# Visualization and Analysis of Software Clones

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Muhammad Asaduzzaman

©Muhammad Asaduzzaman, January 2012. All rights reserved.

# Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

# Abstract

Code clones are identical or similar fragments of code in a software system. Simple copy-paste programming practices of developers, reusing existing code fragments instead of implementing from the scratch, limitations of both programming languages and developers are the primary reasons behind code cloning. Despite the maintenance implications of clones, it is not possible to conclude that cloning is harmful because there are also benefits in using them (e.g. faster and independent development). As a result, researchers at least agree that clones need to be analyzed before aggressively refactoring them. Although a large number of state-of-the-art clone detectors are available today, handling raw clone data is challenging due to the textual nature and large volume. To address this issue, we propose a framework for large-scale clone analysis and develop a maintenance support environment based on the framework called VisCad. To manage the large volume of clone data, VisCad employs the Visual Information Seeking Mantra: overview first, zoom and filter, then provide details-on-demand. With VisCad users can analyze and identify distinctive code clones through a set of visualization techniques, metrics covering different clone relations and data filtering operations. The loosely coupled architecture of VisCad allows users to work with any clone detection tool that reports source-coordinates of the found clones. This yields the opportunity to work with the clone detectors of choice, which is important because each clone detector has its own strengths and weaknesses. In addition, we extend the support for clone evolution analysis, which is important to understand the cause and effect of changes at the clone level during the evolution of a software system. Such information can be used to make software maintenance decisions like when to refactor clones. We propose and implement a set of visualizations that can allow users to analyze the evolution of clones from a coarse grain to a fine grain level. Finally, we use VisCad to extract both spatial and temporal clone data to predict changes to clones in a future release/revision of the software, which can be used to rank clone classes as another means of handling a large volume of clone data. We believe that VisCad makes clone comprehension easier and it can be used as a test-bed to further explore code cloning, necessary in building a successful clone management system.

# Acknowledgements

First of all, I would like to express my heart-felt and most sincere gratitude to my respected supervisors Chanchal K. Roy and Kevin A. Schneider for their constant guidance, advice, encouragement and extraordinary patience during this thesis work. This thesis would not have been possible without them.

I would like to thank Julita Vassileva, Christopher Dutchyn, and Shahedul A. Khan who have generously given their time and expertise to better my work.

I am indebted to the members of the Software Research Lab for their support and inspiration. In particular, I would like to thank Minhaz Fahim Zibran, Ripon Kumar Saha, Mohammad Asif Ashraf Khan, Md. Sharif Uddin, Md. Saidur Rahman, Khalid Billah, Manishankar Mondal, and Avigit Saha.

I am also grateful to Department of Computer Science, the University of Saskatchewan for their generous support through scholarship, awards and bursaries that helped me to concentrate more deeply on my thesis work.

I would like to thank all of my friends and other staff members of the Department of Computer Science who have helped me in one way or another along the way. In particular, I would like to thank Janice Thompson, Gwen Lancaster, Maureen Desjardins, and Heather Webb.

I express my gratefulness to my family members and relatives especially my mother Ayesha Akhtara Begum, my father Md. Abdul Jalil Miah, my brother Muhammad Ahasanuzzaman, my uncle Motahar Hossain, Atahar Hossain, and Mozahar Hossain.

Last but no means least, I thank my friends in Bangladesh for their constant support and encouragement. In particular I would like to thank Matiur Rahman, Sazzad Hasan, Ikhtear Sharif, Ashraf Mohammed Iqbal, and Mizanur Rahman.

For those that I have not listed explicitly, thank you for being a part of this thesis and helping me grow as a person and a researcher.

This thesis is dedicated to my mother Ayesha Akhtara Begum, who always be the source of inspiration to me.

# Contents

**Permission to Use** i

**Abstract** ii

**Acknowledgements** iii

**Contents** v

**List of Tables** viii

**List of Figures** ix

**List of Abbreviations** x

**1 Introduction** 1
- 1.1 Motivation . . . 1
- 1.2 Problem Statement . . . 2
- 1.3 Contributions of the Thesis . . . 3
- 1.4 Publications . . . 3
- 1.5 Outline of the Thesis . . . 4

**2 Background and Related Work** 5
- 2.1 Clone Terminology . . . 5
  - 2.1.1 Code Fragment . . . 5
  - 2.1.2 Code Clone . . . 5
  - 2.1.3 Clone Types . . . 6
  - 2.1.4 Clone Relations . . . 7
- 2.2 Why are there Clones in Software Systems? . . . 8
- 2.3 Negative Impacts of Clones . . . 9
- 2.4 Clone Detection . . . 12
  - 2.4.1 Text-based Approaches . . . 12
  - 2.4.2 Lexical Approaches . . . 12
  - 2.4.3 Graph-based Approaches . . . 13
  - 2.4.4 Syntax-Tree based Approaches . . . 13
  - 2.4.5 Metric-based Approaches . . . 13
  - 2.4.6 Hybrids . . . 13
- 2.5 Supporting Clone Analysis and Visualization . . . 15
- 2.6 Code Clone Evolution . . . 16
  - 2.6.1 Clone Genealogy Model . . . 16
  - 2.6.2 Clone Evolution Analysis . . . 18
  - 2.6.3 Supporting Clone Evolution Comprehension . . . 20
- 2.7 Conclusion . . . 23

**3 Supporting Large Scale Clone Analysis: A Pragmatic Approach** 24
- 3.1 Introduction . . . 24
- 3.2 A Framework for Clone Analysis . . . 25
  - 3.2.1 Import Clone Candidates . . . 26
  - 3.2.2 Global Filtering . . . 26
  - 3.2.3 Categorization and Presentation of Clone Data . . . 27
  - 3.2.4 Goal Selection . . . 27

3.2.5 View Data . . . 27
3.2.6 Analyze and Annotate the Clone Data . . . 27
3.2.7 Refine the Goal . . . 28
3.2.8 Local Filtering . . . 28
3.2.9 Make Decision . . . 28
3.3 From Design to Implementation . . . 28
3.3.1 Supporting Multiple Clone Detectors . . . 30
3.3.2 Visualization of Code Clones . . . 31
3.3.3 Metrics Supported by VisCad . . . 34
3.3.4 Filtering . . . 36
3.4 Analyzing Clones with VisCad . . . 38
3.4.1 Subject Systems and Research Questions . . . 38
3.4.2 Clone Detection . . . 39
3.4.3 Preparing for Analysis . . . 39
3.4.4 Answering Questions . . . 40
3.5 Usability Evaluation . . . 44
3.6 Related Work . . . 44
3.7 Conclusion . . . 46

**4 Improving Clone Evolution Comprehension: A Visual Endeavour 48**
4.1 Motivation . . . 48
4.2 Clone Genealogy Extractor . . . 49
4.2.1 Methodology . . . 49
4.2.2 Text Similarity . . . 51
4.2.3 Combine Similarity . . . 51
4.2.4 Mapping Clone Classes . . . 51
4.2.5 Mapping Clone Fragments . . . 52
4.2.6 Automatic Change Pattern Classification . . . 52
4.3 Requirement Identification . . . 53
4.4 Data Collection and Metrics Computed . . . 54
4.5 Mapping Data to Visual Entities . . . 54
4.5.1 Clone Class Evolution . . . 55
4.5.2 Clone Fragment Evolution . . . 56
4.5.3 Clone File Evolution . . . 57
4.5.4 Line-based Evolution . . . 58
4.5.5 Developers Cloning Profile . . . 58
4.6 Data Filtering . . . 60
4.7 Case Study . . . 62
4.8 Related Work . . . 64
4.8.1 Software Evolution Visualization . . . 64
4.8.2 Clone Evolution Visualization . . . 66
4.9 Conclusion . . . 67

**5 Which Clones to Look at First? 68**
5.1 Introduction . . . 68
5.2 Motivating Scenario . . . 69
5.3 Code Clone Genealogy . . . 69
5.4 Subject Systems . . . 70
5.5 Data Collection . . . 71
5.6 Predicting the Changes of a Clone Class . . . 73
5.6.1 Attribute Subset Selection . . . 74
5.6.2 Evaluating the Predictor . . . 76
5.6.3 Discussion . . . 77
5.7 Related Work . . . 80

5.8 Threats to Validity . . . 81
5.9 Conclusion . . . 82

**6 Conclusion** **83**
6.1 Summary . . . 83
6.2 Future Work . . . 85
6.2.1 Improvement of VisCad . . . 85
6.2.2 Controlled Experiment . . . 85
6.2.3 IDE Integration . . . 85
6.2.4 Clone Filtering . . . 85
6.2.5 Extended Empirical Study . . . 86
6.2.6 Clone Genealogy Extractor . . . 86
6.2.7 VisCad as a Generic Framework . . . 86
6.3 Conclusion . . . 86

**References** **87**

**A VisCad User Guide** **98**

# List of Tables


2.1 Categorization of clone presentation techniques . . . . . . . . . . 14

3.1 Features supported by VisCad . . . . . . . . . . 30

4.1 Metrics considered in Clone Evolizer . . . . . . . . . . 55

5.1 The set of studied subject systems . . . . . . . . . . 71
5.2 Categorization of the clone classes . . . . . . . . . . 74
5.3 Top ranked 10 attributes selected by the attribute selection model . . . . . . . . . . 74
5.4 Precision and recall value for the classifiers . . . . . . . . . . 75
5.5 Weighted average precision and recall values while considering top 20 ranked attributes . . . . 79
5.6 Attributes categorized into two groups . . . . . . . . . . 79
5.7 Rules used to create a binary classifier . . . . . . . . . . 79
5.8 Weighted average precision and recall for a binary classifier . . . . . . . . . . 80

# List of Figures


2.1 An example of a code clone . . . 6
2.2 A Type 1 clone pair . . . 7
2.3 A Type 2 clone pair . . . 8
2.4 A Type 3 clone pair . . . 9
2.5 A Type 4 clone pair . . . 10
2.6 An example of a code clone genealogy . . . 17
2.7 A clone genealogy marked with different change patterns . . . 18

3.1 Clone Analysis Framework . . . 26
3.2 VisCad Interface . . . 29
3.3 VisCad Input File Format . . . 31
3.4 Scatter plot . . . 32
3.5 An effect of applying higher degree of abstraction in clone detection . . . 38
3.6 Analyzing cloning relation with a scatter plot . . . 40
3.7 Cloning relation of NetBSD5.1/dev/ic subdirectory . . . 41
3.8 A treemap showing the distribution of clones for a clone class . . . 42

4.1 Clone Genealogy Extractor . . . 50
4.2 Data collection and view generation . . . 56
4.3 Encoding different metric values within an entity . . . 57
4.4 A clone class evolution view . . . 58
4.5 A fragment evolution view . . . 59
4.6 A line evolution view . . . 60
4.7 Developers cloning profile . . . 61
4.8 An example of a clone class evolution view with a filter composition panel . . . 63
4.9 An example of a fragment evolution view . . . 64
4.10 An example of a clone file evolution view . . . 65

5.1 Building and using the predictor . . . 78

# List of Abbreviations

| | |
|---|---|
| CGE | Clone Genealogy Extractor |
| LCS | Longest Common Subsequence |
| LOC | Lines of Code |
| DQ | Dynamic Query |

# Chapter 1

# Introduction

## 1.1 Motivation

In today's information-based community, software has become a ubiquitous entity and is an integral part of our life. Due to the advancement of technology, writing source code for a software system is no longer the most difficult part of software development. Rather, software maintenance and software evolution have become the more challenging parts, often characterized as inevitable activities. The term software maintenance refers to the modification of a software product after delivery to correct faults, to improve performance or other attributes or to make changes to adapt it into a new environment [16]. On the contrary, evolution refers to the process of updating a software rapidly after its initial development for various reasons. The inevitable characteristics of evolution are also documented in the literature [101]. Although for small systems, maintenance and evolution may not be an issue, for large software systems their effects cannot be ignored. It has been found that 90% of the costs of developing a typical software system is spent on the maintenance phase [35], which indicates there is a need for state-of-the-art techniques, methods and tools to support maintenance and evolution.

Programmers often copy code fragments or other artefacts (such as design documentation) and then paste them with or without modification. Such similar code fragments or artefacts are known as software clones or code clones. Due to the prevalent copy-paste programming practices of developers, clones are inevitable in software development. Previous studies have reported that the amount of cloning in software systems ranges from 5-15% [135] and can be even 50% of the code base [126]. Although a number of positive impacts of clones have been identified [88], their negative impacts cannot be ignored (e.g. increased program size, update anomalies). If a code fragment contains a bug, all other fragments that are copied from it may also suffer from the same problem. Fixing the bug requires the developer to check and update all copied locations as necessary. Enhancing a code fragment also requires the developer to look for its duplicated code fragments to ensure that changes are propagated to all desired locations, which also multiplies the work need to be done [103]. Thus, clones are considered a "bad smell" in software products and are a major contributor to project maintenance difficulties [28, 83].

Despite considerable debate to the positive and negative impacts of clones, there is a general consensus that, at least, clones need to be detected and analyzed. Thus, clone detection and analysis has become an

integral part of software maintenance. It has been also found that the evolutionary characteristics of clones may provide possibilities to better comprehend and to manage code clones [91]. As a result clone evolution analysis has also become an important part of managing code clones.

## 1.2 Problem Statement

Despite the improvements on clone detection techniques and the availability of many state-of-the-art clone detectors, their benefits cannot be realized without the proper support of clone analysis, which starts from where the detection phase ends. Such support not only helps in understanding the cloning status of the systems rapidly, but also allows maintenance engineers to make proper decisions when managing clones. By surveying recent work on clone detection and analysis, we have identified the following research problems:

- Although a number of techniques and tools have been proposed for analyzing clones [135], there is a lack of a generic framework that captures the various aspects and requirements of clone analysis, and that can guide us in analyzing clones.

- The results obtained from clone detectors are textual in nature and large in volume. These two properties pose significant challenges for clone analysis. Although a number of support environments are available [87, 150, 159], they are either incomplete or specific to a clone detection tool. These force developers or researchers to select a particular clone detection tool regardless of its strengths and weaknesses for a particular task at hand. By separating analysis from detection, we can give freedom to the researchers and tool developers to work with the clone detector of their choice.

- Although supporting software evolution through visualization has long been studied, there is a marked lack of clone evolution visualization. During our work on clone genealogies of 17 open source large software systems [138] we faced the problem of comprehending genealogies due to the lack of such support. Adding support for answering questions related to clone evolution helps us better understand trends and changes in evolution and to correlate them with the metrics from a software project.

- A previous study by Kim et al. [91] suggests that removing clones from a system requires informed decisions. Many clones in the systems are short-lived and are not suitable candidates for refactoring, while many have undergone repetitively the same editing operations or never changed during evolution. Recently, Cai and Kim [37] found that the evolutionary characteristics of clones can help us predict the survival time of clones in a system. However, it is not yet clear whether the evolutionary changes of clones can also be predicted, and if the answer is yes, which set of variables can help us best in making the decisions or to what extent they can support our decisions. Can we predict the evolutionary change patterns of clones that can guide developers or maintenance engineers in selecting those clones that require special attention?

## 1.3 Contributions of the Thesis

The contributions of the thesis are three-fold, as follows:

- First, to support large-scale clone analysis, we propose a framework that captures various aspects and requirements of clone analysis and based on the framework we develop VisCad, a support environment for clone analysis with which users can visualize and analyze large volume of raw cloning data in an interactive fashion.

- Second, to support clone evolution analysis we propose a set of visualization techniques and filtering mechanisms. We also develop a clone genealogy extractor (CGE) that takes the clone detection result of CCFinder and maps clone classes across the versions. We have also extended VisCad with the proposed visualizations and filtering mechanism that works on top of the raw clone evolution data obtained from our CGE.

- Third, we present an empirical study to determine how clones are changed during the evolution of a software system using five open source Java systems. We consider a large number of metrics and determine their relation to the evolutionary change patterns of clones. Our study reveals the possibility of developing a prediction model to determine the type of evolutionary changes that affect the clones in the future. We believe that the results can be used in ranking clones, and thus allow developers or maintenance engineers to look for clones that need more attention during software development, maintenance and evolution.

## 1.4 Publications

The early work on VisCad, our clone analysis support environment has already been published. In this publication, I was the primary author and conducted the research under the supervision of Chanchal K. Roy and Kevin A. Schneider.

- VisCad: flexible code clone analysis support for NiCad. Published in 5th international workshop on Software clones, 2011 [24].

In addition to the above article, I also co-authored several publications during my thesis work related to code clones. The challenges faced and lessons learned from these publications motivated me in addressing the various problems discussed in this thesis.

- Analyzing and forecasting near-miss clones in evolving software: An empirical study. Published in 16th IEEE International Conference on Engineering of Complex Computer Systems, 2011 [113].

- Evaluating Code Clone Genealogies at Release Level: An Empirical Study. Published in 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), 2010 [138].

## 1.5 Outline of the Thesis

In this chapter we have motivated the research problems in supporting software clone analysis and visualization along with the contributions of the thesis. The remaining chapters of the thesis are organized as follows:

- In Chapter 2, we introduce relevant terminology and outline the related work that build the foundation of the thesis.
- Chapter 3 presents a framework for supporting large scale code clone analysis and describes VisCad, the prototype tool we develop based on the framework.
- In Chapter 4, we describe a clone genealogy extractor (CGE) that maps clone classes across program versions. To support clone evolution analysis we propose a set of visualization techniques. Furthermore, we extend VisCad by implementing those techniques that work on top of our developed CGE.
- Chapter 5 presents an empirical study on a set of open source Java systems that describes a technique to develop a predictive model supporting selection of clones that require careful observation.
- Chapter 6 concludes the thesis along with some directions for future research.

# Chapter 2

# Background and Related Work

This chapter presents related terminology and summarizes previous studies on code clones in the literature that build the foundation of our work.

## 2.1 Clone Terminology

Clone detection tools usually accept source codes as input and report the clone detection results either in clone pairs or group similar clone fragments into clone classes. A clone relation is established between two code fragments if they are related by some definition of similarity [32]. In the following, we define a set of terminologies that frequently appear in code clone literature.

### 2.1.1 Code Fragment

A code fragment consists of a sequence of lines of code (with or without comments and blank lines). It can be identified by the tuple $(f,\ s,\ l)$ where $f$ is the file name, $s$ is the start line number, and $l$ is the total number of lines. Thus, the end line number of the code fragment is calculated as $s + l - 1$. Alternatively, a code fragment can also be identified with the tuple $(f,\ s,\ e)$ where $e$ is the end line number. In the later case, the length of the code fragment is calculated as $e - s + 1$. When intermediate representation of the source code is employed (such as the token representation), code fragment is identified with file name, start and end token numbers.

### 2.1.2 Code Clone

Two code fragments that are similar to each other by some definition of similarity are called clones to each other [32] (see Figure 2.1). Similarity between the fragments can be defined based on their textual representations (what the code fragment contains) or on their functionality (what the code does). The granularity of the code fragments also dictates the granularity of the clones and can be function, arbitrary block, and structural block. Two clone fragments may or may not be disjunct. It is possible that they have some regions common between them. Let, $CF_1 = (f,\ s_1,\ l_1)$ and $CF_2 = (f,\ s_2,\ l_2)$ are two clone fragments. They can be partially overlapped or any one of them can be located completely within another. The length of the shared region can be calculated as $|min(s_1 + l_1,\ s_2 + l_2) - max(s_1,\ s_2)|$.

Source: jabref-2.7.1/src/java/net/sf/jabref/EntryEditor.java
Start Line: 1504 End Line: 1508

```
public ChangeTypeAction(BibtexEntryType type, BasePanel bp) {
            super(type.getName());
            this.type = type;
            panel = bp;
        }
```

Source: /jabref-2.7.1/src/java/net/sf/jabref/RightClickMenu.java
Start Line: 364 End Line: 368

```
public ChangeTypeAction(BibtexEntryType type, BasePanel bp) {
            super(type.getName());
            this.type = type;
            panel = bp;
        }
```

**Figure 2.1:** An example of a code clone. Clones were detected by NiCad-2.7 and the example is taken from JabRef system (Version 2.7.1)

### 2.1.3 Clone Types

When the similarity in code clones is defined through the textual representation, three types of clones are identified [136]. The fourth type of clone is defined through functional similarity. They are listed as below.

- Type 1: Code fragments that are exact similar (identical in their textual representation) to each other ignoring the variations in comments, whitespace and layout. They are also known as exact clones. Figure 2.2 shows an example of a Type 1 clone pair.

- Type 2: Identical code fragments except for variations in identifier names, their types, literals, comments, whitespace and layout. An example of a Type 2 clone pair is shown in Figure 2.3.

- Type 3: In addition to the variations in identifier names, their types, literals, comments, whitespace and layout, further modifications in code fragments such as line additions, deletions and modifications are also supported (see Figure 2.4).

  It should be noted that Type 2 and Type 3 clones are collectively known as near-miss clones and are often created through the copy-paste programming practices.

- Type 4: Clones that are functionally equivalent (do the same thing), although the textual representation of the code fragments might vary significantly. Figure 2.5 shows an example of a Type 4 clone pair.

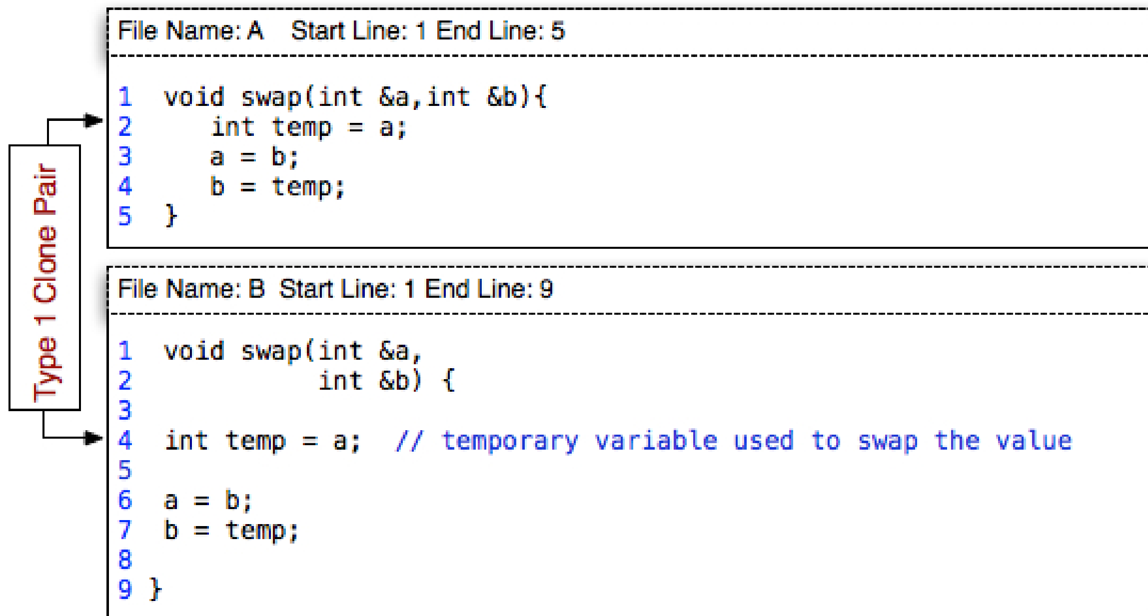



**Figure 2.2:** An example of a Type 1 clone pair. The first clone fragment is shown in the upper part of the figure and belongs to the file A. The start and end line numbers are 1 and 5 respectively. The second clone fragment is shown in the bottom of the figure. It resides in the file B and ranges from line number 1 to 9. The textual content is identical ignoring the variations in comment, whitespace and layout.

### 2.1.4 Clone Relations

Clone detection tools either report individual clone fragments or use clone relations to group small code fragments into large entities, so that maintenance engineers can have a better understanding of the cloning status of the system instead of being overwhelmed by the small clones. The following clone relations are predominant in code clone literature.

**Clone Pair:** Two code fragments form a clone pair if their exists any clone relation between them. If textual similarity is used to define the code similarity, code fragments must be either identical or near similar to each other. Usually, clone detection tools use a predefined threshold value to determine the clone pairs that are near similar to each other. For example, NiCad can be configured to detect two code fragments as a clone pair if at least 90% of the pretty-printed text lines are the same between them.

**Clone Class:** A clone class is the maximal set of code fragments where any two fragments form a clone pair. Most clone detection tools report clones by grouping them into clone classes, also known as clone group or clone cluster.

**Super Clone:** The set of clone classes that belong to the same region form a super clone, also known as clone class family. In other word, super clone is the aggregation of the clone classes cross-cut in the same region. The region can be a file, directory, function, class or a package.

