

In compliance with the  
Canadian Privacy Legislation  
some supporting forms  
may have been removed from  
this dissertation.

While these forms may be included  
in the document page count,  
their removal does not represent  
any loss of content from the dissertation.



**SOFTWARE CLONES**  
**FOR SOFTWARE RE-ENGINEERING AND MAINTENANCE**

by

Irina Padioukova


Diploma Metallurgical Engineering, Moscow State Institute of Steel and Alloys, 1991

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF  
Master of Computer Science  
in the Graduate Academic Unit of Computer Science

Supervisors: Colin Ware, Ph.D., Computer Science  
Andrew McAllister, Ph.D., Computer Science

Examining Board: John DeDourek, MS., Computer Science, Chair  
Prabhat Mahanti, Ph.D., Computer Science and Applied Statistics  
Mary Kaye, Dept. of Electrical and Computer Engineering

This Thesis is Accepted

  
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

February, 2003

©Irina Padioukova, 2003



National Library  
of Canada

Bibliothèque nationale  
du Canada

Acquisitions and  
Bibliographic Services

Acquisitons et  
services bibliographiques

395 Wellington Street  
Ottawa ON K1A 0N4  
Canada

395, rue Wellington  
Ottawa ON K1A 0N4  
Canada

*Your file* *Votre référence*  
*ISBN: 0-612-87535-0*  
*Our file* *Notre référence*  
*ISBN: 0-612-87535-0*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

# Canada

University of New Brunswick

HARRIET IRVING LIBRARY

This is to authorize the Dean of Graduate Studies  
to deposit two copies of my thesis/report in the  
University Library on the following conditions:

**(DELETE one of the following conditions)**

✓(a) The author agrees that the deposited copies of this thesis/report may be made available to users at the discretion of the University of New Brunswick

OR

(b) The author agrees that the deposited copies of this thesis/report may be made available to users only with her/his written permission for the period ending

\_\_\_\_\_  
**JUSTIFICATION:** \_\_\_\_\_

\_\_\_\_\_  
After that date, it is agreed that the thesis/report may be made available to users at the discretion of the University of New Brunswick\*

\_\_\_\_\_  
Date

February 28, 2002

\_\_\_\_\_  
Signature of Author

\_\_\_\_\_  
Signature of Supervisor

\_\_\_\_\_  
Signature of the Dean of Graduate Studies

\* Authors should consult the "Regulations and Guides for the Preparation and Submission of Graduate Theses and Reports" for information concerning the permissible period of restricted access and for the procedures to be followed in applying for this restriction. The maximum period of restricted access of a thesis is four years.

**BORROWERS** must give proper credit for any use made of this thesis, and obtain the consent of the author if it is proposed to make extensive quotations, or to reproduce the thests in whole or in part.

## *Abstract*

Code cloning complicates maintenance and hampers evolution of large software systems as it degrades their design and structure. Systematic management of software clones has the potential to translate into significant budget savings. Although various aspects of clone management have been addressed by academic research, practical application has been hampered by the lack of adequate tools and processes.

This thesis research defines, implements and tests a comprehensive process for analyzing software clones in large bodies of source code. This process provides software practitioners with a necessary set of practical tools to detect, analyze, categorize and remove clones.

A solution to extending a text-based clone identification technique to detect partial clones is described and integrated with an existing clone detection tool.

A prototype of an interactive visual clone management tool that analyzes detected clones, clusters them and presents them to the user is introduced. This tool enables software practitioners to view, analyze, and utilize clone data to pursue their possible elimination.

This thesis evaluates the described process by applying it to a commercial software system and analyzes the results.

## *Acknowledgments*

I would like to express my sincere appreciation to those who helped me during my work:

- Dr. Colin Ware and Dr. Andrew McAllister, my advisors, for their support, guidance, and helpful feedback.
- Eric Falkjare, my manager, for his constant encouragement and support.
- Innovatia Inc., my company, for accommodating my hardware needs.
- Linda Sales, our graduate secretary, for all her kindness and care.
- My friends, my family, and especially my partner, Kevin Murphy, for their love, inspiration, continuous encouragement, and tolerating my constant unavailability.

## *Table of Contents*

<b>Abstract</b> .....	<b>ii</b>
<b>Acknowledgments</b> .....	<b>iii</b>
<b>Table of Contents</b> .....	<b>iv</b>
<b>List of Tables</b> .....	<b>vii</b>
<b>Table of Figures</b> .....	<b>viii</b>
<b>Chapter 1 - Introduction</b> .....	<b>1</b>
1.1 Thesis Objectives .....	3
1.2 Thesis Organization .....	6
<b>Chapter 2 – Software Clones</b> .....	<b>8</b>
2.1 Code Cloning .....	8
2.2 Implications of Code Cloning .....	12
2.3 Management of Software Clones .....	14
2.4 Software Clones Revisited .....	17
2.5 Clone Identification.....	20
2.5.1 Direct Metrics Comparison Approach .....	21
2.5.2 Abstract Syntax Trees Comparison Approach.....	22
2.5.3 Text Based Comparison Approach .....	24
2.5.4 Choice of Approach .....	26
2.5.5 SelArt – a Tool for Identifying Redundancy in Source Code.....	27
<b>Chapter 3 – Information Visualization</b> .....	<b>33</b>
3.1 Origins of Information Visualization .....	33
3.2 Software Visualization.....	36
3.3 Visualization in Maintenance and Re-engineering .....	37
3.4 General Design Guidelines for Program Visualization.....	38



<b>Chapter 4 – Near Clone Identification with SelArt .....</b>	<b>43</b>
4.1 Background .....	43
4.2 Near Clone Detection with SelArt .....	44
4.3 Step 1: Pre-processing Transformation.....	46
4.3.1 Scanner Design.....	53
4.3.1.1 The Token Set .....	54
4.3.1.2 State Diagram Construction .....	54
4.3.1.3 Transition Tables.....	57
4.3.1.4 Token Recognition Procedure.....	58
4.3.2 The Parser Engine Design.....	60
4.3.3 Implementation of the Pre-processing .....	61
4.3.3.1 The Parser Module .....	61
4.3.3.2 Discovery of Directory Structure .....	62
4.3.3.3 Preservation of the Line Structure.....	63
4.3.3.4 Supporting Statistics .....	64
4.4 Step 2: Clone identification with SelArt .....	65
4.4.1 Line-Oriented to Stream Input Conversion.....	65
4.4.2 SelArt Parameters.....	66
4.5 Step 3: Post-processing of Clone Identification Results .....	67
4.5.1 Original Result Presentation .....	67
4.5.2 Filtering Information for Future Analysis.....	68
4.5.3 Conversion of Clone Boundaries .....	68
4.5.4 Implementation Details .....	69
4.6 Closing Remarks .....	71
<b>Chapter 5 – Clone Visualization Prototype: CloneMaster.....</b>	<b>73</b>
5.1 Motivation .....	73
5.2 Related Work .....	75
5.3 Formulating Requirements for the CloneMaster Tool.....	82
5.4 CloneMaster Design and Implementation Highlights.....	91
5.4.1 Usage Scenarios .....	91

5.4.2 CloneMaster Data Organization.....	94
5.4.2.1 Database manipulation with DBManager .....	95
5.4.3 Graphical User Interface Organization .....	96
5.4.3.1 System Loading.....	100
5.4.3.2 Menus, Navigation, and Interaction Techniques.....	102
5.4.3.3 Query Support .....	107
5.4.3.4 Comments on Implementation .....	108
<b>Chapter 6 – Industrial Experience and Evaluation .....</b>	<b>111</b>
6.1 Selecting the Case Study.....	111
6.2 Choice of Source Code Test Case.....	112
6.3 Evaluation Procedure .....	113
6.4 Clone Detection Results.....	117
6.5 Nature of Clones and Their Occurrences .....	123
6.6 Clone Analysis with CloneMaster .....	125
6.7 Conclusions .....	127
<b>Chapter 7 – Conclusions and Future Work.....</b>	<b>131</b>
<b>References .....</b>	<b>136</b>
<b>Appendix A – Parser Design .....</b>	<b>141</b>
<b>Appendix B – CloneMaster Data Model .....</b>	<b>148</b>
<b>Appendix C – Experimental Results .....</b>	<b>151</b>
<b>Appendix D – Support Tools “Help” Page .....</b>	<b>159</b>
<b>Vita</b>	

## *List of Tables*

Table 4.1: C++ tokens with associated actions (full set of participating tokens is listed in Table A.1 Appendix A).....	49
Table 6.1: Comparison of pre-processing configurations between parsed1 and parsed2 systems. ....	116
Table 6.2: Summary of major statistics from the experiments. ....	117
Table 6.3: Classification of typical duplication patterns with possible restructuring solutions. ....	124
Table A.1: Selected Tokens .....	142
Table A.2 Token definition .....	145
Table A.3 Selected Character Classes.....	147

## *Table of Figures*

Figure 1.1: High Level Diagram of Clone Analysis Process.....	4
Figure 2.1: An example of system partitioning by file clusters.....	20
Figure 2.2: Parameterized match verification.....	25
Figure 2.3: A set of snips of length 10 characters generated to represent the source code text (arrows indicate end of lines).....	28
Figure 2.4: Set of snips of length 3 lines generated for the fragment of code of Figure 2.3.....	29
Figure 2.5: Combining raw matches.....	31
Figure 2.6: Combining and splitting raw matches.....	32
Figure 3.1: Visualization model. Mapping data to visual form.....	35
Figure 4.1: Data flow of the clone identification process.....	45
Figure 4.2: Diagram illustrating mechanism of pre-processing.....	47
Figure 4.3: Example of pre-processing transformation.....	50
Figure 4.4: Unifying action of pre-processing.....	52
Figure 4.5: Transition diagram for ‘identifier’ token (TokenCode = 1).....	55
Figure 4.6: Transition diagram of a character constant token (TokenCode = 11) defined as one or more characters enclosed in single quotes (Table A.2).....	56
Figure 4.7: A fragment of state diagram that facilitates recognition of the following token types: ‘C-style comment’ (TokenCode=7), ‘inline comment’ (TokenCode=6), and ‘division assignment’ (TokenCode=50).....	57
Figure 4.8: Token recognition algorithm.....	59
Figure 4.9: Line tracking algorithm.....	64

Figure 4.10: An example of a ‘.pos’ file..	69
Figure 5.1: A scatter plot generated by Baker to visualize clone occurrences in a file. .	77
Figure 5.2: View of the Cluster #1948 Page.....	82
Figure 5.3: Main use case scenarios of the CloneMaster visualization tool.....	93
Figure 5.4: DBManager Graphical User Interface.....	96
Figure 5.5: CloneMaster GUI display (main window). .....	99
Figure 5.6: Icons used in CloneMaster.. .....	99
Figure 5.7: Using color-coding for highlighting.....	99
Figure 5.8: ‘Load System’ dialog.....	101
Figure 5.9: Pie diagram showing amount of cloned code vs. non-cloned code.....	103
Figure 5.10: A histogram showing clone distribution by size built with bucket width of 20 lines.. .....	104
Figure 5.11: Pop-up menus.. .....	104
Figure 5.12: Example of clone instance information.....	106
Figure 5.13: Example of a dynamic query based on clone entity. ....	108
Figure 5.14: CloneMaster architecture: conceptual view. ....	110
Figure 6.1: Experimental process.....	114
Figure 6.2: Clone clusters break down by size. ....	120
Figure 6.3: Duplication within the same file vs. duplication between different files. ....	121
Figure 6.4: File span distribution of clone clusters. ....	121
Figure 6.5: Distribution of file clusters by size (i.e., file count).....	122
Figure C.1: Example of a clone consisting entirely of preprocessor directives.....	151
Figure C.2: Example of an exact clone.....	152

Figure C.3: An example of two near clones typical to ‘parsd1’.. ..... 153

Figure C.4: An example of a near clone.. ..... 154

Figure C.5: Counterpart of the clone from Figure C.4..... 155

Figure C.6: An example of exact clone reported in ‘original’ and ‘parsed1’, but missed  
in ‘parsed2’. ..... 156

Figure C.7: Same code fragment (Figure C.6) after pre-processing (tokenized)..... 157

Figure C.8: Example of restructuring.. ..... 157

Figure C.9: Example of restructuring.. ..... 158

## *Chapter 1 - Introduction*

Software development is an evolving process resulting in ever larger and more complex systems. However, most of the cost of developing these systems comes not in developing the initial release, but in changing and adapting them over time.

Maintenance<sup>1</sup> of existing systems has become the most expensive part of software life cycle (50-70% of the total cost [Boehm 1981, p. 533]).

A noteworthy but often overlooked problem that adversely affects maintainability of large software systems arises from existence of repetitive patterns of duplicated code (also known as software clones). Previous research of commercial code revealed redundancy levels of up to 10% [Baker 1992][Baxter 1998][Dagenais 1998] and even greater [Mayrand 1996] [Baxter 2002].

Duplicated code tends to be introduced into software systems as modifications are made to add new functionality or to fix bugs. As clones proliferate, they degrade the design and the structure of software compromising such important software qualities as readability and adaptability [Kontogiannis 1996][Dagenais 1998][Baxter 1998][Monden 2002]. Conversely, identification of clones may significantly reduce maintenance effort

---

<sup>1</sup> Software Maintenance is the “modification of a software product after delivering to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment” [ANSI 1983]

and improve overall quality: Once clones are identified and analyzed, the code can be often restructured<sup>2</sup> and considered for proper reuse [Fowler 1999] [Baxter 2002].

Clone detection technology and the use of clones for software comprehension and restructuring are the two cornerstones of clone management, an emerging area of software engineering that strives to improve quality and maintainability of existing software. Since meaningful clones may not always be exact copies of each other, the clone detection technology is expected to provide feasible but accurate solutions to partial matches. Previous research has proposed several approaches to identifying partial matches [Baker 1992][Mayrand 1996][Baxter 1998]. However to date, none of the existing techniques adequately fulfill the need for partial matching.

Raw clone data, usually delivered in a textual report form, can be confusing, overwhelming and difficult to work with. One feasible approach to explore in order to enhance its usability is that of visualization [Baker 1992] [Johnson 1994a]. Software visualization (SV) is a type of information visualization that combines aspects of computer graphics, data mining, human-computer interaction, and animation to facilitate users' understanding of the software system. SV strives to support the decision-making process by presenting users with the 'right' information and helping them to make the most sense of this information. Applied in software development environments, SV has been shown to deliver many benefits [Ball 1996][Linos 1994].

---

<sup>2</sup> Restructuring is the modification of software that improves its internal structure, while preserving external behaviour (functionality, semantics). Recognized benefits of improved structure are in facilitation



In fact, presenting users with thoroughly organized multi level visual display showing different occurrences of duplicated code, complemented with a convenient means of navigation between these occurrences, could have several advantages. From a re-engineering<sup>3</sup> perspective, this functionality may facilitate understanding of the possible nature of these occurrences, and, consequently, open opportunities for restructuring code and devising improved architectures. From the maintenance perspective, this functionality will help to ensure consistent maintenance: in many cases, it may be appropriate to propagate any changes made to a particular instance of a clone (i.e., fixing a bug) through the entire set.

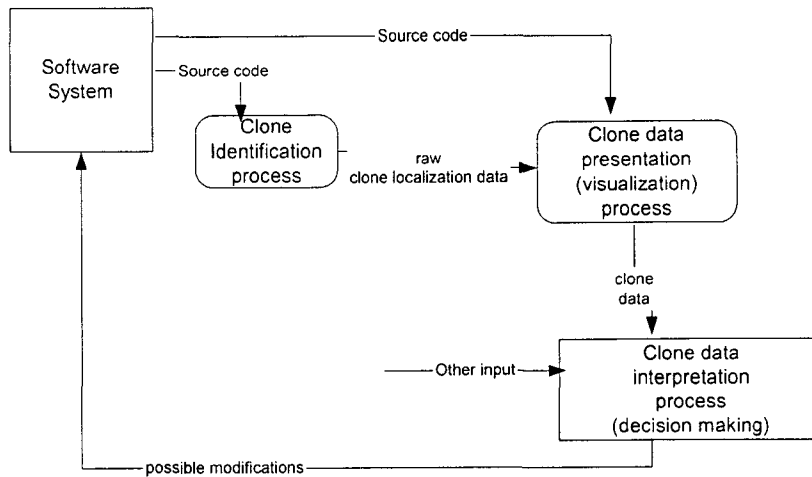
### ***1.1 Thesis Objectives***

To date, the main focus of redundancy analysis research has been on the algorithmic aspects of clone identification or on clone-based restructuring, whereas little work has been done on the practical application of these techniques in the context of software maintenance and re-engineering. Recognizing this deficiency, this thesis attempts to define and implement a comprehensive process of analyzing of software clones in large bodies of source code by integrating the three main stages of redundancy analysis: clone identification, clone data presentation, and clone data interpretation (Figure 1.1).

---

of subsequent extension and long-term maintenance [Chikofsky 1990].

<sup>3</sup> Software Re-engineering is the “examination of a subject system to reconstitute it in a new form and the subsequent implementation of the new form” [Chikofsky 1990].



**Figure 1.1:** High Level Diagram of Clone Analysis Process

The objectives of the research can be summarized as follows:

- Explore the concept of partial matches, - ‘near’ clones.
- Investigate possible approaches to identification of meaningful near clones in the bodies of large software systems. In this area, work will be centered on SelArt [Johnson 1993] [Johnson 1994a] [Johnson 1994b], an existing tool for locating exact duplications of text, in order to extend its capabilities to handling of near clones.
- Analyze the nature of clones and their occurrence in an industrial size software system.
- From the practical perspective (software understanding and maintenance), evaluate the potential benefit of extending the clone identification process to accommodate near clones.
- Design and implement a prototype clone management tool, CloneMaster. Based on a graphical interface, CloneMaster is intended to provide software developers,

software engineers, maintenance engineers, and other professionals involved with software systems at different stages of their lifecycle with powerful yet intuitive means to extract, view and manage information on software clones and their distribution within the system.

The following objectives apply to the CloneMaster system:

- Propose an approach to clone visualization to adequately address the needs of software re-engineering and especially maintenance;
- Develop adequate visual means of presenting clone related data;
- Investigate ways of producing effective displays by supplementing information conveyed via visual abstractions with different types of non-visual information (statistical info, source code view, etc.);
- Allow user interaction. Focus on achieving effective and smooth navigation within clone information spaces.

CloneMaster is designed to be a single- task-oriented tool to serve in specific field of identification of software duplications for the purposes of their understanding and possible elimination. The functionality of the tool and its user interface is designed so that clone information (programming-in-the-large) does not get scattered or even lost in large amounts of local, trivial, or irrelevant information (programming-in-the-small).

Since maintenance is such a cost and resource demanding part of software life cycle, it makes good economic sense to attempt to automate at least some of its aspects (clone

identification, in this particular case) while providing maximum computer-aided support for its other aspects (visual tool for clone analysis).

## ***1.2 Thesis Organization***

This remaining part of the thesis is structured as follows:

Chapter 2 is about software redundancy. It introduces the problem of code cloning, explores its origins and potential impact on the quality of software systems as well as suggests some possible benefits of systematic clone management. The chapter formally defines the term ‘software clone’ and other related terms to be used throughout the thesis. This chapter analyzes some prominent approaches to clone identification and rationalizes the choice of a particular clone identification technique for the purposes of this thesis research.

Chapter 3 covers Visualization topics. It begins with a general discussion of Information Visualization and its role in human cognition. It introduces Software Visualization as a sub field of Information Visualization and presents cases of successful application of visualization concepts and techniques in software engineering and maintenance. The chapter concludes with a discussion of some generalized design guidelines for program visualization.

Chapter 4 develops ideas in the domain of near clone accommodation using SelArt. It describes a three-step integrated process of exact and ‘near’ clone identification and elaborates on design and implementation issues of each of the three steps. However, the main focus of the chapter is on a pre-processing step intended as a solution for SelArt’s deficiency to detect ‘near’ clones in addition to exact clones.

Chapter 5 covers the design and implementation of the CloneMaster visualization tool. It gives a brief overview of the current state of the clone visualization field followed by the discussion of requirements that have been identified for the tool to satisfy. The bulk of this chapter focuses on design and implementation aspects of CloneMaster.

To verify hypotheses developed throughout earlier chapters, Chapter 6 presents the results of applying the proposed integrated process of clone analysis to a commercial software system. The following topics make up the discussion: assessment of the pre-processing and its role, clones found and possible solutions to their elimination, applicability of the approach, effectiveness of the visual tool.

Finally, Chapter 7 presents some conclusions and suggests directions for future work.

## *Chapter 2 – Software Clones*

This chapter is about software redundancy. It dissects the problem of code cloning, explains its origins, its potential impact on the quality of software systems, and argues towards potential benefits of systematic clone management. The chapter defines the term ‘software clone’, differentiating between such distinct kinds of clones as ‘exact’ clones and ‘near’ clones. It examines several distinct approaches to clone identification to justify adaptation of the text-based comparison approach for the purpose of the current dissertation. The chapter concludes with a brief discussion of SelArt.

### *2.1 Code Cloning*

The problem of redundant code is often underestimated. On average, a modern industrial-sized legacy system contains 5-10% of repetitive code [Lague 1997]. However, individual subsystems can exhibit even higher redundancy (up to 30%) [Baxter 1998]. Software clones may occur for a number of reasons [Kontogiannis 1996][Johnson 1994b][Baker 1992][Dagenais 1998]:

- Code reuse by copying existing code fragments

A widely practiced approach to introducing new functionality is through ad hoc reuse of existing code. Programmers simply find some code fragment performing a similar computation to the one desired, copy it, and then do any necessary alterations in place. It is easy to see why this approach is popular: making a copy and modifying it is much easier than trying to exploit commonality by virtue of

generalized procedures. Another argument in favor of copy-and-modify reuse is that it is safer than more major revisions involving structural changes to working sections of the code, especially if the programmer making changes is not the one who wrote the code originally. The 'copy-and-paste' feature, universally supported by screen text editors, also contributes to the popularity of this type of reuse.

Furthermore, there exist cases when the above method becomes adopted as a standard way of producing variant modules. For example, in device drivers: only the code responsible for interaction with the hardware changes, whereas parts of the driver talking to the operating system remain the same and, therefore, can be copied entirely. Another example comprises applications spanning multiple operating systems, such as web browsers, multimedia applications, etc. In these cases, the 'copy-and-paste' technique can have an advantage of reducing the amount of testing needed for it reuses previously tested modules.

- Coding styles

Maintaining a certain coding style may cause a regularly needed code fragment (such as error reporting routines, user interface displays, etc.) to be scattered around the system. In fact, Baker [Baker 1992] reported that a substantial part of the clones found in the source for the X Window System and a large AT&T software system was related to error checking and handling.

- Coding schemas and cliches

Some computations, due to their intrinsic simplicity and universality, are encountered so frequently that they almost become definitional (e.g., linked list insertion, array sorting, etc.), and programmers develop mental macros for coding them. Even when ‘copy-and-paste’ is not used, these mental macros may produce clones differing only in irrelevant details (i.e. variable names, ordering of statements).

- Failure to properly identify/use abstract data types

Some clones are in fact complete duplicates of functions intended for use on another data structure of the same type (insertion sort on an array, for instance). In this case, the data type operation should have been supported by reusing a library function rather than pasting a copy.

- Unavoidable cloning

Sometimes cloning cannot be avoided because the other subsystem(s) may not be modified for a variety of reasons. They may belong to a different department, be part of a different product, stored in non-volatile memory in embedded systems, be frozen after lengthy testing, be already released, etc.

- Name clashes

Often, a module has to be replicated with names changed due to name clashes at link time.



- Efficiency considerations

To avoid the overhead of generalized routines in time critical applications, the code for frequently performed computations is replicated every time the computation is invoked.
- Time constraints

Scheduling pressures contribute to code redundancy by allotting no time for code generalization.
- Maintaining multiple versions

Keeping the code for multiple versions in separate files may be preferable to working with a lot of `#ifdef`'s.
- Developer qualifications

Poor understanding of abstraction, inheritance, composition, etc. among developers limits their reuse approach.
- Lack of support for formal reuse process

Reusable components are not documented or made readily available to the developers.
- Programmer productivity

Evaluating the performance of a programmer by the amount of code produced gives a natural incentive for copying code.

Cloning activity can happen for code fragments of a few lines or for modules of thousands of lines. It can happen in procedural code as well as in object-oriented code. It can happen in documentation. Although code duplication can have its justifications, it is considered bad practice. Some of the negative consequences of software cloning are addressed in the next section.

## ***2.2 Implications of Code Cloning***

As clones accumulate in the system, one can expect a decline in code quality and growth in its sheer size, requiring more personnel, more effort and more resources to maintain its working condition and to respond to market pressures. Some specific problems caused by clone proliferation are considered below [Lague 1997][Kontogiannis 1996][Johnson 1994b]:

- When a bug has been found in one copy, a bug fix may be made to the place where the bug was found, but not to the corresponding parts of other copies simply because the programmer is not always aware that these other copies exist. Consequently, the same bug reoccurs throughout software despite many local fixes. Moreover, multiple unique fixes for the same bug are produced.

- Since the internals of the module are not completely understood (because of time pressure, etc.), unnecessary artifacts of the previous usage can be preserved in the current code (“dead code”).
- As the structure of a software system degrades, there becomes much more code to analyze and maintain. Multiple modules exhibiting the same functionality may be confusing because the reason for their existence is not clear.
- Testing costs increase since there are more modules to test.
- Code duplication increases the size of the code, extending compile time and expanding the size of executables. Code reviews are needlessly extended.
- Code duplication masks potential design problems like missing inheritance or missing procedural abstraction. Without being addressed, such design flaws could hamper the addition of new functionality.
- Proliferating of clones may eventually lead to the loss of design rationale: nobody can explain how a system could be designed this way; so, “instead of logically comprehending the system, it is treated as a living, organic mess that grows in cost and size of its own accord” [McCabe 1990].

A study described in [Monden 2002] used metrics to quantitatively clarify the relation between code clones and module maintainability and reliability. The study found clone-containing modules to be considerably less maintainable and less reliable than clone-free modules.

Clearly, if not properly addressed, software redundancy may create a lot of problems for both software providers and their clients. The next section discusses some aspects of software clone management.

### ***2.3 Management of Software Clones***

Often viewed as overhead, systematic management of software clones may translate into significant budget savings [McCabe 1990] [Lague 1997]. Detection and removal of clones promises an immediate decrease in software maintenance costs of possibly 5-10% magnitude [Baxter 1998]. However, even larger savings could follow from improved architectures (i.e., readability, changeability) and proper reuse [Fowler 1999].

Several academic researchers and practitioners have contributed to clone management in the past years. Some focused on clone detection techniques, some explored software restructuring actions based on clone detection, others worked on developing preventive measures to avoid cloning altogether.

Baxter et al. suggested that applying clone detection as a part of “never-ending mining and refactoring operation could reveal missing abstractions and significantly mitigate the risk of cut-and-paste programming” [Baxter 2002]. They argued in favor of automated clone detection followed by automatic clone elimination by replacing clones with better structured code (i.e., include files, copylibs, typedefs, macros, inlined procedures, etc.) [Baxter 1998].

[Kane 1997] proposed systematic merging of clones into a common baseline, at least once every release, as a mechanism to exercise control over clone proliferation.

Dagenais et al. [Dagenais 1998] suggested that in cases when removing clones via replacing them with reusable components is not feasible, links between the clones could be added to ensure consistent maintenance.

Based on the premise that duplication in software implies latent abstractions, [Fanta 1999] [Balazinska 1999] [Kataoka 2001] explored opportunities for clone based refactoring of object-oriented systems.

Kontogiannis et al. [Kontogiannis 1996] showed that clone detection could positively contribute to systematic reuse. They identified often-cloned functionality as prime candidates for generalization and repackaging for repositories of reusable components to facilitate future development.

McCabe [McCabe 1990] emphasized that design structure surrounding redundant code could be also redundant. Therefore, studying clone occurrences could help to pinpoint design flaws and suggest improved architectures.

Mayrand et al. [Mayrand 1996], researching clones at the function level, developed a scale comprising 10 levels for evaluating the “goodness” of clones, with more acceptable clones residing at higher levels. Based on this scale, they devised a set of clone removal strategies. For instance, level 1 clones, or exact duplicates, can be removed at low risk by creating common libraries of functions, whereas higher level clones could be tackled by means of parameterization (function parameters, preprocessing macros, function templates, etc.).

To reduce maintenance costs and to minimize risk of failures, Lague et al. [Lague 1997] developed two techniques, Preventive Control and Data Mining, to control clone proliferation. Preventive Control ensures against introducing unnecessary new clones into the system. Data Mining, on the other hand, focuses on consistent management of existing clones.

Clone management strives to increase both quality and maintainability of software systems. It emphasizes the importance of a systematic approach to redundancy analysis and places increasing demands on the clone detection technology that is expected to provide efficient, feasible but accurate solutions to the problem of clone identification.

Literature research indicated a lack of adequate clone identification techniques to support successful clone management. Consequently, clone detection has been identified as one of the main areas of interest within the scope of this thesis work.

Section 2.5 of this chapter examines some prominent clone identification techniques currently used by the industry and presents argumentation in favor of choosing a text-based clone identification tool to fulfill the requirements of this thesis. The purpose of the next section is to further clarify the meaning of the term ‘software clone’ and to define some additional terminology to be used in future discussions.

## ***2.4 Software Clones Revisited***

In the context of this thesis, the term ‘software clone’ is used to refer to a program fragment that is identical to another fragment or is a variation of it. The former corresponds to ‘exact’ clones; the latter defines ‘near’ clones. In other words, a software clone is a copy of existing piece of code that underwent (possibly empty) modification.

