represents all the nodes of the
    function and more than one terminating nodes are found where the
    edge splits:
        Group the functions associated with these terminating nodes
        together
/* Additional check on duplicate groups */
For each group of duplicate functions:
    Check whether constant values and variable positioning match each
    other

```

A.2 Implementation Details

A.2.1 Microsoft Phoenix

Our clone detection tool is implemented as a plug-in for the Microsoft Phoenix framework. Although this framework has been offered to academia to aid in the research of compilers and software analysis tools, it is also targeted for developers of production-level compilers and tools. Phoenix is a joint project between the Visual C++, Microsoft Research, and .NET Common Language Runtime groups.

Phoenix is primarily a framework for the backend of a compiler, where optimization and code generation tasks are performed in a customized manner. In Phoenix, the compiler tasks are divided into phases that are executed sequentially according to a specified list. By separating tasks into phases, Phoenix allows customization in any part of the sequence. Custom analysis tools can be developed and included as a specific phase in the process. A new phase is inserted into Phoenix by way of a Dynamic Link Library (DLL) module that is a plug-in for the compiler. Our clone detection tool is written as a custom phase that is plugged into Phoenix. Figure A.5 is a graphical representation of the process.

In Figure A.5, an example program called `example.c` is consumed by the C/C++ Frontend, which is included in Phoenix. This produces `example.ast`, which contains the AST of the source code in `example.c`. This AST file is then consumed by the Phoenix Backend. In addition to the AST file, a plug-in is included that represents our clone detector. The output from the Phoenix Backend is a report of clones found in `example.c`.

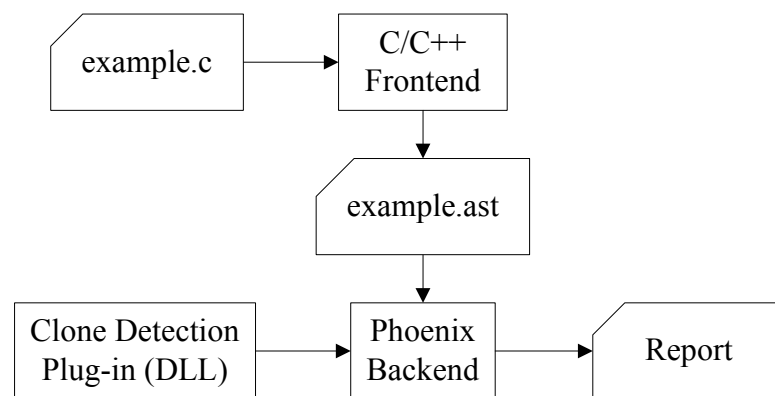


Figure A.5 – Clone detection plug-in for Phoenix.

A.2.2 Detecting Code Clones

A major part of the clone detection tool is the implementation of a mechanism to generate and evaluate suffix trees. The method used to generate the suffix tree follows a naïve approach. This approach takes $O(m^2)$ time to build a suffix tree, where m is the length of the string or sequence. Linear-time construction of suffix trees can be done using either Ukkonen's or Weiner's method [Gusfield, 1997]. Building suffix trees with these methods requires more complex procedures. However, they could be implemented if it is desired to reduce the processing time of suffix tree construction.

The program representation in the form of an AST is obtained from an object provided by the Phoenix compiler. The AST consists of *Node* objects that contain information about the node such as name, type, and value. The names of the nodes are used to generate the suffix tree. The type and value of the node is used in the second step of the process when the clone detector examines groups of reported clones to find exact matching clones.

While traversing the AST, any encounter with a “function definition” node is noted. When the end of the function definition is reached in the sequence of nodes, a special terminating node is inserted into the sequence. This terminating node is a custom node that is not part of the node collection provided by Phoenix. A suffix tree is generated from the sequence of nodes (including terminating nodes). Because each suffix of the sequence is evaluated, there will be a path from the root to a leaf for each suffix. The suffixes whose first nodes are the starting nodes of the functions are of particular interest. The paths that represent these suffixes are traversed from the leaves to the root. Paths that converge (at the top of the tree) into a single edge that represents all the nodes

of one or more functions are considered duplicates. These clones are stored together to be evaluated further.

The next step of the process constructs a separate suffix tree of the nodes for each group of duplicate functions reported in the first step. The difference in the construction of the suffix tree compared to the first construction is that not only are the node names compared, but also their type, value, and variable position. Nodes are considered identical if their types are the same, their values are the same, and the variable represented by the node is at the same position in the functions. These checks are applied on the nodes where applicable. The result of these two steps is a list of functions that are exact duplicates of each other.

A.3 Case Study Example

Our clone detection implementation was experimentally applied on two C programs varying in size. The first program was Abyss [Abyss, 2010], which is a small web server written in approximately 1500 LOC. The second program was Weltab [Weltab, 2010], which is an election results program written in approximately 11K LOC. Weltab was used as part of the evaluation of clone detection software at the *First International Workshop on the Detection of Software Clones* [IWDSC, 2002].

In the evaluation of Abyss, the clone detector found five groups of duplicate functions. However, only two are related to Abyss, while the rest are duplicate functions in predefined header files. The first group related to Abyss consists of the functions *ConfGetToken* (in *conf.c*) and *GetToken* (in *http.c*). These two functions represent 23 lines of code that are exact matches of each other. The second group consists of the

functions *ThreadRun* (in *thread.c*) and *ThreadStop* (in *thread.c*). These two functions represent 5 lines of code that call different functions in their return values, but have similar return types.

In the evaluation of WelTab, the clone detector found six groups of duplicate functions. Two are related to duplicate functions in predefined header files. The remaining four groups are functions scattered in different files where only their “main” functions differ. The following lists the groups of clones excluding the ones found in the predefined header files.

Group No. 1: Function *canvw* in files *canv.c*, *cnv1.c*, and *cnv1a.c*

Group No. 2: Function *lhead* in files *lans.c* and *lansxx.c* and function *rshead* in files *r01tmp.c*, *r101tmp.c*, *r11tmp.c*, *r26tmp.c*, *r51tmp.c*, *rsum.c*, and *rsumxx.c*

Group No. 3: Function *rsprtpag* in files *r01tmp.c*, *r101tmp.c*, *r11tmp.c*, *r26tmp.c*, *r51tmp.c*, and *rsum.c*

Group No. 4: Function *askchange* in files *vedt.c*, *vfix.c*, and *xfix.c*

Initially, the second category of clones was not allowed to have different types. By relaxing this requirement on the second step of the detection process, three additional clone groups were found. These groups contained pairs of duplicate functions that dealt with the conversion of two types of values: `int` and `long int`. The pairs were all found in *baselib.c* and are functions *cvci* and *cvcil*, functions *cvic* and *cvicl*, and functions *cvicz* and *cviczl*.