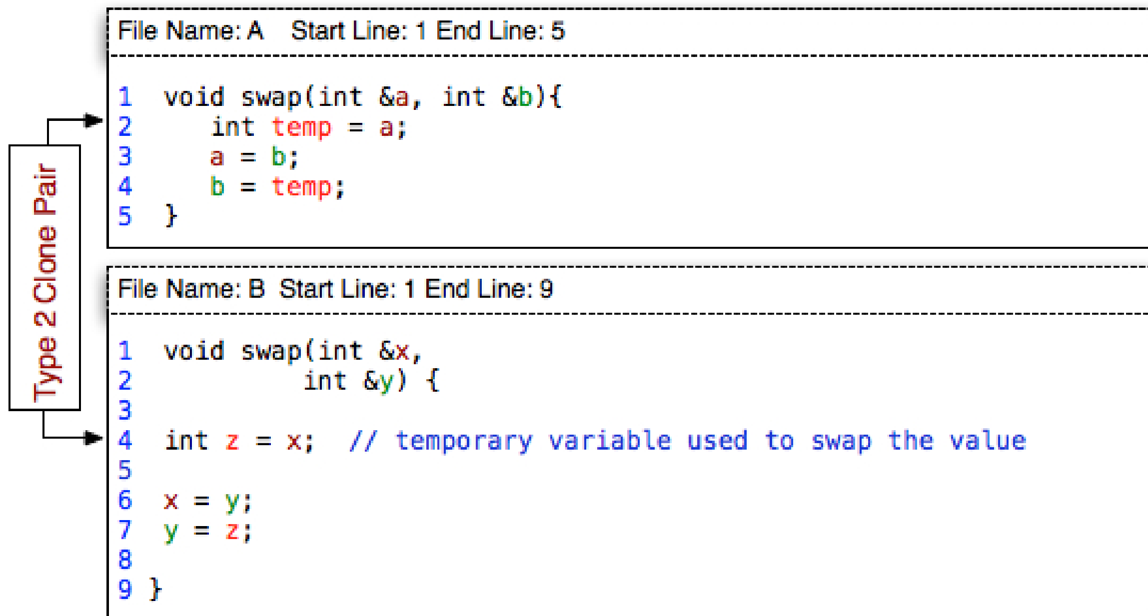



**Figure 2.3:** An example of a Type 2 clone pair. In addition to the variations in comment, whitespace and layout, the variable names are consistently changed in the second code fragment.

## 2.2 Why are there Clones in Software Systems?

In general, software systems contain a significant amount of cloned code and the amount of cloning varies depending on the domain and origin of software systems [135]. There are various factors for which clones can be introduced in a system [27, 32, 90, 93, 125]. Developers may create clones to meet project deadlines because they may not have enough time to create proper abstractions or the short-term cost of creating those abstractions may outweigh the benefits of creating the duplicated code. Developers may also create clones to solve the same problems they encountered before but reluctant to do abstraction because they do not understand the solution, do not have enough time or simply because they do not care about the impact of making clones. Toomim et al. [148] identified a set of cases that makes the abstraction costly and leads programmers to leave the cloned code instead. Kapser et al. [88] identified a set of eight cloning patterns that explain the motivation of cloning, and also listed their advantages and disadvantages. A comprehensive list of factors that introduce cloning can be found in the survey by Roy and Cordy [135]. They categorized the reasons for cloning in the following four groups.

#### Development strategy

Clones can be created due to simple copy-paste programming practices or reusing code, design, logic and functionality, or because of the programming approaches employed during software development.

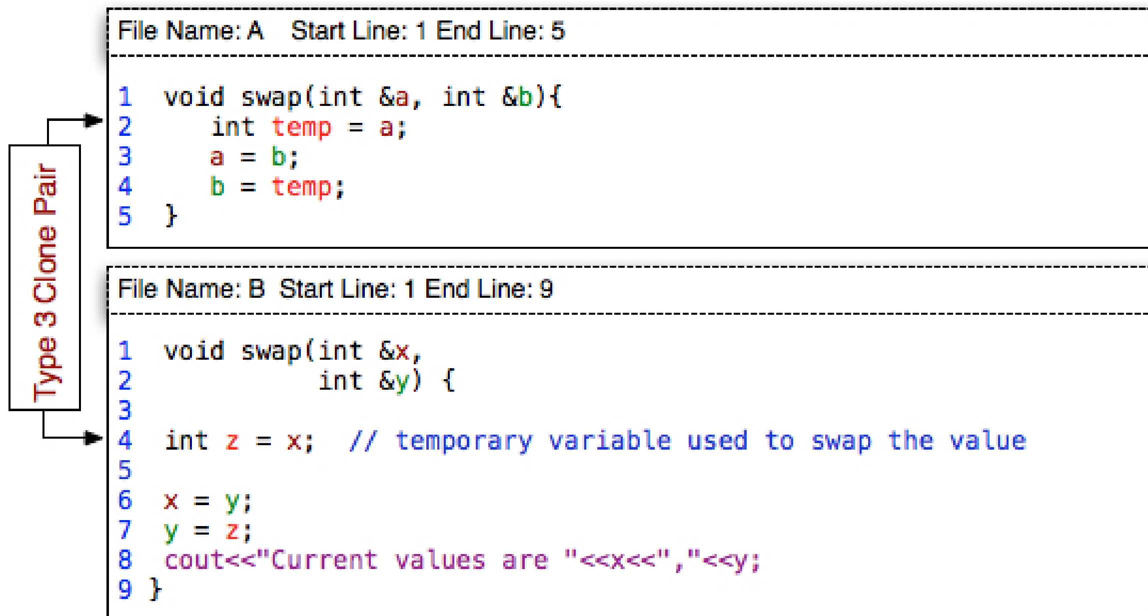



**Figure 2.4:** An example of a Type 3 clone pair. In addition to the variations in comment, whitespace, layout and variable name, a new line (line number 8) has been added to the second code fragment.

#### Maintenance benefit

Clones are sometimes purposely created to obtain maintenance benefits, such as to avoid risk in developing new code [42], particularly where creating a new solution is expensive (such as in financial software), to make software architecture understandable and to speed-up the maintenance.

#### Overcoming underlying limitations

Some programming languages may not have sufficient abstraction mechanism or programmers may have limitations (such as lack of knowledge, time constraint, lack of code ownership, lack of understanding) that lead to cloned code [31, 118].

#### Cloning by accident

Programmers may repeat a common solution, unknowingly create a clone due to the side effect of programmer's memory. Clones can also be created through implementing the same logic or because of following the programming language protocols to use a particular set of APIs. These are examples of cloning by accident.

## 2.3 Negative Impacts of Clones

In the previous section we described a set of reasons for cloning and identified a number of cases where cloning has positive impacts. Despite the positive effects of clones, code duplication may have maintenance

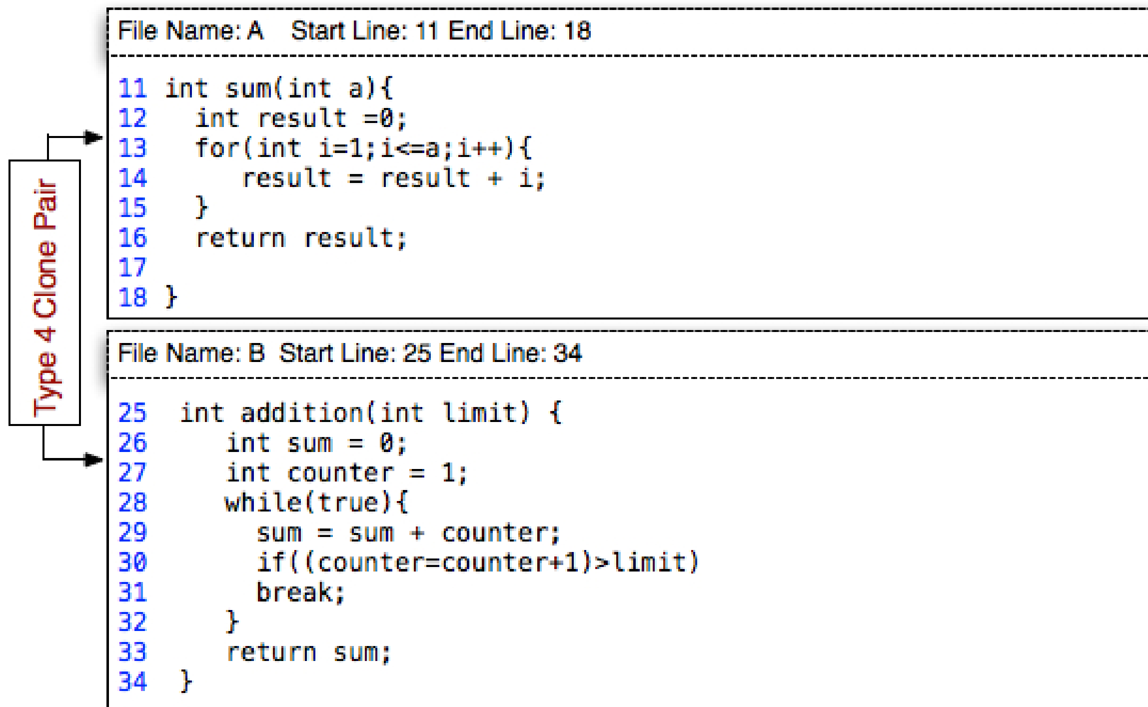



**Figure 2.5:** An example of a Type 4 clone pair. Each code fragment represents a function. Although the textual content varies, they does the same thing. Given a positive integer number as input, both functions calculate the sum of the numbers starting from 1 to that of the input number, and return the sum to the caller.

implications. In the following, we describe a set of common impacts of clones that make software maintenance challenging.

#### Bug propagation

If a code fragment contains a bug, reusing that code fragment in different parts of a system with or without minor modification increases the probability of bug propagation [78,103]. Li et al. [103] used a clone detection technique to identify instances where programmers copy and paste code fragments without renaming the variables consistently. On average 13% of the instances found by them turned out to be real errors for the industrial systems.

#### Anomalies in change propagation

If a cloned fragment needs to be changed because of a bug fixing or a feature improvement, all other fragments similar to it possibly need the same set of changes, or to the least need to be examined to be sure about that. Failing to do so may allow software bugs to be resided in the code for a long time. Thus, cloning increases effort in change propagation and leads to update anomalies.

#### Difficulties in system modification

Since code duplication increases code base size, maintenance engineers or developers need additional time and effort to understand the system. Therefore, system modifications require more time and become more challenging. Moreover, changing a part of a system also requires to detect instances of duplication of that part in other places of the system because chances are high that they also require similar set of changes.

#### Design fault

Code duplication may introduce faulty design, lack of abstraction or bad inheritance structure. It also introduces hidden dependencies among the duplicated parts and as a result, reusing previously developed components or parts of the solutions become difficult in the future project.

#### Increase challenge in refactoring

One possible approach to handle code duplication problem is to refactor them. A number of automatic refactoring [29, 32] or aiding developers to manually refactor clones have been proposed [71]. However, Kim et al. [91] found that aggressive refactoring is not a good solution to manage clones because a large number of clones in the system are short lived and refactoring those clones may not be worthwhile because they are likely to diverge from one another in near future revisions. Refactoring also introduces the risks and dependencies. Thus, the presence of clones makes refactoring a challenge. Recently, Zibran and Roy [160,161] have proposed techniques for scheduling code clone refactoring.

#### Increase code comprehension effort

The larger the size of a code base, the more time it requires to understand where such understanding is essential for successfully adapting necessary changes. Since multiple instances of the same code fragments are located in different places in the systems, they need to be fully analyzed to understand the differences among them [77]. Thus, cloning increases the effort to understand the code base. Moreover, the purposes of copying and pasting fragments of code differ and are not well documented, which also makes the code comprehension a difficult task.

#### Increase resource requirement

Cloning increases the growth rate of a subject system size. This not only increase the storage requirement but also introduces delay in compilation. Although, these might not be a problem for many systems due to the advancement in technology, others (such as telecommunication switch or compact devices) may require costly hardware and software upgrade. Johnson [78] described the overall effect as software aging where small changes in the architectural level become very difficult to implement in the code level.

**Increase code base size**

Code cloning leads to bloated code base, increases the size of binary executables and requires more storage space. Due to the advancements in storage capacity, this may be of little impact for systems with large storage. However, for systems with limited storage capacity like hand-held devices, this would be a big issue.

## 2.4 Clone Detection

Considering the maintenance implications of code clones, a number of techniques and tools have been developed to detect code clones. Depending on the type of representation used and the algorithm applied, clone detection tools can be divided in the following categories. More details about these techniques can be found elsewhere [135, 136].

### 2.4.1 Text-based Approaches

Text-based techniques mostly use raw source code to detect duplicated code fragments and thus do not require parsing. Johnson pioneered text-based clone detection. His approach [77, 79] uses *fingerprints* on substrings of the source code to detect clones. Manber [109] also used *fingerprints* to detect similar files in a large file system. Ducasse et al. [55, 125] developed a language-independent technique that uses line based transformation along with scatter plot visualization to detect duplication visually. Lee et al. [100] developed an algorithm named SDD (Similar Data detection), another text based technique, that uses N-neighbor distance concept to detect similar code fragments. Wettel and Marinescu [156] also developed a text based technique that can detect near-miss clones by combining small isolated fragments into a large duplication chain. Marcus and Maletic [110] applied latent semantic indexing (LSI) to identify high-level concept clones in the source code.

### 2.4.2 Lexical Approaches

Lexical (also known as token based) clone detectors transform the source code into a sequence of tokens and then determine the duplicate sequences of tokens. The first of this kind is the DUP [26, 27], that can detect code fragments that are textually identical or those that are identical after replacing the variable name and the constants with another. The technique can detect Type 1 and Type 2 clones, and these groups of clones can be combined to detect Type 3 clones where gaps between the pair of clones is used as a threshold. Kamiya et al. [84] improved the technique with language specific transformation rules that led to the development of CCFinder. CP-Miner [103] uses a frequent subsequence mining technique to determine the similar sequence of tokenized statements and can also detect copy-paste related bugs. Cordy et al. [45] also developed a token and line-based technique to detect near-miss clones. Clones found by token-based techniques may overlap because syntax is not taken into account. Using pre-processing [45, 62, 72, 129] or post processing we can

find clones corresponding to different syntactic units if block delimiters are known or a lightweight syntactic analysis [116] is employed.

### 2.4.3 Graph-based Approaches

Some clone detectors use graph-based technique to detect code duplication [61, 92, 95]. In this case, source code is transformed into a program dependency graphs (PDG). While the nodes within the graph represent statements and expressions, the edges represent control flows and dependencies. Isomorphic sub-graphs are then identified to detect clones. Although PDG-based approaches can partly handle statement reordering, insertion, deletion and non-contiguous code, they are not scalable [91].

### 2.4.4 Syntax-Tree based Approaches

Syntax tree based detection techniques transform the source code into a parse tree or an abstract syntax tree, and then determine the duplicated code fragment using tree matching algorithm. Baxter et al. [32] developed CloneDr that transforms the source code into an abstract syntax tree and then determine the similar code fragments with tree matching algorithm. Hashing is also used to reduce the number of tree comparisons. Alternative tree representations are also used to improve performance of clone detection. Koschke et al. [94] used suffix trees to determine clones in the abstract syntax tree representation of a program. Tairas et al. [145] developed a technique to detect function clones using abstract syntax trees and suffix trees. Recently, Jiang et al. [74] developed another clone detection tool called Deckard, that uses the tree representation of source code and identifies similar subtrees using certain numerical vectors in an Euclidean space. Locality Sensitive Hashing (LSH) is used to cluster similar vectors to find clones.

### 2.4.5 Metric-based Approaches

Metric-based approaches collect a set of metrics that characterize the code fragments, called fingerprinting functions, and then determine the similar code fragments by comparing the metric vectors [111, 112]. Clone detectors based on this approach find clones for syntactic units such as a class, a function or a method since the fingerprinting functions are often defined for these syntactic units. Metric based approaches have been applied to detect duplicate web pages and clones in web documents [38, 108].

### 2.4.6 Hybrids

In addition to the above techniques, there are also a number of clone detection techniques that uses a combination of syntactic and semantic characteristics [102]. NiCad [47] is an example of a hybrid technique and has been used in a number of empirical studies [114, 115, 128, 133, 134]. To overcome the limitation of the text-based approaches it takes advantage of the AST-based detection techniques and exploits the power of source transformation system to accurately detect near-miss clones. NiCad is also used in incremental

**Table 2.1:** Categorization of clone presentation techniques (taken from [76] and then extended)

| Visualization Technique | Entity | Clone Relation |
|---|---|---|
| Scatter Plot [69, 126, 150] | File, Subsystem | Clone Pair |
| Hasse Diagram [79] | File | Clone Class |
| Metric Graph [150] | Code Segment | Clone Class |
| Hyper-Linked Web Page [45, 80] | Code Segment | Clone Class |
| Dependency Graph [87] | Subsystem | Clone Pair |
| Duplication Web [126] | File | Clone Pair |
| Duplication Aggregation Tree Map [126] | File, Subsystem | Clone Class |
| System Model View [126] | File, Subsystem | Clone Pair |
| Clone Class Family Enumeration [126] | File | Clone Class |
| Clone Coupling and Cohesion [76] | Subsystem | Super Clone |
| Clone System Hierarchy Graph [75] | File, Subsystem | Clone Pair, Clone Class |
| Clone Visualizer View [146] | Code Segment, File | Clone Class |
| Stacked Bar Chart [159] | Code Segment, File | Clone Class |
| Line Graph [159] | Code Segment, File | |
| Clone Cluster View [60] | Code Segment | Clone Class |

mode to check near-miss clones in a software system [46]. Recently, Uddin et al. [149] have evaluated the effectiveness of simhash [39], a fingerprint based data similarity measurement technique, using a modified version of NiCad. The result reveals that the technique is effective in detecting both exact and near-miss clones.

Techniques have been developed to detect clones beyond source code [81]. The first of this kind is developed by Sæbjørnsen [137] that can detect clones in binary executables. Davis and Godfrey [50, 51] developed clone detectors that convert Java source code to one or more class files, C/C++ source code to assembler, and then detect clones in them. Deissenboeck et al. [52, 53] presented an approach to detect clones in graphical models. Nguyen et al. [117] developed a structural characteristic feature extraction approach that can detect clones in source code and models. Pham et al. [119] presented ModelCD, a tool for detecting both exact and near-miss model clones in Matlab/Simulink model. Domann et al. [54] presented a study of cloning in requirement specifications. Later, Juergens et al. [82] performed another study where they applied clone detection to 28 real-world requirement specifications and evaluate the effective of using clone detection techniques to assess the quality of requirement specifications. Liu et al. [104] proposed technique to detect duplication in UML sequence diagrams. Later, Storrle et al. [144] designed and implemented a set of algorithms and heuristics to detect clones in UML domain based models.

## 2.5 Supporting Clone Analysis and Visualization

A major challenge in identifying useful cloning information is to handle the large volume of textual data returned by the clone detectors. To mitigate the problem, a number of visualization techniques, filtering mechanisms and support environments are proposed in the literature. Jiang et al. [76] categorized the proposed clone presentation techniques based on two dimensions. The first dimension refers to the level at which the entities are visualized (such as at the code segment level or file level or subsystem level). The second dimension refers to the type of clone relation addressed by the presentation (whether clones are showed at the clone pair level or grouped into clone classes or super clones). Table 2.1 shows the categorization of clone presentation techniques.

Johnson [79] used the popular Hasse diagram to represent the textual similarity between the files. Later, he also proposed hyper-linked web pages to explore the files and clone classes [80]. Cordy et al. [45] used HTML for interactive presentation of clones where overview of the clone classes is presented in a web page with hyperlinks and users can browse the details of each clone class by clicking on those links. Although such representations offer quick navigation, they cannot reveal the high level cloning relations. A set of polymetric views [126] were also proposed in the literature that permit encoding of a number of code clone metrics to the visual elements. Among various visualizations, scatter plot is the most popular and capable of visualizing inter-system and intra-system cloning [44, 105]. However, the size of the scatter plot depends on the size of input rather than the amount of cloning. Thus, using a scatter plot for visualizing cloning relation of a large software system may become challenging due to the large size of the plot. Moreover, non-contiguous sections that contain the same clone cannot be group together. To overcome this, Tairas et al. [146] proposed a graphical view of clones (also known as Visualizer view) that represents each source file as a bar and clones within the files are represented with stripes. Clones belong to the same class are encoded with the same color.

Jiang and Hasan [76] extend the idea of cohesion and coupling to code clones and proposed a visualization that uses shape and color to encode the metric values. They also developed a framework for large scale clone analysis and proposed another visualization, called a *clone system hierarchical graph* that shows the distribution of clones in different parts of a system [75]. Fukushima et al. [60] developed another visualization using graph drawing technique to identify diffused clones. Here, nodes represent the clones. Those nodes that are located in the same file are connected with edges to form a clone set cluster. Nodes that connect different clone set clusters are called diffused clones (have cloning relation in different files implementing different functions).

Gemini [150] is an example of a clone support environment that uses CCFinder for clone detection and can visualize cloning relation using scatter plots and metric graphs. Kapser and Godfrey [87] developed CLICS, another tool for clone analysis. CLICS can categorize clones based on their previously developed clone taxonomy [86] and support query based filtering. However, it is limited to only C/C++ and Java source

code. Tairas [146] et al. developed an Eclipse plug-in that works with CloneDR, a clone detection tool and implements the visualizer view along with general information and detected clones list views. Third in this group is the Clone Visualizer [159], an eclipse plugin that works with Clone Miner, a clone detection tool. In addition to supporting clone visualization through stacked bar chart and line graph, it supports query based filtering. The recent addition to this group is CYCLONE [17]. It supports single and multi-version program analysis and uses RCF (Rich Clone Format) [68] file as an input. RCF is a data exchange format capable of storing clone detection results.

## 2.6 Code Clone Evolution

Software evolves due to various reasons including bug fixing, increase reliability, performance improvement, and above all, addition of new features to meet the demands of the users. Clones in a software system also evolve with the system. Clone evolution analysis is concerned with understanding the changes of clones, causes and the effects of those changes. Results of such analysis can help us to reason about cloning, improve software processes for managing clones, support developers in copy-paste programming practices and allow maintenance engineers to make informed decisions about removing clones. Thus, clone evolution analyses receive much attention in the research community.

### 2.6.1 Clone Genealogy Model

Kim et al. [91] first proposed the clone genealogy model that works as a basis of many clone evolution analysis including ours. This section introduces related terminologies and the evolutionary change patterns associated with the model.

#### Clone Genealogy

A clone genealogy describes how a clone class evolves during the evolution of a software system. It consists of a set of lineages. In other words, a clone genealogy is a collection of clone classes that are originated from the same clone class. It approximates how programmers create, change and manage code clones. An example of a clone genealogy is shown in Figure 2.6 that consists of two clone lineages.

#### Clone Lineage

A clone lineage is a directed acyclic graph that describes the evolution history of a particular clone class, also known as the sink node. Figure 2.6 shows two clone lineages that describe the evolution of two clone classes, $CC_{51}$ and $CC_{52}$. Although the source node ($CC_{11}$) of $CC_{51}$ contains two code fragments, it has experienced a number of changes including addition, deletion and consistent changes of the code fragments before reaching a sink node.

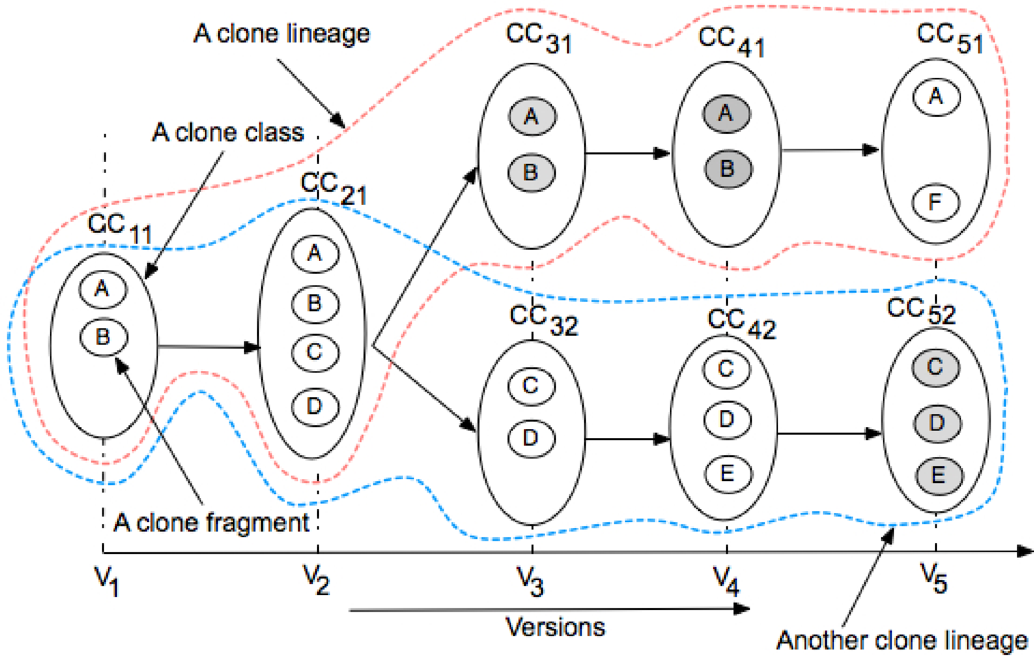



**Figure 2.6:** An example of a code clone genealogy. Each ellipse represents a clone class and the circles inside an ellipse represent the clone fragments that belong to that class. The darker the circles are, the more they are changed from the previous version. The figure shows the evolution of $CC_{11}$ during five consecutive versions of a software system. $CC_{11}$ starts with two clone fragments and two new clone fragments are added in version $V_2$. However, in the next version it is split into two clone classes ($CC_{31}$ and $CC_{32}$) and they evolve independently. Lineages are marked with dotted lines. For example, the lineage of the clone class $CC_{51}$ consists of $CC_{11}, CC_{21}, CC_{31}, CC_{41}$, and $CC_{51}$. Here, $CC_{11}$ represents the source node and $CC_{51}$ is the sink node.

### Evolution Patterns

An evolution pattern classifies the type of changes occurred in a clone class that evolves from a version to the next one (see Figure 2.7).