Exact clones, products of ‘cut-and-paste’ activity with no further customization, are straightforward. Near clones are much more subtle. There exist two potential sources of them: cut-and-pasting and mental schemas<sup>4</sup>. In the case of ‘cut-and-paste’, a code fragment is copied and then edited. However, empirical observations show that changes

---

<sup>4</sup> Baxter refers to them as mental macros [Baxter 1998]

introduced into the copy are usually cosmetic and do not alter its structure. Typical changes include:

- Modifications of identifier names, constants and numbers.
- Addition or removal of comments.
- Reorganization of source code page layout (formatting and whitespace).
- Interchanging of commutative operands in arithmetic expressions (unlikely).
- Addition or deletion of statements (unlikely).
- Rearrangement of the sequence of statements (unlikely).
- Adding an additional block structures such as 'IF...THEN...ENDIF' (unlikely)

Consequently, a near clone is not identical to its original counterpart; nevertheless, the two exhibit a high level of syntactic similarity. Near clones of mental schema origin, on the other hand, produce a considerably harder case. Not based on copying, they normally have lower degree of syntactic correlation and thus are harder to identify. To the best knowledge of the author, none of the currently available methods of clone identification is capable of handling them adequately.

Every clone instance can be characterized by the attributes of contents, file, boundaries, length, and cardinality. The contents attribute refers to the actual source code comprising the clone; the file attribute is the file the clone instance resides in; boundaries reflect clone's positioning within the file (beginning/ending offset); the length attribute corresponds to the physical size, measured in number of lines/characters, of the clone instances. In case of exact clones, all instances of the same clone will share the same

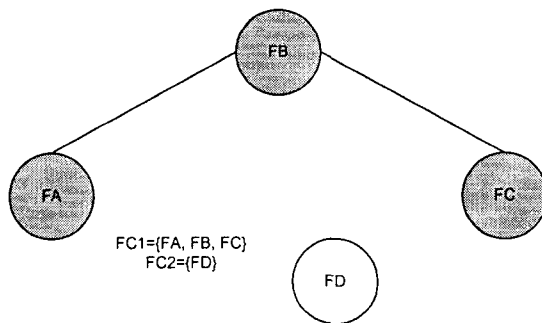


contents and the same length. In case of near clones, however, the contents and the length attributes might vary from instance to instance. Cardinality of a clone is defined as the number of instances of that clone.

At this point, it is customary to introduce two more terms to be used in further discussion: 'clone cluster', and 'file cluster'. A clone cluster is a set of all instances of the same clone and has two main attributes: 'size' and 'file span'. A size of a clone cluster refers to the number of clone instances comprising it, whereas a file span of a clone cluster is taken to be a set of all files sharing that clone.

If a clone cluster is a set of matches, a file cluster is a maximal set of files that have matches in common. The only attribute of a file cluster is its 'size'. A size of a file cluster is equal to the number of comprising files. File clusters consisting of a single file are often referred to as 'singletons'. File clusters that span two files are called 'pairs'. File clusters involving more than two files are called 'complex'. For example, file FA contains clone C1, file FB contains clones C1 and C2, file FC contains clone C2, and file FD contains two instances of clone C3. This example contains 2 file clusters:  $FC1 = \{FA, FB, FC\}$  with size = 3 and  $FC2 = \{FD\}$  with size = 1. FC2 is a singleton, while FC1 is a complex cluster. File cluster identification is an important part of redundancy analysis for it reveals an additional layer of dependencies not usually captured by conventional analysis (i.e., control flow graph, data flow graph, structure chart, etc.). In other words, if a system file structure were represented as a graph, file clusters would correspond to connected components of this graph (Figure 2.1), and thus effectively

partition the system into independent regions. It is apparent that different types of file clusters reflect different influence of clones on software quality. For instance, pairs and complex clusters implicitly indicate inter-module coupling, whereas singletons do not.



**Figure 2.1:** An example of system partitioning by file clusters.

Lastly, there are occasional code fragments that are just accidentally identical, but are not at all clones (not intended to carry out the same computation). However, Baxter et al. concluded, “as size [of clones sought] goes up, the number of accidents of this type drops off dramatically” [Baxter 1998].

## 2.5 Clone Identification

Strictly speaking, determining whether two arbitrary pieces of code are clones of each other (i.e. compute identical result) is known to be undecidable in the general case (a variation of the halting problem) [Kontogiannis 1996]. However in practice, since most of the clones are results of the ‘copy-and-paste’ editing process, detecting complete semantic equivalents is not necessary. Even if modifications take place, the structure of the fragment is rarely changed. Therefore, clone detection can quite legitimately be

based on the observation that clone instances intrinsically exhibit a high degree of syntactic similarity [Baxter 1998].

A number of methods and techniques for identifying replicated code have been described in the literature, varying widely in underlying approach, accuracy, and performance. Techniques based on statistical comparisons of style characteristics (use of operators, use of special symbols, the order in which procedures are referenced, etc.) [Jankowitz 1988], techniques based on comparing arrays of specific software metrics [McCabe 1990] [Kontogiannis 1996] [Mayrand 1996] [Dagenais 1998], techniques exploiting compiler technologies [Baxter 1998], techniques based on matching of the character strings composing the code [Baker 1992], [Johnson 1993], and even techniques based on neural networks [Barson 1995], just to name a few. Of these approaches, the most relevant to the current work are analyzed below.

### *2.5.1 Direct Metrics Comparison Approach*

In the metrics approach, the system is broken down into components, and then a set of selected metrics<sup>5</sup> is computed for each component and used (via numerical comparison) to estimate distance between them. Components that are close together are assumed to be clones.

---

<sup>5</sup> These metrics relate to aspects of sequences of instructions such as their layout, the expressions inside them, their control flow, the variables used, the variables defined, etc.

One of the most important characteristics of a clone detection technique is its granularity, i.e. the minimum module considered for analysis. The metrics approach usually works at the function level [McCabe 1990] [Mayrand 1996] [Dagenais 1998].

Detection of near clones is possible due to insensitivity of metrics to fine detail that doesn't alter control and data flow of the code structure, however, at the expense of accuracy: false matches are too frequent (39% [Kontogiannis 1996]). It has been shown [Kontogiannis 1996] that the accuracy of exact matching can be considerably improved by expanding the set of metrics used and by using additional verification techniques. Unfortunately, the solution is not applicable to cases when 'near' clones are being targeted.

Direct metrics comparison is a feasible approach to clone identification because of its affordability, ability to discover near clones, and the ability to measure the degree of similarity of these matches. However, such weaknesses as coarse granularity and high level of false positives, especially for smaller procedures, severely limit its usefulness.

### ***2.5.2 Abstract Syntax Trees Comparison Approach***

This approach is based upon such conventional compiler technology as abstract syntax trees (AST) and operates in following steps:

1. The code is parsed to produce an AST

2. Sub-tree clones are found via comparison of sub-trees for similarity (similarity threshold)

The AST approach lends itself well to identifying clones differing only in lexemes (identifier names, numbers, string literals, etc) and/or formatting since it is not sensitive to irrelevant changes on the lexical level (i.e. comments, spacing, layout) and performing comparisons for similarity, rather than for equality. Detection in terms of program structure allows clones to be automatically factored out and replaced with equivalent preprocessor macros, type declarations, subroutine calls, or inlined subroutine calls. Other strengths of the approach are its capability to find clones in arbitrary code fragments (as opposed to the metrics approach that operates on complete function bodies) and to provide measures of similarity between clones. Although the AST approach has been stated to accommodate a wide spectrum of near clones including those with commutative operators and re-ordered statements [Baxter 1998], evaluation of a particular AST-based implementation, CloneDR<sup>®6</sup>, performed in the course of this thesis yielded only limited support in favor of this claim.

One obvious disadvantage of comparing ASTs is the computational cost. However, some implementations claim to show adequate performance and even outperform some metrics comparison implementations [Baxter 1998]. Another constricting requirement of the AST approach is its dependability upon syntactic correctness of the source.

---

<sup>6</sup> CloneDR<sup>®</sup> is a commercial AST-based clone identification tool created by Semantic Designs Inc.

### *2.5.3 Text Based Comparison Approach*

A text-based comparison approach is a generalization of string pattern matching. The source code text can be represented as a set of strings<sup>7</sup>; thus the problem of finding duplicated sections of code becomes a problem of finding duplication in strings, the ultimate goal of which is to identify all maximal matching sub-strings over a certain threshold. String matching algorithms treat source code strictly as text (program syntax or/and semantics are not considered).

The appeal of the textual approach is due to the following reasons:

- From the implementation point of view, it takes advantage of data structures and efficient algorithms developed for string pattern matching.
- The source code doesn't have to be syntactically correct. In the case of C/C++, it also avoids common problems with preprocessor directives, as their syntax does not always conform to the grammar.
- Text-based approach is language independent.

The major shortcoming of text-based comparisons is their extreme sensitivity to nuances on the lexical level. Near clones differing in spacing, comments, identifier names, numbers or string literals will be totally missed.

---

<sup>7</sup> A string over a certain alphabet is a sequence of symbols, each of which belongs to that alphabet.

One popular approach to alleviate the above limitation is to use parameterized strings instead of regular strings [Baker 1992]. After obtaining parameterized strings from regular strings via substitution of all appropriate token names (i.e. variables, constants, macros, structure members) with a 'p' (i.e.  $x = 3*y$  will translate into  $p = p*p$ ), these parameterized strings are submitted to one of the exact matching algorithms. This adjustment achieves identification of parameterized (i.e. partial) matches, where the code sections match except for a one-to-one correspondence between parameterized tokens. Finally, a simple verification procedure is applied to ensure that sections of code encompassed by the match are either identical (exact match) or related by the systematic renaming (Figure 2.2).

<pre> x = y - z; if (y &gt; z)     m = 1; h = f(x); y = x; </pre>	<pre> x = b - c; if (b &gt; c)     n = 1; h = f(x); c = x; </pre>
---	---

**Figure 2.2:** Parameterized match verification. Consider two code fragments shown in the boxes above. Matching on p-strings reported them as exact matches. The four first lines require pairings  $y = b$ ,  $z = c$ , and  $m = n$ . However, the fifth line requires a pairing  $y = c$ , which conflicts with the previous pairings. Consequently, only first four lines should be reported as a partial match.

Traditionally, text-based matching relies on line-based comparison on the premise that the line structure of the original is preserved in the clone. Alternatively, Johnson [Johnson 1993] succeeded in the identification of exact matches in a stream input (no line boundaries, one long line). Johnson also successfully attempted near clone identification: Ignoring all whitespace other than line separators revealed some new

matches overlooked by exact matching [Johnson 1994b]. Johnson's research revealed the suitability of the approach for near clone detection for it allows (via special arrangements) making most formatting changes (except for statement order and commutative operands) virtually irrelevant. Johnson's technique is discussed in more detail in section 2.5.5.

#### ***2.5.4 Choice of Approach***

For the purpose of this thesis work, the text-based approach is used. The rationale behind this choice is explained below:

- All considered clone identification approaches offer comparable results in terms of exact matching while providing limited support for approximate matching.
- A text-based tool for identifying exact clones, SelArt [Johnson 1993] [Johnson 1994a] [Johnson 1994b], was readily available to us<sup>8</sup>.
- The text-based approach is simple; it produces accurate results when exact matches are sought, shows adequate performance, and scales well.
- The text-based approach is language independent: language extensions, syntactically incorrect constructs or incomplete code fragments can be easily accommodated. In case of languages that support preprocessor directives, clone detection can be performed without having to expand these directives.

---

<sup>8</sup> Use of SelArt (in executable form) for the purpose of this research was authorized by its creator, Dr. Howard Johnson of NRC Canada.



- The fact that in its raw form the text-based comparison technique is totally intolerant towards near clones provides scope to explore different enhancements to this clone detection technique in order to allow for near clones. It is important to emphasize that the main focus of this work is assessing potential benefits achieved by these changes, rather than assessing the clone detection technique itself.
- The results report produced by SelArt is both human-readable and lends itself well to automated processing.

The extension of SelArt’s functionality to handling approximate matches is covered in Chapter 4. The next section gives a brief overview of SelArt.

### ***2.5.5 SelArt – a Tool for Identifying Redundancy in Source Code***

SelArt is a research prototype of a tool for locating exact repetitions of text in large bodies of text using “fingerprints” [Johnson 1993] [Johnson 1994a] [Johnson 1994b].

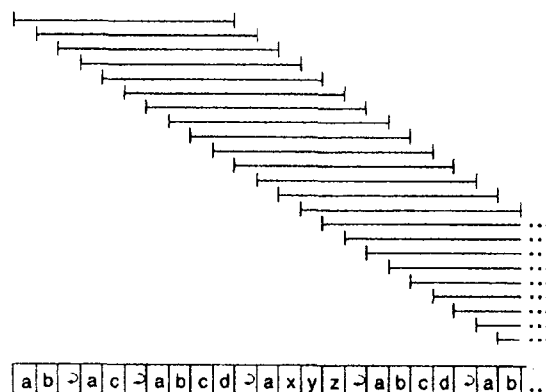
A fingerprint is a short string that can be used to represent a larger data object for comparison or other purposes to improve algorithms’ time and space efficiency.

In general, a fingerprinting function  $f(x)$  maps data objects from some data-object domain  $D$  into a set of fingerprints  $F$  such that  $f(x) \neq f(y)$  implies  $x \neq y$  with extremely high probability, and  $f(x) = f(y)$  implies  $x = y$  with extremely high probability

[Johnson 1993]. Thus, an equality test between two data objects can be performed by comparing their corresponding fingerprints.

Another useful notion in the context of the current discussion is that of a snip. A snip is “a sequence of characters in a source file. It is identified by file name, beginning offset, ending offset, and has as content the substring so identified” [Johnson 1993]. Thus any source code can be represented as a set of snips. Two snips match if their contents agree and, consequently, their fingerprints are identical.

To obtain fingerprints, SelArt uses the Karp-Rabin algorithm [Karp 1987]. Each sequence of  $n$  characters is considered to be a numeral in some base  $r$  with the value computed by multiplying the integers stored in individual bytes by the appropriate powers of  $r$  and sums them up. A fingerprint of the sequence (snip) is the remainder of the division of the above sum by some large prime number  $p$ . Thus a fingerprint is an integer value ranging from 0 to  $p-1$ .

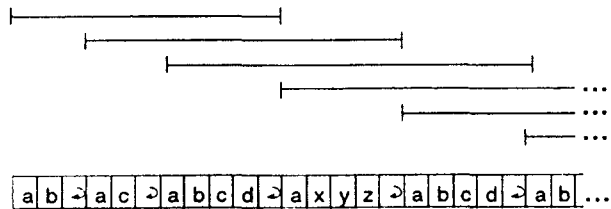


**Figure 2.3:** A set of snips of length 10 characters generated to represent the source code text (arrows indicate end of lines).

For instance, a fingerprint of the following snip of  $n$  characters  $c_i c_{i+1} \dots c_j \dots c_{i+n-1}$  starting at the position  $i$  will be the following integer value  $f_i$ :

$$f_i = (c_i * r^{n-1} + c_{i+1} * r^{n-2} + \dots + c_j * r^{(i+n-1)} + \dots + c_{i+n-1}) \bmod p.$$

Figure 2.4 illustrates a set of snips generated in cases of very long lines or no recognizable lines (stream). However, if source code is organized in relatively short lines, seeking matches in terms of number of lines is possible (Figure 2.4).



**Figure 2.4:** Set of snips of length 3 lines generated for the fragment of code of Figure 2.3.

In order to give the user necessary control over the matching process, SelArt uses the following four parameters:

- $l$  - desired number of lines in a snip
- $M$  - maximum allowed number of characters in a snip
- $m$  - minimum allowed number of characters in a snip
- $c$  - cull parameter ranging from 1 to  $M$ ; As  $c$  increases, amount of culling decreases;  $c = M$  disables culling.

These parameters determine the set of snips to be fingerprinted. A snip is an entity whose size is determined by the triplet of values  $(l, M, m)$  and that is produced independent of the context. Setting  $M = m$  yields character strategy; setting  $m$  to zero

and  $M$  to very large number results in pure line count case. Culling causes certain degradation in precision of clone identification. At worst, all matches longer than  $2 * M - c$  are guaranteed to be found.

After all individual matching snips have been identified, bigger matches are constructed out of them. The combine-and-split strategy of SelArt is best illustrated by example:

**Example 1:**

There exist two matching snips with content  $x$  that are each followed immediately (touching or overlapping) by matching snips with content  $y$ , given that there are no other snips with either  $x$  or  $y$  content which have been encountered in the system (Figure 2.5a). Combining each pair of overlapping (touching) snips will yield two larger snips instead of four (Figure 2.5b). No information about location of matches is lost.

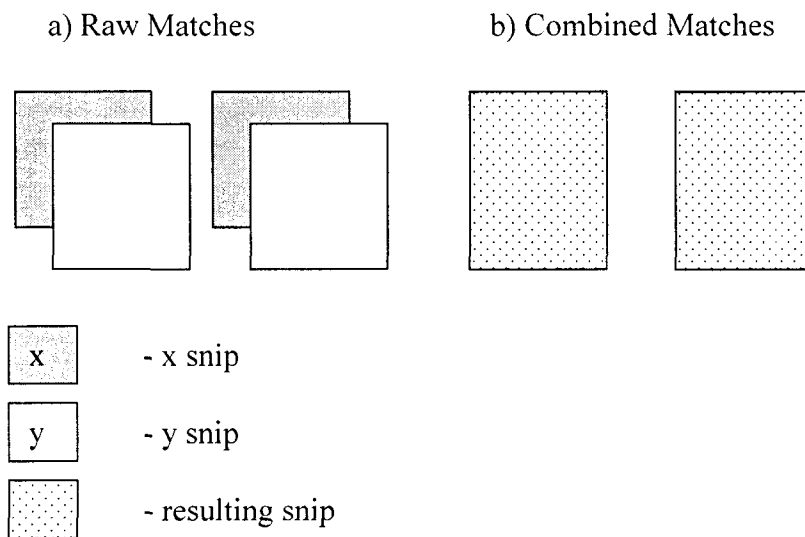
**Example 2:**

Consider the situation depicted in Figure 2.6a with a third snip of content  $x$  occurring elsewhere in the source, not followed by a snip with content  $y$ . SelArt will keep the  $x$  snips but shorten the  $y$  snips, such that  $x$  and  $y$  do not overlap (Figure 2.6b).

Consequently, a three-way  $x\_length$  match and a two-way match sized  $(y\_length - overlap)$  will be constructed. In this case, information about a two-way match of size  $(x\_length + y\_length - overlap)$  will be lost.

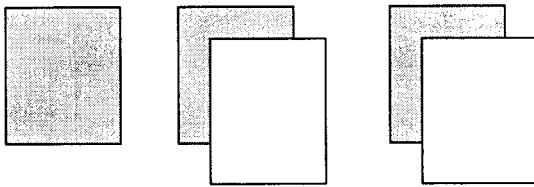
By combining and splitting matches, a partition of the files into disjoint snips is constructed in such a way that all the information about matches obtained in earlier stages is represented as matches on these disjoint snips. This information is organized as a set of records in a flat ASCII file. All record fields and field separators are clearly defined thus making the file suitable for automated processing.

The SelArt tool has shown adequate performance when applied to software systems of substantial size (up to 500 MB) [Johnson 1994a].



**Figure 2.5:** Combining raw matches

a) Raw Matches



- x snip

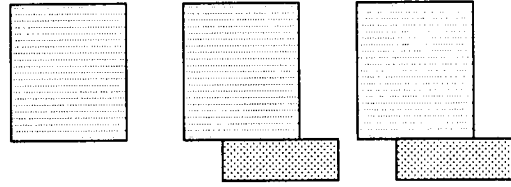


- y snip



- resulting snips

b) Combined and Split Matches



**Figure 2.6:** Combining and splitting raw matches

## ***Chapter 3 – Information Visualization***

The focus of this chapter is on information visualization. It provides a brief introduction into the topic with special emphasis on software visualization. The chapter concludes with a discussion of some design principles to facilitate creation of efficient program visualization systems.

### ***3.1 Origins of Information Visualization***

The progress of human civilization has proven that visual artifacts aid thought. From writing to mathematics, to maps, to printing, to diagrams, to visual computing, - visual artifacts have profound effects on peoples' abilities to assimilate information, to understand it, to create new knowledge. Information visualization is just about that – “exploiting the dynamic, interactive, inexpensive medium of computer graphics to devise new external aids that enhance cognitive abilities” [Card 1999]. According to Ware [Ware 1999], integration with computer technologies has allowed visualization to become “an external artifact supporting decision making”.

Card et al. argued that visualization can enhance “cognitive effort by several separate mechanisms” and described six major ways in which information visualization supports cognition [Card 1999]:

- 1) Increasing the memory and processing resources available to the user: high-bandwidth hierarchical interaction, parallel perceptual processing, offloading

work from cognitive to perceptual system, expanded working memory, expanded storage of information.

- 2) Reducing the search for information: grouping information used together, high data density, locality of processing, hierarchical search.
- 3) Using visual representation to enhance the detection of patterns: recognition instead of recall, simplification and organization of information through abstraction and omission, organization to reveal patterns.
- 4) Enabling perceptual inference operations: visual representations make some problems obvious, facilitates hypothesis formation.
- 5) Using perceptual attention mechanisms for monitoring: organizing visual displays such that events of interest stand out by appearance or motion.
- 6) Enabling dynamic exploration and navigation of information space by encoding information in a manipulable medium.

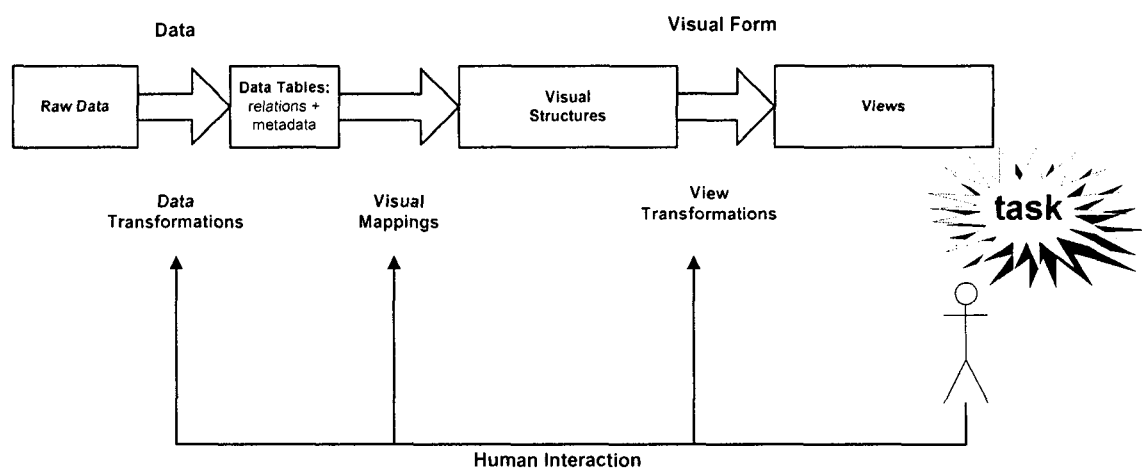
The ultimate goal of information visualization is to change the way we present, manipulate and understand large complex data sets by transforming the information and knowledge into visual form, leveraging people's natural abilities of rapid visual perception and pattern recognition.

In contrast to scientific visualization, which focuses on physical data, information visualization focuses on nonphysical information, which is often abstract and doesn't automatically map to the physical world and, therefore, lacks natural and obvious physical representation (i.e., financial data, business information, collections of



documents, abstract concepts). Software, for example “ is intangible, having no physical shape or size. After it is written, code disappears into files kept on disks” [Ball 1996]. Consequently, creating effective mappings of nonspatial information into visual form (i.e., symbols) is one of the major tasks and challenges of information visualization. As emphasized by Ware [Ware 1999], establishing and following consistent graphical conventions with regards to symbol interpretation is equally important for it reduces “the labor of learning new meanings”.

The diagram in Figure 3.1 presents a simplified model of information visualization that is based upon adjustable mappings from data to visual form to the human perceiver (adapted from [Card 1999]). Arrows from data to the human indicate a series of data transformations. Arrows from the human into the transformations indicate the potential for user input.



**Figure 3.1:** Visualization model. Mapping data to visual form.

The purpose of visualization “is insight, not pictures” [Hamming 1973]. Information visualization is only useful to the extent that it amplifies our cognitive abilities. Therefore, emphasis of any information visualization, and this thesis work especially, should be on the use of the picture to give rapid insight into the data, rather than on the quality of the graphics.

### ***3.2 Software Visualization***

Software Visualization is an emerging field in the fast developing discipline of Information Visualization. Baecker et al. referred to SV as “a branch of software engineering that strives to aid programmers in managing the complexity of modern software” [Baecker 1998].

There have been numerous definitions of SV proposed since it started to emerge in the early eighties. In their milestone paper “A Principled Taxonomy of Software Visualization” [Price 1993], Price et al. defined software visualization as “the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software.” Others proposed similar definitions of software visualization. Domingue et al. suggested that “software visualization describes systems that use visual (and other) media to enhance one programmer's understanding of another's work (or his own)” [Domingue 1992]. Muthukumarasamy and Stasko described SV as “the use of visualization and animation techniques to help people

understand the characteristics and executions of computer programs”  
[Muthukumarasamy 1995].

At present, the term Software Visualization is still quite broad, encompassing almost all forms of visualization concerned with representing any aspect of a software system. To date, a number of taxonomies and surveys of the software visualization field have been published, however, their discussion is beyond the scope of this thesis work.

### ***3.3 Visualization in Maintenance and Re-engineering***

The amount of legacy code accumulated and its poor condition require powerful specialized software tools to support various maintenance and re-engineering activities [Chikofsky 1988]. During the last decade, a number of software tools have been developed to explore the application of various visualization techniques in an attempt to enhance program representation, presentation, and appearance to the user.

The majority of recent tools (NestedVision3D [Ware 1993] [Parker 1998], Rigi [Storey 1998] [Rigi 1999] [Martin 2000] [Storey 2000], Visual Reengineering ToolSet [McCabe 1999]) perform interactive structural visualization of code using graphs where nodes corresponded to software artifacts (variables, classes, data types, functions, methods, files, modules, etc.), and directed arcs corresponded to relationships between them (function calls, data dependencies, inheritance relationship, etc.). Others see value in preserving low-level detail of the actual source code through enhancing its textual

appearance (SeeSoft [Eick 1992] [Ball 1996] [Eick 1998]). However, it is not always clear how to present program information visually in a way that could significantly boost understanding.

### ***3.4 General Design Guidelines for Program Visualization***

Although researchers have been exploring the matter of creating effective visual displays for quite some time, human visual perception of visually displayed information is not fully understood, and most of the existing visualization systems are based merely on heuristics and trial-and-error. However to date, enough research and practical experience has been accumulated to isolate successful trends as well as to attempt some generalization. This section presents some general heuristic design guidelines for building successful information-conveying visualizations of abstract data [Eick 1995] [Ware 1999], particularly software [Storey 1997].

1. **Task specific:** Focusing on the task fosters better understanding of the visual system's requirements and therefore leads to engineering displays and representations to suit these needs.
2. **Reduced representation:** A reduced overview to display the entire target system on a single screen should be provided to serve as navigation guide and coordination mechanism for the finer, more specific views of the data set. This overview arrangement has to be pleasing, informative, and context-preserving, using appropriate representations for the underlying data set.

3. **Data Encoding:** Color and other visual cues such as position, size (area, length, height), shape (orientation), motion (blinking), etc. could be used to represent information. Of all the ways to encode information, color is most powerful. It allows for high information density displays, requires little training to enable subjects to utilize information conveyed through the medium, can be processed pre-attentively (Pre-attentive processing occurs prior to conscious attention), and is easy to implement. However, some peculiarities of color should be carefully considered for they can impair its effectiveness. Although it is possible to display many millions of colors, only a small number of them (13) can be rapidly discriminated. Color resolution is not uniform along different axes (i.e., resolution on a black white axis is much higher than along the yellow blue axis). Some perceptual distortions are possible due to simultaneous contrast (e.g., background – foreground contrast, color and brightness contrast). Another noteworthy issue is color-blindness that affects about 8% of men and 0.5% of women. Most of them have reduced abilities in distinguishing reddish from greenish shades.
  
4. **Metaphors:** Use only familiar or readily inferred visual metaphors for the behavior being presented to lower the cognitive load imposed on the user and increase the rate of comprehension. Use metaphors drawn from nature and everyday life in addition to specific application domains.

5. **Filtering:** Use interactive filters to focus the display. By turning irrelevant bits of information off, interactive filters can reduce visual and conceptual complexity of the system. An example of a filter would be exclusive highlighting (re-coloring) bits of information that are of interest.
  
6. **Drill Down:** “Drill down” techniques are useful for obtaining details about particular items. Upon locating an interesting pattern, the user should be able to access the actual underlying data values.
  
7. **Multiple Linked Views:** Presenting information in multiple views, each showing one aspect of the data and answering a specific question, can be more effective than extracting all of the needed information from a single, usually overloaded, view. To be effective, these multiple views should be tightly linked to each other, such that an operation performed in one view (for example, color manipulations) is instantly propagated through the rest of them.
  
8. **Source Code Browsing:** For many programmers, the source code of the software system is the most trusted form of documentation. According to the integrated model of software comprehension [Mayrhauser 1995], programmers frequently switch between top-down and bottom-up approaches. To facilitate programmer’s ability to rapidly switch between a high-level view of the software and low-level source code, source code views must be seamlessly integrated into higher-level architectural browsing.

9. **User Interface:** To increase the effectiveness of visualization, allow the user some degree of control over the display by enabling direct manipulation of any item on the screen. An adequate interface, albeit intuitive to use, must provide constant and continuous feedback.