Let $CC_i$ denotes a clone class in the version $i$ and $CC_{i+1}$ denotes another clone class in the version $i+1$. If the clone class $CC_{i+1}$ evolves from $CC_i$, then it can experience the following change patterns during its evolution.

- **Same:** All clone fragments in $CC_i$ are present in $CC_{i+1}$ without any changes.
- **Add:** At least one clone fragment is newly added to $CC_{i+1}$ which is not part of $CC_i$. For example, a programmer copied and pasted one code fragment in a different location in the next version.
- **Delete:** There exists at least one clone fragment that is part of $CC_i$ but does not belongs to the $CC_{i+1}$. For example, a programmer removed one clone fragment.
- **Consistent Change:** All clone fragments that are part of $CC_i$ have experienced the same editing operations and thus belong to the $CC_{i+1}$. For example, a programmer added one line to all code fragments in the next version.

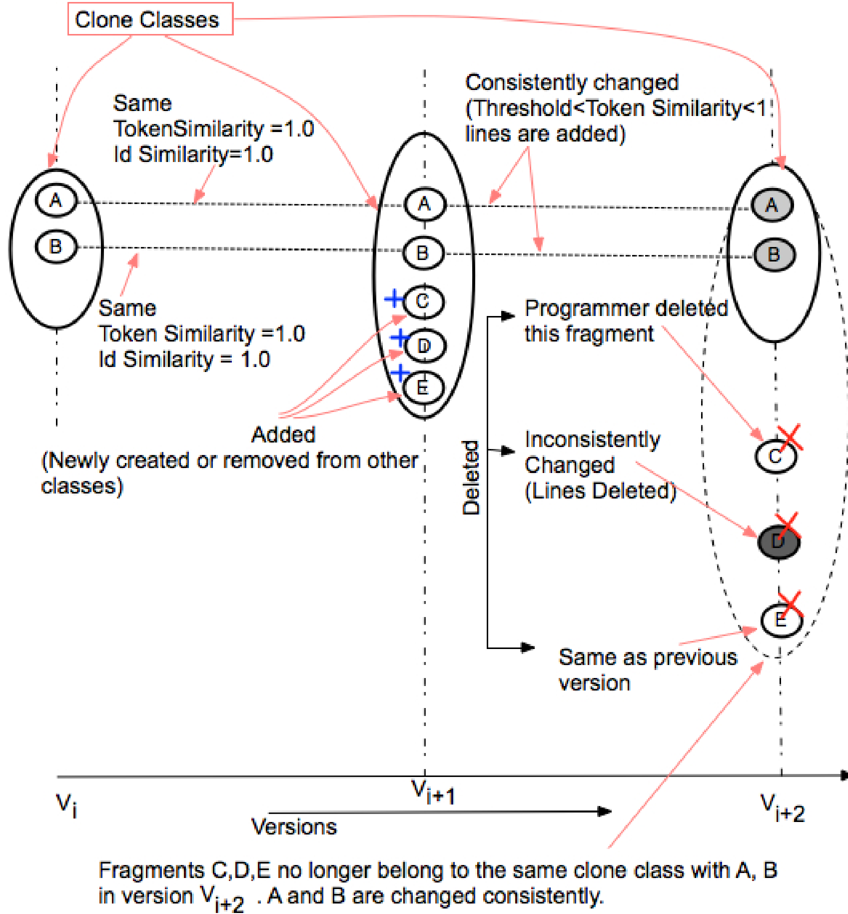



**Figure 2.7:** A clone genealogy marked with different change patterns. It shows the evolution of a clone class during three consecutive versions of a software system. The clone class in version $V_i$ consists of two clone fragments ($A$ and $B$). They are propagated in the next version ($V_{i+1}$) without any change and thus belong to the same clone class. In addition, three new clone fragments ($C$, $D$ and $E$) are added to it. These clone fragments have undergone different degrees of changes before reaching version $V_{i+3}$. Fragments $A$ and $B$ are changed consistently and thus belongs to the same clone class. Fragment C is deleted by the developer. While fragment $E$ is not changed, fragment $D$ is changed inconsistently. Both fragments are no longer belong to the same clone class. Thus, the clone class has undergone with same, inconsistent and delete change patterns. It should be noted that the evolutionary change patterns are not mutually exclusive.

- **Inconsistent Change:** At least one clone fragment that is part of $CC_i$ does not belong to $CC_{i+1}$ because not all fragments are changed consistently in the next version. For example, a programmer added one line to all fragments of $CC_i$ except one.

### 2.6.2 Clone Evolution Analysis

Clone evolution was first addressed by Laguë et al. [98] where they studied the evolution of code clones in a large telecommunication system at the function level and assessed the impact of two changes in the software development process. Their study reveals that preventive support can control the growth of clones whereas problem mining can provide an opportunity to discover and correct problems before the customers experience them. Antoniol et al. [23] studied the evolution of code clone coverage in Linux Kernel and found

that clone coverage stabilizes over time, though increases at the beginning. Kim et al. [91] analyzed the evolution of two small Java systems. Their study revealed the facts that many clones in the system are short lived and thus not suitable for aggressive refactoring since they can be changed from one form into another, many are propagated with the same repetitive edits. Their study was the first that introduced clone genealogy model and formally defined the clone evolution change patterns. Aversano et al. [25] extend Kim's clone evolution patterns by grouping inconsistent changes into independent evolution and late propagation. Their study combined the analysis of Modification Transactions with clone evolution to determine whether clones are maintained properly. They concluded that the majority of clone classes is maintained properly and exceptions occur when clones evolve independently to implement different functionalities. Krinke [96] analyzed a large number of revisions of five open source systems and found that inconsistent changes to clone groups are frequent and are rarely appeared as consistent during the evolution of a software. In another study [97] on clone evolution at the revision level, he found that cloned code is more stable than non-cloned code. Bettenberg et al. [33] argued that clone evolution analysis at the revision level captures many short term changes made by the developers. To ignore the effect of short term changes, they studied the evolution of two Java systems at the release level and found that only a small portion of clones are related with software defects. Lozano et al. [106, 107] studied the maintenance implications of clones. Although they found evidence that cloning may increase maintenance effort of some functions, they could not reveal any systematic relationships between cloning and maintenance effort. Bakota et al. [28] applied a machine learning algorithm to map clone fragments across versions to define *clone smells* whose presence point to suspicious behaviours of clones. They also found a number of bad smells using 12 versions of Mozilla Firefox.

Göde [63] presented a model for describing the evolution of clone fragments and also analyzed the evolution of type-1 clones in nine open source systems. While he concluded that over time the ratio of clones decreased on average and found that many inconsistent changes were remained inconsistent due to late propagation, he did not draw any general conclusion about the inconsistent changes of code clones. Thummalapenta et al. [147] studied the evolution of four different Java and C software systems to determine the extent of consistent changes or independent evolution. Their study revealed that more than 70% of the clone classes are propagated consistently or evolve independently. Moreover, bug fixing changes are largely associated with clone classes that underwent a late propagation. Foyzur et al. [123] analyzed the relationship between clones and defects. They found little or no relation of clones with bugs. Moreover, their study also revealed that cloned codes are less buggy and larger clone classes are not necessarily contributed more to the defects. Recently, Göde et al. [64] studied three software systems to identify the frequency and risks of changes to code clones. They found that clones are rarely changed and unintentional inconsistent changes are also small, suggesting careful selection of clones to manage or to refactor to avoid unnecessary maintenance effort. Cai and Kim. [37] analyzed the evolution of seven Java software systems to identify the correlation between survival time of clone genealogies with code clone metrics. They found that the evolutionary characteristics are highly correlated with the survival time and those metrics can be used to predict their life time.

### 2.6.3 Supporting Clone Evolution Comprehension

Clone evolution can be better understood through visualizations since they can provide a high level overview. If interactions are properly added, they can also guide users to drill down and can help perceive low level details. Although a number of visualization techniques have been developed for supporting software evolution analysis, a few work have been listed in the literature for supporting clone evolution analysis. In this section we first describe the previous work in software evolution visualization because much of the ideas in clone evolution visualization have come from there.

#### Software Evolution Visualization

One of the earliest works in visualizing software changes during the evolution was proposed by Ball and Eick [56]. They used a set of visual metaphors to encode the change patterns in the software including matrix view, cityscape view, bar and pie charts, data sheets and network views. Since each view has its own strengths and weaknesses, they also described how different views can be linked together (also know as creating perspectives) to uncover maximum information. Collberg et al. [41] developed GEVOL, a system to visualize the evolution of a software system using graph drawing techniques. It takes the advantage of the CVS version control system and collects inheritance graph, call graph and control-flow graph to reconstruct the changes in those graphs during the evolution of the system. Finally, changes are encoded with color, and query support is added to reduce the size of the graph. During software evolution, changes are inevitable but understanding the reasons behind those changes aid in better comprehending software evolution. With this aim, Rysselberghe and Demeyer [151] proposed a simple visualization that maps the changes in time to the files in a software. Files are arranged in the horizontal axis and are sorted so that files that are in the same directory appeared close together while the vertical axis refers to the time. A dot in the visualization refers to a change in the corresponding file in a given time. Despite its simplicity, it can help in identifying unstable components, coherent entities, design and architectural evolution, and fluctuations in team productivity. Wu et al. [158] developed a color-coded visualization technique called an Evolution Spectrograph (ESG) to depict the sudden and discontinuous structural changes during the software evolution. The spectrograph uses metrics and gradient colors to portray the changes. Although three different open source systems were analyzed with the proposed visualization, no user study was reported to indicate the comprehensibility and the effectiveness of the technique. To determine the development effort and to compare the distribution of effort among various developers, fractal figure was developed [49]. Although fractal figures have some resemblance to a treemap, they show numerical data in sorted order without any structure. Four different development patterns (one developer, few developers with balanced effort, many developers but unbalanced effort, many developers but balanced effort) can be easily identified by scanning a figure. Although the authors applied the fractal figure to analyze development effort within a version, the fractal value (determine how fractalized a fractal figure is) can be a better indicator to reflect the changes in the development effort during the evolution of a software system.

Although the previous approaches use 2D visualizations to capture different aspects of software visualization, 3D visual models are also developed. Wettel and Lanza [155] developed a set of interactive 3D visualizations based on city metaphor to depict the evolutionary characteristics of a software. The techniques employed can handle a large number of versions of the real-world systems. Metrics provide important insights and quantify the characteristics of a software. Pinzger et al. [120] proposed a visualization of evolution metrics to capture large volume of data in a condensed view and rapidly identify the trends using Kiviat diagrams and graphs. Although powerful, the visualization lacks overview and detail on demand support. Byer and Hassan [34] proposed the evolution storyboard that captures the dynamic nature of evolution. The complete lifetime of a software is divided into a set of time periods. For each time period, a dependency graph is used to visualize the structure of the system in a panel. The panels are displayed sequentially to animate the changes in software structure.

The evolution matrix is proposed by Lanza [99], which combines both visualization and software metrics to show the evolution of a software system. The core idea is to encode metrics value as the height and width of a rectangular box that represents a software unit (such as a class). The boxes are arranged in a two dimensional grid structure where each row represents the evolution of a single unit and each column represents a version. The size of the boxes change as the metric values change and the change patterns are classified into eight different categories that are not mutually exclusive. The patterns help developers to identify entities in a software that require attention. For example, the evolution matrix can guide maintainers to locate classes that persist during the entire evolution of the software or classes that have very short life time or those that suddenly change their characteristics. Two small case studies were reported to explain the benefits of the evolution matrix. Although the presented visualization is useful, identifying patterns require manual effort and thus becomes difficult when there are a large number of classes. If a class changes its name, the evolution matrix reports that as the deletion and addition of a new class. Moreover, the effectiveness of the matrix depends on the number and the distance between the versions. In case of a few number of versions many patterns are difficult to locate. Moreover, if the versions are too distant from one another, many details may get lost. The matrix treats every class as a separate entity without considering the relationship between the classes. Finally, overview and filtering mechanisms are not presented which are crucial for comprehending the evolution.

Line-based software visualization techniques can be a useful source of information for developing techniques of clone evolution visualization since many statistics related with code clones are also line oriented. SeeSoft [57] is the first visualization system that maps line level statistics obtained from the source code to visual entities. The key ideas that were introduced are as follows:

i) Reduced representation

ii) Represent statistics through colors

iii) Direct manipulation

iv) Ability to refer actual code

Reduced representation is achieved by representing the files in columns and lines as thin rows. The color of a line can represent a statistic of a source code line it represents (e.g. code authorship information). Direct manipulation is achieved through brushing that facilitates locating useful patterns. Finally, users can investigate the source code by moving a virtual magnifying box over the reduced representation. Although the tool can successfully unfold changes in a particular version of a software, it does not provide change information during its evolution. To resolve this problem, Voinea et al. developed CVSscan [152] using a new form of line oriented display that represents versions using column and are arranged in order of time from left to right. A number of views are also linked together that can display various statistics collected from a version control system and the corresponding source code. However, the visualization is only file-based and can show the evolution of one file at a time. Using the same file-based evolution visualization, a number of techniques have been proposed to visually mine data stored in a version control system. The techniques have been implemented in a tool called CVSgrab [153] that collects and visualizes evolution from CVS like repositories. The usefulness of the tool has been demonstrated through case studies and also incorporated in an open framework to mine data from repositories [154].

Gonzalez et al. [65] proposed a visualization of software evolution that combines both structural and metrics information. The visualization consists of four views linked within a single design environment. The time-line view uses a circular ring layout to represent various characteristics of the revisions in different time scale, allow maintenance engineers to drill down the data from coarse (years) to fine (days, hours) level. The structure evolution view can be used to compare the packages or classes during software evolution. The metric view helps identifying changes and analyzing trends in metrics. The last view represents indirect class coupling integrated with source code viewing which not only helps in discovering coupling relationships for a target class but also enables to examine their source codes seamlessly. If a class A has a reference to class B and that class has a reference to another class C, then A has a direct coupling relation with B and indirect coupling relation with C. Although, the proposed visualization lacks a formal user study, we believe that combining multiple views within a single environment is useful when they provide information at different levels of detail, meet the requirements of different categories of users (project managers, developers, maintenance engineers),and focus on different aspects of the system.

**Clone Evolution Visualization**

Adar and Kim [19] developed SoftGUESS, a system for clone evolution exploration that supports three different views. SoftGUESS is developed on top of Guess [18], the graph exploration system which models the evolution of a software system using graphs. The genealogy browser offers a simple visualization of clone evolution where nodes represent clones, arranged from left to right, and the those that belong to the same class are arranged vertically in the same position. Thus, each column represents a version. Link between a pair of node reflects the predecessor and successor relationship during the evolution of the software. The

encapsulation browser shows how clones within a clone group are distributed in different parts of a system and how they fit in the hierarchical organization of the software system by visualizing the containment relationship through a tree structure. Finally, the dependency graph describes how the nodes (package, class or method) within a version are evolved from other nodes and how they evolve in the next version. In addition, SoftGUESS also supports charting and filtering mechanisms based on Gython, a SQL type query language. However, SoftGUESS lacks overview feature and requires users interaction for data reduction through queries. Although a query is a powerful mechanism to identify important patterns of cloning, formulating query could be difficult as this requires more cognitive effort from the developers.

Recently, Harder and Nils [68] developed a multi-perspective tool for clone evolution analysis, called CYCLONE. It offers five different views to analyze clone data stored in a RCF file, where RCF is a binary format to encode clone data including the evolutionary characteristics. The evolution view in CYCLONE visualizes clone genealogies that uses simple rectangles and circles to denote software entities. Each circle represents a clone fragments arranged in a set of rows where each row represents a particular version of the software. The clones that belong to the same clone class are packed within a rectangle. Finally, lines represent the evolution of a clone fragments. In addition, the view employs colors to distinguish types and the changes of the clones. Although the view highlights many important evolutionary characteristics, the volume of data produced by the genealogy extractor still limits its usefulness, thus call for overview and filtering mechanisms.

Saha et al. [140] presented an idea for clone evolution visualization using the popular scatter plot. In their proposed approach, scatter plots show the clone pairs associated within a pair of software unit (file, directory or package). Based on the type of clone genealogies they are associated with, clone pairs are rendered with different colors. Selecting a clone pair through user interactions (double clicking on a clone pair in the scatter plot) shows the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone classes in a particular version and then provide a way to call for genealogy browser to dig deeper. However, it does not provide overall characteristics of the genealogies. Moreover, due to the large number of clone pairs, selection and useful pattern identification in such a scatter plot are difficult. As a result, researchers developed variations of the traditional scatter plots [44].

## 2.7 Conclusion

In this chapter, we have provided further motivation for this thesis along with background material and related work. After defining the terminology of clones, we explained various reasons behind cloning. Next, we briefly described various effects of clones that can negatively impact development activities. We reviewed techniques for clone detection, analysis and visualization. We then explained the terminology of clone evolution followed by brief overview of clone evolution analysis. Finally, we reviewed studies of software and clone evolution visualization.

# Chapter 3

# Supporting Large Scale Clone Analysis: A Pragmatic Approach

## 3.1 Introduction

Clones are similar code fragments, often created through copy-paste programming practices. Moreover, writing code for solving similar problems may result in similar code fragments. Such unintentional code clones are also produced as programmers write code using application frameworks, API libraries, or design patterns. Two code fragments that are similar to each other form a clone pair, and a clone class is the maximal set of code fragments where any two are clones of each other. Various clone detection techniques have been proposed in the literature [135,136] and many state of the art clone detectors are also available that can detect clones in a system within a reasonable amount of time. However, the volume of data produced by them is too large to analyze manually. For example, NiCad [47] detected 19,926 clone pairs and 6,728 clone classes for the Linux Kernel system (version 2.6.24.2) [10], even when we set the minimum length of a clone to five lines. Although the detection technique, granularity of operation, size and type of the detected clones vary significantly, most of them produce clone detection results in textual format with file names, starting and ending line numbers of the clone fragments. For small systems, such textual results alone might be adequate. However, for a system with a huge number of clones, such results might be inefficient because of the following two main reasons.

- Since the volume of data is too large, manual inspection of the reported clones would be tedious, error-prone and expensive.
- In-depth analysis on distributions and relationships of the clone fragments or regions becomes an overwhelming and complex task due to the textual nature of the large amount of data

Some clone detection tools including NiCad generate interactive HTML pages to navigate the clone fragments. However, such output lacks the overview of cloning information, does not support interactive exploration of the data set to identify cloning relationships and the volume of data limits its usefulness. This justifies the need of a clone analysis support environment.

In this regard, this thesis make the following contributions.

1. We provide a framework for clone analysis that captures different aspects and requirements of clone analysis.

2. Based on the framework we developed VisCad, a clone analysis and visualization tool that supports various clone detectors (and is easily adaptable to others), visualization techniques, metrics and data reduction techniques to help users efficiently analyze code clones of even large systems such as the Linux Kernel.

3. Finally, an inter-project clone analysis is used to explain how VisCad implements the framework and provides better support in clone analysis.

Given the importance there are also a great many visualization and analysis tools proposed in the literature [135] (Section 2.5). However, as far as we know only a few of them [4], [87], [68], [150] are available for public use, and they mostly focus on a particular area of visualization and or analysis. For example, CLICS [3] is for analyzing and categorization of clones based on a taxonomy of clones, ConQAT [4] is with an integrated IDE, CloneMiner [30] is for structural design analysis from simple clones, Gemini [150] is only based on CCFinder [84] and so on. It is also not clear how those tools can be adapted to other clone detection tools of interest. Ours is the first attempt in the clone detection area to offer a broad range of analyses, metrics, visualizations and other important features in one integrated platform for supporting comprehensive clone analysis and visualization, which is specifically designed to be adaptable to third-party clone detection tools regardless of the types of clones they return. We also released VisCad as open source [14] so that even the researchers and practitioners outside of clone community can reuse the rich set of libraries and features of VisCad in their own domains.

The rest of the chapter is organized as follows. In Section 3.2, we briefly discuss the framework for clone analysis, also acts as the conceptual design of VisCad. Section 3.3 describes features of VisCad including the supported metrics and data filtering. Section 3.4 presents an inter-project clone analysis between two different BSD operating systems (FreeBSD 8.2.0 and NetBSD 5.1) kernels that not only reveals important insights about cross cloning between these systems but also reflects the importance of VisCad in large-scale clone analysis. Section 3.6 presents brief description of the relevant research and compares ours work with others. Finally, Section 3.7 concludes this chapter.

## 3.2 A Framework for Clone Analysis

We were interested in identifying the important requirements of a clone analysis support environment since they can guide us selecting necessary components or features of it. We have used three sources of information to collect such requirements. While previous studies on both code clone analysis and visualization acted as a primary source of information, feedback was also collected from a number of individuals through informal discussions who have experiences on clone analysis. Finally, we also studied existing tools to identify their

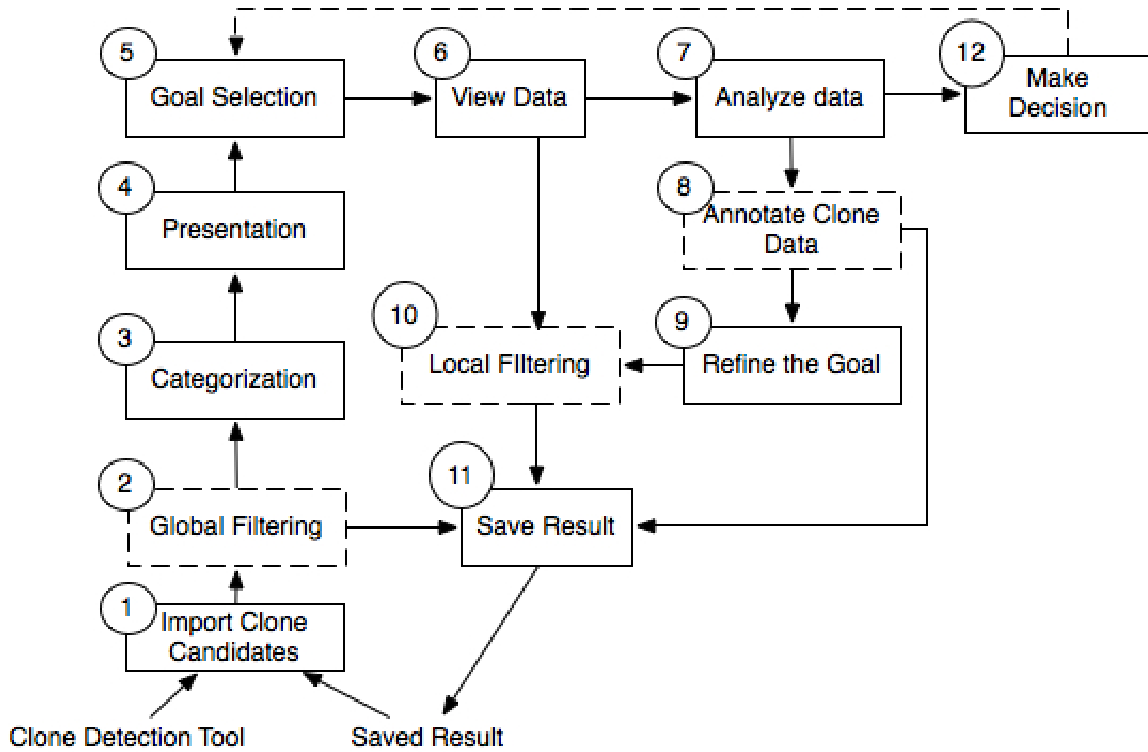



**Figure 3.1:** Clone Analysis Framework. Rectangles with dashed lines refer to the optional steps.

availability, portability and degree of support towards clone analysis. Based on this information, we have developed a framework for clone analysis (that need to be addressed by the support environments) as shown in Figure 3.1. In this section, we briefly describe various parts of the framework.

### 3.2.1 Import Clone Candidates

Clone candidates are the code fragments that are reported as similar code fragments by the clone detection tools. A robust clone analysis tool should support importing clone detection results from a large set of clone detection tools since every clone detection tool has its own strengths and weaknesses, and more importantly the selection of a tool may vary depending on the objectives of the analysis. For example, if the objective is to analyze the Type 3 clones of a subject system, we need to select a clone detector that has the ability. Only a few clone detection tools [47, 162] can detect Type 3 clones.

### 3.2.2 Global Filtering

Filtering is an operation that helps in removing noise from the data. The noise can be false positive clones or can be true clones that may not be useful or interesting to the users depending on the objective of an analysis. Filtering can be applied prior to the categorization and presentation of clone data, and we refer this as the global filtering. For example, if a user wants to analyze cross cloning between a set of systems,

then clones that are spread across the systems constitute the set of interesting clones. We can remove all intra-project clones from the analysis.

### 3.2.3 Categorization and Presentation of Clone Data

The next phase is to group the clones and present them to the users in such a way that ease the analysis. For example, clones can be grouped into clone pairs and clone classes, which are two most frequent categorizations of clone fragments and readily available through the clone detection results. Mapping clones in terms of directory structure of the subject system or to the regions of source code [86] can also help users to relate cloning with the physical or logical structure of the system and provide alternative views for analyzing them. The categorized clones need to be presented through user interface components so that user can select, visualize and perform query operations on them.

### 3.2.4 Goal Selection

It is also required to develop a goal that drives the nature of the analysis. Expressing the idea concretely may be difficult in the first place. Thus, the goal at this stage need to be too specific rather can be refined in the subsequent steps. As a starting point, lets say the goal is that we want to know whether the system contains a significant amount of clones or not.

### 3.2.5 View Data

The clone data can be described in various ways (such as using tabular representations or visualization techniques). One simple but effective technique is to use sortable tables where several metric values of the target entities are displayed using tabular representation where values can be sorted in ascending and descending order. Although powerful, sortable tables have the limitations of being unable to support overview feature. Moreover, it is difficult to locate patterns in clones. The alternative solution is to use information visualization techniques built on visual Information Seeking Mantra [143]. Every view should answer one or more specific questions about code cloning. This helps users learn different aspects of cloning using different views. For example, scatter plots can help us identifying the subsystems that are correlated in terms of clone pairs, whereas a treemap can help us locate the subsystems that mostly contribute to the code cloning.

### 3.2.6 Analyze and Annotate the Clone Data

Analyzing the clone data is the most important step since it helps us gain knowledge or make a decision about code cloning. The process varies depending on the type of objectives of the analysis, support environment used and the view employed. As a part of the analysis, we may annotate the clone data, so that those data can be queried and further explored later. The term 'annotation' refers to the additional pieces of information that can be added to the clone data. For example, if the objective is to get feedback from the developers

to know the reasons of code cloning, developers can view and annotate the clone pairs that indicates their opinion. In annotating the clone data, possible options can be provided or users can add their own note.

### 3.2.7 Refine the Goal

Since the volume of clone data is large, clone analysis starts with the overview of the result. As time passes, a user gains better understanding about cloning and can refine her goal. A user can use a treemap to gain knowledge about how the subsystems contribute to the clone LOC. Later, she can refine her goal and re-analyze the clone files within the subsystem that has the highest contribution to cloning.

### 3.2.8 Local Filtering

During the analysis of clone data, the user may have an interest on particular data set. Filtering at this phase can separate those data from the rest and called local filtering. This can help us to find the outliers or to locate information that are difficult to find otherwise. For example, we may have a list of clone files within a directory but we might be interested in those files that are mostly clone. Filtering helps us locate those files. Different metric values can be used to obtain different aspects of clone data and also can be used for filtering. Support for saving the filtered data set is also needed so that they can be analyzed later. For example, before starting an in-depth analysis, we might check the quality of the data and remove those that are false positives by manually inspecting the clone pairs. After removing the redundancy, we need to save the data for future use.

### 3.2.9 Make Decision

We may iterate through steps 6 to 10 until we have sufficient knowledge about cloning. We can then select another goal or can import candidate clones from a different clone detector for the same subject system (or vice versa) and continue the analysis.

## 3.3 From Design to Implementation

We have developed a clone analysis tool, called VisCad that implements the clone analysis framework as discussed in the previous section. It takes the clone detection results and the code base of the subject system as input for supporting analysis at the source code level. In response to the user's interaction, VisCad populates a number of graphical views and interfaces through which users can further interact and investigate code clones at different levels of abstraction. It has been implemented in Java and runs on systems with Java Runtime Environment (JRE) 6.

The main interface of VisCad consists of five parts. Figure 3.2 shows different parts of the interface that are labelled with different numbers. The top-left part displays the distribution of clones over the directories

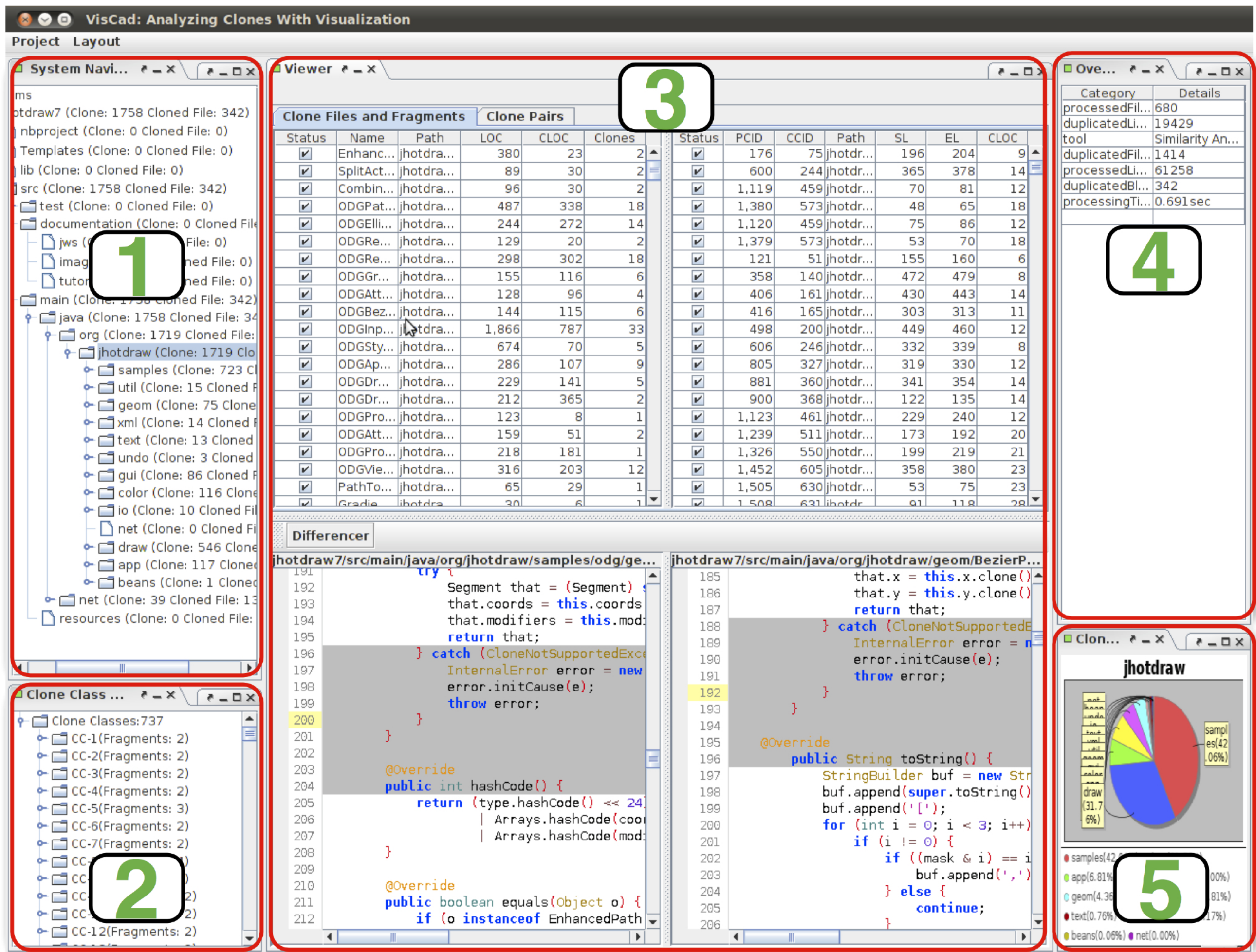



**Figure 3.2:** VisCad Interface

and subdirectories in the subject system, known as 'System Navigation Tree' (labelled with *1*). The bottom-left part, located at the bottom of the clone browser, lists all clone classes and the number of clone fragments in each class, called 'Clone Class Tree' (labelled with *2*). The middle part (labelled with *3*) of VisCad accommodates different views in separate tabs. The top-right window shows the clone detection specific information (e.g. the name of the clone detector, the set of configurations used to detect clones) VisCad obtained while parsing the result file for the selected subject system (labelled with *4*). For any selected directory in the 'System Navigation Tree', the bottom-right window (labelled with *5*) shows the distribution of clones in its subdirectories through a pie-chart.

The various features supported by VisCad are summarized in Table 3.1. The rest of this section explains the visualizations, metrics and the data filtering operations in detail.

The following section explains the set of visualizations, metrics and the data filtering operations supported by VisCad in detail.

**Table 3.1:** Features Supported

| Feature Dimension | Currently Supported Features |
|---|---|
| **Types of Clones** | Any ( if supported by clone detection tool). |
| **Granularity of clones** | Function, Block (Arbitrary + Structural) |
| **Clone Relation** | Clone Pairs, Clone Class, RCF [68]. |
| **Adaptability to Tools** | Already adapted to CCFinder, Simian, SimScan, NiCad, and iClones. Easily adaptable to tools that report clones with source and line locations. |
| **Adapt. to Languages** | Any (if supported by clone detection tool). |
| **Visualizations** | Scatter plot, Treemap and Hierarchical dependency graph. |
| **Metrics** | Clone class and clone system metric sets. |
| **Filtering** | Manual, metric based, textual similarity based, overlapping. |
| **Source Code Viewer** | Display clone fragment or file, support syntax and difference highlighting. |
| **Scalability** | Even the large systems such as the Linux Kernel releases. |

### 3.3.1 Supporting Multiple Clone Detectors

Developing a clone analysis tool that can work with different clone detectors is challenging since there is no standard format for reporting the detected clones and different clone detection tools use different formats. For example, CCFinder produces clone detection results by reporting the clone pairs. On the other hand, NiCad and Simian [13] report clone fragments by grouping them into clone classes with different output formats. Despite the differences, the source files and line locations of the candidate clones along with their clone pair/class information can be obtained for the majority of the clone detectors. We have used an XML-based file format for reporting clones that acts as an input to VisCad. This format allows VisCad to work with any clone detector that minimally reports the detected clone fragments with their source file names and begin-end line numbers in the files. For each of the currently supported clone detectors, a parser is provided with VisCad that parses the result file and converts it to the XML file format that VisCad accepts as input. The reasons behind choosing XML-based file format is that it is somewhat human readable, supports platform independence, and a number of parsers are readily available. Working with a clone detector that is not currently supported by VisCad requires that its clone detection results need to be converted into VisCad input file format.

The input file of VisCad consists of two parts. The first part contains information specific to the clone detection, and may contain tool name, configuration of the current detection, timing information and an overview of the clone detection results. Although the first part is optional, we keep this because we noticed

This part may contain tool name, configuration of the detection and overview of the result

```
<cloneDetectionResult>
<info tool="Similarity Analyser 2.3.31 - http://www.harukizaemon.com/simian"
processedLines="61258"    processedFiles="680"    duplicatedLines="19429"
        duplicatedFiles="1414"  duplicatedBlocks="342"   processingTime="0.691sec" />

<class ccid="1" nlines="142" nfragments="2">
<source file ="... jhotdraw7/src/main/java/org/jhotdraw/samples/mini/DnDMultiEditorSample.java"
        startline="33"  endline="41"  pcid="1"></source>
<source file =".../src/main/java/org/jhotdraw/samples/mini/MultiEditorSample.java"
        startline="30"  endline="38"  pcid="2"></source>
</class>
.
.
.
.
</cloneDetectionResult>
```

Clone Class Information

**Figure 3.3:** VisCad Input File Format

that many clone detectors report the configurations of detection as a part of the result. Moreover, such information is valuable to the user since the quality of the clone data varies with changes in the configurations. The second part contains the detected clone fragments grouped into clone classes. Each clone class has an integer number associated with it, called ‘ccid’ that uniquely identifies it. Similarly, another integer number called ‘pcid’ uniquely identifies each clone fragment within a particular clone class. In addition to the ‘pcid’, the source file name and the begin-end line numbers are also associated with each clone fragment. An example of input file for VisCad is shown in Figure 3.3. To save space, we only show the basic structure of the file. Recently, a kind of similar file format, called Rich Clone Format (RCF) [68] has been proposed. Considering its importance, VisCad has been also adapted to accept RCF format as well.

### 3.3.2 Visualization of Code Clones

Analyzing and exploring large volume of clone data is a complex procedure, often intractable without proper support. In this regard, information visualization and visual data mining can be the effective solution to deal with information flooding. The advantage of visual data mining processes is that it actively involves users in the data mining process [89]. Because of this, code clone visualizations are added into VisCad with following objectives:

- To provide high level overview.
- To empower users to drill down.
- To remove uninteresting clones where possible.

At present, VisCad supports three different visualizations which are scatter plot, treemap and hierarchical dependency graph. In the following, we briefly describe each of the visualizations.

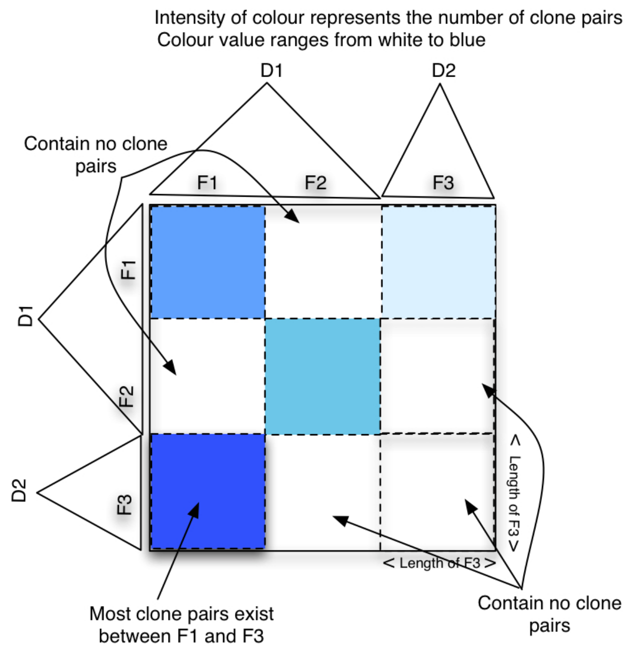



**Figure 3.4:** Scatter plot

**Scatter Plot**

Among various visualization techniques proposed in the literature for code clones, a scatter plot has been widely used and found very effective [105,135]. In VisCad, we also adopt scatter plots as a primary mechanism for clone visualization. It can generate scatter plots of a subject system which are suitable for identifying cloning patterns difficult to spot in other ways. A scatter plot can be viewed as a two dimensional matrix where each cell represents the cloning status between a pair of files or directories. For example, the cell can represent the number of clone pairs common between a pair of files or directories. In that case, a diagonal line in the cell represents the clone pair and both the width and height of the cell represent the length of the related files or directories. Instead of individual clone pairs focus may be on the contribution of the subsystems to the cloning status of a system. To meet this need, the cell can render the cloning status using a colour heatmap. Cells are also labelled in the horizontal and vertical axes.

In a traditional implementation of a scatter plot, the labels in the horizontal and vertical axes are the same, representing the same set of subsystems. This representation does not consider the fact that users may be interested in analyzing clone pair relation across the subsystems only. To address the issue, we have added adjustable axes support in the scatter plot where users can set the subsystems for both axes separately, resulting 50% reduction of axes size in the best case. We have found the technique best suited for inter-project clone analysis where the axes represent the subsystems of two different software systems.

Figure 3.4 shows an example of a scatter plot implemented in VisCad, where the number of clone pairs is represented using a colour heatmap. This form is more suitable for comparing subsystems for various metric values. For example, the cell colour can represent the number of cloned LOC (lines of code) common between

a pair of subsystems.

In VisCad, the following set of operations are supported with scatter plots.

- Selecting any cell and inspecting the corresponding clone pairs and the source files in the source code browser. It is also possible to examine the differences between a pair of source files or code fragments using the integrated *diff* utility.
- Reducing the length of the scatter plot by considering only those files or subsystems that contain clones.
- Reducing cell size to a fixed value, which also helps to reduce the size of the scatter plot.
- Identifying the files or directories involved with a cell using a tooltip by holding the mouse pointer on the cell.

**Treemap**

A treemap [141] has the property of being able to display tree-structured data while making efficient use of space. While its nodes are represented as a set of rectangles, the area and colour property of a rectangle allow us to encode different pieces of information. In our implementation, the treemap preserves the hierarchical structure of subject systems where each rectangle represents a file or a directory. The rectangles representing the files are aggregated to indicate the cloning status of a directory in the system hierarchy. Users can select any rectangles and perform the zoom-in or zoom-out operations. Moreover, the map can be customized so that the rectangles can represent different kinds of information. For example, a user can set criteria so that the area of a rectangle refers to the number of cloned LOC or to the number of clones.

**Hierarchical Dependency Graph**

Clones are more problematic when members of a clone class are scattered in different parts of a software system because this requires changes need to be made in different parts of the system in the event of working with any of the clone fragments. Thus, it is necessary to discover how clone fragments are distributed across subsystems or directories. Moreover, understanding cloning relationship among different subsystems can also reveal their dependencies. VisCad can render the hierarchical organization of a software system along with the distribution of clones using a hierarchical dependency graph. The graph consists of a set of nodes and edges. Nodes are represented as ellipses and edges with straight lines. While nodes can represent files or directories, edges represent containment relationships. The thickness of an edge represents the degree of cloning between two nodes. The width, height and fill colour of an ellipse can be used to represent three different aspects of code cloning. The shape of a node and the thickness of an edge will change based on the cloning status they represent.

To better utilize space, VisCad represents the graph using a radial layout where nodes are arranged in circles and the root of the hierarchical graph is placed at the centre. Nodes that are at the same depth

of the hierarchy, appear at the same distance from the centre. For a large system, representing the entire hierarchy at a time is not useful since this takes a large amount of space. With limited space, the nodes and edges can be so dense that it becomes difficult to identify them clearly. To avoid the problem, the following steps are taken. Instead of visualizing the entire graph, VisCad allows users to select a particular directory. Once a user selects a directory, VisCad only visualizes the graph by filtering those nodes that do not have any cloning relation with the selected directory. Second, nodes representing clone files are collapsed with their parent directory. If a node is selected by clicking the mouse, it is expanded and the nodes that were previously collapsed are displayed in it. The nodes are labelled so that users can identify the directory or file that they represent. Selecting a node by double clicking on it also opens a table that contains various metric values about the selected directory or file. Right clicking over a node, opens a menu. Users can then explore the clone fragments or clone files associated with the node in the source code browser by selecting the 'inspect' menu item.

### 3.3.3 Metrics Supported by VisCad

For supporting in-depth clone analysis, VisCad can compute a set of metrics adapted or reused from previous studies [76, 134]. These metrics can be divided into two groups. The first set of metrics (clone system metric set) relate clones with the organizational structure of a subject system and can be computed for different system boundaries, such as for the entire system, for subsystems/directories or for source files, as per the user's choice. The next set of metrics (clone class metrics set) deals with the clone classes. Results of metric computations can be exported in CSV (comma separated values) format for further processing.

The following metrics are supported by VisCad.

- $TCS$ (Total Clone Snippets) refers to the number of clone snippets within a given boundary. A snippet can be a function or a block of code. The number of clone snippets may fail to reflect the state of code cloning in the target system. A system can have a small number of clone snippets but the snippets may have large number of cloned lines of code. Thus, VisCad includes a set of metrics with respect to the lines of code (LOC).

- $TLOC$ (Total Lines of Code) counts the total number of LOC within a given boundary (i.e., a file or a directory).

- $TCLOC$ (Total Clone Lines of Code) denotes the total number of those LOC that are part of only clone snippets in a given boundary.

- $TCLOC_p$ (Percentage of Total Clone Lines of Code) refers to the percentage of TCLOC over TLOC computed as

$$TCLOC_p = \frac{TCLOC}{TLOC} \times 100.$$

To support investigation on the distribution of clones, VisCad includes another set of metrics at the file and directory/subsystem levels.

- $TF$ (Total Files) computes the total number of files within a given directory or a subsystem.
- $FAWC$ (Files Associated With Clones) For a given boundary, $FAWC$ is the total number of files that contain one or more clone snippets.
- $FAWC_p$ (Percentage of Files Associated With Clones) is the percentage of those files that have at least one clone snippet, which is calculated as

$$FAWC_p = \frac{FAWC}{TF} \times 100.$$

- $CRFL$ (Clone-Ratio of File for LOC) refers to the percentage of cloned LOC in a given file, which is calculated as follows:

$$CRFL = \frac{\text{Total number of cloned LOC in a file}}{\text{Total number of LOC in the file}} \times 100$$

- $QFCL(n)$ counts the number of qualifying files (within a given directory) for which the CRFL value is not less than $n$.

For determining the cloning relationships among files or directories/subsystems, VisCad uses two more metrics, *clone-cohesion* and *clone-coupling* which extend the idea of cohesion and coupling to the domain of code clones. *Clone-cohesion* measures to what extent the code snippets in a file or a directory are clones to the snippets within the same file or directory. *Clone-coupling* of a file or a directory on the other hand measures to what extent the snippets of a file or a directory are clones to the code snippets residing outside of the concerned file or directory. A set of code snippets are said to form a clone class if any two fragments in the set are clones of each other. A clone class is said to be associated with a file or a directory, if at least one of its members is from the concerned file or directory. In terms of software maintenance, clones may be more problematic if most of the clones within a file or a directory have clone pair relations with other clones that are dispersed in the subject system. On the contrary, if most of the clones belong to a directory or a file have clone pair relations within the directory/file, then it is expected that the clones share domain specific similarity, and possibly are less problematic than the other classes.

- **Clone-cohesion:** A file or subsystem/directory is said to be highly clone-cohesive, if most of the members of the associated clone classes are located inside the concerned file or directory/subsystem. Highly clone-cohesive subsystems are desirable since the clones are typically created for reusing implementations of similar functionality. Clone-cohesion (denoted by $cch$) is computed using the following equation:

$$cch = \frac{\sum_{i=1}^{n} \frac{Total\ number\ of\ local\ members\ in\ clone\ class\ i}{Total\ number\ of\ members\ in\ clone\ class\ i}}{Total\ number\ of\ clone\ classes}$$

Here, $n$ denotes the number of those clone classes associated with the concerned file or directory/subsystem. The term "local members" refers to those members of the clone classes that reside within the concerned file or directory/subsystem.

- **Clone-coupling:** For a file or subsystem/directory, high clone-coupling indicates that the members of the associated clone classes have clone-pair relations with many distinct clone classes distributed beyond the boundary of interest. Low clone-coupled subsystems are desirable as they minimizes the effort to understand subsystem functionality in dealing with the local clones. Clone-coupling, denoted by *ccp*, is calculated as follows:

$$ccp = \frac{\sum_{i=1}^{n} \frac{Total\ number\ of\ foreign\ members\ in\ clone\ class\ i}{Total\ number\ of\ members\ in\ clone\ class\ i}}{Total\ number\ of\ clone\ classes}$$

Here "foreign members" denotes those members of the associated clone class, which are not located within the file or directory of interest.

For each clone class, VisCad can compute the following set of metrics.

- $CCS$ (Clone Class Size) refers to the number of clone snippets within a clone class.

- $CCL$ (Clone Class Length) is the total number of cloned LOC for a clone class.

- $CCR$ (Clone Class Radius) indicates how the snippets of a clone class are distributed over the system. A high value indicates that the snippets are scattered widely and difficult to maintain as we need to make changes in different locations. Let us consider that a clone class ($CC$) consists of two snippets ($s_1$ and $s_2$). If they reside in two files $f_1$ and $f_2$, then the $CCR(C)$ is calculated as the average length of paths from the files to the lowest common ancestor directory of $f_1$ and $f_2$. We differ from Gemini by taking the average path length since taking the maximum value may mark a clone class as problematic even when all other snippets are closely located.

Although metrics are important for quantitative analysis, identifying important patterns from a large set of data is difficult. To avoid such difficulties, users can plot the metric values which helps in identifying an anomaly within clone patterns easily. For example, a directory of interest may contain thousands of source files, but users may be interested to identify whether or not the majority of the source files have a large number cloned LOC. The $CRFL$ metric values of those files can provide the answer, and by plotting the metric values for the files we can easily get the answer.

### 3.3.4 Filtering

The first and foremost challenge in clone analysis is the large volume of clone detection results returned by the clone detectors. Not all clones are useful to the user and the objective of the analysis at hand governs the set of the useful clones. Here, the term 'useful clones' refers to those clone fragments that the maintenance

engineers are looking for or are interested in. VisCad supports a set of filtering operations to remove clones that are not useful/interesting to the users. The set of filtering methods supported by VisCad is discussed below.

#### Metric Based

The set of metrics supported by VisCad can be used to perform filtering. For example, clones may be filtered based on the number of cloned LOC. VisCad allows users to specify the condition(s) of filtering (e.g. remove all clones that do not have a minimum of five LOC) and then remove those clones that meet the criteria. Filtering of this kind allows us to remove getter/setter methods that are common place in programming languages (such as Java). Depending on the metrics, the granularity of the filtering operation changes as well. For example, clone files can be removed based on the number of clone fragments or cloned LOC. Similarly, clone classes can be filtered based on their sizes, lengths or radii.

#### Textual Similarity

Although the state of the art clone detection tools can quickly detect clones in a subject system, many of the clone pairs could be false positives. For example, CCFinderX reports two code fragments as a clone pair that have similar structure in variable and function declarations (i.e., similar token-sequences). Developers frequently need to follow specific programming protocols in solving problems that also results in similar code fragments. CCFinder also reports them as clones [76] if their normalized token-sequences are similar. Another clone detection tool, NiCad also exhibits similar characteristics, although it overcomes many of the limitations by extracting structural blocks and applying contextual abstractions/normalizations followed by a line based text comparison. It also supports various configurations for clone detection including blind and consistent renaming of identifiers, statements normalizations/abstractions and filtering, and a dissimilarity threshold in comparison. Depending on the configuration, it applies different degrees of abstraction on the code-base while detecting clones. We experienced that higher degrees of abstraction produces clones that are only structurally similar, and often human judges do not consider them as clones. Figure 3.5 shows an example of such a clone pair detected by NiCad between *Kawa* and *Padclipse Lite* systems in an inter-project clone analysis for text editors. Clones were detected at the block level by applying consistent-renaming of identifiers with a dissimilarity threshold of 30%.