Again, developing highly interactive visualization systems requires adequate hardware support and careful software design to achieve acceptable degree of responsiveness.

10. **Animation and Motion:** For data sets with a temporal aspect, animation can be used efficiently to show the evolution of that data. One of the most successful examples of animation so far is algorithm animation in ‘Sorting out Sorting’ [Baecker 1981]. Another good example is NestedVision3D-Trace [Parker 1998]. To be effective, animations must be smooth and continuous: jerking stands out perceptually and thus is distracting.
11. **Graph Layout:** The layout of a graph (the relative sizes and positions of nodes and arcs) strongly affects readability. For software visualization, the graph layout must be designed both to facilitate analysis of the program and to reduce visual clutter.
12. **3D vs. 2D:** The third (depth) dimension of 3D graphics can be utilized to considerably increase information density of the screen without overloading it. 3D

representations augmented with proper viewing techniques (3D rotations, variable view angles, etc.) and depth cues (motion, lighting, stereo, etc) generate more efficient spatial layouts and reduce the number of crossing and intersections. Ware and Franck [Ware 1994] showed that “a [3D] static perspective image may add little in comparison with a 2D diagram and adding real time rotation is considerably more important.” In the same study, combining stereo with motion (head coupling) resulted in a 200% increase in the amount of information that could be understood. Unfortunately, 3D graphical representations suffer from a number of additional problems (i.e., occlusion, disorientation, spatial complexity) and can get overly complex for large graphs (only at higher threshold). Some tasks, such as pattern recognition, are better supported by the 2D representation; 2D is also more compatible with paper presentation.

The purpose of the above guidelines has been to summarize the experience that has been garnered in the area of program visualization. These design guidelines are of particular interest to the current thesis work for they will be relied on in the process of design and implementation of CloneMaster, the clone visualization system (Chapter 5).



## ***Chapter 4 – Near Clone Identification with SelArt***

This chapter describes a three-step integrative solution devised to extend the functionality of SelArt and to enable it to detect near clones. The chapter elaborates on design and implementation details of each of the three steps together with some major integration issues. Empirical validation of the developed solution can be found in Chapter 6 that presents results of a redundancy analysis performed on an industrial software system.

### ***4.1 Background***

When it comes to seeking near clones, the idea of parameterized matching is by no means new, nor is the idea of ignoring all white space (except for new line characters) [Baker 1992], [Johnson 1994a], [Kontogiannis 1996], [Ducasse 1999].

Traditionally, near clone detection is a three-step process. First, the code is transformed into some intermediate format (parameterization). Second, a more or less sophisticated comparison algorithm is executed to identify exact matches (code sections that match in their parameterized form). Finally, a verification process examines all identified matches to ensure that sections of code encompassed by each match are either identical (exact clones) or related via systematic renaming (near clones); matches that fail such verification are rejected (false positive).

## *4.2 Near Clone Detection with SelArt*

Based on the condition that no modifications to SelArt per se were possible (no access to the source code), considering successful accounts of using parameterization reported in previous work [Baker 1992] [Ducasse 1999], and based on our definition of the term ‘near’ clone (section 2.4), the following integrated solution is proposed:

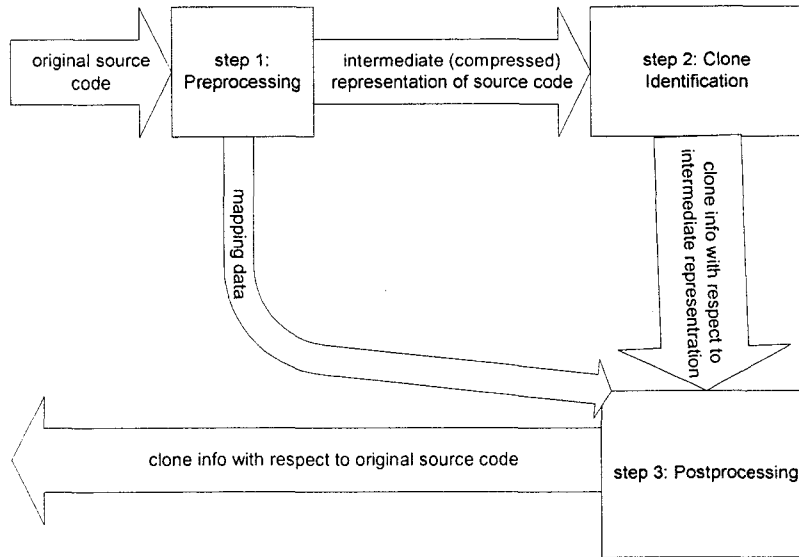
- **Step 1.** Pre-processing<sup>9</sup>: Prior to invoking SelArt, apply a text-to-text transformation of the source code to discard characters not to be considered for matching (section 4.3). Such a transformation is not language independent and requires specialized parsers.
- **Step 2.** Clone identification with SelArt:
  1. Utilize the ability of SelArt to seek exact clones in a stream (as apposed to line oriented) input on the premise that in most modern programming languages line structure is irrelevant and, thus, can be considered a matter of formatting.
  2. Provide proper configuration of SelArt (via its external parameters) to ensure adequate correspondence between line-based and stream-based matching.
- **Step 3.** Post-processing: Since the data considered for matching undergoes pre-processing transformation, results must be mapped back onto the original data set (original source code). This step doesn’t have any bearing on the clone

---

<sup>9</sup> In the context of this thesis, the term ‘pre-processing’ is used to refer to an operation preceding invocation of the clone detection algorithm, whereas ‘preprocessing’ is used to refer to the 1<sup>st</sup> stage of a compiler that extends preprocessor directives.

identification process itself, but it is necessary to relate two different reference systems: source code before and after the pre-processing transformation.

Figure 4.1 clarifies the relationships between the steps by depicting the data flow between them.



**Figure 4.1:** Data flow of the clone identification process

The current solution doesn't contain a verification step traditionally inherent to parameterized matching. This step is omitted based on the hypothesis (yet to be verified) that when matches of considerable length are being sought (very short matches are mostly noise anyway), occurring of false positive is quite unlikely. Furthermore, false positive is more acceptable to us as it would be for some automatic solutions to clone removal (i.e., [Baxter 1998]). Based on user interaction<sup>10</sup>, our approach alleviates the complexity of analysis necessary with an automatic approach while giving more flexibility to the user. Besides, defining the threshold of precision in clone identification

is virtually impossible due to the quite imprecise nature of the underlying assumption itself: if two code fragments can be generated by the same patterns then they could be clones [Ducasse 1999]. False negatives (missed clones), on the other hand, produce a challenging case since they are much harder to uncover.

The intent of this implementation is to provide a means for materializing ideas, experimenting with them, and to facilitate verification/rejection of the underlying hypotheses. It is intended to be a ‘prove-of-concept’ prototype, rather than a full-fledged product.

### ***4.3 Step 1: Pre-processing Transformation***

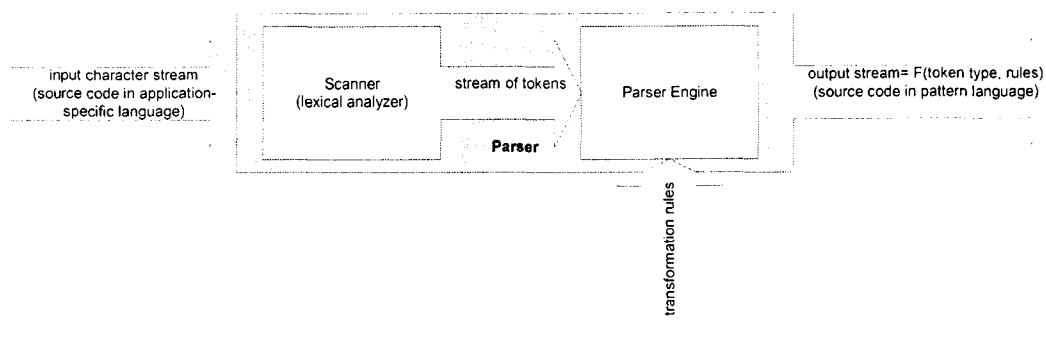
Pre-processing is a text-to-text transformation with the purpose of preserving high-level (clones’) information from low level code by filtering out irrelevant detail.

The pre-processor is a set of transformations specified using rules. It converts a program written in an application-specific language into a program written in more general-purpose pattern language by applying these rules. The pattern language is defined as a set of symbols that can be used as substitutes for lexical entities in the programming language and, therefore, is an extension of the application-specific language. Tokenizing on a lexical level produces a sequence of tokens; each token type has a rule associated

---

<sup>10</sup> All identified clones are presented to the user for manual examination via the CloneMaster interface. At this point, it is up to the user whether to accept or to reject a match.

with it; this rule determines the kind of action to be performed upon that token. The mechanism of pre-processing is summarized schematically in Figure 4.2.



**Figure 4.2:** Diagram illustrating mechanism of pre-processing.

Although one might argue for superiority of the syntactic approach, the lexical approach was chosen because of its simplicity and sufficiency. The advantage of simplicity is important, particularly in multilanguage systems: constructing separate lexical analyzers would require significantly less effort than constructing separate parsers, especially if dealing with nonstandard language extensions or proprietary languages (quite common in legacy systems).

For the purpose of this thesis, C++ is used as a target (application-specific) language. From this point on, the discussion is language specific. Nevertheless, the underlying ideas remain language independent and could be extended to accommodate any language.

In languages with textual-level preprocessors (e.g., C, C++, PL/1), unprocessed source code should be considered for clone analysis to prevent possible loss of structure (i.e., manifest constants, inline functions, sharing of inclusions) due to expansion of macros and other inclusions. Besides, expansion of preprocessing directives causes the code to grow in size that adversely impacts performance of the clone identification algorithm.

The following set of rules is used to govern the pre-processing transformation:

Rule #	Token Type	Token Code	Action
1.	inline comment	6	discard
2.	C-style comment	7	discard
3.	blank	9	discard
4.	string literal	10	output 'L'
5.*	decimal constant	4	output 'INT'
6.*	real number	5	output 'FLOAT'
7*.	octal/hexadecimal constant (zero remains zero)	63	output 'OX'
8.	character constant	11	output 'C'
9.	identifier	1	output 'I'
10.	preprocessor directive	8	discard
11.	#if 0 block	57	discard

Rule #	Token Type	Token Code	Action
12.	escape	66	discard
13.	continuation sequence	69	discard
14.	EOF	64	discard
15.*	decimal constant or real number or octal/hexadecimal constant	4 or 5 or 63	output 'N'
16.	direct component selector ‘.’	34	output ‘^’
17.	indirect component selector ‘→’	38	output ‘^’
18.	any other type		output ‘as is’

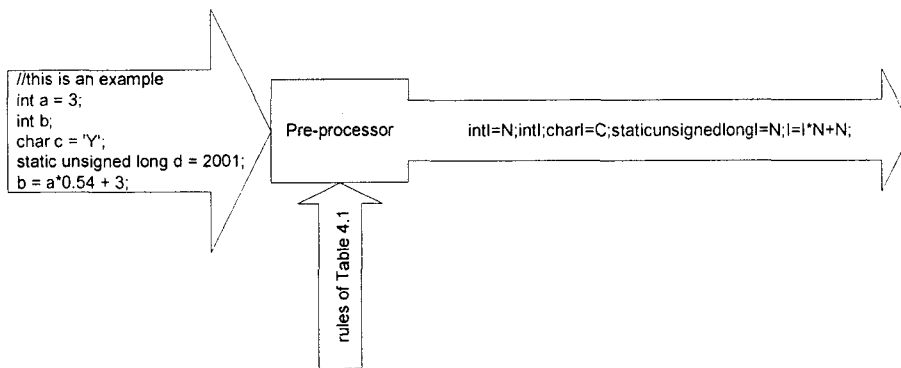
**Table 4.1:** C++ tokens with associated actions (full set of participating tokens is listed in Table A.1 Appendix A).

For instance, if a token is recognized as a comment or a blank (Table 4.1), it is ignored and won't appear in the output. If token is a string literal, its lexeme<sup>11</sup> is changed to 'L' and that is how it will appear in the output stream. If a token is of a type that has the 'output as is' rule associated with it (i.e., keyword), it is relayed from the input stream to the output stream with intact lexeme. However, for case insensitive languages (i.e., VB) 'output as is' rule would have an additional step of formatting the token lexeme using letters of the same case (uppercasing or lowercasing).

\* Rules 5, 6, 7 and rule 15 are mutually exclusive. When rule 15 is enabled, rules 5, 6, and 7 must be disabled.

<sup>11</sup> Lexical analysis converts strings in a language into a list of tokens. Each token has a type and a lexeme. The tokens are then passed to the parser for syntactic analysis. For example, 'char': token type is 'keyword', token lexeme is "char"

A more concrete example of one possible pre-processing transformation is depicted in Figure 4.3. This example also illustrates the compacting effect of the pre-processing transformation that positively affects performance of the clone detection algorithm.



**Figure 4.3:** Example of pre-processing transformation

The effect of applying transformation rules is a lot like parameterization used by Baker et al. [Baker 1992]. However, it is broader than just parameterization, and is best referred to as ‘unification’. In chapter 2, the term ‘software clone’ is defined as “a copy of existing piece of code that underwent (possibly empty) modification”. Consequently, the purpose of unification is to somewhat reverse this potential modification by representing the code via a unified pattern language, thus neutralizing the impact of the original modification.

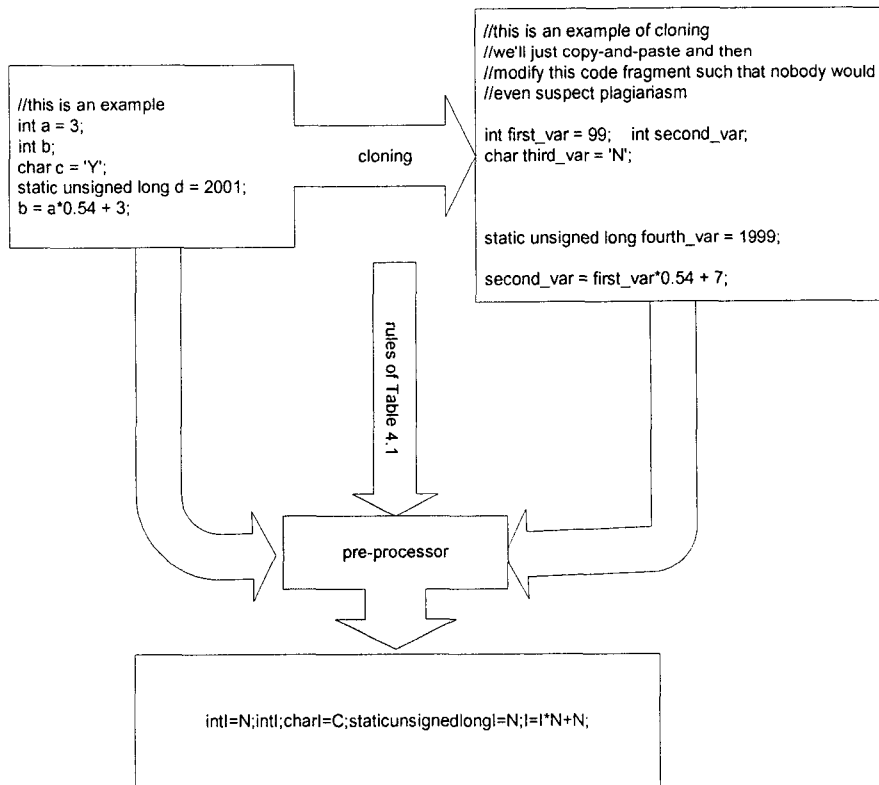
Figure 4.4 further illustrates the unifying action of pre-processing. The majority of changes generally associated with cut-and-pasting (i.e., modification of identifier



names/constants/numbers, addition/removal of comments, reorganization of source code page layout via formatting) can be adequately addressed by pre-processing based on lexical analysis. However, there exist other possible modifications (interchanging of commutative operands in arithmetic expressions, rearrangement of the sequence of statements) that may require more sophisticated (syntax) analysis. These modifications are quite unlikely to occur [Baker 1992][Kontogiannis 1996] and are beyond the scope of the current work.

Since all analysis takes place on the lexical level, no information on name scope resolution is preserved. One might argue that such loss of information could potentially lead to an increased probability of false positives. However in practice, type information (that is being preserved) could be used to reduce detection of accidental clones, as many such clones use different types in their computation.

Our current implementation does not distinguish between library identifiers and user defined identifiers. Dealing with the source before the preprocessor has run makes it impossible to trace nested library includes. Besides, the benefit of having library name resolution is unclear.



**Figure 4.4:** Unifying action of pre-processing

The pre-processing transformation is implemented via a parser<sup>12</sup> (Figure 4.2). This parser is able to extract lexical information from C++ files before the pre-compiler has run and to perform a set of predetermined actions as discussed earlier. The parser takes in a C++ source/header file, parses it, converts extracted information into intermediate format (pattern language), and spits it out to a text file. The parser consists of two integral parts – a scanner (lexical analyzer) and a parser engine. The scanner is responsible for reading the input characters and grouping them into lexical tokens. The parser engine's job is to handle these tokens and output them to a file.

<sup>12</sup> The term 'parser' here is used in its generic meaning and doesn't imply building a parse tree.

The parser is designed to be robust in a sense that it does not give up when it encounters constructs it can't parse; rather it proceeds, building an 'error' token until it finds a construct that makes sense (a well-formed token).

### ***4.3.1 Scanner Design***

In order to accommodate a wide variety of legacy systems, a superset of the three most popular C++ implementations (MS VC++, Borland Turbo C++, and GNU gcc 2.4 C++) was considered, as opposed to any single C++ implementation [Gcc] [Ellis 1990] [MSDN 2000].

Although standard tools (i.e., Lex or Flex) could've been successfully used to generate a C++ lexical analyzer [Aho 1986], it was chosen to build one from scratch. The constructed scanner is based on the algorithm proposed by DeDourek et al. in [DeDourek 1980] for it is conceptually simple, easily extendible and lends itself well to implementation.

The adopted approach is table driven and follows a 'one character look-ahead' scheme. It consists of the following four steps each of which is further elaborated on in subsequent sections:

1. Identification of a list of tokens to be recognized;
2. Construction of state diagram for these tokens;
3. Use state diagrams from the previous step to build transition (driver) tables;

4. Implement the scanning algorithm.

#### ***4.3.1.1 The Token Set***

The complete set of tokens selected for recognition is listed in Table A.1 in Appendix A. It is substantially wider than a conventional C++ token set and was influenced by the following factors:

- C++ language specifics;
- Considerations of clone identification;
- Extendibility provisions;
- Implementation specifics.

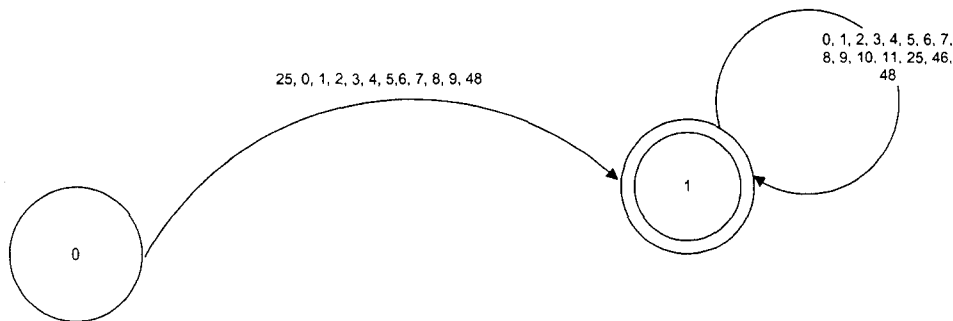
Definitions of identified tokens are listed in Table A.2.

#### ***4.3.1.2 State Diagram Construction***

In this step, state (transition) diagrams to represent tokens are constructed. A state diagram consists of a set of nodes called states represented by circles, and a set of directed edges joining these states. Each edge is labeled with character class names. It is required that this state diagram be deterministic (i.e., no more than one edge leaving a given state is labeled with the same character class).

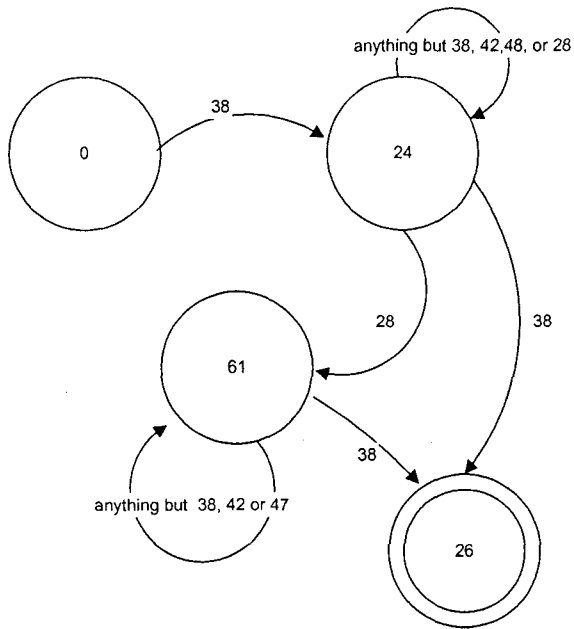
Character classes are defined by partitioning the set of all valid C++ characters such that every character belongs to exactly one class. The partitioning is also based on tokens defined in order to facilitate their recognition. Identified character classes are listed in Table A.3.

One state is designated as the initial state (state 0) where all recognition initiates. The directed edge indicates the flow. If one is currently in state  $s$  and there is an edge labeled with character class  $k$  joining it to state  $t$ , one moves from state  $s$  to  $t$  if the next character read is a member of character class  $k$ . Some states (depicted by double circles) are ‘final’ states that represent possible identification of a token. A string of characters is a token of a given type if and only if there is a path from state 0 to a final state for that token type. As the source code is parsed, tokens are extracted in a way that the longest possible token from the character sequence is selected.



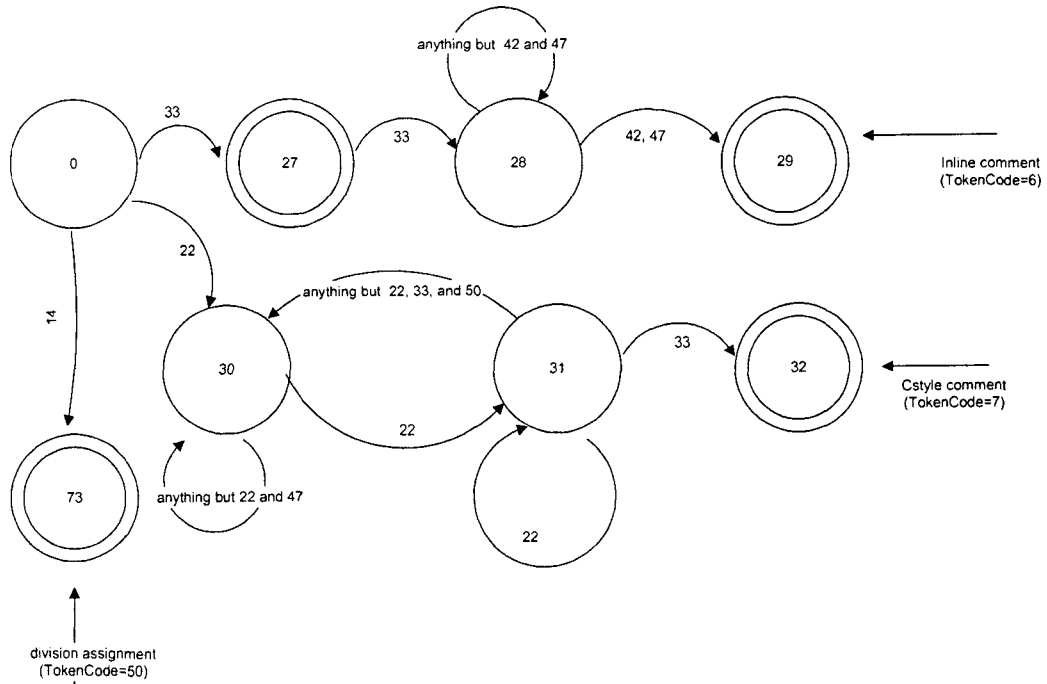
**Figure 4.5:** Transition diagram for ‘identifier’ token (TokenCode = 1). According to its definition (Table A.2), identifier can start with a letter [a-zA-Z], an underscore or a \$ followed by any number of letters, underscores, or \$.

First, for each token identified in step 1, a state diagram needed to be built. Some simple cases of these state diagrams are presented in Figures 4.5, 4.6 for illustration purposes. These ‘individual token’ state diagrams are then merged to form a combined state diagram of the entire token set. This combined state diagram made use of over a 100 states. A fragment of it is presented in Figure 4.7.



**Figure 4.6:** Transition diagram of a character constant token (TokenCode = 11) defined as one or more characters enclosed in single quotes (Table A.2).

Once a state diagram encompassing all identified tokens has been constructed, it has to be converted into ‘computer friendly’ form. In DeDourek’s et al. approach, this is accomplished by assembling two lookup tables: the NEXT STATE table and the OUTPUT table [DeDourek 1980].



**Figure 4.7:** A fragment of state diagram that facilitates recognition of the following token types: ‘C-style comment’ (TokenCode=7), ‘inline comment’ (TokenCode=6), and ‘division assignment’ (TokenCode=50).

#### 4.3.1.3 Transition Tables

When fully constructed, both the NEXT STATE table and the OUTPUT table contain one row per each state (plus ‘EOF’ and ‘Error’ rows) and one column per each character class. In the NEXT STATE table, a reading from row  $s$  column  $c$  intersection indicates which state  $t$  to transition to next (hence the name) if we’re currently in state  $s$  and a character of character class  $c$  has been read. The corresponding entries in the OUTPUT table indicate whether or not character just read should be included in the token currently being formed. Zero entries indicate that the token is still under construction. Non-zero values, on the other hand, mean two things:

1. The character just read is the beginning of a new token;

2. Token code of just recognized token.

#### ***4.3.1.4 Token Recognition Procedure***

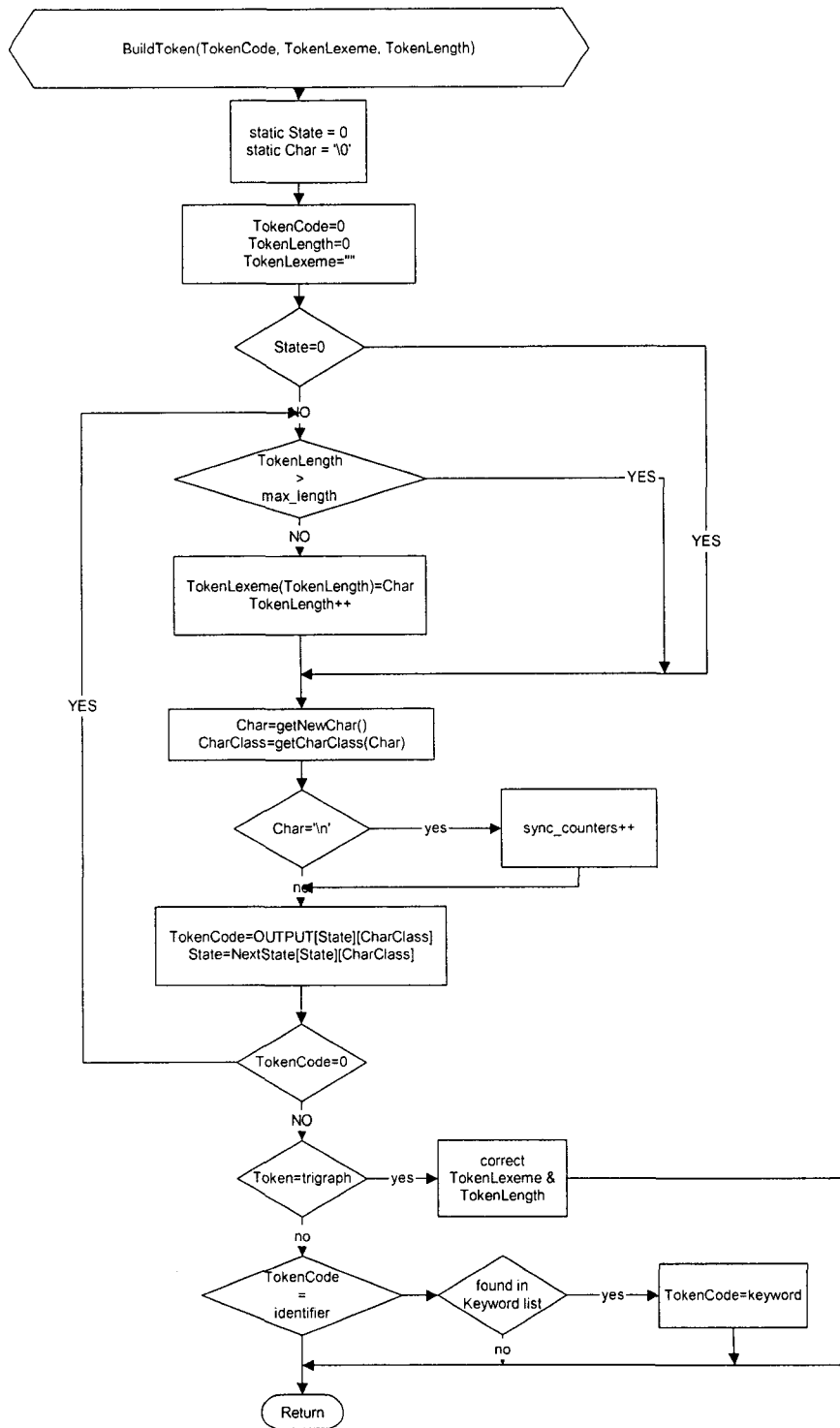
An algorithm used to convert an input stream of characters into a stream of C++ tokens is presented in Figure 4.8. Being an extension of the original algorithm described by DeDourek et al, it utilizes tables constructed in the previous step to determine whether the character just read belongs to the token being built or indicates the beginning of a new token. Additions to the algorithm were necessary to accommodate trigraphs<sup>13</sup> and to track occurrences of new line character.