We have thus incorporated textual filtering in VisCad to remove such clones depending on a users' choice. For each clone class, VisCad determines the clone fragment that maximizes the sum of the textual similarity to all other fragments of that class. We call this fragment as the 'leading clone fragment' for that class. If the textual similarity between the 'leading clone fragment' and any other clone fragment in the clone class falls below a given threshold value, we remove the fragment from the analysis. We discard an entire clone class from the analysis when the textual similarities between the leading clone fragment and all other non-leading clone fragments of that clone class fall below the threshold value. The textual similarity between two clone

| kawa-cvs-20030716/kawa/kawa/lang/SyntaxRule.java Lines 371 - 377 | padclipse-lite-0/org.eclipse.ui.ide/src/org/eclipse/ui/internal/ide/actions/OpenWorkspaceAction.java Lines 299 - 304 |
|---|---|

```
{
    out.writeObject(pattern);
    out.writeObject(pattern_nesting);
    out.writeObject(template_program);
    out.writeObject(literal_values);
    out.writeInt(max_nesting);
  }
```

```
if (property == null) {
   result.append(CMD_DATA);
   result.append(NEW_LINE);
   result.append(workspace);
   result.append(NEW_LINE);
           } else {
```

**Figure 3.5:** Two code fragments that are detected as clone pair due to applying higher degree of abstraction in clone detection

fragments, $A$ and $B$ is calculated by determining the common token sequence between them using the longest common subsequence algorithm. Equation 3.1 describes the similarity function.

$$\text{Similarity}(\text{A, B}) = \frac{2 \times |\text{A} \cup \text{B}|}{|\text{A}| + |\text{B}|} \tag{3.1}$$

Here, $|A \cup B|$ denotes the number of ordered tokens common between $A$ and $B$. $|A|$ and $|B|$ refer to the number of tokens in A and B respectively. The threshold value ranges from 0 to 1 and can be set by the user.

#### Overlapping

Clones may be overlapped each other and removing those overlapping clones also reduces the size of the result set. For example, VisCad detected 695 cases in JHotDraw (version 7.6) [9] system where two clone fragments overlap each other, and the clones were detected using Simian. When we set the threshold value (30%) for filtering out the overlapped clones, VisCad removed clones from the pairs of clone fragments that were overlapped by more than 30%. Among the two overlapping clone fragments, VisCad removes the one that has the larger overlapped cloned LOC. With these settings, we reduced the result set by 23% (a total of 413 clones and 173 clone classes were removed).

## 3.4 Analyzing Clones with VisCad

This section explains how VisCad can be used in clone analysis with a case study.

### 3.4.1 Subject Systems and Research Questions

In this case study, we used NetBSD (version 8.2.0) [11] and FreeBSD (version 5.1) [6] as the subject systems and they were selected because of their non trivial size of the source code. While the current version of the NetBSD contains 6,475 C source files, version 5.1 (latest stable release at the time of writing) of FreeBSD contains 3721 C source files (we considered only the kernel source code). The sizes of the systems were

calculated after removing comments and black lines. Both FreeBSD and NetBSD are Unix like operating systems and derive from the same predecessor, the 386BSD. As a result, it is expected that they share a large number of similar code fragments. We addressed the following three research questions and using VisCad, we seek answers for them.

- RQ1: Do FreeBSD and NetBSD share any copied code fragments from the same source at the function level? If yes, what are they?
- RQ2: Is it the case that the similar clone fragments form a large number of small size clone clusters? Is there any exception to this?
- RQ3: The overall similarity observed between these subject systems by Kamiya et. al. [84] was a decade ago. From then to now, both systems have gone through considerable changes. Do they still share the same level of similarity in their most recent releases? A previous study [43] answered a similar question but they used Linux Kernel and NetBSD as the subject systems. We deliberately select this question to show the flexibility of answering such questions with VisCad.

### 3.4.2 Clone Detection

To answer the first two research questions, clones were detected using the clone detection tool NiCad (version 2.6.3). NiCad was applied to detect function clones of at least five lines in their pretty-printed format where the UPIT (Unique Percentage of Item Threshold) value was set to 0.3. NiCad uses the UPIT as a dissimilarity threshold to determine whether two code fragments are similar to each other. When the UPIT is set to 0.0, two code fragments that are exactly similar in their pretty-printed line based comparison are reported as a clone pair. To detect near-miss clones (code clones that are not exact similar but share some degree of similarities.) we need to set UPIT value more than 0. Setting the UPIT value 0.3 allows two code fragments reported as a clone; allowing 30% dissimilarity in their pretty-printed (optionally normalized/filtered) line based comparison.

### 3.4.3 Preparing for Analysis

At first, we instructed VisCad to import the clone detection result and specified the location of the code base of the subject system. Since our objective was to analyze cross cloning between the systems, we also informed VisCad to load cross-clones only by simply selecting a checkbox on the import dialog box. This was the filtering we applied in the first place. After loading the result set, clones were grouped into clone classes and also mapped to the directory structure of the subject system. Finally they were displayed through the *Clone Class Tree* and *System Navigation Tree.* This completes the categorization and presentation phases. From this point, the user gains the full control of analysis and she needs to select the view that best matches with her goals.

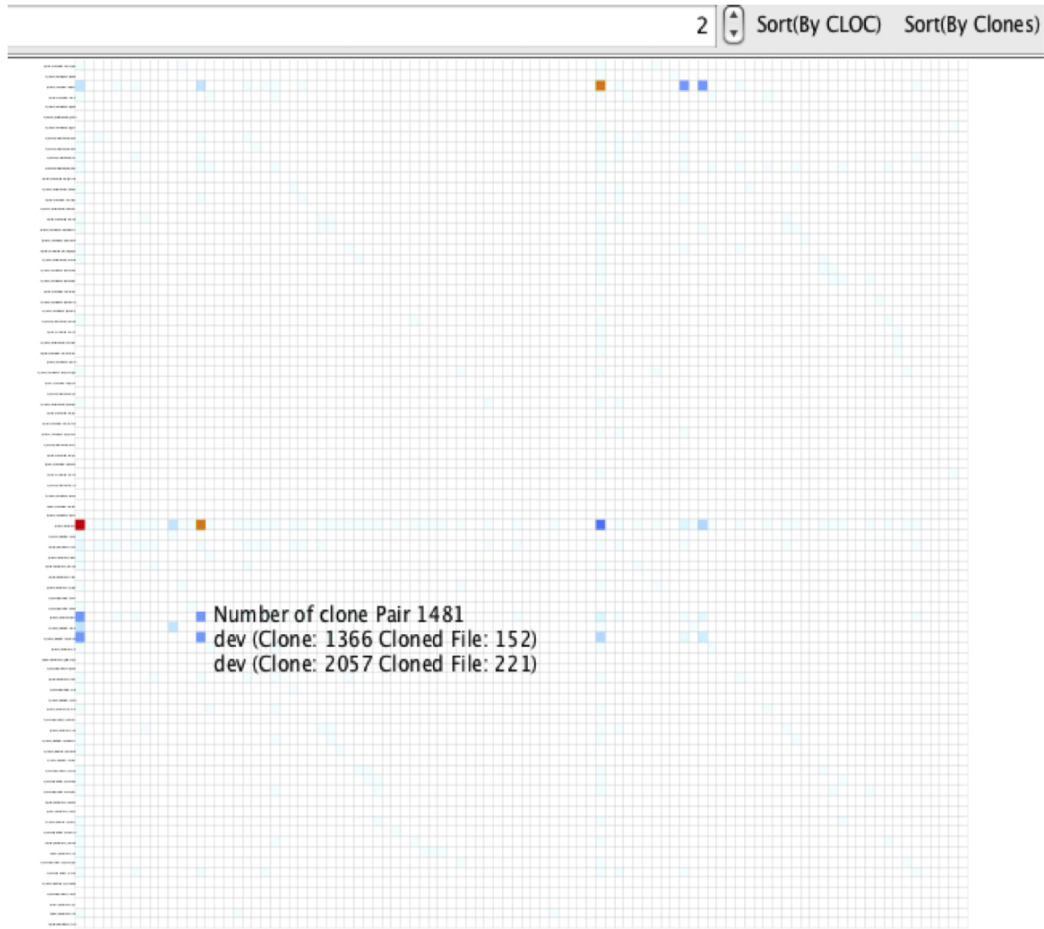



**Figure 3.6:** Analyzing cloning relation with a scatter plot

### 3.4.4 Answering Questions

The following sections describe how VisCad helped finding answers to the research questions.

#### Answer to the RQ1

Due to the increased number of the open source projects available today, developers tend to search for existing solutions on the web before trying to implement by their own. Both NetBSD and FreeBSD have the same predecessor and open source in nature. It is expected that the developers of the systems may copied code from one system to another or from a common source. To identify this, we have used the clone detection result at NiCad's UPIT 0.3, since it allows us to detect code fragments that are copied verbatim (as an exact clone) and also those that are adopted with modifications (as a near-miss clone). We used the scatter plot to identify the cross-cloning between the systems and we found a number of cross-cloning between them which provide positive result to the first question (see Figure 3.6). For example, the *arch* folder of NetBSD has clones with several different directories of the FreeBSD. Hovering the mouse pointer over the cells allows us to identify the related directories of the FreeBSD system. We found that *arch* has cloning relations with

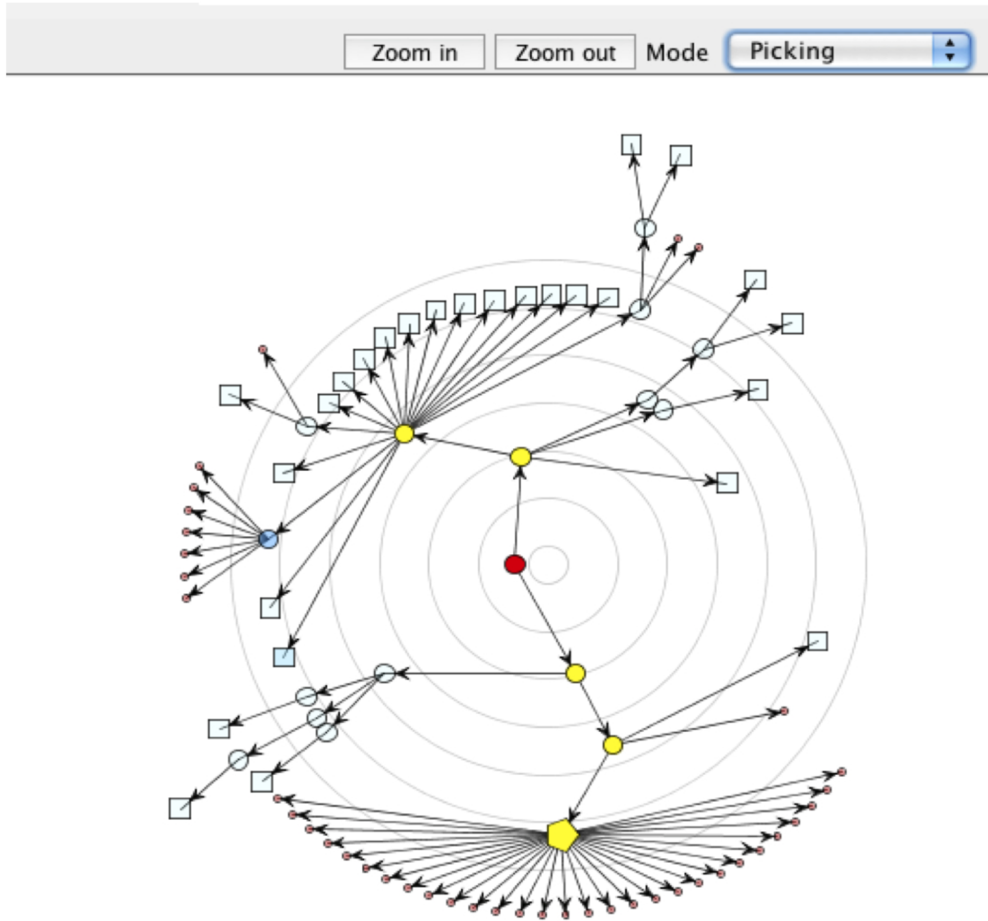



**Figure 3.7:** Cloning relation of NetBSD5.1/dev/ic subdirectory (the pentagon shaped node) with other files and directories.

*powerpc*, *i386*, *sparc64*, *amd64* which contain code for porting the operating system to different platforms, and this is not unexpected.

Using the scatter plot we examined the sub-directories inside the *arch* folder and found that the code related with porting the system on to different platforms (such as *atari*, *playstation2*, *amd64*, *hp300*) are organized into different folders inside the arch. This reflects the difference in their source code organization. We found 1481 clone pairs between *FreeBSD8.2.0/dev* and *NetBSD5.1/dev* directories (they contains code specific to various device drivers). The *FreeBSD8.2.0/dev* has mostly clones with *pci* and *ic* (this provides core functionality for the device drivers) sub-directories and both reside within the *NetBSD5.1/dev* directory. Right clicking on the cell bring up a popup menu and we selected the 'explore' command to analyze the clone pairs spread between these places. This allow us to identify and analyze the source code of the clone pairs with the source code browser. Interestingly, we found evidence of source files that are copied and adopted to the FreeBSD from NetBSD. For example, the dp83932.c file is copied from NetBSD and adopted in FreeBSD. We were also interested to see what are the files and directories of FreeBSD that have cross cloning with the ic directory of NetBSD. Although it is possible to obtain the answer using the same scatter plot, but it has the

**Figure 3.8:** A treemap showing the distribution of clones for the clone class 9034. Files that contain the clones are rendered with red colour.

limitation that it cannot group related files. This limitation becomes more notable when the size of a scatter plot is very large. This made scatter plots not suitable for the current objective. We used the hierarchical dependency graph to get the answer. While the mouse is on a cell representing cross cloning between *ic* and other directories, we brought up the popup menu by right clicking on it and selected the 'view hierarchical dependency graph' command. This helped us to identify target information on the fly without switching context (see Figure 3.7). We found that the *ic* directory has cross-cloning with the *sparc64*, *aic7xx*, *usb* and *ath* directories of the FreeBSD systems. All other directories that do not have any cloning relations with the *ic* directory were removed from the graph. Except *sparc64*, all others are located within *FreeBsd8.2.0/dev* directory. We found that only one file (lsi854.c) under the *sparc64* is involved in cross cloning. The graph helped us to view the file in the source code browser and we identified that this is also another file copied from NetBSD. The *NetBSD5.1/dev* directory contains the source.

While *FreeBSD5.1/dev* directory exhibits little cross-cloning with other directories of the FreeBSD 8.2.0, we found that *NetBSD8.2.0/dev* directory has higher cross-cloning with the *NetBSD5.1/external* directory, which is indeed highest in number(1569). After reading documentation, we found that the directory contains several third party software. We found that most of the files are copied verbatim although the way they are organized differs between the systems. Scatter plots also allow us to locate notable cross cloning between *fs*, *net*, *compat* and *contrib* directories of FreeBSD system with *fs* (contains code for file system functionalities), *net* (contains routines for networking protocols), *compat* (provide code for emulating binary compatibility with various non-NetBSD operating systems) and *dist* directories of NetBSD.

### Answer to the RQ2

NiCad groups similar code fragments into clone classes where any two fragments are clone to each other. Examining the clone classes is also an alternative way to analyze code cloning. To obtain the list of clone classes along with their metric values, we selected the subject system from the clone class tree, right click to bring up the popup menu and selected the 'clone class metrics' command. We examined the list of clone classes that VisCad provides along with the metric values. We sorted the clone classes by their length and quickly identified that most of them contain a small number of fragments (majority of the cone classes contain two clone fragments). However, there are a few exceptions and among them the largest clone class contains 17 clone fragments. We then used the treemap to see how the clone fragments of that class are scattered in different parts of the systems. We opened the treemap of the subject system and selected the target clone class. The files that contain those clones were marked with red colour in that treemap (see Figure 3.8). It is also possible to identify the files by simply pointing the mouse pointer on a cell representing a file or directory in the treemap. Surprisingly, the distribution of the files are similar for both subject systems and the name of the directories are same for both systems (*ar5212* and *ar5416*). However, at the top level, those files are located in *dev* (for FreeBSD 8.2.0) and in *external* (for NetBSD 5.1) directories. The treemap view helped us to locate the fact that in both the systems the same files are involved in cross-cloning for the target clone class except the ar9280.c file, which is not present in NetBSD 5.1.

### Answer to the RQ3

Using CCFinder as the clone detection tool, the previous study [84] on these two systems reported 25,621 clone pairs where around 40% files of FreeBSD and 36% files of NetBSD are involved in cross-cloning. We were interested to see whether there is a notable change in cross-cloning in the latest stable releases of these systems since the size of the system has increased significantly. To answer the first two research questions, we used NiCad as a clone detection tool. Even at the function level, NiCad detected 636(9.82 %) clone files in the NetBSD5.1 and 585 (15.72 % ) clone files in FreeBSD 8.2.0 (VisCad provided us the value as a part of the metric calculation). We obtained lower values compared to Kamiya et al. [84] due to the fact that we used different clone detection tool and only detected clones at the function level whereas they used CCFinder that detects clones at the unstructured block level and works differently. Thus, to compare the results with the previous finding, we need to detect clones with the same tool and settings. Since VisCad can import clone detection result from CCFinder, comparing the results with the previous finding was only a distance ahead. We detected clones between the FreeBSD 5.1 and NetBSD 8.2.0 using the same clone length for CCFinder. Later, we used VisCad to get the clone coverage value for files and the results reveal a significant increase in the value.

## 3.5 Usability Evaluation

Once we developed a version of VisCad, we conducted a user study to identify the usability issues. This was important to further improve VisCad before conducting a formal comparative study. VisCad was evaluated in a user study with five participants. All are graduate students (at the M.Sc. levels of study), male and had ages between 24 to 32. Moreover, all participants had prior programming experiences and three of them were aware about the code clones and performed some degrees of clone analysis. We arranged a training session of one hour on a particular day where we explained VisCad to the participants followed by a live demonstration of clone analysis with the tool using JHotDraw as a subject system. Then they used the tool in a laboratory setting for another hour and completed five tasks. After that, they filled out an evaluation questionnaire that asks participants about different aspects of VisCad on a seven point Likert Scales, also contained open questions to identify the usefulness and the limitations of the tool. All of the participants liked the idea of analyzing clone detection results of different detectors in a single environment. The scores from the usability evaluation indicate that the tool is easy to use and can help to understand the cloning status of a system. However, some issues need to be addressed. For example, the views lack customizability, do not support filtering on the fly, and switching from one view to another is not that much intuitive.

## 3.6 Related Work

There are a number of techniques and tools that have been proposed in the literature to visualize code clones. A detailed list of them can be found in Roy and Cordy's technical report [135]. Johnson [79] applied *Hasse diagram*, a graph based visualization to represent textual similarity between files. The diagram shows the clones and source files as nodes and their relationships with edges. Later, he proposed exploring the files and clone classes via hyper linked web pages [80]. Although hyperlinks allow quick navigation, they lack the overview and selection features. Rieger et al. [126] proposed a set of *polymetric views* to identify the duplication within a subject system. Among them, *scatter plot* is the most popular approach for clone visualization, which helps visualizing both inter and intra system clones. Higo et al. [70] proposed an enhanced scatter plot to overcome the scalability problem of the classical one, and later Livieri et al. [105] visualized clones detected from 400 millions lines of code using colour heat map, another variant of scatter plot. Ueda et al. [150] developed a visualization tool, Gemini [150], which incorporates CCFinder [84] for clone detection. While Gemini provides *scatter plot* and *metric graph* views for clone visualization and analysis, VisCad incorporates *scatter plot*, *treemap*, and *hierarchical dependency graph* and can analyze clone detection results of a large set of detectors including CCFinder.

Tairas et al. [146] developed an Eclipse plug-in that integrates CloneDR, a clone detection tool, to Eclipse and produces graphical views of the clone detection results. They visualized the clones through the *visualizer view* that represents clones within source files using bars, stripes, and kinds. Bars represent the source files,

clones within source files are presented with stripes, and clone classes are identified with kinds where kinds are distinguished with different colours. Similar to their plug-in, VisCad also allows users to locate and inspect source files that contain clones of a particular clone class. Moreover, VisCad can analyze cloning relationships between subsystems/directories, which is not supported by their tool. Since CloneDR also reports the detected clones using source file names and line locations, it can be easily adapted to VisCad.

Jiang et al. [75] extended the classic idea of cohesion and coupling to code cloning. They developed a clone recovery framework that uses the clone detection results from CCFinder, and graphically represents the clone cohesion and coupling between the subsystems. Later Jiang and Hasan [75] proposed a framework for large scale code clone analysis. They also presented a visualization called the *clone system hierarchical (CSH)* graph which helps in identifying how clones are scattered across a software system. While their *CSH* graph uses tree layout, the *hierarchical dependency graph* of VisCad provides the same information with radial layout utilizing the display area more efficiently. Furthermore, the graph can be configured through a wizard to display different cloning information. The source code of the clone fragments or clone files can be directly accessible from the graph and can be examined in the source code browser.

Kapser and Godfrey [86] developed a tool CLICS [3], which categorizes the clones according to a documented taxonomy and enables query based clone exploration. CLICS can display the cloning relationships between subsystems with a hierarchical containment graph and the visualization is performed through a program called LSEdit. VisCad significantly differs from CLICS both in terms of objectives and supported features. ConQAT [4] is a quality assessment toolkit for software systems that supports clone detection and analysis. For clone detection, ConQAT has its own engine and for analysis it offers treemap, clone visualizer and family visualizer views. Although the last two views can group and visualize related clone files, they do not support visualization of cloning dependency among subsystems. VisCad allows users to analyze the cloning dependency among the subsystems through *hierarchical dependency graph* and supports both clone class and clone system metric sets, many of which are not available in ConQAT.

Zhang et. al. [159] proposed a clone query system that stores clone detector's output in a relational database and allows human expert to define a set of filters in terms of standard SQL queries to locate target clones. They developed the system as an Eclipse plug-in, called Clone Visualizer that uses the output of the *Clone Miner*, a clone detection tool. Standard graphical views, such as bar charts are used to visualize the result of clone queries. Although the benefits of the clone query system cannot be ignored, forming complex queries are difficult. Similar to the Clone Visualizer, VisCad also uses sortable tables. Moreover, the source code browser in VisCad lists the set of clone files, clone fragments and clone pairs associated with a clone class or a subsystem/directory. Users can also change the selection of clone files or clone fragments to analyze the target clone pairs.

Fukushima et al. [60] developed an alternative code clone visualization technique that uses a graph to locate diffused clones. Each node in the graph represents a clone set. Nodes that they are loctated in the same file are connected with edges and form a clone set cluster. Nodes that connect different clone set clusters

are called diffused clones that exist in different files implementing different functions. Although VisCad's purpose is different from their work, it can easily be adapted to VisCad. Moreover, using VisCad one can locate clone classes that are associated with a large number of files and these clone classes represent the clones that are diffused in different files.

Harder and Göde [68] introduced a clone data exchange format, called Rich Clone Format (RCF) capable of storing clone detection result for both single and multiple versions of a program. The data model covers information reported by state-of-the art clone detectors and can be extended to meet any future requirements. A set of Java API is provided to access, modify and to calculate code clone metrics from the stored data. To support clone inspection and understand the evolution of clone fragments, they developed CYCLONE [17,68] that uses a set of views to analyze clone data stored in RCF. Although there are several common features between VisCad and CYCLONE, it is tailored toward clone evolution analysis. VisCad is also adapted to use the clone data stored in RCF.

Adar and Kim [19] provided a code clone exploration system called SoftGUESS by extending GUESS [18], a Graph Exploration System. While SoftGUESS focuses on understanding the evolution of clones, VisCad focuses on understanding the cloning status within a subject system through different metrics and visualizations, and serves as an analysis environment.

As we see although a great many clone visualization techniques and tools have been proposed, they focused on different directions for different purposes. Also most of them are based on a particular clone detection tool. Furthermore, as far as we know only a few of them are available [4], [86], [68], [150] for public use. Ours is the first attempt to provide a general set of visualizations and metrics including many filtering options in a common framework that can even work with the results from different clone detectors.

## 3.7 Conclusion

Code clone analysis is a challenging task as this involves identifying useful cloning facts from a large volume of data produced by the clone detectors. In this regard, VisCad can help a human expert to locate useful clones through visualizations, removing clones that are not useful using various filters, and with a large set of metrics. During clone analysis, users can have an overall picture of cloning status through *scatter plots*, *treemaps* or *hierarchical dependency graphs* and then explore deeper into directories and files of interest. During such exploration, users may inspect the clone fragments through the *source code browser*, identify differences at the source code level using the integrated *diff* utility or use metric values to gain insight, and can remove clones from the study that are not useful. Thus, VisCad facilitates, (a) evaluation of the overall system, (b) guidance to explore, and (c) refinement of analysis, which are the requirements of a clone comprehension tool as identified by Kapser and Godfrey [87]. To the best of our knowledge this is the first attempt that enables code clone data from different detectors to be analyzed in a single environment. Thus, VisCad provides an opportunity to work with clone detector of choice while at the same time taking advantage of the features available in VisCad. An early prototype of VisCad that uses NiCad as the clone detector, has been published

as a tool demo at IWSC 2011 [24]. This new version described here extends VisCad in many ways and the tool is available for public use as an open source program so that researchers not only can use the tool with other clone detectors but also can reuse VisCad features and visualizations techniques in other domains of interest. Interested readers are referred to the VisCad homepage [14].

# Chapter 4

# Improving Clone Evolution Comprehension: A Visual Endeavour

## 4.1 Motivation

Clone evolution analysis is helpful to understand the nature, effects and reasons for cloning. The data collected from the evolution of clones also helps us to make better decisions in managing clones. For example, the fact that aggressive refactoring is not the proper solution for solving problems of code clones was identified by Kim et al. [91] by analyzing the evolution of code clones in two Java systems. Studying the evolution of clones can guide us detecting changes in clones that can help to separate potentially harmful clones from the others. As a result clone evolution analysis has gained much importance in the research community and a number of genealogy extractors are available that can track clones across different versions of a program. In the previous chapter, we discussed the challenges in analyzing the clones in a version of a program. Such a situation becomes worse when the objective is to analyze the evolution of clones. The challenges are handling the large volume of clone data produced by the extractors and to comprehend the evolutionary changes properly. In this regard, visualization can assist maintenance engineers to have an overview of clone evolution, identify useful patterns and look for low level details. Although a number of visualization techniques and tools are available for analyzing clones in a single version, very few tools and techniques are available for clone evolution analysis and visualization. To the best of our knowledge, Adar and Kim [19] first provided clone evolution analysis support through SoftGUESS, that supports three different visualization means to comprehend clone evolution. Recently, Harder and Nils [68] present visualization aids for clone evolution analysis with CYCLONE. However, these tools fail to answer many typical questions related with evolution and does not consider the rich set of change information available in source code repositories. Although a large volume of work has been done to support software evolution analysis [99, 152], there is a marked lack of support in clone evolution analysis.

The objective of this work is to provide support for clone evolution analysis. To assist developers in clone evolution analysis, we have developed a clone genealogy extractor using AIST CCFinderX as the underlying clone detection tool, which is a major improvement over CCFinder [84]. CCFinder is considered as a state-of-the-art clone detection tool that has higher precision, although the recall is lower than some other tools [36].

The genealogy extractor supports automatic classification of evolution patterns. We have extended VisCad to build the genealogy extractor. We hypothesize that developers need to look for the genealogies that have change patterns other than same since they have the potential to cause maintenance implications. We present a set of visualization aids combined with data aggregation and filtering mechanism that can provide high-level overview of evolution in the first place, allow developers to drill down if required and can apply filters to reduce the volume of data need to be analyzed. In general, we make the following contributions.

1. We have proposed a set of visualizations along with a data filtering mechanism for clone evolution analysis that uses both evolutionary and temporal metrics to guide developers in clone evolution analysis.
2. We have extended VisCad to develop a genealogy extractor that supports automatic classification of a number of evolution patterns.
3. We have implemented the proposed visualizations along with the filtering mechanism by further extending VisCad. In this thesis, we refer to the component of VisCad that is responsible for clone evolution visualization as *Clone Evolizer*.

The rest of this chapter is organized as follows: Section 4.2 explains the clone genealogy extractor. Section 4.3 introduces *Clone Evolizer*. The set of metrics, visualizations and the filtering mechanism are discussed in Section 4.4, 4.5 and 4.6. Section 4.7 describes a case study. Section 4.8 outlines the relevant research and Section 4.9 contains the conclusion.

## 4.2 Clone Genealogy Extractor

This section describes our clone genealogy extractor (see Figure 4.1). The data collected from the extractor is used by the *Clone Evolizer* to visualize code clone evolution. Kim et al. [91] identified six evolutionary change patterns that can describe the changes of a clone class. Although our genealogy extractor follows the same model, the definition of some change patterns have been adapted since our extractor uses modified technique to map the clone classes with higher accuracy. In the following, we explain the methodology for detecting clone genealogies and the evolutionary change patterns considered.

### 4.2.1 Methodology

A clone genealogy extractor is a software component or a tool, that takes a number of versions of a software system as input and determines the set of clone genealogies. Our clone genealogy extractor maps clone classes across the versions and explains how clone classes evolve with the evolution of a software. The genealogy extractor works as follows:

1. It takes as input a set of versions for the subject system.

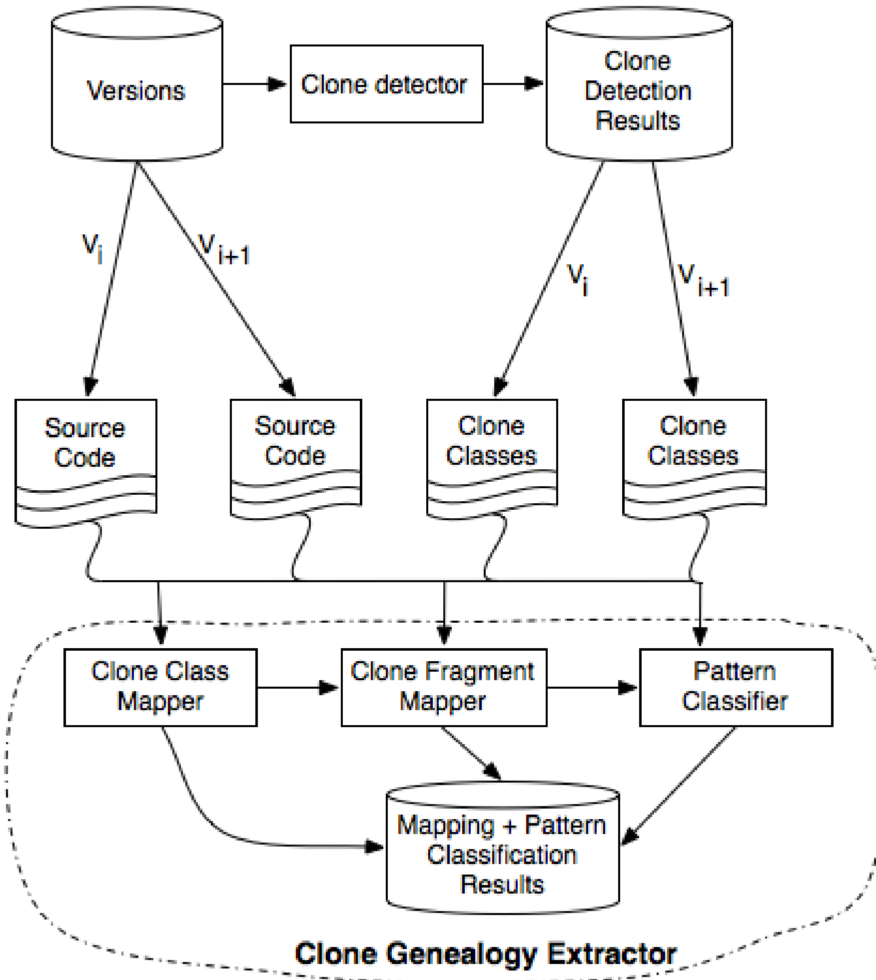



**Figure 4.1:** Clone Genealogy Extractor

2. Next, we use a custom script to batch run CCFinder on these versions to detect clones and to collect the pretty-printed result files. By default, CCFinder generates the result on a binary file which is converted to the pretty-printed result file using a command-line instruction.

3. The set of result files, the intermediate files generated by the CCFinder and the source code of the versions are fed into the extractor. The CGE then determines the mapping between the clone classes for each two successive versions.

4. The generated mapping produces the clone genealogies for the set of versions. Given any version and a clone class (sink) of that version, CGE can identify the corresponding lineage that identifies the source of the lineage and explains how the clone class evolves from the source to the sink.

To map the clone classes in the successive versions, we use both *TextSimilarity* and *CombineSimilarity* functions. We keep the *TextSimilarity* function same as of Kim et al. [91], but change the *LocationOverlapping* function with the CombineSimilarity function. The location overlapping function uses a file and line-based

location tracker built on top of *diff*. The problem is that, the location overlapping function did not work well when lines are reordered because *diff* cannot detect such changes. The objective of the *LocationOverlapping* function is to map a clone class from its current version to the next. To fulfill the same objective we have used the file and line independent *CombineSimilarity* function. In the following sections, we briefly describe the two similarity functions.

### 4.2.2 Text Similarity

Let $C_1$ and $C_2$ are the two code fragments. The text similarity between $C_1$ and $C_2$ is determined by calculating the common token sequences between them using Longest Common Subsequence (LCS) algorithm. The value is normalized and ranges from 0 to 1 where 0 means no similarity and 1 indicates the highest similarity. The *TextSimilarity* function can be described by equation 4.1. Here $|C_1|$ and $|C_2|$ refer to the token sizes of the code fragments $C_1$ and $C_2$ respectively. $|C_1 \cap C_2|$ is the number of common ordered tokens between the code fragments $C_1$ and $C_2$, calculated using the longest common subsequence (LCS) algorithm. A threshold value for the text similarity is used to map the clone classes between two successive versions. Although adjustable, we have tested the mappings with similarity thresholds ($Sim_{th}$) of 0.3 and 0.8.

$$TextSimilarity(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1| + |C_2|} \tag{4.1}$$

### 4.2.3 Combine Similarity

We use the *CombineSimilarity* function to mitigate the limitations of the *LocationOverlapping* function that takes advantage of the findings by Reiss et al. [124]. His empirical study reveals that source code lines can reasonably be tracked by considering both context and content. To determine the context similarity between two code fragments $C_1$ and $C_2$, we consider four lines up and four lines down of both code fragments, determine the similarity and normalized the value between 0 and 1. We also consider the lines of the two code fragments and determine their content similarity and normalized the value between 0 and 1. A combine similarity is used that takes 0.6 times of the content similarity and 0.4 times of the context similarity. We use 0.5 as the threshold value for mapping two code fragments using the *CombineSimilarity* function since the same value is suggested by Reiss [124] and we also found good results using it.

### 4.2.4 Mapping Clone Classes

We map the clone classes that have highest *TextSimilarity* and *CombineSimilarity* scores. If the highest *TextSimilarity* score differs from the highest *CombineSimilarity* score, we map both of them to avoid the ambiguity. Since the token sequence is same for all fragments that belong to a clone class, to determine the text similarity between two clone classes, we consider only the first fragment as the representative of that clone class and determine the text similarity between the representatives. However, to determine the combine similarity score, we consider all possible pairs of clone fragments between two clone classes.

### 4.2.5 Mapping Clone Fragments

The token similarity value is the same for any two clone fragments belonging to two different clone classes that are mapped previously by the genealogy extractor. As a result, we did not use the text similarity function for mapping individual clone fragments. Let, for two consecutive versions the genealogy extractor maps clone class $CC_i \in V_i$ to another clone class $CC_{i+1} \in V_{i+1}$. To map individual clone fragments of these two clone classes, we use the *CombineSimilarity* function. We set the origin of clone fragment $CF_{i+1} \in CC_{i+1}$ to that clone fragment of $CC_i$ that has the highest combine similarity value. If the same combine similarity value appears more than one cases then we use the file name to break up a tie.

### 4.2.6 Automatic Change Pattern Classification

Our genealogy extractor supports automatic change pattern classification, which is one of the important requirements of a clone genealogy extractor. We should note that at this point, we have mappings of both clone classes and individual clone fragments of two mapped clone classes. In this section, we describe how the pattern classifier classifies different evolution patterns and how we can define them in terms of similarity value.

- Same: There are no changes between a pair of mapped clone fragments. We can define the change pattern in terms of the similarity value in the following way,
  $TextSimilarity(CF_1, CF_2) = IdSimilarity(CF_1, CF2) = 1.0$

  Here, *IdSimilarity* function is similar to that of *TextSimilarity* function, but instead of tokens it considers the ordered sequence of identifiers only. In this case, *IdSimilarity* is determined without applying the consistent renaming of the identifiers.

- Add: Detecting add pattern is fairly straightforward as this requires us to find the clone fragments in the current version that do not have mapping to the previous version. Add pattern can result from either the creation of a new clone fragment or due to the deletion of a clone fragment from a clone class in the previous version that has been detected in another clone class of the current version.

- Delete: The unmapped clone fragments in the next version are reported as deleted in the next version. Similar to add, detecting this pattern is also simple as this requires to track the clone fragments of the current versition to the next version.

- Consistent and Inconsistent Changes: Among various change patterns, determining these two are the most difficult due to their subtlety. If all clone fragments in the previous version are present in the next version with text similarity 1.0, we cannot simply say that they are the same since the developers may have changed the identifiers consistently. Similarly, with $Sim_{th} < tokensimilarity < 1.0$, it cannot be said that the clone fragments are changed consistently since the developers may have changed the fragments with different set of identifiers. We have considered two different cases. First, we consider the

case where we have two clone classes that are mapped with text similarity value of 1.0. This requires to further investigate their similarity at the identifier level. If the $IdSimilarity(CF_1, CF2) < 1.0$ then we need to check whether the developers change the identifiers consistently. If yes, we mark this as an id-consistent change pattern. Otherwise, it is marked as an id-inconsistent change pattern. For this purpose, we apply consistent renaming to both code fragments and measure the similarity value again. If the result is 1.0 then we can decide that all identifiers are consistently changed in the next version, which is expected in copy-paste programming practices. Failure to do so often produces bugs reported in the previous studies [73, 103]. Second, we consider another case where two clone classes are mapped with $Sim_{th} < TextSimilarity < 1.0$. In that case, we check whether the same set of changes are applied in all the mapped fragments in the next version. We apply consistent renaming again and determine the differences between each pair of mapped fragments using the *diff* algorithm. If the differences are same, then the clone class is marked as consistently changed. Otherwise, it is marked as consistently changed.

Consistent renaming technique identifies the set of unique identifiers in a code fragment and replaces all occurrences of those identifiers with a fixed naming scheme. For example, all occurrences of the first identifier are replaced with $x1$, all occurrences of the second variables are replaced with $x2$ and so on. As a result, if we apply consistent renaming on two code fragments and the programmer renames all the identifiers of a fragment in $CC_i$ consistently in $CC_{i+1}$. we will get similarity value 1.0 at the identifier level. This can be expressed in the following way,
If $TextSimilarity(CF_i, CF_{i+1}) = IdSimilarity(CF_i, CF_{i+1}) = 1.0$, where, *IdSimilarity* is determined after applying consistent renaming, then all the variables of $CF_i$ are consistently changed in $CF_{i+1}$

## 4.3 Requirement Identification

We realized the need of a tool supporting clone evolution analysis during our previous work on clone genealogies at the release level [138]. Although that experience guided us in developing the current system, we also explored previous work on both clone and software evolution analyses to identify possible room for improvements. One of the major challenges in analyzing code clone evolution is the large volume of raw data produced by the clone genealogy extractors capturing various aspects of the evolution. To deal with the problem, we have added visualization techniques that can provide both high-level overview of the evolutionary data and also portray low-level details based on user selections. Although the high-level view supports data aggregation, we have also added data filtering support as a primary means of removing unnecessary information and to guide maintainers to find useful information. Although it is possible to formulate powerful query through SQL like interface, we opt for dynamic queries (DQ) [22, 142] since formulating useful queries may be difficult and may require much time for the novice users. We have collected both spatial and evolutionary metrics (such as code authorship information obtained from the version control repository) that can provide

useful insights about the system under investigation. We found that working at the clone class level provides opportunity for data reduction and helps maintainers to find possible opportunities for refactoring.

We have implemented the proposed visualizations along with the filtering and target pattern identification mechanisms in a tool called *Clone Evolizer*. The tool has been implemented by extending VisCad, our framework for supporting large scale clone analysis. The tool can be used to answer the following questions:

- How does a clone class evolves during the evolution of a software system?
- What are the clone evolution change patterns observed during the life time of a clone class?
- Who are responsible or related with code cloning?
- In which points clone genealogies are discontinued?
- How do clone files are evolved?
- How do developers copy-paste programming practices change during the evolution of a software?

## 4.4 Data Collection and Metrics Computed

Version control systems capture the evolutionary changes of a software system. Such historical information can aid in comprehending the system and in identifying important patterns in the development. As a result, it has been widely used in many software evolution visualizations to build the underlying data model. In this thesis, we also make use of the data available in there. Although it is possible to use any other version control system with little modifications, current implementation uses Subversion (SVN) to primarily show the effectiveness of our visualizations. We consider both spatial and evolutionary metrics related with both clone classes and clone files, which are the central entities in our visualizations. The set of metrics considered are listed in Table 4.1.

Multiple versions of a software are collected from a SVN repository. Users need to select the time interval between two successive versions, the start, and the end version numbers. *CloneEvolizer* then automatically collects and stores the target versions in the local machine. In addition, we also collect the revision date, developer information associated with each version using the SVN *blame* command. The evolutionary clone data is collected from our clone genealogy extractor that maps the clone classes across the versions of a software system and uses AIST CCFinderX as the underlying clone detection tool (see Figure 4.2).

## 4.5 Mapping Data to Visual Entities

We sought for 2D representations for visualizing evolutionary information because of their simple interface and free of occlusion problem that are observed in 3D view [152]. We hypothesize that such interface requires less cognitive effort to be comprehended by the developers because of their simplicity. We consider different attributes of a software entity to encode different metric values as shown in Figure 4.3. In the following, we

**Table 4.1:** Metrics considered in Clone Evolizer

| Entity | Metric | Description |
|---|---|---|
| Clone Class | Pop | Number of fragments in a clone class |
| | Length | The total number of lines |
| | Radius | Radius refers to the longest distance from the clone files associated with a clone class to the lowest common ancestor directory of those files |
| | FAWC | Number of files associated with a clone class |
| | DAWC | Number of developers edited a clone class |
| | TSP | The total number of same evolution patterns in its lineage. |
| | TAP | The total number of add evolution patterns in its lineage. |
| | TDP | The total number of delete patterns in its lineage. |
| | TIDCP | The total number of Id-consistent changes in its lineage. |
| | TIDIP | The total number of Id-inconsistent change pattern in its lineage. |
| | TCP | The total number of consistent change patterns in its lineage. |
| | TIP | The total number of inconsistent change patterns in its lineage. |
| | TUC | The number of times a clone class evolves without same change pattern from the source node |
| Clone File | LOC | Number of lines of code |
| | CLOC | Number of clone lines of code |
| | ELA | Number of clone lines added from its creation to the current point |
| | ELD | Number of clone lines deleted from its creation to the current point |
| | ELC | Number of clone lines changed in a file from its creation to the current point |
| | ELNC | Number of clone lines left unchanged in a file |

present a set of views and explain how developers or the project managers can interact with them to better understand different aspects of cloning in a software system.

### 4.5.1 Clone Class Evolution

We use rectangular boxes to represent clone classes and they are arranged in a number of columns. The width and height of a clone class can be encoded with two different pieces of information. The horizontal position of the clone classes depend on the version they belong to and software versions are arranges from left to the right. If a clone class($CC_a$) belongs to the version $V_a$ and maps to another clone class ($CC_b$) which belongs to version $V_b$, then $CC_a$ appears on the left of $CC_b$ if $V_a < V_b$. This forms a two dimensional matrix where columns indicate versions and each column consists of the set of clone classes that belong to that version. The vertical axes are labelled with version number so that user can easily identify which clone class belongs to which version.

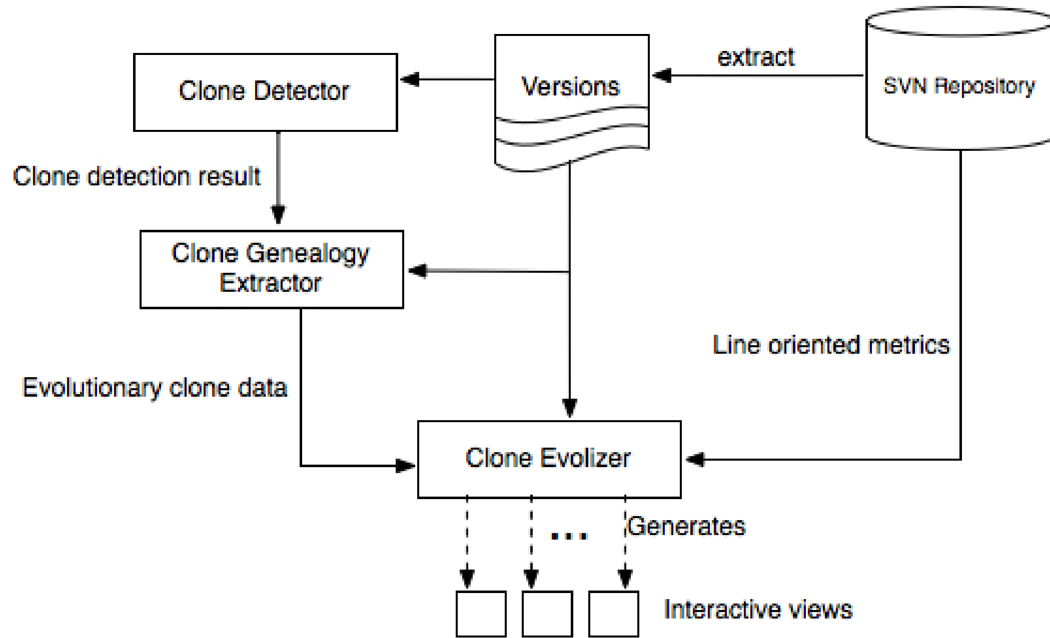



**Figure 4.2:** Data collection and view generation

An example of a clone class view is shown in Figure 4.4. The mapping of the clone classes between two successive versions is depicted with a line that connect the boxes representing those clone classes. The vertical arrangement of the clone classes is such that no two lines cross each other, which assist in understanding the evolution at ease. Since changes like the appearance/disappearance of a clone class are more interesting from the analysis perspective, we use different colours to encode this information. For example, blue color depicts a new born clone class while a red color indicates that the clone class is removed in the next version and thus ends a genealogy. The dead clone classes are appeared at the end of the column because they lack any connection in the next version. This assists to quickly locate the places where clone genealogies disappeared during the evolution of a software system.

In order to identify which clone classes are edited by which developers, each clone class is divided into a set of pixel lines where the height of the line indicates the percentage of lines of code contributed by a developer and the number of lines indicate the number of developers created the clone class. The lines are arranged in such a way that the more a developer contributes to the cloning, the more she appears at the top. Each developer is encoded with a colour value that is used to fill the pixel lines.

### 4.5.2 Clone Fragment Evolution

Visualizing at the clone class level provides the opportunity to reduce clone data since the classes aggregate the details of the clone fragments. However, this comes with the cost of losing fine grain details about what evolutionary patterns are observed and how the clone fragments change with those patterns. To mitigate this problem, the clone class evolution view is complemented with a clone fragment evolution view (see Figure 4.5)

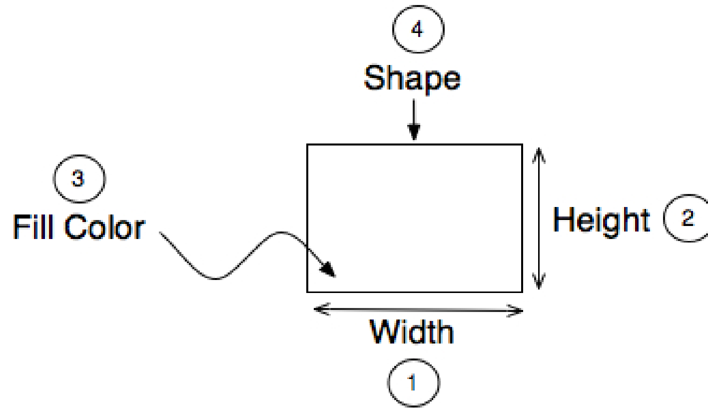



**Figure 4.3:** Encoding different metric values within an entity

that makes use of the brushing technique. Sweeping the mouse over a clone class show the complete genealogy by visually describing the evolution patterns of the clone fragments.

Here, each clone class and the clone fragments it contains are represented by the same rectangular boxes. The clone fragments in the successive clone classes are connected with edges to determine their evolution and different colours are used to depict different evolution patterns. For example, edges representing same change pattern is displayed by green color while id-inconsistent changes are represented by red color. Similar to the clone class view, the added or deleted cloned fragments are highlighted with blue and red color. We use a customizable color map for highlighting the change patterns that can be adjusted. Since developers are more often interested to see the changes at the source code level, a code view is attached with this view that highlights the changes between a pair of code fragments whenever the users point the mouse over an edge.

### 4.5.3 Clone File Evolution

This view is similar to that of the clone class evolution view with the exception that instead of clone classes, each rectangular box represents a clone file. The horizontal width of the boxes can be configured to represent different metric values, whereas each rectangular box is horizontally split into a number to pixel lines to depict the developer information. Instead of both clone and non-clone lines, we consider only clone lines and display how they are edited by different developers because of two reasons. First, instead of each clone lines, we use the percentage of the lines contributed by the developers within a file to make the view scalable. Visualizing both clone and non-clone lines will take a large amount of space and provide greater details whereas our objective is to use this view to depict high-level details and complement with another view that provides the low-level details. Second, trying to visualize a lot of details within a single view may distract users concentration and might increases the cognitive effort requirement.