Handling trigraphs can be seen as another example of unifying action of pre-processing: When a trigraph is encountered, it is recognized as the punctuation mark it stands for, and is replaced with a single character ( '[' for '??(', '~' for '??-', etc.). Conversion of trigraphs was intentionally built into the token recognition routine as opposed to accommodation via yet another transformation rule.

---

<sup>13</sup> Trigraphs are sequences of three characters (beginning with two consecutive question marks) that are used to represent certain punctuation characters in C/C++ source files with character sets that don't contain graphic representation for them. Trigraph sequences allow C/C++ programs to be written using only the ISO Invariant Code Set that is a subset of the 7-bit ASCII character set.





**Figure 4.8:** Token recognition algorithm

### *4.3.2 The Parser Engine Design*

The parser engine part of the pre-processor (Figure 4.2) is an agent that enforces transformation rules defined in section 4.3. It is made configurable, such that a (sub)set of rules to be used can be easily controlled by the user. Such flexibility allows the user to experiment with various combinations of transformation rules and helps to make the clone detection process as precise or imprecise as the user wants it to be since various types of approximate matching can be accommodated by discarding different parts of the input.

Rules 3, 12, 13, and 14 (see Table 4.1 for details) are made unconditional. Other rules can be turned on and off upon user's discretion. In cases when rules 1, 2, 10, or 11 are turned off, lexemes of corresponding tokens will be preserved after extraction of all occurrences of new line character, blanks, and/or escape sequences within these tokens.

The error token deserves special attention. Currently, there is no transformation rule defined for it and, therefore, it is preserved. Under the assumption that syntactically correct code is being parsed (a pretty safe assumption in case of legacy systems), errors should never occur. However, error tokens could be caused by oversights during construction of state diagrams or unaccounted transitions, therefore they should be preserved.

### ***4.3.3 Implementation of the Pre-processing***

Implementation of the pre-processing process was developed using object-oriented C++ on a UNIX workstation (sun4u sparc SunOS 5.6). The resulting system consists of the main module implementing the scanner/parser design discussed above and a number of supporting modules described later in this section. All modules support command line mode as the only available interface.

#### ***4.3.3.1 The Parser Module***

The parser module has the following parameters:

- Input/output buffer size. For efficiency reasons, the parser uses buffered input/output.
- Maximum number of characters allowed in a token. This parameter limits the number of characters that are being remembered for each token and does not affect token recognition process.
- List of C/C++ source and header files to be parsed. Details on creation of this list are covered in section 4.3.4.2.
- Input directory – a path to a directory that is at the root of the source tree.
- Output directory – a path to a directory where output of the pre-processing is stored. The output directory mirrors the hierarchy of the input directory with the only exception that the processed files are marked with an additional suffix '.U'. For example, output corresponding to 'INPUT\_DIR/my\_dir/fileX.C' will be in 'OUTPUT\_DIR/my\_dir/fileX.C.U'.

- Mapping directory – a path to a directory where supplementary information needed for mapping results of clone identification on the pre-processed code back to the original code is saved. The mapping directory mirrors the hierarchy of the input directory with the only exception that the files are marked with an additional suffix ‘.pos’. For example, a file ‘INPUT\_DIR/my\_dir/fileX.C’ will produce the following entry ‘MAPPING\_DIR/my\_dir/fileX.C.pos’.
- Rule configuration parameters. Allow the user to specify transformation rules to be enabled for a pre-processing session.

#### ***4.3.3.2 Discovery of Directory Structure***

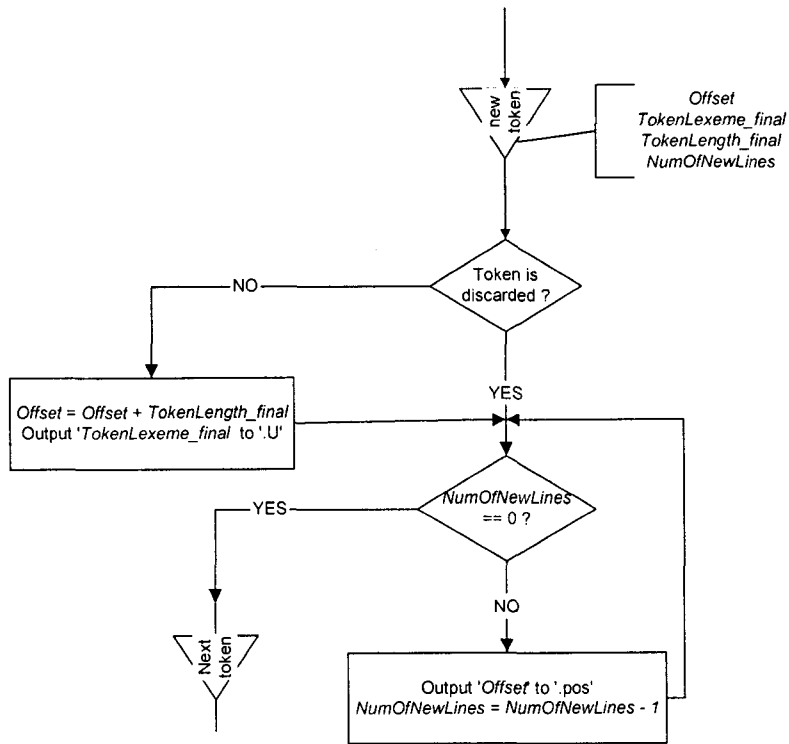
The current implementation of SelArt does not allow for the selection of files for analysis. Consequently, all files found in the specified directory tree are analyzed. Thus, for efficiency reasons, any files that are not of interest from the clone finding perspective have to be removed from the directory tree prior to analysis.

This thesis is concerned with C++ source files only; hence, all supplementary files (i.e., different configuration files, batch files and scripts, makefiles, .html files, .asm files, .def files, .rc files, binary files, .txt files, etc.) are to be ignored. To facilitate this, a command line utility has been created that recursively traverses the specified directory tree and inspects file extensions for each file found. If the extension indicates that the file is indeed a source file (.h, .C, .c, .cc, .cxx, .C++, .cpp), its path is recorded; otherwise, the file is deleted. This list of recorded paths is then used to communicate to the pre-processor the files to be parsed.

#### *4.3.3.3 Preservation of the Line Structure*

One of the major challenges faced during implementation of pre-processing was to come up with a mechanism that would allow relating character offset information in pre-processed code to the line structure of the original (non pre-processed) code (refer to section 4.5.3 for more detail). The solution found is both simple and accurate.

Information on original line structure is preserved via maintaining auxiliary '.pos' files in the MAPPING\_DIR. A '.pos' file contains data (offsets in terms of number of characters from the beginning of the corresponding '.U' file) on where new line characters would have been in the pre-processed source. The mechanism shown in Figure 4.9 is used; For an example of a '.pos' file refer to Figure 4.10.



**Figure 4.9:** Line tracking algorithm. ‘Offset’ - current number of characters in the output (i.e., ‘.U’) file; ‘TokenLexeme\_final’ – lexeme to be output; ‘TokenLength\_final’ – number of characters in TokenLexeme\_final; ‘NumOfNewLines’ – number of new lines originally contained within the token.

#### 4.3.3.4 Supporting Statistics

To facilitate comparability of clone detection results between non-pre-processed and pre-processed source code, a pair of complementary statistics is collected: average number of characters per line (ANCPL) and compression rate (CR). Both statistics are self-explanatory (defined by formulas below) and are used in calculating of SelArt’s configuration parameters (section 4.4.2).

$$ANCPL = \frac{CHARS\_BEFORE}{LINES} \quad (4.1)$$

$$CR = \frac{CHARS\_BEFORE}{CHARS\_AFTER} \quad (4.2) \quad , \text{ where}$$

*LINES* – total number of lines in the system.

*CHARS\_BEFORE* – number of characters before pre-processing

*CHARS\_AFTER* – number of characters after pre-processing.

#### ***4.4 Step 2: Clone identification with SelArt***

As was mentioned in section 2.5.5, SelArt supports both line oriented and fixed length (stream) modes. By default, it works in line mode and only falls back on fixed length mode when the lines are very long or there are no line-end characters. Clearly, systems subject to any degree of pre-processing will enforce stream mode behavior due to a lack of line-end characters.

##### ***4.4.1 Line-Oriented to Stream Input Conversion***

To eliminate the possible impact of the mode variant on the results of clone detection between pre-processed and original source code, a simple line-end elimination operation is used. If no pre-processing is to take place, before invoking SelArt, all files in the

INPUT\_DIR are scanned for new-line characters that are discarded. At the same time, '.pos' files are generated in the MAPPING\_DIR in a manner very similar to the one described in section 4.3.3.3.

#### 4.4.2 SelArt Parameters

Section 2.5.5 described four parameters that SelArt provides to give the user some degree of control over the matching process. In the context of stream mode operation, their purpose is explained below:

- $l$  - target clone size in number of lines. It is used in calculation of other parameters ( $M, m$ ) since size of clones is more naturally to be expressed in terms of number of lines rather than number of characters
- $M$  – determines number of characters in a snip:  $M = \frac{ANCPL * l}{CR}$  (4.3). ANCPL is calculated by formula (4.1), and  $CR$  by formula (4.2). It is guaranteed that all matches of at least size  $M$  will be found.
- $m - m = M$
- $c$  – to eliminate possible inaccuracy introduced by culling, it is turned off by setting  $c = M$ .
- *target path* – absolute path to a directory node; all files residing in this node's subdirectories will be recursively analyzed.

Although the meaning of these parameters is discussed elsewhere (section 2.5.5), calculation of parameter  $M$  needs some further explanation. In line-oriented mode, the



parameter that determines the target clone size is  $l$ . In stream mode, however, that's  $M$ . In order to be able to compare results of clone analysis of the same source code with different pre-processing transformations, it is important to ensure that parameter  $M$  adequately reflects the compacting effect of such transformations. Formula 4.3 is a simplistic but adequate way to achieve this.

### ***4.5 Step 3: Post-processing of Clone Identification Results***

#### ***4.5.1 Original Result Presentation***

SelArt partitions the source input into disjoint snips constructed in such a way that all the information about identified matches is represented as matches on these disjoint snips (SelArt's combine-and-split strategy is discussed in section 2.5.5). This information is organized as a set of records (one per snip) in a flat ASCII file (grpl.l) suitable for automated processing. Each record contains the following fields of interest:

- *Snip Number*. Snips are numbered in the order in which they are found in files.
- *Beginning Offset*. This field indicates the beginning of the snip as a character offset within the file.
- *Ending Offset*. This field indicates the end of the snip as a character offset.
- *Hash Value*. Snips with matching hash values have the same contents.
- *File Number*. This is a number sequentially assigned to each file seen.
- *File Name*. This is a full (absolute) path name of the file.

### ***4.5.2 Filtering Information for Future Analysis***

Although only matches of some size  $M$  and greater are requested, the result file could contain a substantial amount of considerably smaller matches (by-products of SelArt's combine-and-split strategy). Some of them are meaningful matches; some of them are noise.

To give users control over the amount of information preserved for future analysis, a notion of noise threshold is used by the post-processor. Simply put, only clones with sizes equal or greater than the noise threshold are kept during post-processing.

### ***4.5.3 Conversion of Clone Boundaries***

The stream mode of clone identification, especially when preceded by pre-processing, necessitates conversion of information on clone boundaries from the character offset representation in the pre-processed source code back into the line number representation in the original source code. This step is very important because clone boundaries reported with respect to the pre-processed source coordinate system are meaningless in the original source coordinate system. The conversion is a mapping between the two coordinate systems and is achieved by interpreting the contents of corresponding '.pos' files produced during pre-processing under the MAPPING\_DIR directory tree (refer to section 4.3.3.3 for the algorithm used). The *Beginning/Ending Line Number* is

determined by simply counting the number of entries in the '.pos' file with values less than the *Beginning/Ending Offset* (refer to Figure 4.10 for more explanation).

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	19	35	52	69	86	86	104	134	157
185	208	228	254	287	287	287	287	287	287	287	287	287	287	287	287	287	287	287	287	287	323	323	340	340	
382	382	382	382	382	382	382	382	382	382	382	402	420	420	421	454	455	469	470	499						
517	611	616	697	703	721	791	826	827	827	902	926	957	981	1037	1038	1069									
1117	1117	1117	1117	1117	1189	1221	1222	1237	1238	1252	1253	1253	1253	1253	1253										
1273	1325	1326	1401	1424	1439	1483	1497	1498	1498	1518	1594	1595	1595	1670	1693										
1708	1773	1774	1774	1794	1873	1874	1949	1972	1987	2055	2056	2056	2076	2152											
2153	2228	2251	2266	2331	2332	2332	2352	2429	2430	2505	2528	2543	2609	2610											

**Figure 4.10:** An example of a '.pos' file. Each entry indicates a position of a discarded new line character expressed in terms of character offset within the pre-processed source ('.U' files). This file can be read as follows: the first 17 lines of the original source were discarded, 18<sup>th</sup> line was compressed to 19 characters, 20<sup>th</sup> line was compressed to 35 – 19 = 16 characters, ..., 23<sup>d</sup> line was discarded, etc. Conversely, to convert clone boundaries expressed via character offsets to line numbers, just count the number of entries with values less than the corresponding offset. This count is in fact the line number. For instance, character offset of 144 translates into line number 25.

In cases of aggressive pre-processing, some clone boundaries may be miss-reported (at most 2 line miss). Another case of information loss occurs when clones begin/end mid-line. However, these are acceptable in the context of the clone analysis process. Overall, the mechanism showed to be reliable.

#### 4.5.4 Implementation Details

Post-processing is implemented via a simple utility that sifts through the clone identification result file (grpl.l) seeking records corresponding to clones of sizes exceeding some threshold (controlled via a parameter). These records are retained in an intermediate result file of the following format:

- *Clone Number*. Clones are numbered in the order in which they are encountered in the grpl.l file.
- *Beginning Line Number*. Indicates the first line of the clone (converted from character offset).
- *Ending Line Number*. Indicates the last line of the clone (converted from character offset).
- *Hash Value*. Same as in grpl.l file.
- *File Number*. Same as in grpl.l file.
- *Relative File Path*. This is a path to the file relative to the target directory of SelArt's analysis (converted from absolute path).

To facilitate the use of the acquired clone data in subsequent phases of clone analysis (i.e., clone data presentation and clone data interpretation), it makes sense to store the data in a relational database rather than a flat file. Delegating data management to a Database Management System (DBMS) delivers many benefits. The most important of which in the context of this work, is provision of infrastructure for convenient and efficient interaction with the data (retrieving, querying, updating, inserting, or deleting), as well as enforcement of data integrity and elimination of data redundancy. The database design issues are discussed in the next chapter within the context of the clone visualization tool – CloneMaster.

## *4.6 Closing Remarks*

This chapter describes a method for extending SelArt into a tool that can be used to find near, non exact, clones. The strength of the above solution is its simplicity, flexibility, and extensibility. The degree of clone similarity required for a match is easily controlled through configuration. New transformation rules can be defined and easily accommodated based upon user's needs. Currently, only one programming language (i.e., C++) is supported. However, support for new languages can be easily added.

To fulfill the requirements of this thesis, the implementation is based on one particular implementation of a text-based comparison technique, SelArt. Nevertheless, the proposed approach is generic and will work for any representative of the text-based comparison technique.

A high degree of confidence in the effectiveness of the approach and the implemented solution has been gained through extensive experimentation.

Results of the clone identification process are delivered in a form of a textual summary. These summaries could be hard to work with due the abstract nature of the information they contain and the sheer quantity of this information. Issues of efficient clone data presentation are the subject of the next chapter. Chapter 6 provides some experimental analysis of the nature of clones and their occurrences in an industrial sized software system. It also attempts to evaluate, from the perspective of software understanding and

maintenance, some potential benefits of extending the clone identification process to accommodate near clones.

## ***Chapter 5 – Clone Visualization Prototype: CloneMaster***

Many publications propose various ways of identifying cloned components in a software system [Baker 1992][Johnson 1993][Mayrand 1996][Baxter 1998]. However, it is still not clear how a clone detection technology can be applied in an industrial software development process in order to achieve significant savings. This thesis attempts to address the issue by investigating the feasibility of a visual tool that not only delivers clone data to the user effectively but also provides a powerful means of interacting with that data in order to facilitate various software engineering and maintenance tasks.

This chapter introduces CloneMaster, - a clone visualization tool developed as a core part of this thesis work. This chapter briefly explains motivation behind the tool, discusses some related work, defines a set of requirements for the tool, and then proceeds with detailed description of some major design and implementation issues.

### ***5.1 Motivation***

The problem of understanding, navigating through and manipulating complex information spaces is now arising in a wide range of application areas, and it is becoming increasingly important to provide tools that offer sophisticated support to users with these tasks.

In Software Development and Software Maintenance, for example, the systems tend to be large and complex and normally involve a large number of programmers, who, besides maintaining an overall understanding of the system they are working with, are often interested in acquiring more detailed knowledge of some particular aspects of that system. Hence, depending on the task, the system must be analyzed from different perspectives (e.g., control flow, data flow, class structure, memory allocation, input/output, profile information, etc.). To assist software engineers with some of these tasks, various visual tools have been developed (e.g., class browsers, animators, integrated development environments, software visualization tools, etc.). However, these tools are usually of a quite general nature and, thus, fail to adequately support working on particular aspects of program understanding. Any solution that tries to be all things to all users tends to be a poor solution to any one of them plus to be intolerably complex. Therefore, the emphasis should be on being extremely clear about the goals and building specific tools for specific jobs. If desired, these tools can be then bundled together to form multifunctional tool suites. Alternatively, they can be implemented as plug-ins to be used from other applications.

Many cognitive models<sup>14</sup> have been proposed that describe how programmers comprehend code during software maintenance and evolution [Mayrhauser 1995][Storey 2000]. It has been also generally accepted that the comprehension strategy employed depends on a variety of factors dictated by the maintainer, the software system and the task. Therefore, it would be favorable for a tool to support a wide array of



comprehension strategies. Minimally, such a tool must help maintainers with the key activities [Tilley 1996] inherent to any comprehension process:

- Data gathering through static analysis of the code or through dynamic analysis of the executing program.
- Knowledge organization by organizing the raw data by creating abstractions for efficient storage and retrieval.
- Information exploration through navigation, analysis, and presentation.

Ideally, a tool should provide the user with some sort of task oriented ‘information workspace’ [Card 1999]. Centered on one or more visual components (visualizations), such an ‘information workspace’ combines the presentation of carefully selected and organized information related to the task with some effective mechanisms for access, retrieval, and manipulation of this information to facilitate knowledge crystallization.

## ***5.2 Related Work***

Although software clones have attracted a fair amount of attention in the recent years, very few user-friendly tools exist to support their analysis. Existing systems focus primarily on issues related to clone localization, while delivering insufficient support for other aspects of clone management.

---

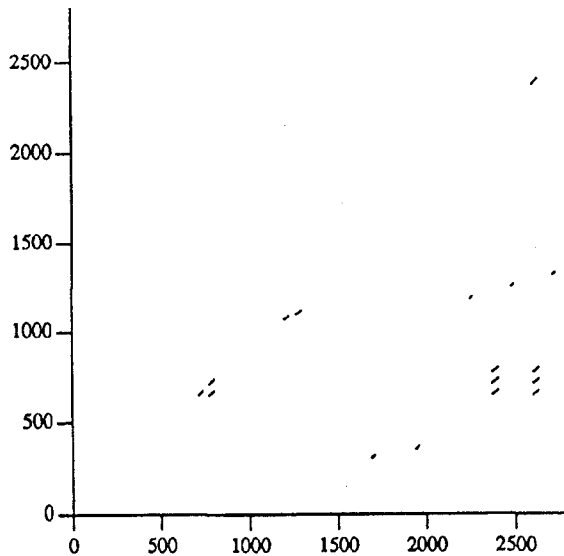
<sup>14</sup> A cognitive model is a program understanding strategy that uses existing knowledge together with the code and documentation to create a mental representation of the program (i.e., Bottom-Up, Top-Down,

Traditionally, results of clone identification are presented via textual summaries, which, due to information overload, tend to be overwhelming, difficult to interpret, and therefore inefficient in solving real world user problems. Yet, some attempts have been made towards exploring other, mainly graphical, means of presentation of such information. Baker [Baker 1992], for example, used simple scatter plots (Figure 5.1) to visually show distribution of clones in a software module. The plots used identical horizontal and vertical axes to depict lines of source code of the module (referenced by their ordinal numbers). Each dot on the plot corresponded to a line of code that had been encountered in the module twice. The corresponding reading off the horizontal axis identified the location of its first occurrence, whereas the reading off the vertical axis gave the location of its clone. Accordingly, matches spanning more than one consecutive line appeared on the plot as line segments, with the length proportional to the size of the match. Exact matches produced line segments parallel to the main diagonal of the plot (a  $45^\circ$  axis); line segments corresponding to approximate matches might not be strictly diagonal due to possible differences in sizes of the originals and their corresponding duplicates<sup>15</sup>.

---

Knowledge-based, Systematic and As-Needed, etc.) [Mayrhauser 1995]

<sup>15</sup> These differences occur because some irrelevant details (i.e., white space, comments, etc.) are ignored during the matching stage, while line numbering remains unaffected by these omissions (i.e., line numbers are the original line numbers in the module).



**Figure 5.1:** A scatter plot generated by Baker to visualize clone occurrences in a file. The axes depict line count; clones are represented via line segments

Baker's graphical representation of the clone data is straightforward, easily interpreted, and cheap to generate. She was able to combine both the 'big' picture of clone occurrences in a software module (number of clones, their size, and distribution within the module) and the detailed picture (right from the chart it is possible to deduce which segments of code has been replicated, how many times, and where the duplicates are located) in the same view. One apparent limitation of her method, however, is its poor scalability. Since the size of the plot is directly proportional to the module's size, large modules produce plots that are difficult to read due to their enormous physical size and information density. The number of plots generated depends upon the number of modules comprising the system. As the number of modules increases, referencing becomes an issue. Other noteworthy weaknesses of Baker's approach include lack of support for source code browsing, navigation, and interactivity.

Nonetheless, visualization proposed by Baker was an important event in clone data presentation for it explored, for the first time, the potential of using visual media to communicate what was traditionally considered textual data.

A similar scatter plot based approach to clone visualization is discussed by Ducasse et al. [Ducasse 1999] who report successful results of visualizing occurrences of exact clones both within the same file and between several files. They claim that generated visual representations are useful for practical software maintenance and re-engineering tasks. Poor scalability and high degree of visual redundancy are identified among the major drawbacks of the system.

Johnson [Johnson 1996] describes “visualization and navigation of textual redundancy based on the technology of HTML and the World Wide Web”. He suggests representing the match data via a network of entities and links, with links based on different relationships between the entities. He uses six basic entity types, such as *file*, *directory*, *snip*, *hash*, *cluster*<sup>16</sup>, and *component*.

- *File* and *directory* entities are self-explanatory.
- A *snip* is defined as a sequence of characters that occur in a file. Snips associated with a file partition that file; that is, the file is a concatenation of the snips it contains.
- *Hash* is a numerical value characterizing a snip. Two snips are said to match if they have the same hash value.

- A *cluster* is a set of files that share a number of matching snips. Each file belongs to at least one cluster (singleton cluster is a cluster that contains a single file; each file has a singleton cluster associated with it).
- A *component* entity is associated with another way to partition the set of files.

Some possible relationships between these entities include: *file* or *directory* to parent *directory*; *snip* to the *file* containing it; *snip* to *hash* value; *file* to a cluster containing it; *hash* to the *cluster* implied by its match set; *cluster* to the *component* containing it; *cluster* to *cluster* containment relation; etc. Each entity possesses a certain attribute (key) through which it can be accessed (i.e., File\_ID, Snip\_ID, Hash\_ID, etc). Actual values of these keys are generated for each instance of the entity during the analysis stage.

The layout of Johnson's tool is as follows: Each instance of the above entities is mapped to an individual HTML page, whereas the relationships among them are implemented via various hypertext links between these pages. Figure 5.2 shows the page for cluster 1948, a typical *multiple-file cluster* page. 'Previous' and 'Next' links point to clusters 1947 and 1949 respectively. 'Files' section enumerates all the files the cluster contains (*cp/parse.c*, *cp/parse.h*, *c-parse.c*, *c-parse.h*, and *objc-parse.c*) and provides links to corresponding *file* pages. 'Superclusters' and 'subclusters' sections provide linkage to all upper and lower level clusters that involve files constituting the cluster #1948 (i.e., sub-clusters increase the size of the matches by reducing the set of files; super-clusters do the opposite). Under the stippled line, the content of the matching snip

---

<sup>16</sup> Johnson's usage of the term cluster is different from the one used throughout this thesis (consult

(ten lines of #defines) for this cluster is shown with a link to the corresponding *hash* page.

Other pages (i.e., *directory* page, *file* page, *hash* page, *snip* page, and *component* page) exhibit analogous structure. For example, a *file* page consists of a list of all snips constituting that file. For each snip, links to corresponding *hash* and *snip* pages are provided together with links to all cluster pages for which there is a match involving this snip.

Organizing information hierarchically allowed Johnson to achieve some degree of information hiding: Upper level pages, such as *component* and *multi-file cluster* pages, give just an overall breakdown of the match structure, while most detail becomes available via exploring lower level pages (*hash*, *file*, *singleton cluster*). Hiding unneeded detail reduces the information load on the user, while making that detail available on demand.

Johnson's approach, best described as text-based visualization, allows presenting large amounts of match data in a systematic and meaningful way. Unlike Baker's case, 'drill down' capabilities together with low level detailed views are supported; yet the global structure of clone proliferation is not explicitly communicated, forcing the user to form it himself by mentally integrating multiple local views (the problem of focus and

---

Chapter 2 for the appropriate definition). What Johnson refers to as a 'cluster', in the context of this thesis, corresponds to the term 'file span' of a clone.

context<sup>17</sup>). However as the complexity of the system grows, such integration quickly becomes virtually impossible. Another apparent deficiency of the Johnson's approach pertains to its navigation technique: It has been previously observed that following hyperlinks may cause disorientation resulting in loss of context, especially when a place of interest is several pages away. Moreover, the underlying data model is far from being intuitive. Some of its parts (*component* entity and all relationships pertaining to it, for example) exhibit complexity levels not appropriate for a generic user.

Although Johnson didn't use any graphics per se, his tool is an important milestone in the yet to be completed quest for effectively organization and presentation of clone data.

In conclusion, presenting clone information visually holds a lot of potential and practical benefits. Although substantial progress in the area of devising efficient visual displays has been made, clone visualization techniques and tools are yet to come of age. One of the objectives of the current research is to contribute to the field by exploring different approaches to presentation of clone information via devising a visual tool, CloneMaster, and evaluating its effectiveness. The rest of this chapter covers the design and implementation of the tool.

---

<sup>17</sup> In system visualization, a conflict between small-scale and large-scale structures is known as the problem of focus and context. Ideally, it should be possible to view details at arbitrary depth level (focus) without losing the high-level perspective of the system (context).

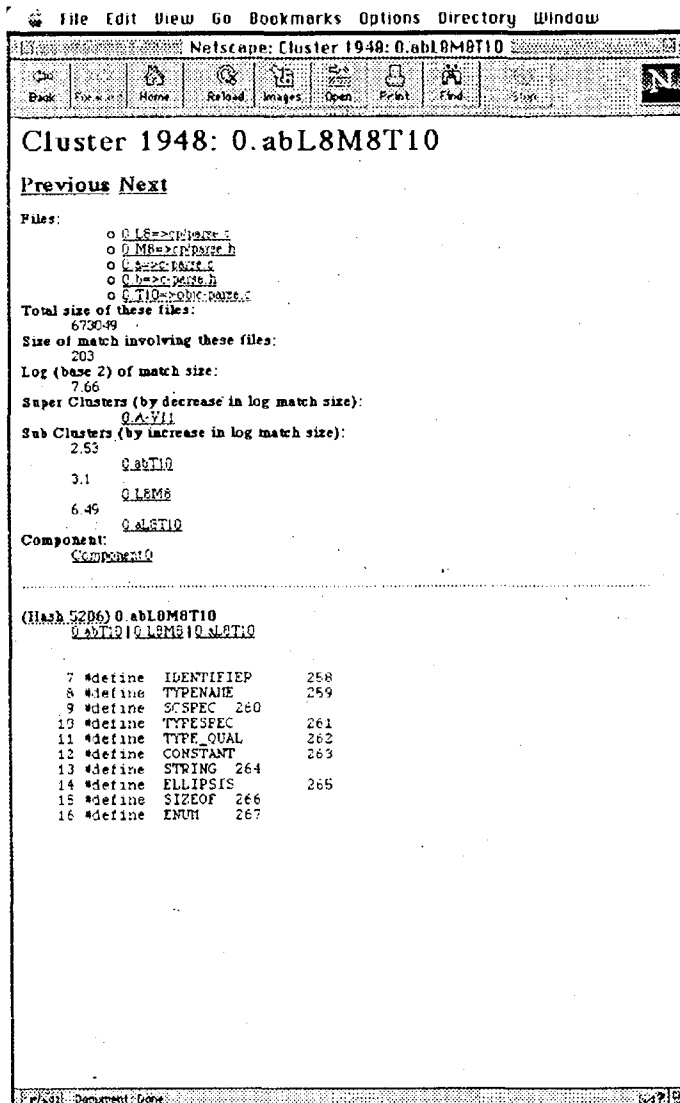


Figure 5.2: View of the Cluster #1948 Page

### 5.3 Formulating Requirements for the CloneMaster Tool

The ultimate goal of this research work is to ensure more reliable and more cost effective re-engineering and maintenance by providing software practitioners with a visual clone exploration environment that would facilitate the clone analysis and clone



management facets of the software development process. The scope of this tool should be navigation, analysis and presentation of the clone data.

Developing functional requirements for a clone visualization tool turned out to be a challenging task on its own. Due to the novelty of the topic of clone visualization and very early stages of development of the clone management industry in general, there has not been accumulated enough experience to allow one to scope out a precise set of requirements for a clone visualization tool that would be clear, sound, complete, and that would ensure a viable solution. Given these conditions, requirements for CloneMaster have been compiled based on pieces of knowledge garnered from multiple sources such as:

- Observations in the areas of clone management, software engineering, software maintenance, and software visualization.
- Knowledge gained from other related work.
- Survey of industrial software development and maintenance experiences.
- Application of best software engineering and maintenance practices.
- Author's experience in software development and maintenance.
- Common sense and imagination.

The following requirements have been identified for CloneMaster to satisfy:

1. Effective presentation of the overall clone information.

The tool's visual display should present the user with an easy to interpret visual image (map) of clone distribution within the system. The organization of the clone map should provide a clear picture of which pieces of the system's code have been cloned and where their cloned counterparts reside. Some form of overview of the entire system should be always kept available while pursuing detailed analysis of some of its parts to help the user stay in context (i.e., not to lose the high-level perspective of the system). It should be possible to identify both individual clones and patterns of clones.

2. Effective use of auxiliary data (e.g., statistical, source code view) to supplement the information pictorially conveyed via clone map.

It is desirable to provide the user with some additional statistics on each clone entity, each cluster entity, and the system itself. These data should be made available upon request, not to overload the display with excessive amount of information. However, access to the details should be easy, fast, and smoothly integrated into the overall context to avoid disruptive attention shifts.

Information available on any clone entity should encompass:

- Length
- file it resides in
- location within the file (starting position, ending position)
- cluster it belongs to
- cardinality

- contents of the clone

Information on a clone cluster should include:

- size
- enumeration of files associated with the cluster (file span of the cluster)

General statistics describing the system as a whole should comprise the following:

- size of the system (number of files, number of lines)
- min match length guaranteed to be found by the clone identification procedure (parameters  $M$ ,  $l$  specified at the invocation of SelArt)
- noise threshold ( 'Split-and-Combine' phase of SelArt can produce clones that are smaller than specified above. Noise threshold is used on the stage of post-processing of the clone results to eliminate noise)
- percentage of found duplications
- number of clones
- clones distribution by size
- number of clone clusters
- clone clusters distribution by size
- number of file clusters
- number of files affected by cloning
- percentage of clones occurred within the same file vs. percentage of clones occurred between different files
- file clusters distribution by size

The tool should support direct navigation to a clone's source code from any instance of clone display to allow the user to inspect its actual content. At any time, it should be possible to have as many open source code views as desired. These source code views should be displayed in separate windows to augment, not to occlude, the global picture of clone distribution in the clone map.

### 3. Analysis based on a file entity.

For any file in the system, it should be possible to determine how it is composed of clones and to track down other files that have matches with it (clone clusters, file cluster).

### 4. Analysis based on a clone entity

Upon choosing any clone entity on the clone map, it should be possible to:

- obtain detailed information about this particular instance
- visually identify in the context of the clone map other instances of the same clone entity
- view information about the clone cluster this clone is part of
- easily navigate between different clone instances in the cluster

### 5. Analysis based on a clone cluster entity

For a certain cluster, it should be possible to activate all of its clone members such that they stand out perceptually in the context of the clone map. Information on this cluster (see above) should be made accessible at this point.

## 6. Navigation issues (cross-reference browsing)

Some means of navigation should be provided to allow the user to maneuver in the clone hyperspace: follow certain logical links between different entities (i.e., files, clones, clone clusters) for the purposes of knowledge acquisition (e.g., discovering cluster structures, exploring file sets sharing certain clones, etc.) or performing some specific task (e.g., propagating a bug fix). Such navigation should make sense to the user conceptually, as well as it should be easy to perform technically. Moving around must not take too many steps, and the route to any target must be discoverable.

## 7. Query support and report generation

It is fundamental to enable dynamic querying capabilities against the context in order to facilitate information retrieval. The results of these queries should be presented in a form that makes best sense and is most effective (i.e., pictorially, as a textual summary, as a table, or an interactive histogram). The minimum set of queries to be supported is listed below with more useful query ideas to evolve in the course of practical application of the tool:

- clone size range search
- clone clusters size range search
- file clusters size range search

## **8. Using views**

Using different views of the same data should be considered where appropriate. Views can highlight pertinent data, show relevant data while hiding irrelevant information, sharpen the focus, clarify the issue, etc.

## **9. Supporting source code browsing and editing**

Since CloneMaster is intended as a maintenance tool, built-in source code browsing is an absolute must. Support for making (and saving) changes to the system's source code should be also supported.

## **10. Handling of false positive**

It is important to give a user an ability to judge for himself and therefore, reject (permanently) a clone instance if a case of false positive is suspected. In this situation, the user should be able to either mark that instance or permanently remove it from consideration.

## **11. Language independence**

CloneMaster is not a programming language centered tool; it, thus, should be highly generalized to deal with any system written in any language.

## **12. Scalability**

The technology should be capable of handling a wide range of software system sizes, especially in the medium to large range.

## **13. Other Requirements**

- overall ease of use
- pleasantness of use
- confidence in results generated

Usability of the tool is critical to its effectiveness. Poorly designed interfaces can induce extra cognitive overhead. Available functionality should be visible and relevant and should not impede the more cognitively challenging task of clone analysis. Where appropriate, meaningful orientation cues should be used (to indicate to the users where they are, how they got there, and where to go next).

## **14. Design Standards**

Use guidelines of the program visualization framework summarized in Chapter 3:

- effective presentation style
- suitable layout algorithms to display graphs in more meaningful manner.
- meaningful visual abstractions and attributes

#### **15. Target audience**

The tool caters towards software professionals with programming background, especially in software maintenance. Thus, strong understanding of basic computer science concepts is assumed. However, no specialized knowledge should be required.

#### **16. Physical Environment Constraints**

An average stand-alone workstation should be sufficient to run the tool. Since just a prototype is being attempted, the physical environment is constrained mostly by the resources available to the author (MS Windows 2000 Professional SP1, 192MB RAM, 18.6GB hard drive).

#### **17. Extensibility**

The tool must support the addition of new views.



CloneMaster is a proof of concept prototype system to explore ideas in the domains of clone visualization, cognition and management, not a full-blown commercial product. Although the list of requirements was developed to be as complete as possible, the intention of the prototype is only to demonstrate some key design and visualization concepts without necessarily attempting to meet all of the requirements.

#### ***5.4 CloneMaster Design and Implementation Highlights***

The following considerations are fundamental to the CloneMaster design:

- The user does not always know what he is looking for and, thus, he may not be able to search for something specific. Consequently, the explorative approach not based on prior knowledge or anticipation must be well supported and facilitated (i.e., browsing, discovery, range searches, proximity searches).
- A requirement of instant and accurate evaluation of the clone situation.
- The sheer volume of data to be presented.
- Multidimensional data analysis (file entity based, clone entity based, clone cluster entity based, file cluster entity based).

##### ***5.4.1 Usage Scenarios***

Functional requirements of the CloneMaster clone visualization system are represented using use cases. An informal and imprecise modeling technique, use cases describe the system at a high level from the user's standpoint. Each use case defines a particular

aspect of what the user wants to do with the system and the most usual course of this action. A set of main use cases identified for the CloneMaster visualization system is presented in Figure 5.3.

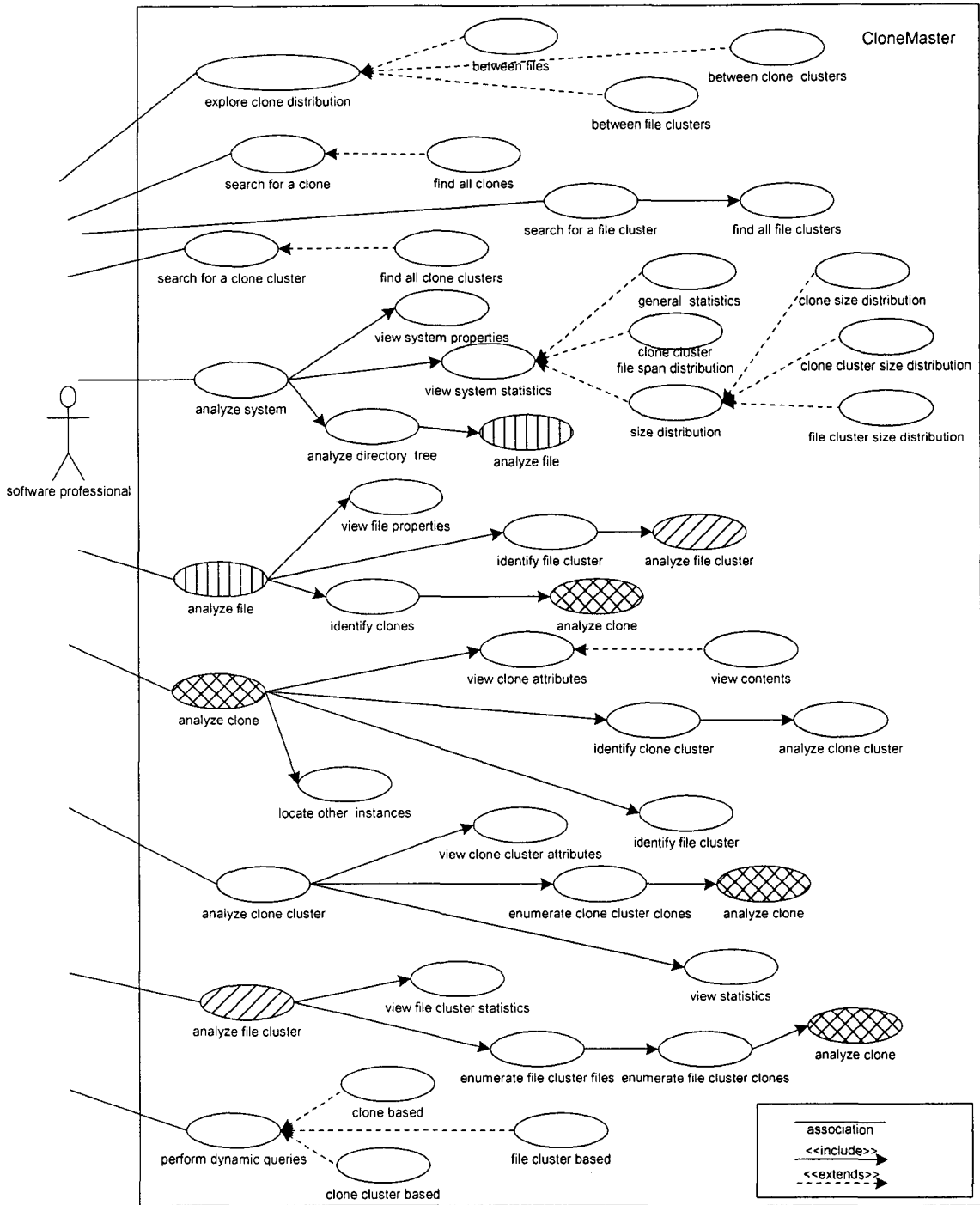


Figure 5.3: Main use case scenarios of the CloneMaster visualization tool

### ***5.4.2 CloneMaster Data Organization***

As discussed in Chapter 4, the underlying clone data is maintained and manipulated using an almost fully normalized relational (MS SQL Server v.7.0) database. The data model of the CloneMaster database is quite simple and was carefully designed to provide as much advantage to the application as possible. It is fully described in Appendix B.

The CloneMaster schema maintains information about clone structure of the analyzed system and is based on such fundamental entities as clone system, file, clone, clone cluster, and relationships between them. While the file, clone, and clone cluster entities were defined in Chapter 2 (section 2.4), the term ‘clone system’ requires further clarification.

Within the context of CloneMaster, the term ‘clone system’ is used to refer to the result set of clone identification process applied to a software system rather than the software system itself. The clone identification process can be uniquely identified in terms of its parameters (i.e., pre-processing configuration, target snip size, and noise threshold). Distinct results (or clone systems) are achieved when the same body of source code undergoes clone identification with different parameters.

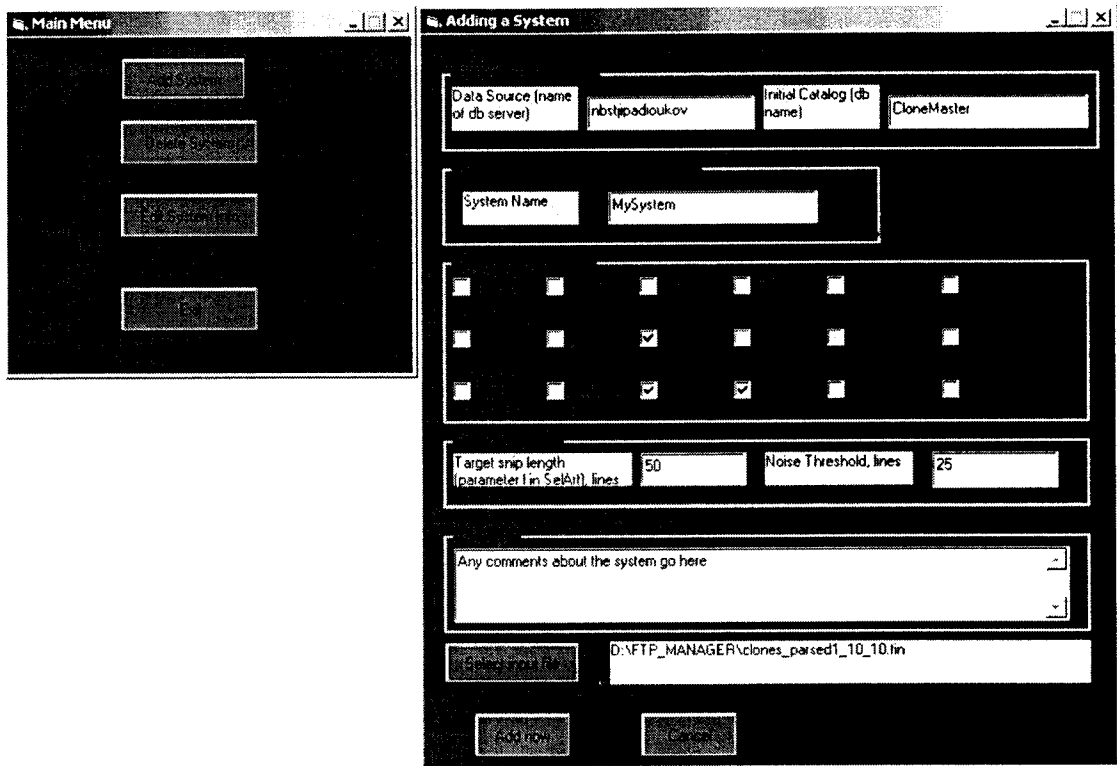
One important characteristic of the data model is that it does not provide complete information about directory tree structure associated with the body of source code of the

analyzed software system. Nor does it store the source code itself. Consequently, the source directory tree has to be available in order for the clone data to be mapped onto it.

Database population with clone data is outside the scope of CloneMaster. However as part of clone analysis process, a supporting utility, DBManager, was created to assist the user with this operation.

#### ***5.4.2.1 Database manipulation with DBManager***

DBManager is a GUI utility that provides the user with a simple interface to the CloneMaster database while keeping the underlying schema completely transparent. These manipulations include adding a new system, deleting an existing system, and modifying an existing system. Figure 5.5 shows 'Main Menu ' and 'Add System' screens. To add a system, the user needs to specify a target database, choose a name for the system being added, provide some relevant information about the system (pre-processing details, target snip size, noise threshold, optional comments), and point to a file that contains results of clone identification. Once complete information is supplied, DBManager parses the file, extracts clone information, and inserts it into tables.



**Figure 5.4:** DBManager Graphical User Interface

### 5.4.3 Graphical User Interface Organization

Figure 5.5 shows the CloneMaster display, which is composed of four panes (left, middle, right, and bottom) and a set of menus:

- The left pane presents a directory tree based or file-based view. It represents source directory tree of a software system and shows a clear picture of clone distribution in the system. This view presents clone information from the point of view of their physical location within the context of software system's source tree. The view renders an explorer-type directory tree with clone nodes hanging off corresponding file nodes. Each clone node is identified by a combination of

that clone's name<sup>18</sup> and the name of its clone cluster. To facilitate pre-attentive processing, files, and recursively folders, that contain clones are color-labeled blue as well as their labels are extended to display the number of clones found underneath that particular node (displayed in brackets).

- The middle pane is a clone cluster<sup>19</sup>-based view. This view shows software system's decomposition in terms of clone clusters. The set of clone clusters identified in a system is organized in a tree-like fashion. Each clone cluster node is labeled by its name with its size showed in brackets. Children of a cluster node represent clones that belong to that cluster.
- The right pane is a file cluster-based view. This view allows the user to investigate file clusters found in the system by analyzing how they are composed of files and what matches these files have in common. Similar to the clone cluster-based view, a file cluster node is identified by its name and the number of files it is composed of (displayed in brackets). In turn, for each file node, its clone structure is revealed.
- The bottom pane (can be collapsed/expanded on demand) provides system description (i.e., parameters of the clone identification process).

---

<sup>18</sup> Name is not an attribute of clone, clone cluster, and file cluster entities. It is just a string generated for referencing convenience.

- Menus supported in CloneMaster include static menus (along the top edge of the window) and pop-up dynamic menus with cross-browsing support.

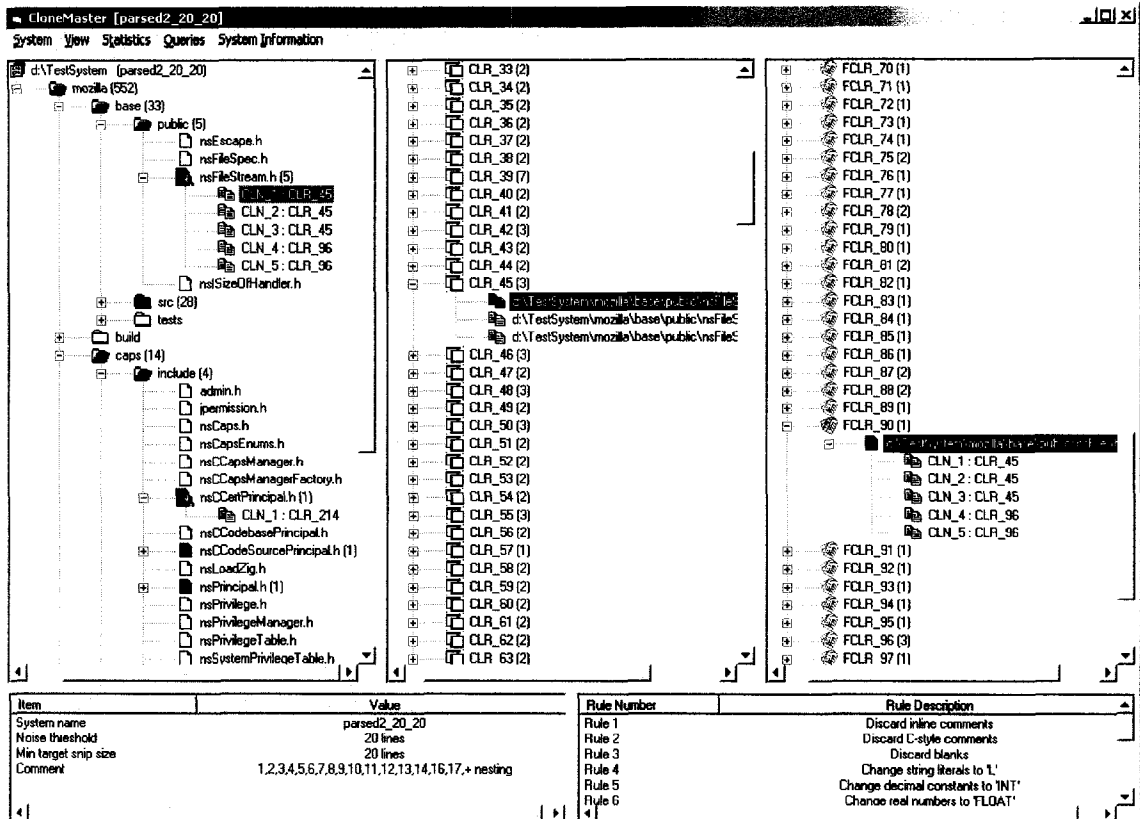
Figure 5.6 illustrates the graphical conventions used in CloneMaster to represent different objects in redundancy analysis. Whenever possible, system metaphors are used. When custom metaphors are used, they are designed to be as intuitive and instantly recognizable to the user as possible.

CloneMaster uses multiple views to present the data to the user from different perspectives and to enable different exploration routes. Distributing data between multiple views also alleviates the problem of overcrowding of a single view. Each view is based on a fundamental entity of redundancy analysis (i.e., clone, clone cluster, or file cluster). Within each view, a tree-based representation leverages a familiar and easily understood common way of depicting hierarchical data as well as the ability to collapse certain graphical structures into a single node (information hiding). All views use the system node as their root node.

---

<sup>19</sup> The notions of 'clone cluster' and 'file cluster' are discussed in section 2.4.





**Figure 5.5:** CloneMaster GUI display (main window).



**Figure 5.6:** Icons used in CloneMaster. From left to right: folder without clones, folder containing clone(s), file without clones, file with clone(s), clone, clone cluster, and file cluster.



**Figure 5.7:** Using color-coding for highlighting. From left to right: cross-referenced clone instance, cross-referenced file, cross-referenced file cluster, and active clone instance.

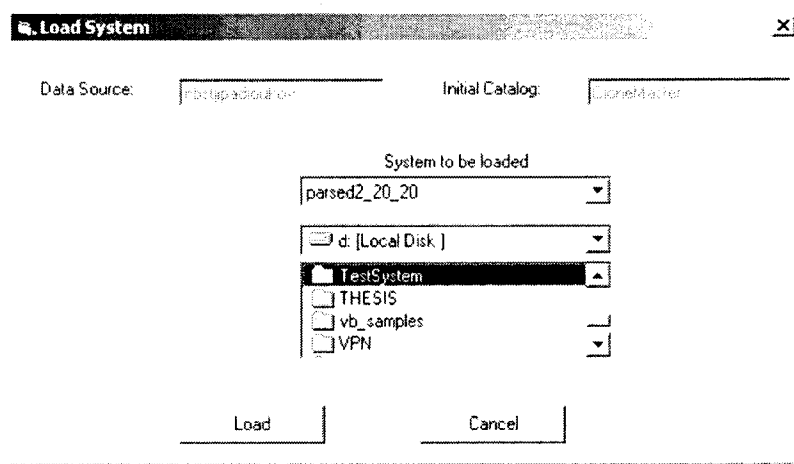
The views are permanently displayed side by side with the file-based view serving as an anchor – an overview window that allows the user to stay in context while exploring other views or navigating between views. The file-based view is best suited for this role because it is the most concrete of the three views on one hand. On the other hand, for most users the natural way of thinking about a software system, especially an unfamiliar one, is in terms of its source tree structure.

CloneMaster is a single-document interface (SDI) application, such that only one system can remain loaded at a time. However, to accommodate cross-system comparison analysis, multiple concurrent instances of CloneMaster are allowed.

#### ***5.4.3.1 System Loading***

As discussed above, the CloneMaster database stores only information on clones and their locations. Therefore, the root node of a directory tree has to be provided during system loading to associate clone information with the actual source directory tree.

To build the file-based view, the loading process performs dynamic discovery and rendering of the source directory tree. For each file found, a database lookup returns information about clone structure of the file that is integrated with the directory structure during rendition.



**Figure 5.8:** ‘Load System’ dialog. Allows the user to pick the system from the list of all available systems and to specify the corresponding source tree.

The clone cluster-based tree is built next, while the file cluster-based tree is built last.

The clone cluster-based tree is built directly from the database. Conversely, file clusters have to be discovered first. A recursive procedure uses the two existing trees to build file clusters before organizing them in a tree-like fashion.

All trees are fully pre-constructed during system loading. Moreover, some additional information is pre-stored within the tree data structure to minimize database access.

Although this makes for a sluggish start, it is a sensible tradeoff for it significantly improves responsiveness of the tool during operation. Rapid feedback is important because it makes the user feel in control of the data exploration process.

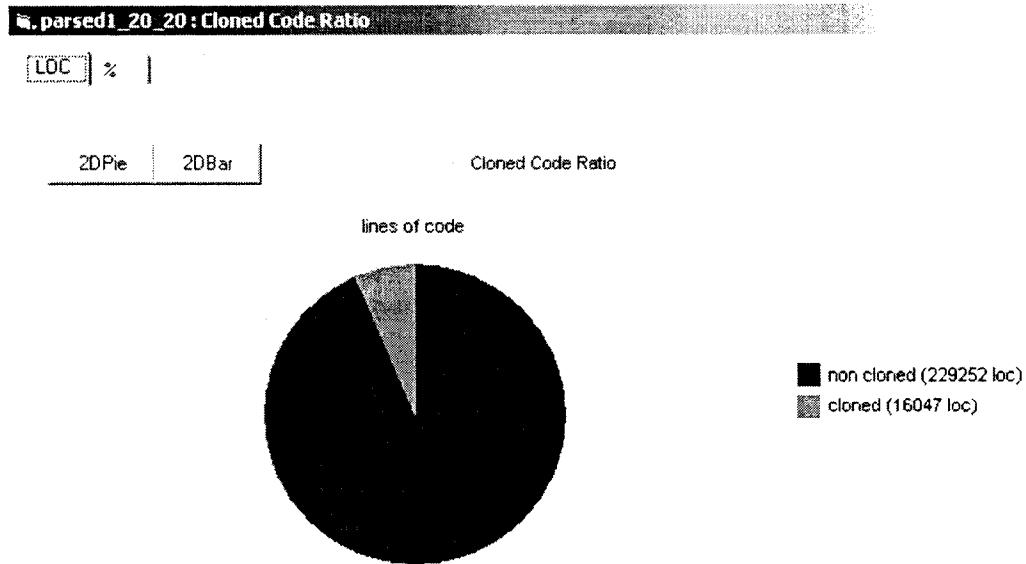
### *5.4.3.2 Menus, Navigation, and Interaction Techniques*

CloneMaster provides the user with a wide array of task specific interactive techniques for working with the data: static menus, dynamic pop-up menus, dynamic queries, and “mouse over” mode.

Static menus provide the following functionality:

- Manipulate a single view or multiple views in terms of extending trees, collapsing trees, selectively extending branches to reveal all clones, reverting color-coded icons to their normal state.
- Calculate and deliver graphically (histogram or pie diagram) system statistics such as cloned code ratio, clone size distribution, clone cluster size distribution, clone cluster file span distribution, and file cluster size distribution. Figures 5.9 and 5.10 show two examples of such statistical analysis.
- Perform dynamic queries against the data. This aspect of CloneMaster’s functionality is covered in the next section.
- Enable/disable the “mouse over” mode. “Mouse over” is a real time interaction technique that allows the user to obtain additional information about any particular node by simply positioning the mouse pointer over it. The mouse position is tracked; the node under the mouse pointer becomes activated, and its label changes to display an extended summary of the node’s attributes. When the mouse pointer moves away from the node, the node becomes deactivated, and its

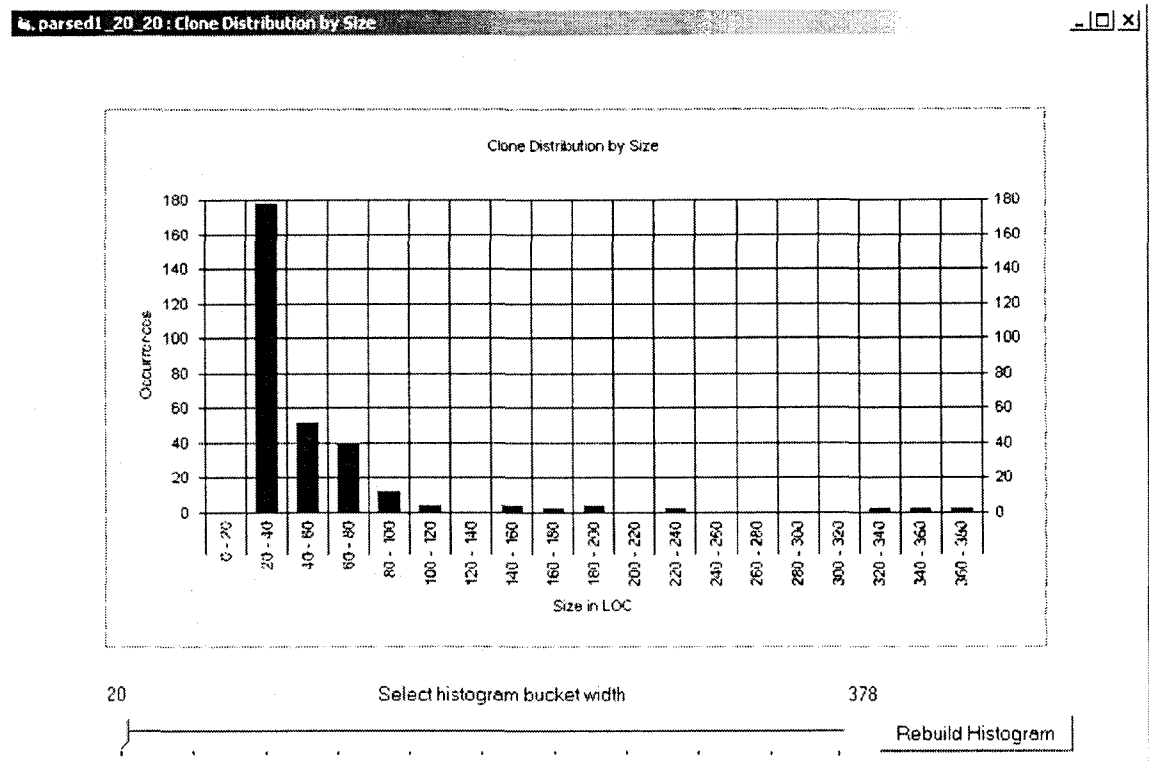
label changes back to the original form. For example, for an activated clone node the following information is displayed: path of the parent file, clone's starting position, clone's ending position, clone's length, corresponding clone cluster, and clone's cardinality.