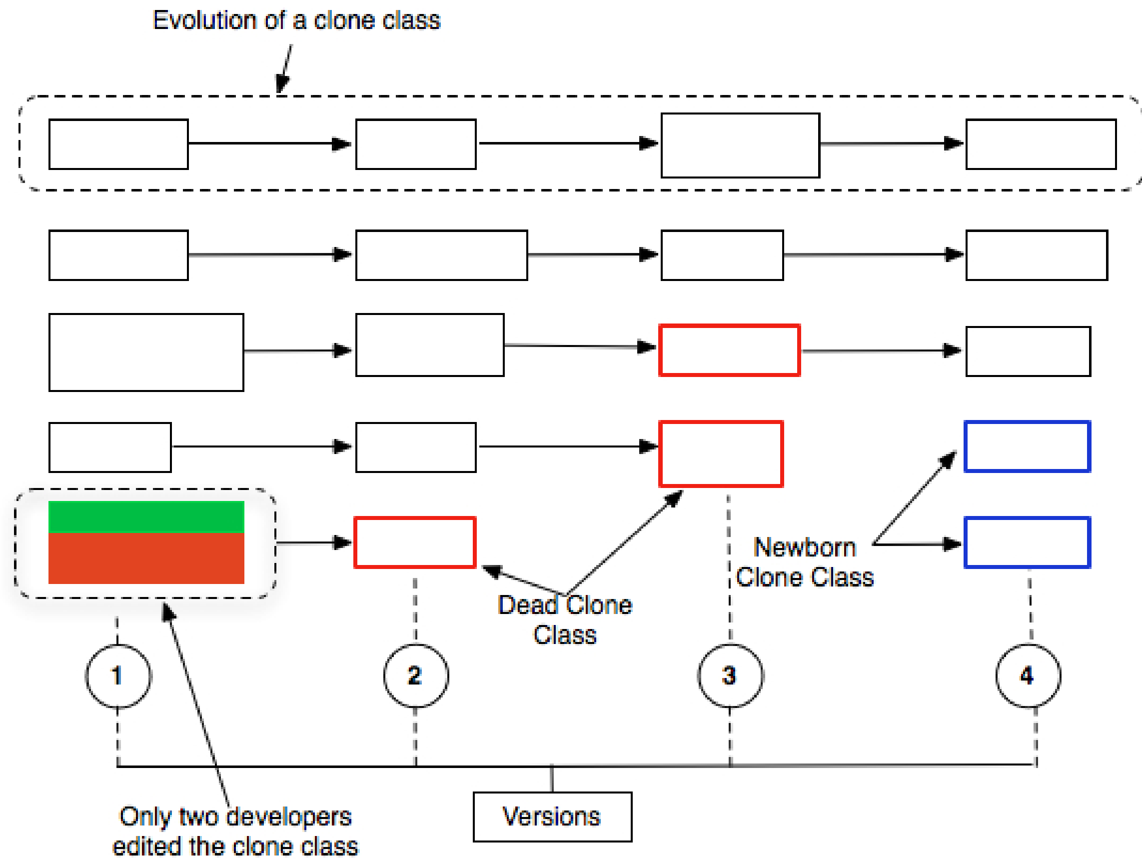



**Figure 4.4:** A clone class evolution view. For simplicity, we show the dense pixel display in only one clone class to encode code authorship information.

### 4.5.4 Line-based Evolution

This view depicts the complete evolution of a single file. It uses a tabular representation where each cell acts for a line of code, each row represents the evolution of a particular line of code and each column refers to a particular version, arranges in order of time from left to right. We use global line position display [152] that allows us to eliminate the edges we use in the previous views to show the evolution of an element and also preserve the order of lines. Although it takes empty spaces, such a view is more suitable to identify which lines are added or deleted or consistent during the evolution of the examined file. While we use colour encoding to depict code ownership information, we use different shapes to encode the cloned and non-cloned lines (rectangular boxes are used to represent cloned lines while diamond shapes are used to represent non-cloned lines).

### 4.5.5 Developers Cloning Profile

We were motivated to create another visualization focusing on the correlation between developers and clones because we believe that identifying when and how developers copy can give us insights in managing clones.

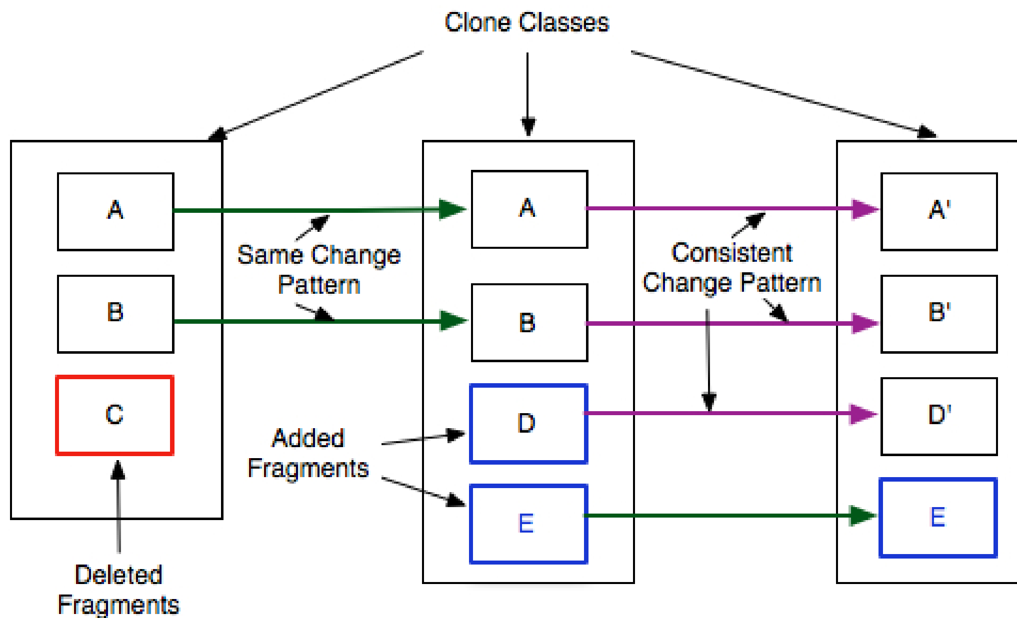



**Figure 4.5:** An example of a fragment evolution view. For simplicity, we consider a hypothetical genealogy of a clone class that spans in three versions only

All of the above views can answer clone code ownership information, which is important for both single and multi-version or evolution analysis. For example, a clone researcher may want to identify the reasons of cloning. She can use these views to determine which clone is created by which person and can interview them to reason about those clones. In another case, a project manager may want to know whether the developers are frequently creating clones or not. However, some questions are not that much intuitive to answer with them because the developer information is scattered in different parts. For example, we may want to know when a developer started cloning or how the copy-paste programming practices of a developer changes during the evolution of a software or it is the case that a few developers are responsible for creating clone codes.

Developers cloning profile can help us answer these sort of questions (see Figure 4.7). In this view, rectangular boxes are again used to represent individual developers and the number of developers within a version are arranged in a column. Colour is used to encode the code ownership information and each developer is assigned with a unique colour value. The color code consistency is preserved throughout all the views. The width and height of a rectangular box can be encoded with different pieces of information. In this case, the width of the box represents the number of clone classes associated with a developer and the height indicates the number of clone line contributed by a developer.

If the same developer also creates clones in the next version, the corresponding rectangular boxes are connected by a line. The boxes can be sorted based on different metrics, such as the percentage of cloned lines. In that case the slope of the line refers to the copy-paste programming practices of a developer.

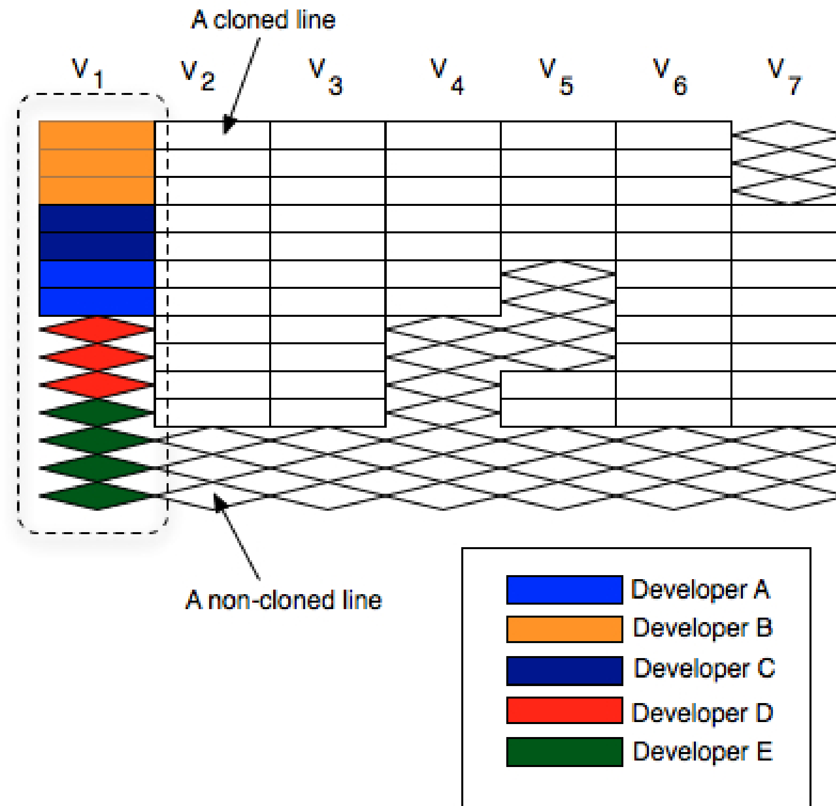



**Figure 4.6:** The figure shows a hypothetical line evolution view of a file. Although the figure shows both cloned and non-cloned line evolution, the view can be configured to show only one of them. Colours are used to encode code authorship information. For example, five different developers (represented by five different colors) were contributed to the file in the first version. For simplicity, we show the code authorship information in only the first version.

## 4.6 Data Filtering

Filtering refers to the process of removing data that are not useful or interesting in the current context of analysis. For example, if a user wants to analyze the evolution of those clone classes that propagates with at least one change pattern other than the same, then clone classes that are evolved without any change can be removed since they are of no more interesting to the user. We have added both temporal and evolutionary metrics that can guide users to look for the target patterns. Although it is possible to specify filters as queries in SQL, we opt for a more simple solution since writing complex queries requires expertise, more cognitive efforts from the users side and may be difficult for the novice users.

As an alternative way of formulating queries by typing in command languages (like in SQL), a dynamic query (DQ) approach has been developed that supports direct manipulation, provides immediate feedback and allows users to rapidly cope with the information overload by removing unwanted information. In a typical application of dynamic query, users specify their queries by adjusting sliders or selecting buttons or using check boxes, which provide both overview of the data and the opportunity to seek target information

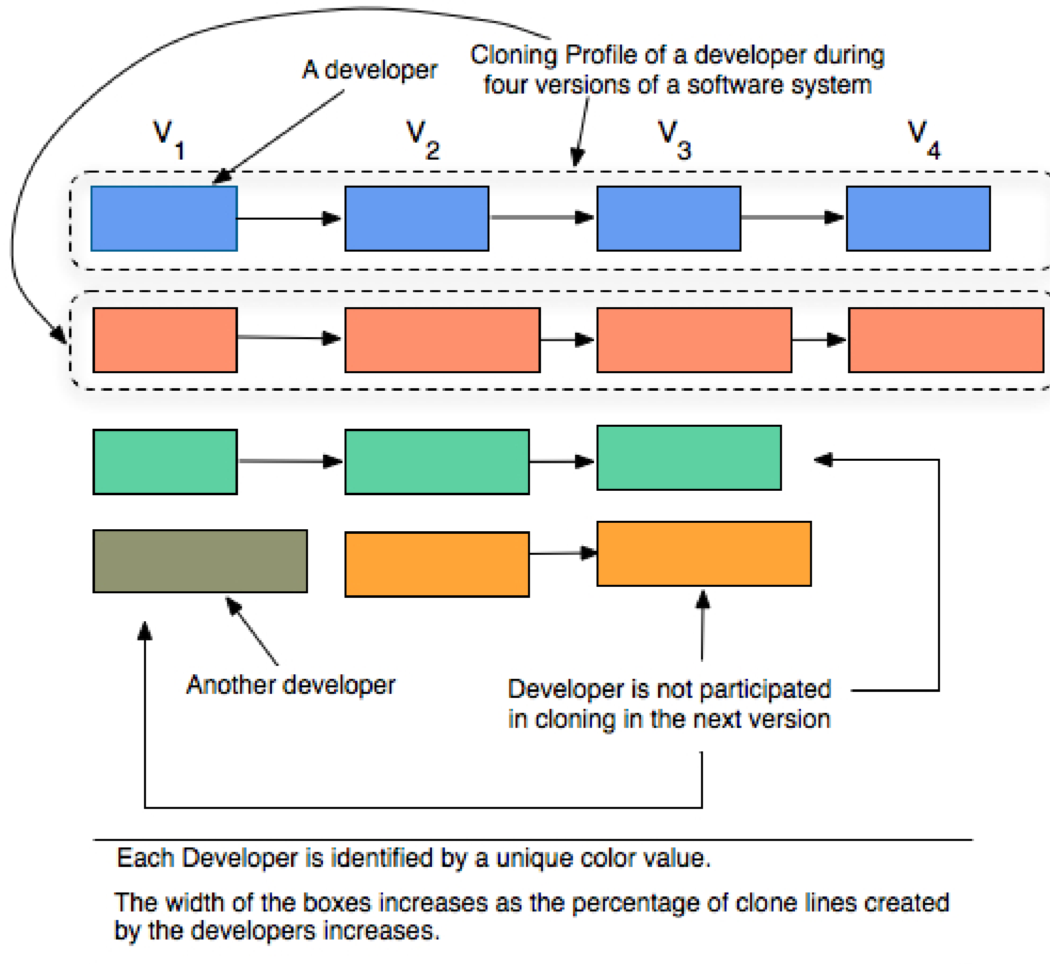



**Figure 4.7:** The copy-paste programming practices of the developers are highlighted through developers cloning profile view

visually. A number of applications and tools have already been developed based on dynamic query, such as Dynamic HomeFinder [157], Dynamaps [121], PhotoFinder [85], FilmFinder [21] and SpotFire [20]. In these applications, the conditions specified by a set of sliders are conjunctive. To increase the expressiveness of dynamic query, a graphical interface has been developed that can express complete set of boolean expressions. Fishkin et al. presented a new interface to construct compound queries by overlapping the multiple magic lens filters [59]. Recently, dynamic query approach has also been applied to support proximity query [40].

We have used dynamic query based interface that allows users to specify queries and apply filtering incrementally, providing more flexibility in formulating complex queries. The query interface is integrated with a *History Manager* component that keeps track of the query applied and the data set affected by the query. This allows a user to roll back to the previous state if she wants to undo the effect of the filter applied most recently. In the following, we explain the use of the dynamic query interface with a hypothetical scenario.

The filter composition panel consists of a set of metrics associated with individual rows. Each row is

also equipped with a dynamic query (DQ) slider, composition options (through a combo box), text boxes for indicating the starting minimum and maximum values, and a NOT option (through a check box) to reverse the effect of the query. For example, to select clone classes of length 100 or more, we need to adjust the minimum value of the DQ slider to 100 corresponding to the Length metric. However, we can reverse the selection (that is selecting the clone classes whose length is less than 100) by selecting the corresponding NOT option. The composition option specifies how the filter in a row can be merged with the filter found in the next row. The row position of a metric can be changed by selecting a row and interacting with the up or down button found in the filter composition panel.

Consider the case, where the maintenance engineer, Sally, wants to analyze the evolution of clone classes of a subject system. She is only interested to investigate those genealogies that have at least one change pattern other than the same. The interface highlights the lowest and highest values of all metrics. We hypothesize that all other change patterns except the same is unsafe due to their potential to cause maintenance implications and the *unsafe_count* metric refers to the number times that a clone class evolves with any unsafe change patterns. Thus, Selly sets the DQ sliders of *unsafe_count* metric to zero to select those genealogies that have only same change pattern. The composition option dictates how multiple queries can be combined. For example, to find clone classes that have more than two clone fragments and associated with only only one file, Selly needs to set the minimum value of the DQ slider corresponding to the *Pop* metric to two and both the values of the DQ slider corresponding to the *FAWC* metric to one. Since among this two metrics, *Pop* appears just before *FAWC* in the filter composition panel, we need to set the composition option of the *Pop* to AND. Filtering can be done incrementally. Whenever the user applies a filter, both the visualization and metric table are updated to reflect the changes. Users can again use the filter composition panel to create another query and apply that one on the recently changed data set. Our experience suggests that such an approach allows users to create complex queries in stages, provides deeper understanding about how queries affect the data set.

Since it might so happen that during the incremental query creation procedure, users apply a query but realize that the target pattern can be better understood by changing the most recent query a little. The *history_manager* component, integrated with the filter composition panel, keeps track of the queries applied along with the data sets affected by the query and thus supports both undo and redo operations to cancel the effect of recently applied queries, rolling back to a previous state or cancel the roll back operation.

## 4.7 Case Study

*Clone Evolizer* has been implemented by extending VisCad. In this section, we briefly describe how the tool can be used in understanding code clone evolution. For this case study, we have used eleven revisions of the dnsjava [5] that are at least one month apart from each other. Since only one developer is involved in the development of the system, we have ignored the involvement of developers, although the tool can be

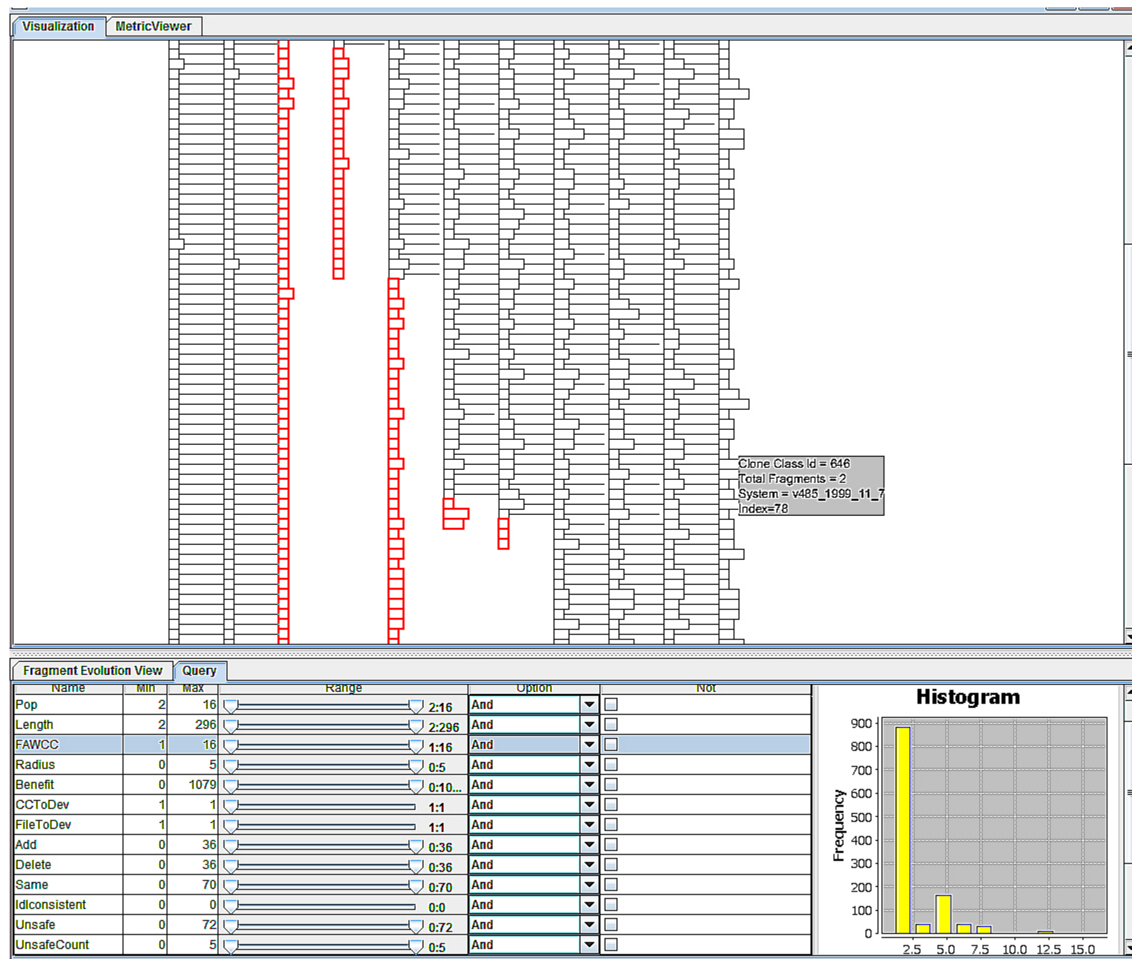



**Figure 4.8:** An example of a clone class evolution view (top panel) and a filter composition panel (bottom panel). Each rectangle in the clone class evolution view represents a clone class and each column represents the set of clone classes in a version. The filter composition panel consists of a set of sliders (bottom-left) to create and apply dynamic queries to the clone class evolution view. It also contains a histogram (bottom-right) that shows the distribution of a metric value for different clone classes.

configured any time to visualize the developer data. Figure 4.8 shows the evolution of clone classes, along with the metric values computed (located on the top panel). The bottom panel helps to create the dynamic query to filter unnecessary data on the fly. For each metric, we can show only the minimum and the maximum values using a DQ slider. However, this fails to provide us an idea about the distribution of metric values for the clone classes. For example, $FAWCC$ metric refers the number of files associated with a clone class. From the figure we can see that the value ranges from one to 16. However, it cannot show whether the majority of the clone classes are associated with a few files. To understand the distribution, we have added another view that shows the histogram of the metric. The figure reveals that the majority of the clone classes are associated with less than three files.

The clone class evolution view can help to identify the genealogies and the point in which they are terminated. The rectangles with red colour indicates the termination of a genealogy in a particular version.

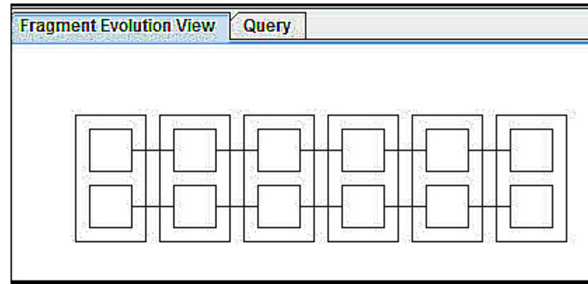



**Figure 4.9:** An example of a fragment evolution view

The figure reveals that a large number of clone classes are disappeared in version 4, which indicates a major restructuring of the code base, whereas in the later versions, clone classes are evolved without any changes or with little modifications. By putting the mouse on a rectangle allow us to identify the clone class (provide us clone class id, total number of fragments in it and the version number). To determine the evolution of a clone class along with its fragments, brushing is required over the rectangles. For example, Figure 4.9 shows the fragment evolution view of a clone class (clone class id 308) in the version eleven. The figure reveals that the clone class evolves without any change with only two code fragments. Here, an outer rectangle identifies a clone class and the rectangles inside it represent clone fragments that belong to that clone class. A higher level abstraction for clone evolution analysis can be obtained through the evolution of clone files. Figure 4.10 shows the evolution of clone files in the dnsjava system for the selected versions. Each rectangle represents a file, each column represents a version, and the colour value indicates the status of the file (whether a file is created in a version or removed in the next version). From the figure, we can easily identify that a large number of clone files removed in version 4 and 5. Brushing over a rectangle (that represents a clone file) shows the evolution of cloned and non-cloned lines of the selected file in a line evolution view and allow developers/maintenance engineers to understand the changes at the source code level.

## 4.8 Related Work

Although clone evolution analysis has gained much interest in recent days, only a few work on clone evolution visualization have been reported in the literature to date. However, software visualization has been an active research area for a long time. When augmented with evolution analysis, they can capture interesting patterns in evolution that are difficult to find otherwise. This section describes software evolution visualization followed by clone evolution visualization.