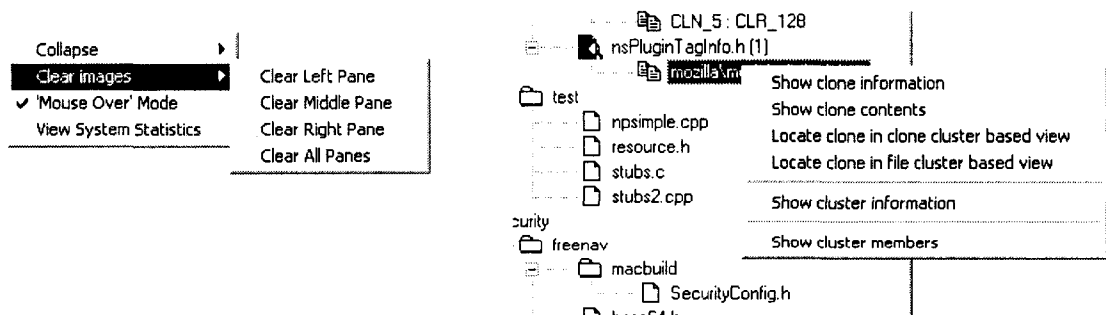
**Figure 5.9:** Pie diagram showing amount of cloned code vs. non-cloned code.

Another type of menus supported is pop-up menus. These menus are activated via right mouse click and are context-sensitive: they offer different choices depending on the object selected. If the user right-clicks on a tree node, a menu specific to this node's type pops up. A typical node menu consists of options to display the node's attributes and to identify this node in the other views (cross-referencing). Right-clicking outside the nodes activates a menu that allows one to manipulate entire view(s) (collapse trees, clear images, enable/disable "mouse over" mode, etc.). Examples of node specific and generic

menus are shown in Figure 5.11. Pop-up menus help to achieve the so called 'drill down' effect where the user can obtain more detail without losing context.



**Figure 5.10:** A histogram showing clone distribution by size built with bucket width of 20 lines. The slider at the bottom allows the user to adjust the width of the bucket and to rebuild the histogram using the new setting.



**Figure 5.11:** Pop-up menus. The menu on the left is a generic menu that appears when the user clicks outside a node. It allows to manipulate the view in general and mirrors the options offered by the 'View' static menu. The menu on the right is a clone node menu that enables actions specific to the clone entity.

For clone nodes and file nodes, CloneMaster supports direct navigation to the corresponding source code. Right clicking on the node and choosing 'Show clone/file content' option from the node specific pop-up menu brings up a separate window displaying the source code. To make cloned fragments stand out perceptually, they are rendered in different color. Source code windows are equipped with caret position tracking (i.e., line and column) that makes it easier to navigate through the code and to perform cross-examination between multiple code windows.

When 'Show instance (i.e., file, clone, clone cluster, etc) information' option is selected from a pop-up menu, this information is displayed in table form in a separate window. Figure 5.12 is a typical example of how the information is displayed for a clone instance. Clicking the 'Show other instances' button extends the window to list all other instances of that clone. Clicking on a row will reveal the location of that particular instance of the clone in each view.

Path	Clone ID	Start Line	End Line	Length	Cluster ID
d:\TestSystem\mozilla\modules\plugin\src\nsplugin.cpp	CLN_13	938	981	44	CLR_57

Path	Clone ID	Start Line	End Line	Length	Cluster ID
d:\TestSystem\mozilla\modules\plugin\src\nsPluginManager.cpp	CLN_6	693	736	44	CLR_57
d:\TestSystem\mozilla\modules\plugin\src\nsPluginManager.cpp	CLN_7	881	924	44	CLR_57
d:\TestSystem\mozilla\modules\plugin\src\nsplugin.cpp	CLN_12	750	793	44	CLR_57

CLN\_13      d:\TestSystem\mozilla\modules\plugin\src\nsplugin.cpp

**Figure 5.12:** Example of clone instance information. A summary of attributes of clone CLN\_13 located in 'd:\TestSystem\mozilla\modules\plugin\src\nsplugin.cpp' file is displayed.

Since entities may simultaneously appear in more than one view, the ability to cross-reference them becomes important in order to link them together. CloneMaster supports interactive identification of an object selected in one view by highlighting it in all views in which it appears. This is achieved via two mechanisms: the corresponding branches are extended to ensure the target node's visibility, plus the node itself is highlighted. These visual clues point the user to where to go next and, therefore, are central to view navigation.

CloneMaster uses color-coding to achieve persistent highlighting (Figure 5.7). When the highlighting is no longer needed, it can be cleared through the 'Clear Images' menu option.



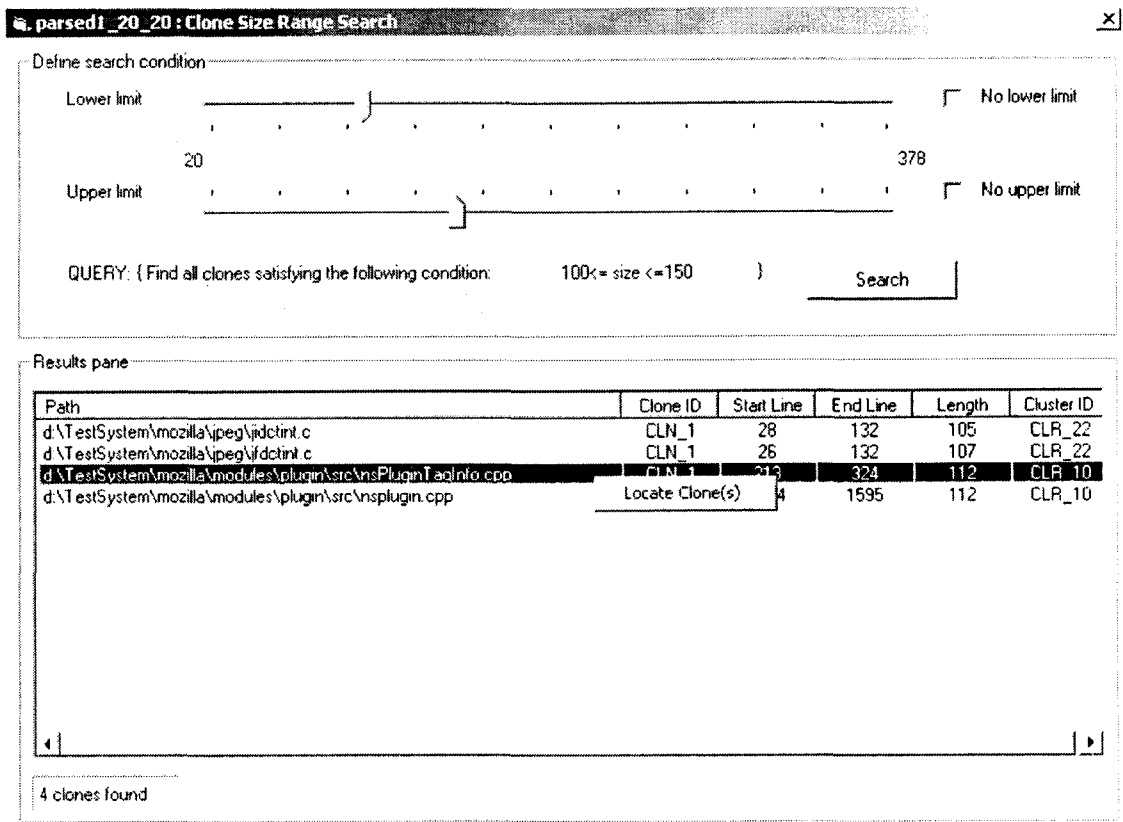
### 5.4.3.3 Query Support

One prominent feature of CloneMaster is its ability to perform interactive queries against the data. Queries aid in the data exploration process. By partitioning the data set according to certain criteria, they allow the user to focus on specific aspects of the data.

CloneMaster currently supports the following groups of queries:

- Based on clone entity: clone size range search.
- Based on clone cluster entity: cluster size range search, file span range search.
- Based on file cluster entity: file cluster size range search, file cluster proximity search.

Figure 5.13 depicts an example of a dynamic query based on clone entity. The two sliders allow the user to select search criteria. In this case, all clones with sizes between 100 loc and 150 loc were sought. Four clones that satisfied this condition are listed in the results pane. The results pane is made interactive to provide “jump and show” capability: choosing the ‘Locate Clones’ menu option will visually identify highlighted clones in all views that they appear.



**Figure 5.13:** Example of a dynamic query based on clone entity.

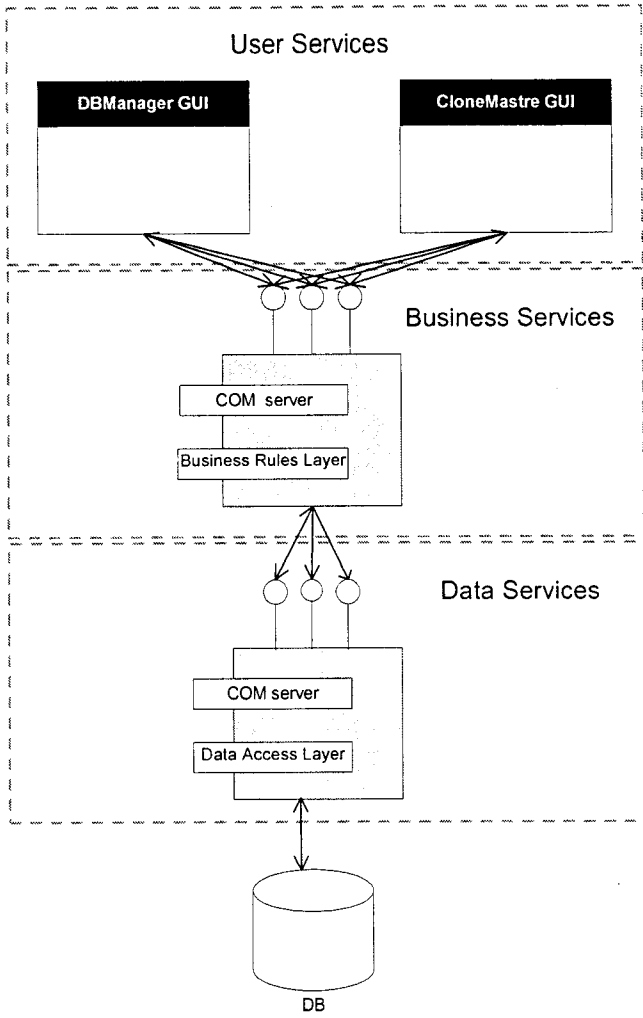
#### 5.4.3.4 Comments on Implementation

CloneMaster is built in Visual Basic 6.0. VB offers strong support for developing GUI applications, and, more importantly, allows creation of user interfaces with a look and feel of a traditional MS Windows interface. This consistency allows to leverage user's familiarity with the environment that, in turn, significantly improves the chances of the application acceptance by the user. Another factor in favor of Visual Basic is its relative "friendliness" in terms of programming effort and, thus, productivity.

CloneMaster is implemented as a conventional three-tiered solution consisting of the user services layer, the business services layer, and the data services layer. Figure 5.14 illustrates the architectural model of CloneMaster. The three layers are briefly summarized below:

- User services layer– provides visual interface for presenting information and gathering data.
- Business services layer– manages requests from the user to execute a business task, maintains business rules that dictate policies for manipulating data.
- Data services layer– possesses extensive knowledge of database organization; maintains, accesses, and updates data.

To ensure data integrity, all multi-table operations that involve data updates are implemented using transactions. In case a failure occurs, the changes are automatically rolled back leaving the data in a consistent state.



**Figure 5.14:** CloneMaster architecture: conceptual view.

## ***Chapter 6 – Industrial Experience and Evaluation***

This chapter presents a study designed to evaluate the clone analysis process developed and implemented throughout this thesis. To carry out this evaluation, a large body of well-known public domain source code was chosen.

### ***6.1 Selecting the Case Study***

The study encompasses all stages of the clone analysis process (i.e., clone identification, clone data presentation, and clone data interpretation). It focuses on the following goals:

1. Prove the potential of our approach, - the comprehensive clone analysis process.
2. Study the effect of different degrees of pre-processing on the results of redundancy analysis.
3. Evaluate potential benefit of extending the clone identification process to accommodate near clones.
4. Analyze the nature of clones and their occurrences.
5. Evaluate the usefulness of the visual clone management tool (CloneMaster).
6. Evaluate the scalability of the approach under two aspects:
  - Does it scale given the size of the source code itself?
  - Does it scale given the amount of duplication identified?
7. Identify potential deficiencies and devise strategies for their elimination.

## *6.2 Choice of Source Code Test Case*

For the purpose of this evaluation, the source of one of the mini-releases of the Mozilla browser (v.1.0) was used. Mozilla, available via <http://www.mozilla.org>, is an open-source web browser known to the world as the Netscape Communicator client. Mozilla is implemented in C++, JavaScript, and some embedded assembler. Mozilla C++ source code is highly modular and follows the rules of OOP. After filtering out all irrelevant files<sup>20</sup>, approximately 7.5MB (246,000 lines of code) of source code distributed between 689 files in 98 directories remained to be analyzed.

The Mozilla software system was selected based on the following criteria:

1. It is well known and freely available in source form.
2. It is of sufficient size.
3. The majority of it is written in C++.
4. It seemed likely to contain source code clones for the following reasons:
  - Mozilla is a cross-platform application (Windows, Mac OS, Unix (Solaris, Irix, Linux)) that supports multiple web technologies and protocols.
  - Mozilla underwent numerous successive releases.
  - Mozilla was partially developed by numerous independent developers (one of the benefits of being an open source).

---

<sup>20</sup> As discussed in Chapter 4, current implementation of pre-processing supports only C++ implementation files.

5. Mozilla is still evolving and, therefore, makes good material for subsequent releases analysis.

### ***6.3 Evaluation Procedure***

The procedure used for the evaluation is depicted in Figure 6.1. This procedure derives from the clone analysis process outlined in Chapter 1 (Figure 1.1) and contains the following stages:

#### Stage I – Clone Identification:

First, different degrees of pre-processing (discussed in the next section) were applied to the original source code. Next, SelArt was invoked (with appropriately adjusted parameters<sup>21</sup>) on each system to perform clone detection. The minimum size of match to be found was set at 20 lines.

#### Stage II – Clone Data Presentation:

Result reports produced by SelArt for each system were further processed to extract information about clones of size no less than 20 lines and to convert information about clone boundaries into a meaningful form (line numbers). During the next step, this information was transformed into a set of database tables.

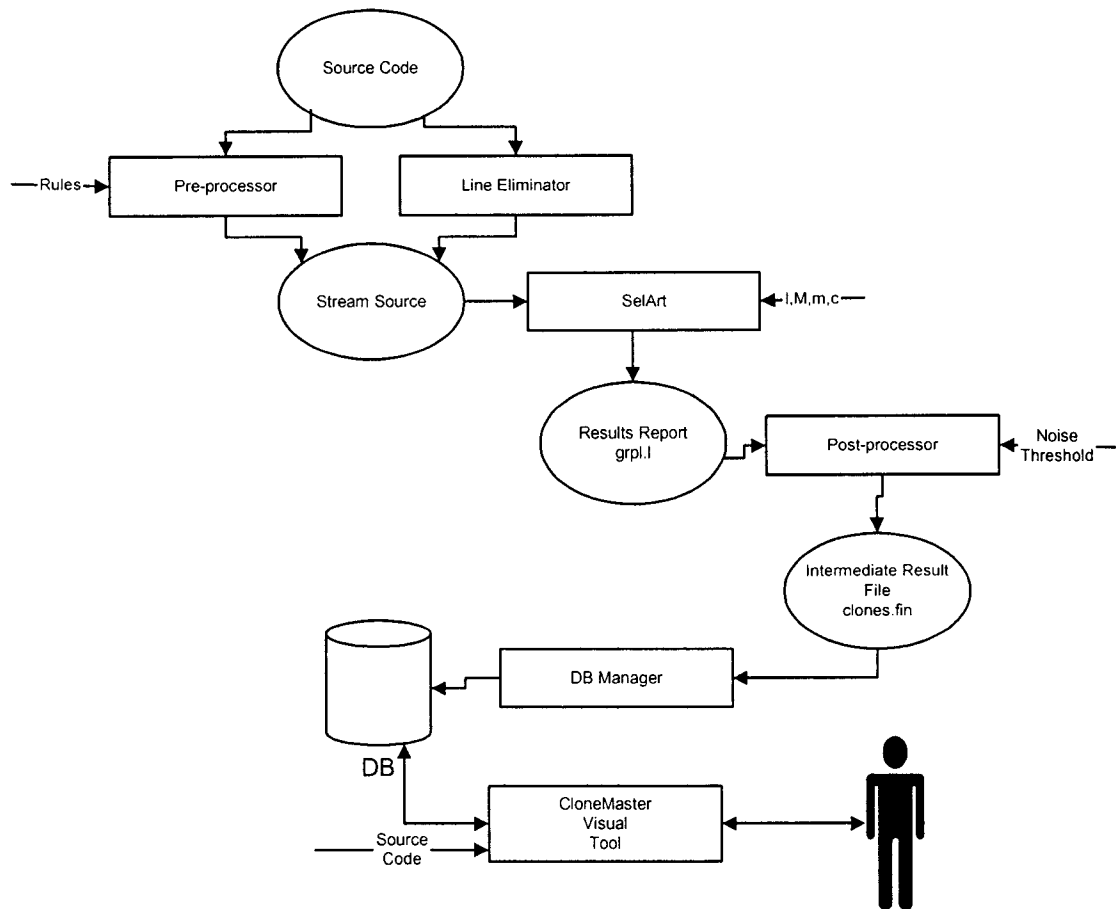
Finally, each system was loaded into a separate instance of the CloneMaster GUI.

#### Stage III – Clone Data Interpretation:

---

<sup>21</sup> These parameters are discussed in section 4.4.2.

The author performed careful examination of each system as well as comparison between different systems via the visual interface of CloneMaster, leveraging its navigation capabilities, dynamic query support, and reporting mechanisms.



**Figure 6.1:** Experimental process

Pre-processing, clone identification and post-processing were performed on a UNIX workstation with two 400MHz UltraSPARC-II processors with 2GB of RAM running Solaris 2.6. The rest of the experiment (including hosting the data base) was conducted on a Pentium III 800MHz laptop with 192MB of RAM running Windows 2000.



Two different pre-processing configurations were applied to the original Mozilla source code to produce two distinct systems 'parsed1' and 'parsed2'. These pre-processing configurations are described in Table 6.1.

To provide a baseline for comparison, one more system was created by eliminating only new line characters from the Mozilla source code. From this point on, this system will be referred to as 'original'.

Pre-processing Details		Systems	
Rule #	Rule Description	parsed1	parsed2
1.	Discard inline comments	X	X
2.	Discard C-style comments	X	X
3.	Discard blanks	X	X
4.	Change string literals to 'L'		X
5.	Change decimal constants to 'INT'		X
6.	Change real constants to 'FLOAT'		X
7.	Change octal/hexadecimal constants to 'OX'		X
8.	Change character constants to 'C'		X
9.	Change identifiers to 'I'		X
10.	Discard preprocessor directives	X	X
11.	Discard '#if 0 ...#endif' blocks	X	X
12.	Discard escape sequences	X	X
13.	Discard continuation sequences	X	X

Pre-processing Details		Systems	
Rule #	Rule Description	parsed1	parsed2
14.	Discard 'EOF'	X	X
15.	Change all numeric constants to 'N'		
16.	Change direct component selector '.' to '^'		X
17.	Change indirect component selector '→' to '^'		X
	Pre-processing Time, sec	108	109

**Table 6.1:** Comparison of pre-processing configurations between parsed1 and parsed2 systems.

The initial hypothesis was that pre-processing would help to find more clones:

- in addition to 'exact' clones, 'near' clones would also become detectable;
- higher degree of pre-processing would reveal more 'near' clones.

Therefore, the amount of identified duplication was expected to increase as we moved from 'original' system to 'parsed1' and then to 'parsed2' respectively.

To reduce the amount of data for further analysis, any identified clones with sizes under 20 lines were ignored. Thus, all data and inferences presented below only apply to a particular subset (i.e., 20 lines and greater) of clones present in the Mozilla system and may change if a different subset of clones is considered.

## 6.4 Clone Detection Results

This section presents and interprets results of clone identification performed under similar conditions after three different degrees of pre-processing had been applied to the Mozilla code. The three systems described above (i.e., ‘original’, ‘parsed1’, and ‘parsed2’) were loaded into the CloneMaster tool and studied using the facilities provided by the tool. The outcome of this analysis is discussed below.

	<b>original</b>	<b>parsed1</b>	<b>parsed2</b>
Degree of pre-processing	none	little (discarding)	extensive (discarding + tokenizing)
Type of clones identified	‘exact’	‘exact’ + ‘near’	‘exact’ + ‘near’
Min target clone size, lines	20	20	20
Noise threshold, lines	20	20	20
Number of files analyzed	689	689	689
LOC analyzed	245299	245299	245299
Number of clones	310	303	552
LOC cloned	16783	16047	26520
Percentage of cloned code	6.8	6.5	10.8
Number of clone clusters	126	139	252
Number of file clusters	40	45	105

**Table 6.2:** Summary of major statistics from the experiments.

Table 6.2 summarizes the major statistics obtained from the experiments, the type of pre-processing performed for ‘parsed1’ (Table 6.1) actually caused a slight decrease in both the number of clones found and the amount of code shared between these clones. Yet, the number of clone clusters and the number of file clusters identified in ‘parsed1’ was larger than in ‘original’. ‘Parsed2’, as expected, showed a significant increase in all categories of measured characteristics with respect to both ‘original’ and ‘parsed1’.

The three clone systems were scrutinized with the help of CloneMaster to verify and interpret these results. The results of this analysis are summarized below:

- The ‘original’ system produced a set of all exact clones in the Mozilla code that were longer than 20 lines.
- For the most part, clusters of exact clones identified in ‘original’ were also found in ‘parsed1’. Some clones in ‘parsed1’ had bigger sizes than their counterparts in ‘original’ due to the elimination of editing idiosyncrasies and/or discrepancy in comments. However, when a clone in ‘original’ started or ended with a comment, it came out shorter in ‘parsed1’ exactly by the size of the comment (i.e., comments were ignored). In some cases, removing the comment made the clone shrink below the minimum target snip size and, thus, to be completely missed during the clone identification phase or rejected at the post-processing phase as noise.
- Some clones identified in ‘original’ consisted entirely of comments, especially in the beginning of the file, preprocessor directives, or combination of the two (Figure C.1). These clones were missed in ‘parsed1’ and ‘parsed2’. This was

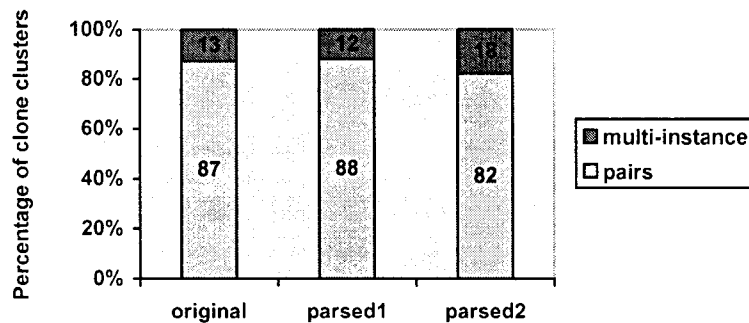
quite typical since many files in Mozilla system started with a standardized header (enforced by the Mozilla Coding Style Guide) that was longer than 20 lines. For instance, just 3 clone clusters in ‘original’ encompassed 39 clones (27, 7, and 5 respectively) that were of that nature. Consequently, the number of clones in ‘parsed1’ was slightly lower than in ‘original’, whereas the number of clone/file clusters actually was higher.

- ‘parsed1’ revealed some new clusters that were not detected in ‘original’. These clusters were near clones differing just in editing detail, comments, or blocks of conditional compilation (`#if 0 ... #endif`). See Figure C.3 in Appendix C for an example.
- ‘parsed2’ produced a large increase in the number of new clone clusters (113) containing more interesting matches. Clones revealed in ‘parsed2’ system were mostly cut-and-paste copies of each other that underwent successive modification (name changes, value changes, addition/deletion/modification of comments, editing changes). A typical example of such near clones is depicted in Appendix C Figures C.4 and C.5.
- Some exact clone clusters reported in ‘original’ and ‘parsed1’ did not appear in ‘parsed2’. This was typical when periodic strings (i.e., ones that contain a repetition such as ‘ABABABABAB...’) were produced during tokenizing. A known bug in SelArt caused combining and splitting such periodic strings to produce a series of short matches (i.e., ‘AB’) as opposed to one long match. Figures C.6 and C.7 further illustrate this scenario. Quite infrequent with

untokenized source, the problem intensifies when the source is aggressively filtered.

- In terms of individual length of cloned fragments, the three systems were quite consistent. The majority of clones were under 100 lines, however, matches longer than 300 lines were also found. Figure 5.9 from the previous chapter shows clone distribution by size for ‘parsed1’.

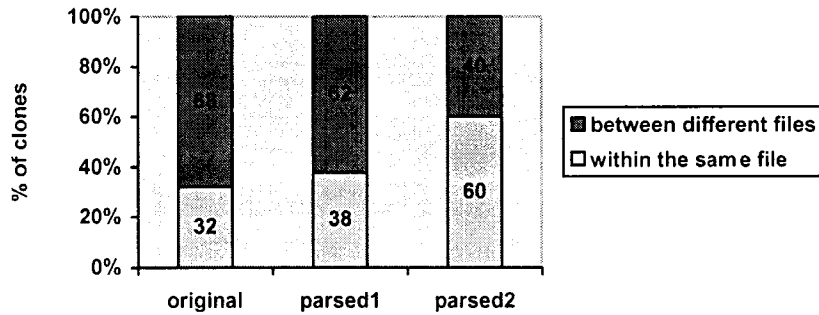
Clone cluster size examination revealed that the majority of clones in all three systems had a cardinality of 2 (i.e., had just two instances). However, increasing the degree of approximate matching allowed one to uncover more multi-instance clone artifacts (Figure 6.2).



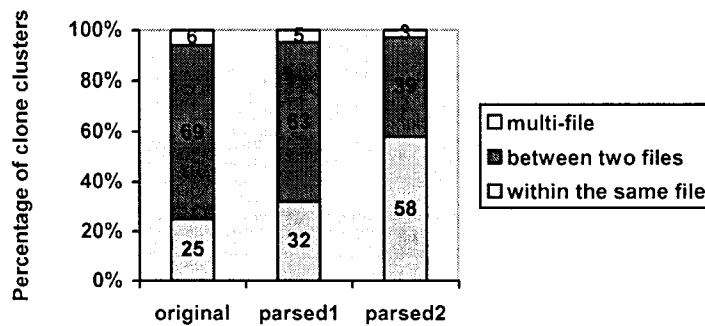
**Figure 6.2:** Clone clusters break down by size.

Figure 6.3 presents the distribution of clones found within the same file versus between different files. The majority of near clones happened within the same file. Exact clones, on the contrary, showed a tendency to span different files. However, as suggested by

Figure 6.4, the clone cluster file span in Mozilla system rarely exceeded two files. The noticeable decrease in complex (i.e., multi-file) clone clusters from 'original' to 'parsed2' is due to the fact the loss of some exact clones as discussed earlier (i.e., 100% comments, periodicity).



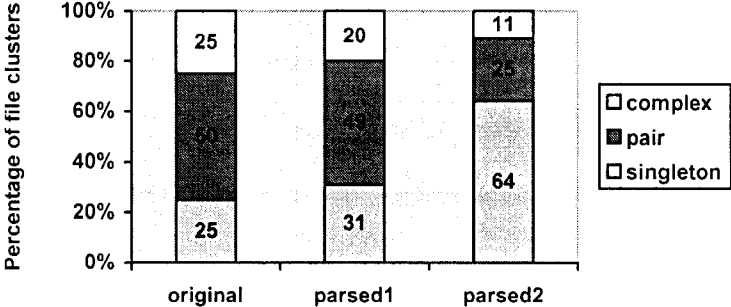
**Figure 6.3:** Duplication within the same file vs. duplication between different files.



**Figure 6.4:** File span distribution of clone clusters.

File clusters reflect system partitioning based on dependencies resulting from cloning. Complex file clusters (including pairs) indicate some degree of functional coupling between files. Singletons, on the other hand, do not imply inter-file coupling. Exact clones appeared to increase coupling as they partitioned the target Mozilla system predominantly into two-file clusters or pairs (Figure 6.5) with a substantial amount of

complex multi-file aggregates and singletons (i.e., single file). Near clones, on the other hand, tended to cause less interdependency between files as they mainly produced independent singletons. However, distributions for ‘parsed1’ and ‘parsed2’ shown in Figure 6.5 may have been skewed due to the loss of information on certain exact clones (see discussion above).



**Figure 6.5:** Distribution of file clusters by size (i.e., file count).

Another noteworthy characteristic of code duplication in the Mozilla system was that, overall, the majority of file clusters (over 75 %) were formed within a single directory. The rest had quite a narrow directory span of 2, and never more than 3. This can be explained with organization of Mozilla’s source tree: Code delivering similar functionality and/or fulfilling common purpose is stored together. However, there are also platform dependent subdirectories that contain platform specific code.

Studying file clusters identified in ‘parsed2’ revealed at least one case of code sharing between different modules. A three-file cluster linked together three different



components through one two-way exact match and two two-way near matches of function fragments.

## 6.5 Nature of Clones and Their Occurrences

Clones are perfect candidates for restructuring that could transform the ad hoc reuse they represent into more disciplined types of reuse. By replacing a clone with a new subroutine or method the duplication can be eliminated, and this may lead to greater maintainability. One of the goals of this evaluation was to analyze detected clones for the purpose of understanding their nature and occurrences, so as to help develop efficient strategies for their removal (i.e., replacing with a single code entity). Some observations are summarized in Table 6.3.

	Clone Type	Match Type	Duplication Context	Possible Restructuring Measures
1	Full/partial class declaration or/and implementation	exact, near	<ul style="list-style-type: none"> <li>▪ Classes representing close/identical concepts (i.e., IOFileStream, InputFileStream)</li> <li>▪ Classes representing symmetrical concepts (i.e., input/output, compress/decompress)</li> <li>▪ Classes implementing same functionality on different platforms (i.e., platform specific data types, API calls)</li> <li>▪ Classes operating on different objects/data types (i.e., single-byte char/wide char, handling different image formats, compression algorithms)</li> <li>▪ Classes representing unrelated concepts but sharing some code</li> </ul>	<ul style="list-style-type: none"> <li>▪ Use of inheritance (i.e., extract a superclass)</li> <li>▪ Use of &lt;Template&gt; classes</li> <li>▪ Factor common code out into a set of helper functions or a helper class</li> <li>▪ Add cross-reference<sup>22</sup></li> </ul>
2	Function (complete)	exact	<ul style="list-style-type: none"> <li>▪ Platform specific implementations</li> <li>▪ Implementation of similar</li> </ul>	<ul style="list-style-type: none"> <li>▪ Create new function FX and change</li> </ul>

<sup>22</sup> There existed cases where a pointer to the original was added via a comment. However, this was rather an exception than a rule and was done in undisciplined manner.

	Clone Type	Match Type	Duplication Context	Possible Restructuring Measures
	F1, F2, ..., FN		<ul style="list-style-type: none"> <li>concepts (i.e., handling different image formats) <ul style="list-style-type: none"> <li>Interface implementation (i.e., IRender, IRender1)</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>F1, ..., FN to call it <ul style="list-style-type: none"> <li>Change all calls to F2, ..., FN into calls to F1 and remove F2, ..., FN</li> <li>Add cross-reference</li> </ul> </li> </ul>
3	Function (complete) <ul style="list-style-type: none"> <li>Differ in return type</li> <li>Differ in data type(s)</li> </ul>	near	<ul style="list-style-type: none"> <li>Platform specific implementations</li> <li>Implementation of similar concepts (i.e., handling different image formats, array types)</li> <li>Interface implementations (i.e., IRender, IRender1)</li> </ul>	<ul style="list-style-type: none"> <li>Extract common code into a function FX; Use parameterization (add arguments, 'type def' return type, etc.) or template function. Change cloned code to calls to FX <ul style="list-style-type: none"> <li>Add cross-reference</li> </ul> </li> </ul>
4	Function (fragment) <ul style="list-style-type: none"> <li>same method</li> <li>different methods (same file, different files)</li> </ul>	exact, near	<ul style="list-style-type: none"> <li>Implementation of similar functionality</li> </ul>	<ul style="list-style-type: none"> <li>Extract shared code into a separate method FX; Parameterize if necessary. Change cloned code to calls to FX <ul style="list-style-type: none"> <li>Add cross-reference</li> </ul> </li> </ul>
5	Fragments of embedded assembler code	exact	<ul style="list-style-type: none"> <li>Image handling</li> </ul>	<ul style="list-style-type: none"> <li>Add cross-reference</li> <li>Encapsulate in a function</li> </ul>
6	Entire file containing declaration and/or implementation of certain classes/interfaces	exact	<ul style="list-style-type: none"> <li>Duplicated between different files in the same directory</li> <li>Files with same name and identical contents located in different directories</li> </ul>	<ul style="list-style-type: none"> <li>Add cross-reference</li> <li>May have to do with build configurations or other aspects of configuration management</li> </ul>

**Table 6.3:** Classification of typical duplication patterns with possible restructuring solutions.

Occasionally, potential clone removal solutions were obvious without having to fully understand the semantics of the code (i.e., example given in section 6.6). Overall, devising restructuring solutions was hampered by the following factors:

- The rationale behind cloning was not exactly clear;
- The code structure was difficult to understand;

- Removing clones could cause a ripple effect through related code [Fanta 1999] that was difficult to assess;
- Manual analysis was cumbersome.

## ***6.6 Clone Analysis with CloneMaster***

This section describes one clone exploration scenario using the CloneMaster visual tool. Its purpose is to verify the potential merit of clone identification in achieving better quality software as well as to demonstrate usefulness of the tool. ‘Parsed2’ system was chosen as a target system because it contained the most interesting matches. No prior knowledge about duplication or the Mozilla system itself was assumed. We decided to start with researching matches contained within the same file and having just two instances.

- Step 1: Loaded ‘parsed2’ system into CloneMaster GUI.
- Step 2: Used ‘Queries’ → ‘Clone cluster instance based query’ → ‘File span range search’ menu option to search for all clone clusters with file span of 1 (i.e., all clones reside in the same file).
- Step 3: In the query results pane, focused on clusters of size 2. Found cluster CLR\_252.
- Step 4: From the query results pane, navigated to CLR\_252 location in the clone cluster-based tree to identify the two clones of interest.
- Step 5: Studied the source code of both clones (Figure C.4 and C.5).

- Step 6: Located the file containing the clones in the file-based tree to identify other three pairs of clones it contained to better understand the context of cloning.
- Step 7: Located the corresponding file cluster in the file cluster-based tree to determine that the parent file did not share any matches with other files.
- Step 8: As a result of Steps 5-7, it became apparent that the two clones were actually made of 4 methods that closely resembled each other. These methods performed some kind of an object retrieval action using same local variables and same parameters. The only differences were in the names of the methods itself, the names of the object to be retrieved, and the path to that object (Figure C.4 and C.5). Both object names and paths were just strings.
- Step 9: Analysis of Step 8 led to a conclusion that such cloned methods could be eliminated by creating a new helper method ‘\_getObject’ that, in addition to the original arguments, would also accept two new string arguments (i.e., ‘object\_name’ and ‘object\_path’) and encapsulate parameterized functionality of the four original methods (Figure C.8).
- Step 10: The bodies of the 4 cloned methods were replaced with the calls to the new ‘\_getObject()’ method with appropriate parameters (Figure C.9). This change should not have caused any ripple effect since it was very localized and disconnected from the rest of the system.

## 6.7 Conclusions

The Mozilla case study described in this chapter demonstrated how the clone analysis process developed throughout this thesis can be successfully applied to a large-scale commercial system to improve the system's internal structure, and, thus, such fundamental software characteristics as changeability and maintainability.

High degrees (up to 10%) of duplication identified in the experiment provided an excellent opportunity to validate the benefits of applying the integrated clone analysis process. These benefits ranged from reliable clone detection to identifying opportunities for improved architectures. Integrating the three separate stages of clone analysis (i.e., clone identification, clone data presentation and clone data interpretation) into one process allowed information obtained during the clone identification stage to be successfully leveraged on later stages of analysis (i.e., clone data interpretation and restructuring) and to enable the user to closely control each stage as well as the information flow between different stages. Analysis of the results of clone identification in 'original', 'parsed1', and 'parsed2' led to the following conclusions:

- Significant fraction of duplication is near clones;
- Ability to identify near clones is important since they are good targets for restructuring;
- The pre-processing technique developed in Chapter 4 to extend the capabilities of SelArt to support near clones adequately fulfilled its purpose and proved to be very reliable. The higher degree of pre-processing was applied, the more near clones were uncovered;

- The effectiveness of pre-processing should not be evaluated by the number of clones found. Pre-processing and clone matching are effected by both, the context and the conditions of experiment (i.e., Selart parameters, noise threshold);
- Pre-processing preserves exact clones. However, aggressive filtering of the source could cause some exact matches to be lost due to a bug in SelArt. Thus, exact matching must always complement partial matching to ensure against clone information loss.
- It was not possible to evaluate scalability of the approach based on just this case study. However, the example tested was substantial and represented the size of many commercial applications.

The case study also provided a good opportunity to evaluate the applicability and usefulness of the visual clone management tool described in Chapter 5:

- CloneMaster proved to be functional and helpful as a visual clone exploration browser in terms of analyzing clone distribution in the system and supporting decision making. Before CloneMaster was developed, all attempts of manual clone analysis undertaken by the author always failed due to the overwhelming nature and amount of clone data;
- The most beneficial features of CloneMaster discovered through the evaluation were the ability to trace different occurrences of clones based on different criteria, the ability to identify clone incurred dependencies, and to see how the system was composed of clones.

- Multiple concurrent views complemented by straightforward and easy navigation between them allowed the flow of analysis to effortlessly maneuver from one view to the other taking advantage of each view's offering. Such interconnectedness of the views facilitated different exploration scenarios depending on the context and user preference;
- Easy access on demand to auxiliary data such as different statistics, dynamic queries, and especially source code browsing proved to add value to the analysis process without overcrowding the visual display;
- CloneMaster showed adequate scalability in terms of the size of the source code itself and the amount of duplication identified. However, after a certain number of open windows was accumulated, navigation between them could become confusing;
- Performing cross-system analysis showed to be tedious, time consuming and error prone. Thus, a better mechanism to fulfill this requirement is needed;
- Since no information on degree of similarity between clones is available, it has to be determined manually through visual cross-inspection of the corresponding sources, - a very time consuming and error prone operation.

The results discussed in this chapter suggested that the proposed clone analysis process is a viable practical solution to the problem of code cloning that, if not addressed, can cause maintenance nightmares. It was shown that clones of different kinds, both exact and near, could be easily and reliably identified even in large bodies of source code.

Visual presentation of clone data, capitalizing on human abilities to rapidly assimilate

and interpret visual information, allowed this clone information to be used efficiently in practical software engineering and maintenance tasks (i.e., restructuring).

The study also helped to identify areas for improvement. Support for the cross-system comparison and the mechanism for discovering differences between clone instances are the most prominent ones. Other areas to be addressed are outlined in Chapter 7.



## *Chapter 7 – Conclusions and Future Work*

This chapter delivers some concluding remarks as it revisits the contributions of this thesis and outlines the potential for future work.

Code cloning complicates maintenance and hampers evolution of large software systems as it degrades their design and structure. Systematic management of software clones has the potential to translate into significant budget savings. Identification of software clones followed by their analysis could often suggest ways to improve internal structure of source code and to clarify its meaning. Although various aspects of clone management have been addressed by academic research, practical application has been hampered by the lack of adequate tools and processes.

A comprehensive process for analyzing software clones in large bodies of source code has been defined, implemented and tested. The approach successfully integrated the traditionally disjointed domains of clone identification and clone-based restructuring. This integration provides software practitioners with a complete set of practical tools that enable them to detect, analyze, categorize and remove duplication.

A major problem with text matching for identification of software clones is that it fails to find clones when minor changes, such as renamed variables or altered comments, have been made. This deficiency of the text-based comparison clone identification technique to detect near clones has been limiting its practical application. A simple, effective and reliable solution to this problem has been developed. The solution is based

on a concept of pre-processing during which the source code is transformed into some intermediate format designed to preserve clone information but eliminate irrelevant detail. Integration with one particular implementation of the text-based comparison technique, a tool called SelArt, was a success and demonstrated the validity and benefits of the approach. Application of the solution to an industrial software system uncovered a large amount of meaningful near clones that would have been missed by a more traditional approach.

A prototype clone management tool, CloneMaster has been designed and implemented. CloneMaster is an interactive visual clone exploration environment that empowers software practitioners with powerful yet intuitive means to view, analyze and manage information on clones and their distribution within the system. The tool analyzes detected clones, clusters them and presents them to the user in a systematic manner. Graphical presentation allows the user to see the global impact of duplication as well as to study each particular clone. Application of CloneMaster in the Mozilla case study showed several benefits and verified CloneMaster's capability to support consistent maintenance and clone-based restructuring.

The Mozilla source code represents an industrial scale body of software written in C++. This was analyzed to evaluate the clone analysis process in the context of software engineering and maintenance. Based on visual analysis (i.e., CloneMaster) of discovered duplications, some concrete possibilities for restructuring were identified. These represented places where clones could be removed or links between them added. On one

hand, the case study showed the unrealized potential of the clone analysis process to provide software professionals with opportunities to significantly improve quality of software systems. On the other hand, it validated such advantages of our approach as the ability to detect near clones, visualization-based approach to management of clone data, and supporting all stages of clone analysis to give the user ultimate control.

One potential limitation of our approach is that it is based on human interaction. The balance between automation of and human involvement in the process of clone-based analysis and re-engineering is skewed to the side of the human. Human involvement is costly, slow, and error prone, especially in larger projects. Although human involvement in clone-based re-engineering is inevitable due to the complexity of the task, the degree of this involvement should be as little as possible.

CloneMaster, through source code viewing, helps the user to evaluate possibilities for clone-based restructuring. However, it does not provide enough support for the restructuring operation itself. Other specialized tools that support restructuring (i.e., usually represent program structure via program entities and relationships between these entities) can be investigated and integrated with CloneMaster.

It is currently not possible to compute any measure of degree of similarity between matches (clones); however, such information could be extremely valuable. This limitation stems from the underlying clone identification method. It would be interesting

to investigate possibilities of delivering this functionality (i.e., in a form of a post-processing add on).

There is a need to implement a mechanism for cross-clone source code comparison to automatically identify and highlight differences between them. Currently, this is a manual task that proved to be time consuming and error prone.

Future development of the CloneMaster tool might benefit from the use of color coding. For example, using color coding to discriminate between exact and near clones, to portray distance between different clone instances, or any other attributes (clone size, file size, cluster size, file span, etc.) could convey a lot of additional information to the user or allow him to see patterns. An industrial-strength version of this tool could offer a mechanism for the user to control the color scheme used for color coding and labeling to accommodate user preferences or limitations (i.e., color-blindness).

In its current form, CloneMaster is not fully optimized for performance. Possible adjustments to improve responsiveness of the system include database access optimization (indexing, denormalization, query refinement), caching, and algorithm optimization.

The current implementation of the clone analysis process consists of multiple separate operations, or steps, performed in predefined order. It makes good sense to integrate these individual steps into one automated process.

Although the described clone analysis process is in its prototype stages, it shows the potential for developing into an effective clone management solution to support software engineering and maintenance.

## References

- [Aho 1986] Aho, A., Sethi, R., Ullman, J., *Compilers Principles, Techniques, and Tools*, Addison-Wesley Pub. Co., 1986.
- [ANSI 1983] An American National Standard IEEE Standard Glossary of Software Engineering Terminology, *ANSI/IEEE Standard 729*, 1983.
- [Baecker 1981] Baecker, R., Sherman, D., Sorting Out Sorting, 16 mm color sound film, 1981, *Shown at SIGGRAPH'81*, Dallas, TX.
- [Baecker 1998] Baecker, R., Price, B., The History of Software Visualization, *Software Visualization*, MIT Press, Cambridge, Massachusetts, 1998, pp. 29-34.
- [Baker 1992] Baker, S., A Program for Identifying Duplicated Code, *Proceedings of Computing Science and Statistics: 24<sup>th</sup> Symposium on the Interface*, 1992, pp. 49-57.
- [Balazinska 1999] Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K., Measuring Clone Based Reengineering opportunities, *Proceedings of the 6<sup>th</sup> IEEE International Symposium on Software Metrics*, Florida, Nov. 1999, pp. 292-303.
- [Ball 1996] Ball, T., Eick, S., Software Visualization in the Large, *IEEE Computer*, Vol. 29(4), April 1996, pp. 33-43.
- [Barson 1995] Barson, P., Davey, N., Field, S., Frank, R., Tansley, D., Dynamic Competitive Learning Applied to the Clone Detection Problem, *Proceedings of International Workshop on Applications of Neural Networks to Telecommunications 2*, 1995, pp.234-241.
- [Baxter 1998] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., Clone Detection Using Abstract Syntax Trees, *Proceedings of ICSM'98*, November 16-19, 1998, Bethesda (Maryland).
- [Baxter 2002] Baxter, I., Churchett, D., Using Clone Detection to Manage a Product Line, *7<sup>th</sup> International Conference on Software Reuse*, April 15, 2002, Texas.
- [Boehm 1981] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.
- [Card 1999] Card, S., Mackinlay, J., Shneiderman, B., *Readings in Information Visualization: Using Vision to Think*, Morgan Kaufmann Publishers, San Francisco, California, 1999.
- [Chikofsky 1988] Chikofsky, E., Rubenstein, B., CASE: Reliability Engineering for Information Systems, *IEEE Software*, Vol. 5(3), 1988, pp. 11-16.

[Chikofsky 1990] Chikofsky, E., Cross, J., Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, Vol. 7(1),1990, pp.13 –17.

[Dagenais 1998] Dagenais, M., Merlo, E., Lague, B., Proulx, D., Clones Occurrence in Large Object Oriented Software Packages, *Proceedings of IBM CAS Conference (CASCON'98)*, Toronto, November 30 – December 3, 1998, pp. 192-200.

[DeDourek 1980] DeDourek, G., Gujar, U., McIntyre, M., Scanner Design, *Software – Practice and Experience*, Vol. 10, 1980, pp.959-972.

[Domingue 1992] Domingue, J., Price, B., Eisenstadt, M., A Framework for Describing and Implementing Software Visualization Systems, *Proceedings of Graphics Interface '92*, Vancouver, Canada, May 1992, pp.53-60.

[Ducasse 1999] Ducasse, S., Rieger, M., Demeyer, S., A Language Independent Approach for Detecting Duplicated Code, *Proceedings of the 1999 International Conference on Software Maintenance*, IEEE Computer Society Press, September, 1999, pp. 109-119.

[Eick 1992] Eick, S., Steffen, J., Sumner, E., SeeSoft – A Tool For Visualizing Line Oriented Software Statistics, *IEEE Transactions on Software Engineering*, Vol. 18(11), November 1992, pp.957-968.

[Eick 1995] Eick, S., Engineering Perceptually Effective Visualizations for Abstract Data, AT&T Bell Laboratories, April 1995.

[Eick 1998] Eick, S., Maintenance of Large Systems, *Software Visualization*, MIT Press, Cambridge, Massachusetts, 1998, pp. 315-328.

[Ellis 1990] Ellis, M., Stroustrup, B., *The Annotated C++ Reference Manual*, Addison-Wesley Pub. Co., 1990.

[Fanta 1999] Fanta, R., Rajlich, V., Removing Clones from the Code, *Journal of Software Maintenance*, 1999, pp. 223-243.

[Fowler 1999] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison Wesley Professional, MA, 1999.

[Gcc] Official GNU GCC Website, Available via <http://gcc.gnu.org/>.

[Hamming 1973] Hamming, R., *Numerical Analysis for Scientists and Engineers*, McGraw-Hill, New York, 1973.

[Jankowitz 1988] Jankowitz, H., Detecting Plagiarism in Student PASCAL Programs, *Computer Journal*, 31(1),1988, pp.1-8.

[Johnson 1993] Johnson, H., Identifying Redundancy in Source Code using Fingerprints, *Proceedings of IBM CAS Conference (CASCON'93)*, Toronto, October 24 - 28, 1993, pp. 171-181.

[Johnson 1994a] Johnson, H., Visualizing Textual Redundancy in Legacy Source, *Proceedings of IBM CAS Conference (CASCON'94)*, Toronto, October 31- November 3, 1994, pp. 9-18.

[Johnson 1994b] Johnson, H., Substring Matching for Clone Detection and Change Tracking, *Proceedings of the 1994 International Conference on Software Maintenance*, September 19-23, 1994.

[Johnson 1995] Johnson, H., Using Textual Redundancy to Understand Change, *Proceedings of IBM CAS Conference (CASCON'95)*, Toronto, November 7 - 9, 1995, (CD-ROM).

[Johnson 1996] Johnson, H., Navigating the Textual Redundancy Web in Legacy Source, *Proceedings of IBM CAS Conference (CASCON'96)*, Toronto, November 12 - 14, 1996, pp. 7-17.

[Kane 1997] Kane, D., Opdyke, W., Dikel, D., Managing Change to Reusable Software, *Proceedings of the 4<sup>th</sup> Pattern Languages of Programs Conference*, Illinois, USA, September 3-5, 1997.

[Karp 1987] Karp, R., Rabin, M., Efficient Randomized Pattern-Matching Algorithms, *IBM J. Res. Develop.* 31(2), March 1987, pp.249-260.

[Kataoka 2001] Kataoka, Y., Ernst, M., Griswold, W., Notkin, D., Automated Support for Program Refactoring using Invariants, *Proceedings of the 2001 International Conference on Software Maintenance*, Italy, November 6-10, 2001, pp. 736-743.

[Kontogiannis 1996] Kontogiannis, K., Demori, R., Merlo, E., Galler, M., Bernstein, M., Pattern Matching for Clone Detection, *Automated Software Engineering*, 3, 1996, pp.77-108.

[Lague 1997] Lague, B., Proulx, D., Mayrand, J., Merlo, E., Hudepohl, J., Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, *International Conference on Software Maintenance*, 1997, IEEE, pp. 314-321.

[Linos 1994] Linos, P., Aubet, P., Dumas, L., Helleboid, Y., Lejeune, P., Tulula, P., Visualizing Program Dependencies: An Experimental Study, *Software – Practice and Experience*, Vol. 24(4), pp.387-403, April 1994.

[Martin 2000] Martin, J., Wong, K., Winter, B., Muller, H., Analyzing xfig Using the Rigi Tool Suite, *Proceedings of the Working Conference on Reverse Engineering*, Brisbane, Australia, November 23-25, 2000.



- [Mayrand 1996] Mayrand, J., Leblanc, C., Merlo, E., Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, *Proceedings of the International Conference on Software Maintenance*, Monterey, California, USA, November 4-8, 1996, pp. 244-253.
- [Mayrhauser 1995] Von Mayrhauser, A., Vans, A., Program Comprehension during Software Maintenance and Evolution, *IEEE Computer*, pp. 44-55, August, 1995.
- [MSDN 2000] *Microsoft Developer Network Library*, October 2000.
- [McCabe 1990] McCabe, T., Reverse Engineering, Reusability, Redundancy: the Connection, *American Programmer*, October 1990, pp.8-13.
- [McCabe 1992] McCabe, T., Williamson, E., An Engineering Approach to Software Maintenance, *CASE OUTLOOK*, Vol. 6(1), 1992, pp. 19-21.
- [McCabe 1999] McCabe & Associates Inc. Official Web Site, Available via <http://www.mccabe.com>.
- [Monden 2002] Monden, A., Nakae, D., Kamiya, T., Sato, S., Matsumoto, K., Software Quality Analysis by Code Clones in Industrial Legacy Software, *8<sup>th</sup> IEEE Symposium on Software Metrics*, Ottawa, Canada, June 4-7, 2002, pp. 87-94.
- [Muthukumarasamy 1995] Muthukumarasamy, J., Stasko, J., Visualizing Program Executions on Large Data Sets Using Semantic Zooming”, GVU Center, College of Computing, Georgia Institute of Technology, *Technical Report GIT-GVU-95-02*, 1995.
- [Parker 1998] Parker, G., Franck, G., Ware, C., Visualization of Large Nested Graphs in 3D: Navigation and Interaction, *Journal of Visual Languages and Computing*, No. 9, 1998, pp. 299-317.
- [Price 1993] Price, B., Baecker, R., Small, I., A Principled Taxonomy of Software Visualization, *Journal of Visual Languages and Computing*, Vol. 4(3), September 1993, pp. 211-266.
- [Rigi 1999] Official Rigi Web Site, Available via <http://www.rigi.csc.uvic.ca/rigi>.
- [Storey 1997] Storey, M., Fracchia, F., Muller, H., Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization, *Proceedings of the Fifth International Workshop on Program Comprehension (IWPC-97)*, IEEE Computer Society Press, Dearborn, Michigan, May 28-30, 1997, pp. 17-28.
- [Storey 1998] Storey, M., Muller, H., Wong, K., Manipulating and Documenting Software Structures, Software Visualization, P. Eades and K. Zhang (eds.), World Scientific Publishing Co., in press. ISBN 981-02-2826-0, Available via <http://tara.uvic.ca/rigi/ref.html>.

[Storey 2000] Storey, M., Wong, K., Muller, H., How Do Program Understanding Tools affect How Programmers Understand Programs?, *Journal of Science of Computer Programming*, Vol. 36, pp.183-207, March 2000.

[Tilley 1996] Tilley, S., Paul, S., Smith, D., Towards a Framework for Program Understanding, *Proceedings of the 4<sup>th</sup> Workshop on Program Comprehension*, Berlin, Germany, March 23-31, 1996, pp. 19-28.

[Ware 1988] Ware, C., Using color Dimensions to Display Data Dimensions, *Human Factors*, 30(2), 1988, pp. 127-142.

[Ware 1993] Ware, C., Hui, D., Franck, Glenn, Visualizing Object Oriented Software in Three Dimensions, *Proceedings of IBM CAS Conference (CASCON'93)*, Toronto, October 24-28, 1993, pp. 612-620.

[Ware 1994] Ware, C., Franck, G., Visualizing Information Nets in Three Dimensions, *University of New Brunswick Technical Report TR94-082*, February 1994.

[Ware 1999] Ware, C., *Information visualization: Perception for Design*, Morgan Kaufman Publishers, 1999.

## *Appendix A – Parser Design*

Token Code	Token Type	Token Description
1	identifier	user defined identifier
2	keyword	a C/C++ keyword or identifier that has special meaning (main, argc, argv, NULL, EOF)
3	library routine name	identifier defined in a standard library file
4	decimal constant	decimal integer
5	real number	floating point number
6	inline comment	from // to end-of-line or EOF
7	C-style comment	/* ... */
8	preprocessor directive	from # to end-of-line, EOF, /*, or //
9	blank	one or more space, tab, or new line char
10	string literal	"..."
11	character constant	'A' or '\n'
13	[	left bracket
14	]	right bracket
15	(	left parenthesis
16	)	right parenthesis
17	&	unary address operator
18	*	binary multiplication or unary indirection operator
19	+	binary addition or unary plus
20	-	binary subtraction operator or unary minus
21	~	bitwise complement (1's)
22	!	unary logical negation
23	%	remainder
24	<	less than
25	>	greater than
26		bitwise OR
27	^	bitwise XOR
28	,	comma separator
29	?	'a ? x : y' = "if a then x, else y"
30	;	semicolon separator
31	:	'a ? x : y' = "if a then x, else y"
32	{	left brace
33	}	right brace
34	.	direct component selector
35	=	assignment operator
36	/	divide
37	.*	pointer to member operator
38	→	indirect component selector
39	++	pre/post increment

Token Code	Token Type	Token Description
40	--	pre/post decrement
41	<<	bitwise shift left or insertion operator (C++)
42	>>	bitwise shift right or extraction operator (C++)
43	<=	less than or equal to
44	>=	greater than or equal to
45	==	equality operator
46	!=	not equal
47	&&	logical-AND operator
48		logical-OR operator
49	*=	multiplication assignment
50	/=	division assignment
51	%=	remainder assignment
52	+=	addition assignment
53	-=	subtraction assignment
54	&=	bitwise AND assignment
55	^=	bitwise exclusive OR assignment
56	=	bitwise inclusive OR assignment
57	#if 0 block	a block of code commented out using conditional compilation: #if 0 statement1 ... statementN #endif
58	::	scope resolution operator
59	->*	pointer to member operator
60	<<=	left shift assignment
61	>>=	Right shift assignment
62	...	ellipsis
63	octal/hexadecimal constant	octal or hexadecimal integer
64	EOF	end-of-file character
65	error	unrecognized unit
66	escape	control sequence (backspace, bell, etc.) excluding what makes up token 9
69	continuation sequence	back slash followed by new line

**Table A.1:** Selected Tokens

Token type	Token Code	Definition
identifier	1	nondigit (nondigit   digit)* nondigit → [a-zA-Z_\$ ] digit → [0-9]
keyword	2	{asm, __asm, __asm__, auto, break, case, catch, _cdecl, __cdecl, cdecl, char, class, const, continue, _cs, default, delete, do, double, _ds, else, enum, _es, _export, explicit, extern, _far, fae, float, for, friend, goto, huge, if, inline, int, interrupt, _loads, long, _near, near, new, operator, _pascal, pascal, private, protected, public, register, return, _saveregs, _seg, short, signed, sizeof, _ss, static, struct, switch, template, this, typedef, union, unsigned, virtual, void, volatile, while, try, throw, typeof, const_cast, static_cast, dynamic_cast, reinterpret_cast, NULL, EOF, main, argc, argv}
library routine name	3	identifier defined in included (standard) library file
decimal constant	4	[digit^0](digit)*(suffix) digit → [0-9] suffix → (suffix1   suffix2) suffix1 → (u   U)?(l   L)? suffix2 → (l   L)?(u   U)?
real number	5	fractional_const(exponent_part)?(suffix)? (digit) <sup>+</sup> exponent_part(suffix)? digit → [0-9] fractional_const → (fr1   fr2) fr1 → (digit) <sup>+</sup> .(digit)* fr2 → .(digit) <sup>+</sup> exponent_part → (e   E)(+   -)(digit) <sup>+</sup> suffix → (l   L   f   F)
inline comment	6	starts with ‘//’ and terminates with end of line or EOF
C-style comment	7	starts with ‘/*’ and terminates with ‘*/’; doesn’t nest
preprocessor directive	8	starts with ‘#’ and terminates with end of line, EOF, ‘/*’ or ‘//’, excluding cases that can be identified under token 57
blank	9	space(space)* space → (sp(32)   nl(10)   ht(9)   vt(11)   ff(12)   cr(13))
string literal	10	sequence (any length) of any characters surrounded by double quotes
character constant	11	one or more characters enclosed in single quotes
[	13	( [ )
]	14	( ] )
(	15	( ( )
)	16	( ) )
&	17	( & )
*	18	( * )

Token type	Token Code	Definition
+	19	(+)
-	20	(-)
~	21	(~)
!	22	(!)
%	23	(%)
<	24	(<)
>	25	(>)
	26	( )
^	27	(^)
,	28	(,)
?	29	(?)
;	30	(;)
:	31	(:)
{	32	({)
}	33	(})
.	34	(.)
=	35	(=)
/	36	(/)
.*	37	(*)
→	38	(→)
++	39	(++)
--	40	(--)
<<	41	(<<)
>>	42	(>>)
<=	43	(<=)
>=	44	(>=)
==	45	(==)
!=	46	(!=)
&&	47	(&&)
	48	(  )
*=	49	(*=)
/=	50	(/=)
%=	51	(%=)
+=	52	(+=)
-=	53	(-=)
&=	54	(&=)
^=	55	(^=)
=	56	( =)
#if 0 block	57	Starting with '#if 0' and terminating with '#endif' with possible nesting of #if ...#endif blocks
::	58	(::)
→*	59	(→*)
<<=	60	(<<==)
>>=	61	(>>==)
...	62	(...)

Token type	Token Code	Definition
octal/hexadecimal constant	63	octal constant: 0(octal_digit)*(suffix) octal_digit $\rightarrow$ [0-7] hexadecimal constant: 0(x   X)(hex_digit)*(suffix) hex_digit $\rightarrow$ [0-9a-fA-F] suffix $\rightarrow$ (suffix1   suffix2) suffix1 $\rightarrow$ (u   U) <sup>?</sup> (l   L) <sup>?</sup> ; suffix2 $\rightarrow$ (l   L) <sup>?</sup> (u   U) <sup>?</sup>
EOF	64	EOF char
error	65	any unrecognizable sequence of chars
escape	66	esc_char(esc_char)* esc_char $\rightarrow$ (bs(8)   bel(7)   etc)
continuation sequence	69	\nl (back slash followed by new line)

**Table A.2** Token definition

Character Class	Character Class Name	Characters
0	LETa	A   a
1	LETb	B   b
2	LETc	C   c
3	LETd	D   d
4	LETe	E   e
5	LETf	F   f
6	LETu	U   u
7	LETl	L   l
8	LETx	X   x
9	LET	[A - Za - z^LETaLETbLETcLETdLETeLETuLETlLETx]
10	ODIGIT	[1-7]
11	RDIGIT	8   9
12	MINUS	-
13	PLUS	+
14	EQU	=
15	TLDA	~
16	EXL	!
17	OTHER	@   `
18	PS	#
19	PERC	%
20	CAP	^
21	AMPS	&
22	STR	*
23	LPER	(
24	RPER	)
25	UNDSC	_
26	LBR	[
27	RBR	]
28	BSLASH	\
29	LBRC	{
30	RBRC	}
31	OR	
32	SEMC	;
33	FSL	/
34	COLON	:
35	DQ	“
36	COMMA	,
37	DOT	.
38	SQ	‘
39	LTHEN	<
40	GTHEN	>
41	QM	?
42	NLINE	nl (ASCII code 10)
43	WHITESPACE	ht(9)   ff(12)   cr(13)   vt(11)
44	ESCAPES	bel(7)   bs(8) {iscontrol() but !isspace()}

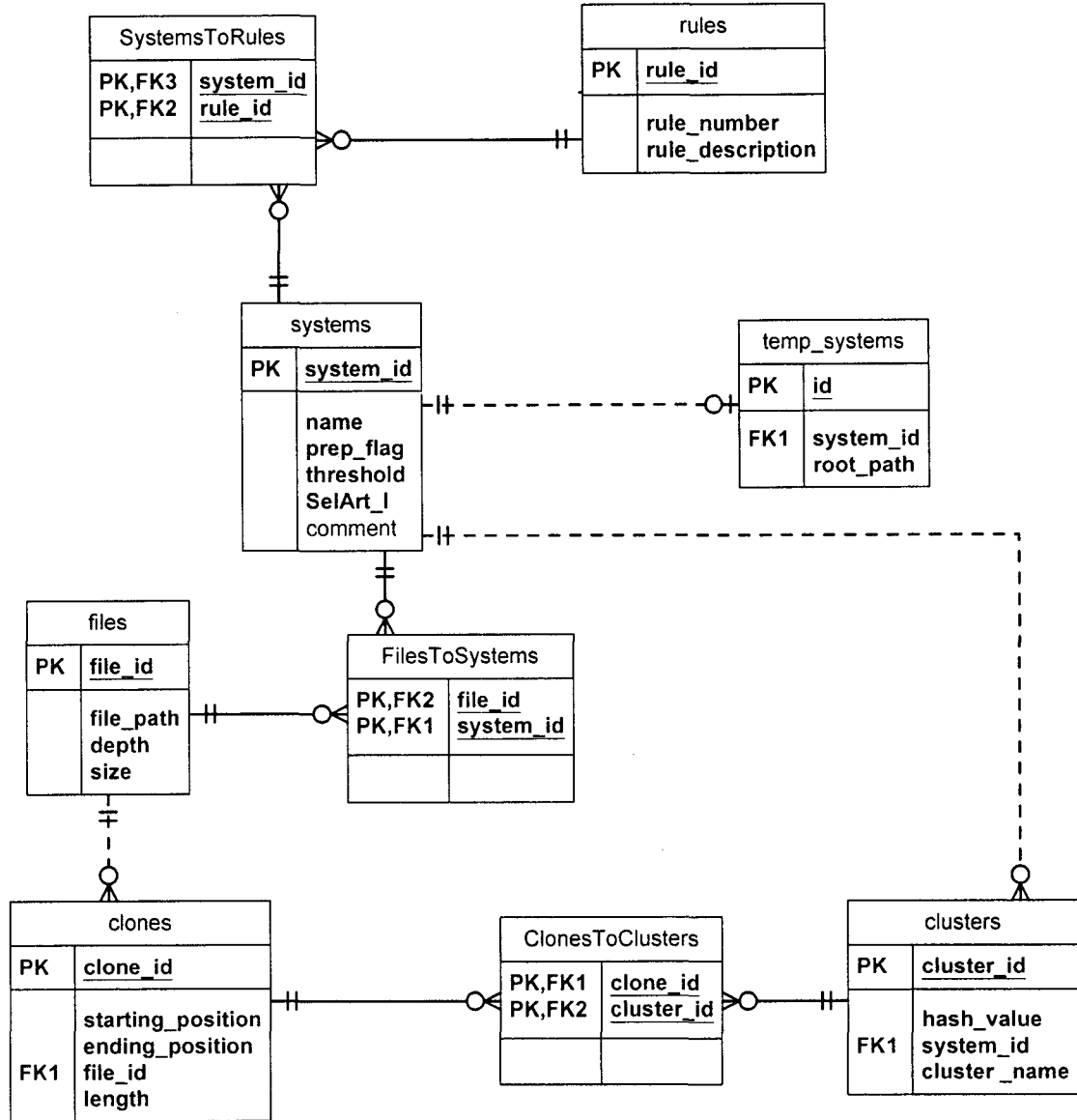


Character Class	Character Class Name	Characters
45	BLANK	sp(32)
46	ZERO	0
47	EOF	eof char
48	\$	\$

**Table A.3** Selected Character Classes

## Appendix B – CloneMaster Data Model

### CloneMaster Data Model



## Data Descriptions

### SYSTEMS table

This table stores information on system entities.

<b>system_id: (PK, Identity)</b>	smallint – Mandatory unique identifier of a system
<b>name:</b>	varchar(256) – Mandatory unique name of a system
<b>prep_flag:</b>	char(1) – Mandatory flag indicating whether or not the system underwent preprocessing {Y, N}
<b>threshold:</b>	tinyint – Mandatory noise threshold (number of lines).
<b>SelArt_l:</b>	smallint – Mandatory target clone size (minimum requested clone size)
<b>Comment:</b>	varchar(1024) – free form comment; Optional

### FILES table

Stores information on file entities.

<b>file_id: (PK, Identity)</b>	int – Mandatory unique file identifier
<b>file_path:</b>	varchar(900) – Mandatory unique path to the file relative to INPUT_DIR
<b>depth:</b>	smallint – Mandatory number of directories in the path
<b>size:</b>	int – Size of file in bytes; Optional

### FILESTOSYSTEMS table

Information on file-system association.

<b>file_id: (PK, FK)</b>	int – Mandatory file identifier
<b>system_id: (PK, FK)</b>	int – Mandatory system identifier

### CLONES table

Stores information about clone entities.

<b>clone_id: (PK, Identity)</b>	int – Mandatory unique clone identifier
<b>starting_position:</b>	int – Mandatory starting line of the clone
<b>ending_position:</b>	int – Mandatory ending line of the clone
<b>file_id:</b>	int – Mandatory file the clone resides in
<b>length:</b>	int – Mandatory length of the clone in number of lines

### CLUSTERS table

Stores information on cluster entities.

<b>cluster_id: (PK, Identity)</b>	int - Mandatory unique cluster identifier
<b>hash_value:</b>	int – Mandatory Unique hash value
<b>cluster_name:</b>	varchar(50) – Mandatory symbolic name to serve as cluster identifier for presentation purposes

**system\_id: (FK)**                      smallint – Mandatory identifier of the corresponding system, Unique

Note: combination of 'hash\_value' and 'system\_id' uniquely identify a cluster

#### **CLONESTOCLUSTERS table**

Information about clone-cluster association.

**clone\_id: (PK, FK)**                      int – Mandatory clone identifier  
**cluster\_id: (PK, FK)**                      int – Mandatory cluster identifier

#### **RULES table**

Stores information on pre-processing rules.

**rule\_id: (PK, Identity)**                      smallint – Mandatory unique rule identifier  
**rule\_number:**                                  tinyint – Mandatory unique rule number  
**rule\_description:**                              varchar(1024) – Mandatory rule description

#### **SYSTEMSTORULES table**

Provides information on pre-processing rules associated with system

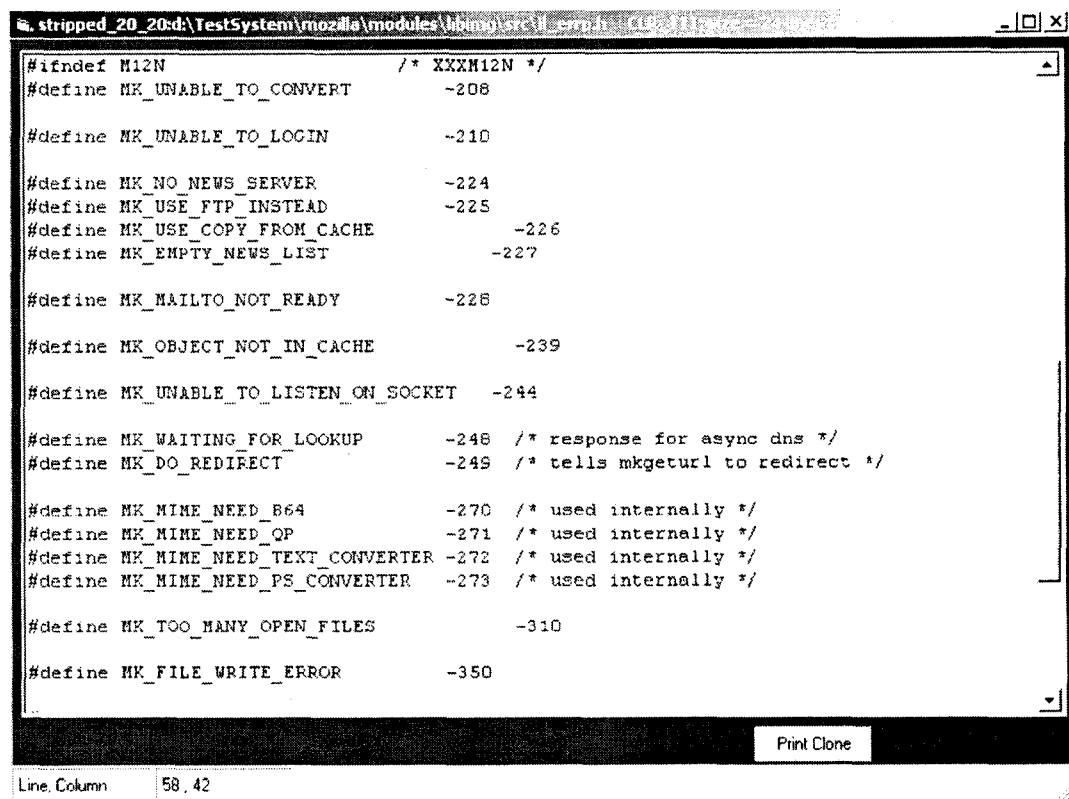
**system\_id: (PK, FK)**                      smallint – Mandatory system identifier  
**rule\_id: (PK, FK)**                              smallint – Mandatory rule identifier

#### **TEMP\_SYSTEMS table**

This dynamic table maintains records on systems currently loaded into visualization tool.

**id: (PK, Identity)**                              smallint – Mandatory unique identifier  
**system\_id: (FK)**                                  smallint – Mandatory identifier of loaded system  
**root\_path:**                                      varchar(900) – Mandatory unique absolute path to the directory where the source code for the loaded system is located

## Appendix C – Experimental Results



```
stripped_20_20sd:\TestSystem\mozilla\modules\libimg\src\il_error.h  C:\C:\Program Files\...
#ifndef M12N /* XXXM12N */
#define MK_UNABLE_TO_CONVERT -208

#define MK_UNABLE_TO_LOGIN -210

#define MK_NO_NEWS_SERVER -224
#define MK_USE_FTP_INSTEAD -225
#define MK_USE_COPY_FROM_CACHE -226
#define MK_EMPTY_NEWS_LIST -227

#define MK_MAILTO_NOT_READY -228

#define MK_OBJECT_NOT_IN_CACHE -239

#define MK_UNABLE_TO_LISTEN_ON_SOCKET -244

#define MK_WAITING_FOR_LOOKUP -248 /* response for async dns */
#define MK_DO_REDIRECT -249 /* tells mkgeturl to redirect */

#define MK_MIME_NEED_B64 -270 /* used internally */
#define MK_MIME_NEED_QP -271 /* used internally */
#define MK_MIME_NEED_TEXT_CONVERTER -272 /* used internally */
#define MK_MIME_NEED_PS_CONVERTER -273 /* used internally */

#define MK_TOO_MANY_OPEN_FILES -310

#define MK_FILE_WRITE_ERROR -350
```

Print Clone

Line, Column 58, 42

**Figure C.1:** Example of a clone consisting entirely of preprocessor directives. It is located in `il_error.h` and defines netlib error return codes to be used by Image Library. Its counterpart resides in `merrors.h` and also defines netlib error return codes.

```
stripped_20_20:d:\TestSystem\mozilla\js\jsd\java\jsd_intvc  CLR 31: 922 70 lines, 2.0 KB
#define ASSERT_RETURN_VOID(x) \
JS_BEGIN_MACRO \
    if(!{x}) \
    { \
        JS_ASSERT(0); \
        return; \
    } \
JS_END_MACRO

#define ASSERT_RETURN_VALUE(x,v)\
JS_BEGIN_MACRO \
    if(!{x}) \
    { \
        JS_ASSERT(0); \
        return v; \
    } \
JS_END_MACRO

#define CHECK_RETURN_VOID(x) \
JS_BEGIN_MACRO \
    if(!{x}) \
    { \
        return; \
    } \
JS_END_MACRO

#define CHECK_RETURN_VALUE(x,v) \
<
```

**Figure C.2:** Example of an exact clone. This clone corresponds to 70 duplicated lines dedicated to declaration of 9 macros. The counterpart of this clone resides in a different directory.

```

};

void TimerImpl::FireTimeout()
{
    if (mFunc != NULL) {
        (*mFunc)(this, mClosure);
    }
    else if (mCallback != NULL) {
        mCallback->Notify(this); // Fire the timer
    }

    // Always repeating here

    // if (mRepeat)
    // mTimerId = gtk_timeout_add(aDelay, nsTimerExpired,
}

TimerImpl::TimerImpl()

```

---

```

};

void TimerImpl::FireTimeout()
{
    if (mFunc != NULL) {
        (*mFunc)(this, mClosure);
    }
    else if (mCallback != NULL) {
        mCallback->Notify(this); // Fire the timer
    }

    // Always repeating here

    // if (mRepeat)
    // mTimerId = XtAppAddTimeOut(gAppContext, GetDelay(),
}

TimerImpl::TimerImpl()

```

**Figure C.3:** An example of two near clones typical to 'parsd1'. The only difference between these two code fragments is the line that is actually commented out (underlined).

```

static jobject
getScriptHook(JNIEnv *env, JSDJContext* jsdjc)
{
    jclass clazz;
    jfieldID fid;

    clazz = (*env)->GetObjectClass(env, jsdjc->controller);
    ASSERT_RETURN_VALUE(clazz, NULL);
    fid = (*env)->GetFieldID(env, clazz, "scriptHook",
                            "Lnetscape/jsdebug/ScriptHook;");
    ASSERT_RETURN_VALUE(fid, NULL);
    return (*env)->GetObjectField(env, jsdjc->controller, fid);
}

static jobject
getInterruptHook(JNIEnv *env, JSDJContext* jsdjc)
{
    jclass clazz;
    jfieldID fid;

    clazz = (*env)->GetObjectClass(env, jsdjc->controller);
    ASSERT_RETURN_VALUE(clazz, NULL);
    fid = (*env)->GetFieldID(env, clazz, "interruptHook",
                            "Lnetscape/jsdebug/InterruptHook;");
    ASSERT_RETURN_VALUE(fid, NULL);
    return (*env)->GetObjectField(env, jsdjc->controller, fid);
}

```

**Figure C.4:** An example of a near clone. The counterpart of this clone is presented in Figure C.5. The differences between these two code fragments are underlined.



```

static jobject
_getErrorReporter(JNIEnv *env, JSDJContext* jsdjc)
{
    jclass clazz;
    jfieldID fid;

    clazz = (*env)->GetObjectClass(env, jsdjc->controller);
    ASSERT_RETURN_VALUE(clazz, NULL);
    fid = (*env)->GetFieldID(env, clazz, "errorReporter",
                            "Lnetscape/jsdebug/JSErrorReporter;");
    ASSERT_RETURN_VALUE(fid, NULL);
    return (*env)->GetObjectField(env, jsdjc->controller, fid);
}

static jobject
_getScriptTable(JNIEnv *env, JSDJContext* jsdjc)
{
    jclass clazz;
    jfieldID fid;

    clazz = (*env)->GetObjectClass(env, jsdjc->controller);
    ASSERT_RETURN_VALUE(clazz, NULL);
    fid = (*env)->GetFieldID(env, clazz, "scriptTable",
                            "Lnetscape/util/Hashtable;");
    ASSERT_RETURN_VALUE(fid, NULL);
    return (*env)->GetObjectField(env, jsdjc->controller, fid);
}

```

**Figure C.5:** Counterpart of the clone from Figure C.4. Both clones reside in the same file.

```

NS_IMETHOD
GetCertData(const unsigned char ***certChain, PRUint32 **certChainLengths, PRUint32 *noOfCerts);

/**
 * Returns the public key of the certificate.
 *
 * @param publicKey - the Public Key data will be returned in this field.
 * @param publicKeySize - the length of public key data is returned in this
 * parameter.
 */
NS_IMETHOD
GetPublicKey(unsigned char **publicKey, PRUint32 *publicKeySize);

/**
 * Returns the company name of the certificate (OU etc parameters of certificate)
 *
 * @param result - the certificate details about the signer.
 */
NS_IMETHOD
GetCompanyName(const char **ppCompanyName);

/**
 * Returns the certificate issuer's data (OU etc parameters of certificate)
 *
 * @param result - the details about the issuer
 */
NS_IMETHOD
GetCertificateAuthority(const char **ppCertAuthority);

/**
 * Returns the serial number of certificate
 *
 * @param result - Returns the serial number of certificate
 */
NS_IMETHOD
GetSerialNumber(const char **ppSerialNumber);

/**
 * Returns the expiration date of certificate
 *
 * @param result - Returns the expiration date of certificate
 */
NS_IMETHOD
GetExpirationDate(const char **ppExpDate);

/**
 * Returns the finger print of certificate
 *
 * @param result - Returns the finger print of certificate
 */
NS_IMETHOD
GetFingerPrint(const char **ppFingerPrint);

```

**Figure C.6:** An example of exact clone reported in ‘original’ and ‘parsed1’, but missed in ‘parsed2’. Figure C.7 shows this block of code in tokenized form as presented to SelArt.

```

ll(constunsignedchar***l,**l,l*l);
ll(unsignedchar**l,l*l);
ll(constchar**l);
ll(constchar**l);
ll(constchar**l);
ll(constchar**l);
ll(constchar**l);

```

**Figure C.7:** Same code fragment (Figure C.6) after pre-processing (tokenized). Note: lines 3 through 7 contain periodic repetition of `<ll(constchar**l);>` substring. SelArt reports 5 short matches of this substring instead of a proper long match containing entire string. Vertical bars on the right hand side in Figure C.6 delimit these detected (near) matches. Depending on the noise threshold value, these matches may be filtered out from the final result set.

```

//helper function
static jobject
_getObject(JNIEnv *env, JSJContext* jsdjc, const char *object_name,
           const char *object_path)
{
    jclass clazz;
    jfieldID fid;

    clazz = (*env)->GetObjectClass(env, jsdjc->controller);
    ASSERT_RETURN_VALUE(clazz, NULL);
    fid = (*env)->GetFieldID(env, clazz, object_name, object_path);

    ASSERT_RETURN_VALUE(fid, NULL);
    return (*env)->GetObjectField(env, jsdjc->controller, fid);
}

```

**Figure C.8:** Example of restructuring. Step 1: extracting a helper function `_getObject()` from clones shown in Figures C.4 and C.5.

```

static jobject
_getScriptHook(JNIEnv *env, JSJContext* jsdjc)
{
    return _getObject(env, jsdjc, "scriptHook", "Lnetscape/jsdebug/ScriptHook;");
}

static jobject
_getInterruptHook(JNIEnv *env, JSJContext* jsdjc)
{
    return _getObject(env, jsdjc, "interruptHook", "Lnetscape/jsdebug/InterruptHook;");
}

static jobject
_getErrorReporter(JNIEnv *env, JSJContext* jsdjc)
{
    return _getObject(env, jsdjc, "errorReporter",
                     "Lnetscape/jsdebug/JSErrorReporter;");
}

static jobject
_getScriptTable(JNIEnv *env, JSJContext* jsdjc)
{
    return _getObject(env, jsdjc, "scriptTable", "Lnetscape/util/Hashtable;");
}

```

**Figure C.9:** Example of restructuring. Step 2: Replacing clones with newly defined helper function.

## ***Appendix D – Support Tools Help***

### **Step 1: Discovery of directory structure**

**Purpose:** Recursively traverse the source code directory tree to:

1. Enumerate all C/C++ header/source files (i.e., .C, .c, .h, .cpp, .cc, .cxx, .c++) encountered in the system.
2. Delete any files other than these from the source code directory tree.

**Usage:** *grinder\_clean\_and\_log* <result\_file>

- <result\_file> - name of the file to store records about paths of C/C++ header/source files encountered in the system.

Upon invocation, the user will be prompted for the location of the source code tree to be analyzed (*INPUT\_DIR*). Upon completion, every file in *INPUT\_DIR* will have a corresponding entry in the *result\_file*. Information collected in the *result\_file* will be used to drive steps 2a/b and 3.

### **Step 2a<sup>23</sup>: ‘Line-oriented’ to ‘stream’ input conversion**

**Purpose:** Converts original (i.e., line-oriented) source code into a stream form suitable for exact clone identification with SelArt. This is achieved simply by discarding all new line characters.

**Usage:** *stripper* <list>

---

<sup>23</sup> Steps 2a and 2b are mutually exclusive. 2a precedes exact matching, whereas 2b precedes near matching.

- *<list>* - name of the file created in step 1 (i.e., *<result\_file>*) containing the list of files to be processed.

Upon invocation, the user will be prompted for the following input:

- *INPUT\_DIR* – absolute/relative path to a source code tree containing actual files to be processed (i.e., *INPUT\_DIR* from step 1).
- *OUTPUT\_DIR* – absolute/relative path to a directory for capturing output (doesn't have to exist). Contains “stripped” source to be used for clone identification. File names/paths are derived from the original file names/paths by appending the '.S' extension.
- *MAPPING\_DIR* – absolute/relative path to a directory for collecting supporting statistics (doesn't have to exist). File names/paths are derived from the original file names/paths by appending the '.pos' extension (section 4.3.3.3).

## **Step 2b: Pre-processing**

**Purpose:** Converts original (i.e., line-oriented) source code into some intermediate stream representation suitable for near clone identification with SelArt. This is achieved via tokenizing on the lexical level and applying different transformation rules to the tokens.

**Usage:** *parser <buffer\_size> <max\_token\_size> <list>*

- *<buffer\_size>* - size of input/output buffers to be allocated for the purposes of lexical analysis (recommended value: 1024).
- *<max\_token\_size>* - max length of a token lexeme to be remembered (recommended value: 1024).

- *<list>* - name of the file created in step 1 enumerating files to be processed.

Upon invocation, the user will be prompted for the following information:

- *INPUT\_DIR* - absolute/relative path to a source code tree containing files to be pre-processed (*i.e.*, *INPUT\_DIR* from step 1).
- *OUTPUT\_DIR* - absolute/relative path to a directory for capturing output (doesn't have to exist). Contains pre-processed source to be used for clone identification. File names/paths are derived from the original file names/paths by appending the '.U' extension.
- *MAPPING\_DIR* - absolute/relative path to a directory for collecting supporting statistics (doesn't have to exist). File names/paths are derived from the original file names/paths by appending the '.pos' extension (section 4.3.3.3).
- '*Nesting Switch*' (*YES/NO*) – allows to control 'nesting of #if0 ...#endif' feature (recommended value: YES).
- '*Transformation Rules Switch*' – allows to control transformation rules. NOTE: Some transformation rules are enabled unconditionally, others need to be turned ON/OFF explicitly.

During operation of step 2a/b, the values of such statistics as the average number of characters per line (*ANCPL*) and compression rate (*CR*) are calculated. They are printed to the standard out upon termination and will be used in step 3.

### Step 3: Clone identification with SelArt

**Purpose:** ‘Exact’ or ‘exact’-plus-‘near’ clone detection.

**Usage:** *clones* <-M###> <-m###> <-l###> <-c###> <target\_directory\_path>

- <target\_directory\_path> - an absolute path to a directory to be analyzed (i.e., *OUTPUT\_DIR* from step 2a/b).
- <-l> - a minimum target clone size expressed via number of lines. All clones of size *l* and greater are guaranteed to be found.
- <-M> - a minimum target clone size expressed via number of characters calculated based on *l* using formula 4.3 and values of *ANCPL* and *CR* from step 2a/b.
- <-m> - *m* always equals *M*.
- <-c> - *c* always equals *M*.

SelArt requires a lot of disk space to run. Otherwise, some bizarre failures could occur. While running, SelArt creates and deletes a number of intermediate files in the current working directory. It also maintains a log file that can be used to monitor the progress (i.e., *tail -f log*). Out of all files left in the working directory after SelArt terminates, only one, *grpl.l*, is useful as it contains clone information; the rest may be deleted.

### Step 4: Post\_processing

**Purpose:** Parse results of clone identification (i.e., *grpl.l* file from step 3) to:

1. Extract information about clones with sizes exceeding some threshold.
2. Convert clones’ boundaries into meaningful representation.



**Usage:** *grpll\_cruncher*

Upon invocation, the user will be prompted for the following information:

- *'grpl.l file'* – a path to a grpl.l file (i.e., from step 3) to be processed.
- *OUTPUT\_DIR* – a path to a directory that has been subjected to clone identification (i.e., *'target\_directory\_path'* of step 3).
- *MAPPING\_DIR* – a path to a directory containing supporting statistics (i.e., *MAPPING\_DIR* from step 2a/b).
- *size\_threshold* (in number of lines) – specifies clone size threshold. Only information about clones with sizes greater than this threshold is retained.
- *output\_file\_name* – name of the file to capture the output of this post-processing (recommended value: *'clones.fin'*). This file will be used in CloneMaster data base population (section 5.4.2).

## *Vita*

Candidate's Full Name: Irina Padioukova

Place and data of birth: Moscow, Russia  
April 13, 1968

Permanent Address: 172-2 Princess street  
Saint John, NB  
E2L 1K9, Canada

Universities attended: Moscow State Institute of Steel and Alloys  
(Technical University)  
September 1985 – February 1991  
Diploma Metallurgical Engineering

University of New Brunswick  
Fredericton, New Brunswick  
September 1996 – January 2003  
Candidate for MCS