### 4.8.1 Software Evolution Visualization

Ball and Eick [56] described a set of views (matrix view, cityscape view, bar and pie charts, data sheets and network view) to hint changes in a software using visual metaphors. Proposals for linking views to increase comprehensibility of software changes are also presented. Collberg et al. [41] developed a system using graph drawing techniques that depicts changes in inheritance graph, call graph and control-flow graph

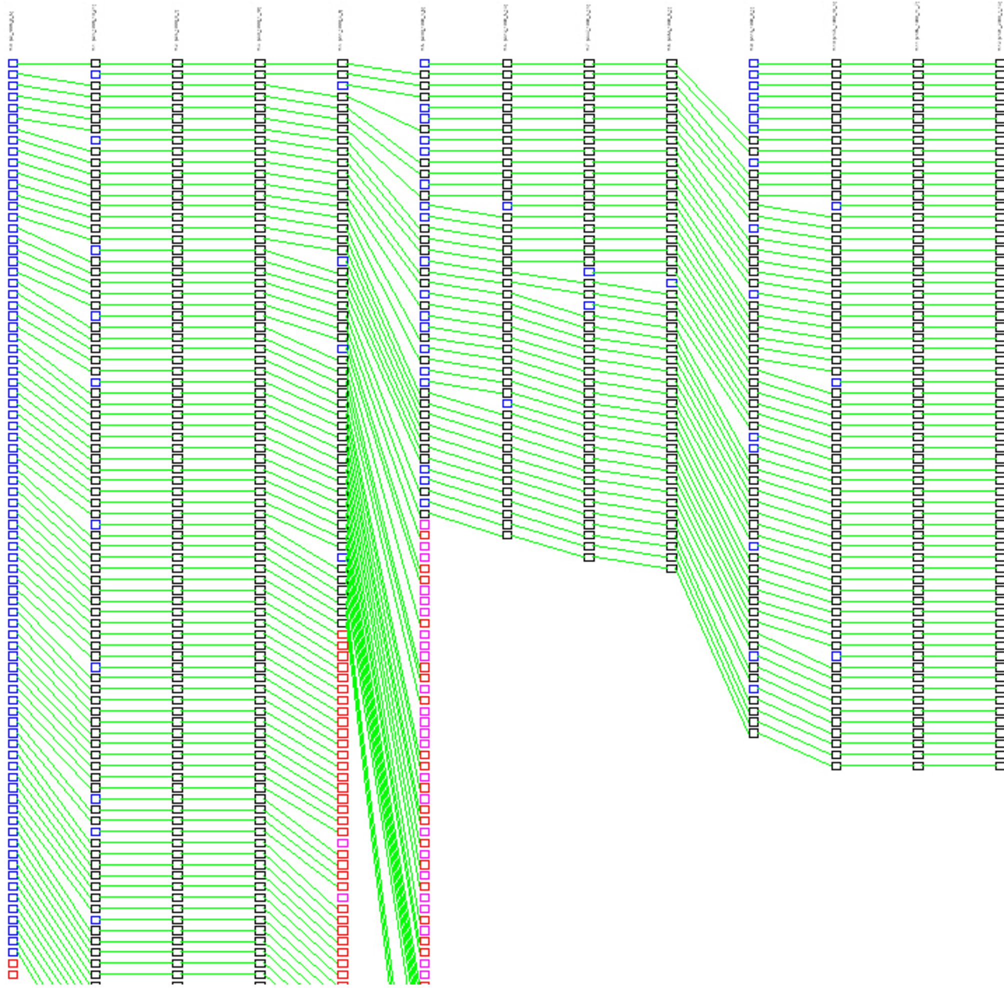


**Figure 4.10:** An example of a clone file evolution view. Each rectangle represents a file and each column represents the set of files in a version. An edge between two rectangles represents the evolution of a file. While blue colour indicates a new file, the red colour indicates that the file has been deleted in the next version.

using colour coding . Rysselberghe and Demeyer [151] proposed a scatter-plot visualization to quantify the changes in files during software evolution. Evolution Spectograph [158], developed by Wu et al. can depict punctuated changes (changes that are sudden and discontinuous). It uses metrics and color coding to portray those changes. Fractul figures [49] facilitate identifying and comparing development effort among the developers. The fractul value obtained from the fractul figure can be a better indicator to portray the changes in effort distribution during the evolution of a software. Instead of depicting software entities through visual metaphors, Pinzer et al. [120] visualized the change and trends in evolution metrics using Kiviat diagrams and graphs. Wettel and Lanza [155] used 3D visualization based on city metaphor to depict the evolutionary characteristics of a software. Gonzalez et al. [65] proposed a set of four views linked within a single environment where each view addresses different aspects of evolution of a system. While the time-line view characterize the revisions in different time scales, the structure evolution view helps to compare software entities during software evolution. The third view (metric view) shows changes in metric values using parallel

coordinates during the evolution and the last view shows indirect class coupling relation. Although it lacks formal user study, the integration of the views can facilitate cross-role tasks.

The evolution matrix, proposed by Lanza et al. [99] forms the foundations of our visualizations that shows the evolution of software entities through rectangles and the changes are depicted by varying the width or height of it. However, we have enhanced the visualization by encoding developers information through dense pixel displays. The information is collected from the version control repository as a line oriented metric value. Line oriented visualization is first addressed in SeeSoft [57], that maps line oriented statistics to visual entities. However, it does not support evolution visualization of the software entities (like files). CVSscan [152] is a system that supports visualization of evolution of a file where colour encoding is used to represent line changes and author information extracted from CVS repositories. To collect evolution information from the repository, CVSscan uses another tool called CVSgrab [153]. Although effective, the visualization is limited to one file only. The file based evolution visualization has been used in other places to facilitate visual data mining [154].

### 4.8.2 Clone Evolution Visualization

Supporting clone evolution analysis was first addressed by Adar and Kim [19] through SoftGUESS, a system for clone evolution exploration. SoftGUESS is developed on top of GUESS [18], the graph exploration system, that models the evolution of a software using graphs. In addition to three distinct views of evolution, SoftGUESS uses Gython, a SQL type query language to dynamically colouring and filtering the graph. Although query is a powerful mechanism to identify important patterns of cloning, formulating the query could be difficult as this requires more cognitive effort from the developers side. Harder and Nils [68] developed a multi-perspective tool for code clone analysis, called CYCLONE that offers visualization of clone genealogies (known as evolution view) that uses simple rectangles and circles to represent software entities. Circles represent clone fragments and the clones of the same class are packed into a rectangle. Lines illustrate how cloned fragments are evolved across versions and color coding is used to represent the changes in the clones. However, the view lacks query mechanism and does not address other dimensions of clone evolution (e.g. structural evolution or changes in code authorship information). Recently, Saha et al. [140] presented an idea for clone evolution visualization using the popular scatter plot. In their proposed approach, scatter plots show the clone pairs associated within a pair of software unit (file, directory or package). Based on the type of clone genealogies they are associated with, clone pairs are rendered with different colors. Selecting a clone pair through user interactions (double clicking on a clone pair in the scatter plot) shows the associated genealogy in a genealogy browser. The proposal facilitates developers or maintenance engineers to identify evolutionary change patterns of the clone classes in a particular version and then provide a way to call for genealogy browser to dig deeper. However, it does not provide overall characteristics of the genealogies. Moreover, due to the large number of clone pairs, selection and useful pattern identification in such a scatter plot are difficult. As a result, researchers developed variations of the traditional scatter plot. Finally, it is still not clear about the usability and usefulness of the technique since no implementation is available yet.

## 4.9 Conclusion

In this chapter, we present the working principles of a clone genealogy extractor we have developed by extending VisCad. This chapter also presents a set of visualizations and a dynamic query based filtering mechanism to gain insight into code clone evolution. In addition, it also introduces *Clone Evolizer*, a component of the extended VisCad that implements the proposed visualizations and the filtering mechanism. Although a large number of software visualizations have been proposed in literature there is a marked lack in clone evolution visualization, and only a few visualizations were proposed in the past. We not only consider different aspects of clone evolution but also correlate the evolution with other information (e.g. code authorship) extracted from a SVN version control repository. We are currently working towards completion of *Clone Evolizer* tool that we plan to make open source. We are expecting that *Clone Evolizer* would help the researchers, developers and maintenance engineers to gain insights about clone evolution and in making informed decisions for properly managing clones. It could be also an integral part of a clone management system.

# Chapter 5

# Which Clones to Look at First?

## 5.1 Introduction

Since clones have both positive and negative impacts, there is an ongoing debate on whether clones are harmful or not. However, researcher agree that to the least clones need to be detected and analyzed to identify hidden problems (such as anomalies in change propagation), and to ensure the safety of the software. In response, a large number of clone detection techniques and tools have been developed [135, 136], and in general, they report the detected clones by grouping similar code fragments into clone classes or clone groups. One of the biggest challenge while analyzing clones is the volume of the raw clone data. Developers or maintenance engineers may not always have sufficient time to analyze all clone fragments due to short deadline or have difficulties about where to start the analysis. This calls for the support of recommending clones that need to be dealt with as a priority.

Several heuristics have been developed to remove false positives from the clone detection results. However, a system with a large clone base may have a considerable amount of clones even when we apply various filtering including textual, metric based or manual approaches to remove the uninteresting clone fragments or clone classes. A previous study by Kim et al. [91] suggests that aggressive refactoring is not the best solution for clone management as many of them are propagated without any changes during the evolution of a software system, some fragments are short lived while many are propagated with same repetitive set of operations. This requires to the least careful selection of clones to be analyzed to best utilize the invested time. We hypothesize that we need to be careful and also analyze those clones that have the possibility to change in a way that might create buggy code. Consider the case where developers have close deadline for releasing the next version of the program but also want to analyze and remove as many clones as possible. In such a case, a recommendation system would be helpful that can guide maintenance engineers to avoid those clones that are less likely to change and those that are more likely to change with patterns that have the potential to cause clone related bugs.

To address this issue, we use evolution patterns as a first class candidate to identify problematic clone classes. Our rationale is that clone classes that encounter evolution patterns other than the same, require more effort to be understood, analyzed and managed. However, not all types of evolution patterns are equally harmful. Clone classes that change inconsistently are more probable to cause maintenance implications than

those that are evolved consistently. Our proposed framework combines the clone evolution technique with statistical methods to predict the changes of clone classes in the future version. Such prediction can also guide users to be careful about a subset of clones that are more likely to change in a way that has the potential to cause trouble. Since a number of spatial clone class metrics are available in literature, we also consider their impact and validate whether their inclusion increases the performance of the prediction in terms of precision and recall. In general, we make the following contributions.

1. We perform an empirical study using five open source Java systems of variable sizes to determine how clone classes evolve including the type and frequency of the changes they encounter. We have used an extended version of VisCad to collect the spatial and temporal attributes of the clone classes.

2. Using that data we built a prediction model that acts as a recommendation system and can rank clone classes based on the potential risks.

3. We also measure the precision and recall of the proposed recommendation.

The rest of the chapter is organized as follows: While Section 5.2 describes a motivating scenario, we explain the code clone genealogy in Section 5.3. Details on the subject systems analyzed and the list of attributes collected are described in Section 5.4 and 5.5 respectively. The construction of the prediction model along with the evaluation and further discussion is presented in Section 5.6. Section 5.7 outlines the relevant research. Section 5.8 mentions some possible threats of our study followed by Section 5.9 that concludes the chapter.

## 5.2 Motivating Scenario

Consider a case, where a developer is editing a duplicated code fragment. Since, developers often create short-term changes, it is not desirable to notify the developer that he is changing a clone fragment every time he starts the editing. Instead, it will be helpful, if we can predict which fragments have the possibility to change in such a way that may lead to defective code in the next version and use that information to notify developers to carefully edit duplicated fragments. Such prediction can also guide maintenance engineers in selecting clones that need special attention and need to be managed.

## 5.3 Code Clone Genealogy

A code clone genealogy describes the evolution of clone classes over multiple versions of a program. In a clone genealogy, for a clone class ($CC_i$) in version $V_i$ we can determine another clone class ($CC_{i-1}$) in version $V_{i-1}$ that is the ancestor of $CC_i$. A clone genealogy consists of a set of lineages where each lineage is a directed acyclic graph that identifies the source of a clone class (sink) and also describes the evolution of the source to the sink. A set of evolutionary change patterns are associated with a clone genealogy that explain the

changes in a clone class from version $V_{i-1}$ to version $V_i$. We have considered the set of all change patterns described below.

- Same: All code fragments that are present in the old clone class (clone class in version $V_{i-1}$), also present in the new clone class (clone class in version $V_i$) without any changes.
- Add: At least one code snippet is added to the new clone class.
- Delete: At least one code snippet of the old clone class does not appear in the new clone class
- Id-Consistent Change: For all code snippets, the identifiers are changed consistently.
- Id-Inconsistent Change: At least for one code snippet the identifiers are not changed consistently.
- Consistent Change: All code snippets of the old clone class have experienced the same set of changes and now belong to the new clone class.
- Inconsistent Change: At least one code snippet of the old clone class has undergone different changes from the other snippets, and thus changed inconsistently.

We built a clone genealogy extractor by extending VisCad that accepts a set of versions of a program as input. It uses the clone detection result of CCFinder [84], maps the clone classes between two subsequent versions and automatically classifies the change patterns. Our genealogy extractor not only maps the clone classes but also maps individual code snippets in a clone class where each mapping is labelled with any of the change patterns. Detail about the genealogy extractor and automatic classification of the evolution patterns are described in Section 4.2.

## 5.4 Subject Systems

We studied five different software systems covering different application domains. Table 5.1 summarizes the size, duration and number of revisions considered for each system. Among them, dnsjava [5] is the implementation of the DNS protocol in Java and provides both high and low level functionality. JabRef [8] is an open source reference management program. It supports importing, editing and organizing popular BibTex files that can store list of bibliography items. iText [7] is a popular Java library to dynamically create and manipulate pdf documents. The largest system ArgoUML [2] is an UML modeling application that supports design, development and documentation of object oriented software systems. Finally, PMD [12] is a static source code analyzer for Java that can detect a number of potential problems including possible bugs, duplicated or overly complicated expressions. It should be noted that the revisions of the subject systems were collected from SVN repositories and two successive revisions were at least thirty days apart from each other.

**Table 5.1:** The set of studied subject systems

| Subject System | Lines of code(LOC) | Duration | # of Revisions |
|---|---|---|---|
| dnsJava | 11498 ~ 23738 | 2003-04-01 ~ 2011-07-07 | 51 |
| JabRef | 59472 ~ 117078 | 2007-04-17 ~ 2011-09-10 | 45 |
| ArgoUML | 128397 ~ 175999 | 2006-03-08 ~ 2011-08-15 | 66 |
| iText | 91517 ~ 80760 | 2007-11-20 ~ 2011-04-12 | 38 |
| PMD | 83377 ~ 123030 | 2005-11-12 ~ 2009-10-17 | 43 |

## 5.5 Data Collection

We used AIST CCFinderX [1] for clone detection which is a major version up of CCFinder. For the rest of the discussion, we will use the term CCFinder to refer to AIST CCFinderX [1]. CCFinder was instructed to detect clones with minimum number of kinds of tokens in a code fragment set to 12. The minimum token length was set to 30 as the same values were used in other studies also [91, 138]. We then determined the mapping between clone classes using our clone genealogy extractor. Next, we collected a set of temporal and spatial attributes for each clone class. For a clone class in version $V_i$, we determine the temporal attributes by considering its lineage and the spatial attributes (e.g. how many clone fragments are there in a clone class) are calculated for version $V_i$ only. This section introduces each attribute we computed for a clone class.

- $m_0$: Number of clone fragments in a clone class.
- $m_1$: Maximum number of lines of code (LOC) of the clone fragments.
- $m_2$: Total LOC of the clone fragments.
- $m_3$: Maximum number of tokens of the clone fragments.
- $m_4$: Total tokens of the clone fragments.
- $m_5$: Files associated with a clone class.
- $m_6$: Radius of a clone class. Let, $F$ be the set of files that contain the clone fragments of a clone class and $D$ is the lowest common directory of all files in $F$. Then the radius is defined as the maximum length of all paths, where each path defines the distance of a file in $F$ to the $D$.
- $m_7$: Clone Class Refactoring Benefit (CCRB). For a clone class, CCRB calculates the number of LOC ($CCRB_{Loc}$) that can be removed if all clone fragments can be replaced by a function call. Clearly, this requires to have one function that need to be called in every occurrence of the clone fragment and is defined in (5.1).

$$m_7 = m_0 * m_1 - ((m_0 - 1) + m_1) \tag{5.1}$$

- $m_8$: Maximum number of internal clones of the files associated with a clone class.
- $m_9$: Maximum number of external clones of the files associated with a clone class.
- $m_{10}$: Maximum number of developers edited a clone fragment.
- $m_{11}$: Total number of developers edited a clone class.
- $m_{12}$: Maximum number of developers edited the files associated with a clone class.
- $m_{13}$: Total number of developers edited the files associated with a clone class.

  We were interested to determine whether all developers are equally involved in maintaining a clone class or contribute to the files associated with a clone class. To determine this, we use the entropy measurement from information theory [48] that can be defined as follows:

$$\sum_{i=1}^{n} -p_i \times log(p_i), \tag{5.2}$$

  Here, $p_i$ is the probability of a modification belonging to developer $i$ and $n$ is the number of unique developers that are associated with the clone fragments or the files associated with a clone class. If a few developers are involved in the maintenance then the entropy value will be low. The more the developers are equally involved in maintaining the clones or the clone files, the more the entropy value will be.

- $m_{14}$: The entropy value of the developers involved in maintaining a clone class.
- $m_{15}$: The entropy value of the developers involved in maintaining the files associated with a clone class.
- $m_{16}$: Total LOC added in the files associated with a clone class.
- $m_{17}$: Total LOC deleted in the files associated with a clone class.
- $m_{18}$: Total LOC changed in the files associated with a clone class.
- $m_{19}$: Total LOC added, deleted or changed in the files associated with a clone class.
- $m_{20}$: The number of files associated with a clone class are changed.
- $m_{21}$: The number of time *same* evolution pattern occurs.
- $m_{22}$: The number of fragments evolve with *same* pattern.
- $m_{23}$: The timing of the most recent *same* evolution pattern in days.

  We also collected metrics for *add* ($m_{24}$ to $m_{26}$), *delete* ($m_{27}$ to $m_{29}$), *id-consistent* ($m_{30}$ to $m_{32}$), *id-inconsistent* ($m_{33}$ to $m_{35}$), *consistent* ($m_{36}$ to $m_{38}$), and inconsistent ($m_{39}$ to $m_{41}$) change patterns.

- $m_{42}$: The number of times files associated with a clone class changes in its lineage.

- $m_{43}$: The number of distinct developers maintaining the clone class in its lineage.
- $m_{44}$: The number of distinct developers maintaining the files associated with a clone class in its lineage.
- $m_{45}$: The number of distinct developers maintaining the files associated with a clone class in its lineage.
- $m_{46}$: The number of times a clone class evolves safely from its source to the sink. If a clone class $CC_i$ of version $V_i$ maps to another clone class $CC_{i-1}$ of version $V_{i-1}$ and only the subset of *same*, *id-consistent* and *consistent* patterns are associated with its evolution, we mark such evolution as *safe*.
- $m_{47}$: In the lineage of a clone class, the number of times it changes with add, delete, id-inconsistent or inconsistent change pattern. We hypothesize that the more such patterns are appeared in a clone class, the more problematic it is and need to be investigated first. Throughout the next of the discussion, we will refers to this attribute as change weight.

  To determine the change weight for a clone class $i$, we use the mapping from clone class $CC_i$ to clone class $CC_{i+1}$ since our objective is to built a predictor that can determine the change weight of a clone class in the future version.
- $m_{48}$: The number of time a clone class evolves unsafely. If a clone class $CC_i$ of version $V_i$ maps to another clone class $CC_{i-1}$ of version $V_{i-1}$ and only the subset of *add*, *delete*, *id-inconsistent* and *inconsistent* patterns are associated with its evolution, we mark such evolution as *unsafe*.
- $m_{49}$: The timing of the last *safe* pattern.
- $m_{50}$: The timing of the last *unsafe* pattern.
- $m_{51}$: Survival time of a clone class in days.

## 5.6 Predicting the Changes of a Clone Class

To predict the changes of a clone class, we attempted to develop a prediction model using several classification algorithms. Here, the dependent attribute is of nominal type that uses the set of rules outlined in Table 5.2 to classify the clone classes based on their changes. The rules are developed considering the possibility of causing troubles by the evolutionary change patterns and based on our previous experience on clone evolution analysis. We further refined it through discussion with other graduate students who have prior experience or currently involve in code clone research. We hypothesize that it is more important to examine those clone classes that are evolved with change pattern that can lead to buggy code. For example, clone fragments that are evolved without any change are less interesting than those where fragments are deleted because the deletion may be due to the inconsistent changes of the clone fragments. It may be the case that such inconsistent changes are made by the developers purposely but at least they need to be examined. Clone classes that are changed inconsistently or evolve with id-consistent change are more probable to cause

**Table 5.2:** Categorization of the clone classes

| Class Label | Change Pattern(s) |
|---|---|
| Safe | Not change |
| Possibly Safe | Id-consistent or Consistent |
| Possibly Problematic | Added or Deleted |
| Problematic | Inconsistent or Id-inconsistent |
| Highly Problematic | Inconsistent or Id-inconsistent+ added or deleted |

**Table 5.3:** Top ranked 10 attributes selected by the attribute selection model (described in Section 5.6.1)

| Subject System | Reduced Attribute Set |
|---|---|
| ArgoUML | $m_8, m_9, m_3, m_{49}, m_{19}, m_{46}, m_{21}, m_{47}, m_{50}, m_{26}$ |
| DnsJava | $m_{23}, m_{49}, m_8, m_{15}, m_{51}, m_{20}, m_{18}, m_9, m_{19}, m_7$ |
| JabRef | $m_{51}, m_{23}, m_{49}, m_{15}, m_8, m_{22}, m_{47}, m_{50}, m_9, m_{38}, m_7$ |
| iText | $m_8, m_{49}, m_{23}, m_{51}, m_{50}, m_4, m_3, m_{21}, m_{26}, m_{13}, m_{44}$ |
| PMD | $m_8, m_9, m_{19}, m_{18}, m_{16}, m_{50}, m_{23}, m_{51}, m_{41}, m_{38}$ |

maintenance implications since previous study shows that such changes are highly correlated with bugs than the other evolutionary changes [103].

We used the Weka [15] toolkit to built a number of models using different classifiers (J48, Random Forest, Random Tree, Naive Bayes, Bayes Network, and Simple Cart). The J48 decision tree classifier is the implementation of C4.5 algorithm [122] in Weka that uses information entropy measurement to classify the nodes. At each node in the tree it chooses one attribute of the data that splits the data set into subsets that results in the highest normalized information gain. Random forest is an example of ensemble classifiers that develops many decision tress. An unclassified new object is put into all the trees and each tree provides a classification. The new object is assigned to that class classified by the most classifiers. A Naive Bayes classifier is an example of a probabilistic classifier that uses Baye's theorem for the classification. We also tried with the Bayes Network classifier with the default settings in Weka. SimpleCart is the implementation of CART in Weka. Since our dependent variable is of categorical type, the algorithm produces a classification tree although it can generate both classification and regression tree depending on the type of the variable.

### 5.6.1 Attribute Subset Selection

We considered a large number of spatial and temporal attributes to built the predictor. However, it is expected that all the attributes do not equally affect predictor's performance. Thus, it is required to reduce

the number of dependent attributes. Techniques like Pearson product moment correlation coefficient [127] can compute correspondence among the attributes to determine the most influential set of attributes. Since our dependent attribute is of nominal type and all other independent attributes are of numeric type, we cannot use such technique. We used the information gain attribute selection technique to determine the top ranked influential attributes. This is considered as one of the fastest and simplest attribute ranking technique. Details about the technique can be found elsewhere [66]. To apply the information gain attribute selection technique, we used the implementation available in Weka. It uses a method developed by Fayyad and Irani [58] for data discretization.

For each of the subject system, the set of top ranked attributes selected by the attribute selector is listed in Table 5.3. The listing of attributes from left of right refer their ranking as determined by the selector. We considered the top ten ranked attributes to build the classification models.

**Table 5.4:** Precision and recall value for the classifiers. The term "Partial" refers to the weighted average precision and recall of a classifier considering all classes except the safe.

| Subject System | J48 | | | | Random Forest | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | | Recall | | Precision | | Recall | |
| | Overall | Partial | Overall | Partial | Overall | Partial | Overall | Partial |
| ArgoUML | 0.916 | 0.61 | 0.936 | 0.15 | 0.905 | 0.42 | 0.929 | 0.18 |
| dnsjava | 0.976 | 0.15 | 0.984 | 0.069 | 0.976 | 0.38 | 0.983 | 0.21 |
| JabRef | 0.91 | 0.43 | 0.939 | 0.076 | 0.908 | 0.29 | 0.928 | 0.16 |
| iText | 0.84 | 0.61 | 0.864 | 0.40 | 0.831 | 0.53 | 0.847 | 0.43 |
| PMD | 0.88 | 0.63 | 0.903 | 0.19 | 0.874 | 0.52 | 0.9 | 0.24 |

| Subject System | Simple Cart | | | | Random Tree | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | | Recall | | Precision | | Recall | |
| | Overall | Partial | Overall | Partial | Overall | Partial | Overall | Partial |
| ArgoUML | 0.918 | 0.65 | 0.936 | 0.14 | 0.895 | 0.24 | 0.904 | 0.21 |
| dnsjava | 0.972 | 0.35 | 0.983 | 0.055 | 0.975 | 0.37 | 0.975 | 0.23 |
| JabRef | 0.908 | 0.407 | 0.939 | 0.064 | 0.906 | 0.22 | 0.91 | 0.19 |
| iText | 0.838 | 0.62 | 0.864 | 0.39 | 0.823 | 0.48 | 0.836 | 0.42 |
| PMD | 0.879 | 0.61 | 0.903 | 0.20 | 0.869 | 0.47 | 0.895 | 0.24 |

| Subject System | Naive Bayes | | | | BayesNet | | | |
|---|---|---|---|---|---|---|---|---|
| | Precision | | Recall | | Precision | | Recall | |
| | Overall | Partial | Overall | Partial | Overall | Partial | Overall | Partial |
| ArgoUML | 0.879 | 0.083 | 0.854 | 0.095 | 0.885 | 0.14 | 0.872 | 0.17 |
| dnsjava | 0.971 | 0.032 | 0.869 | 0.078 | 0.972 | 0.061 | 0.938 | 0.24 |
| JabRef | 0.882 | 0.032 | 0.88 | 0.034 | 0.914 | 0.21 | 0.888 | 0.35 |
| iText | 0.764 | 0.28 | 0.735 | 0.22 | 0.846 | 0.41 | 0.789 | 0.52 |
| PMD | 0.821 | 0.11 | 0.82 | 0.14 | 0.855 | 0.28 | 0.855 | 0.25 |

### 5.6.2 Evaluating the Predictor

Evaluating the accuracy of the predictor is an important part of the framework since the result can guide us comparing different set of classifiers and select the appropriate ones. A number of techniques have been developed to measure the accuracy of a predictor including holdout method, random sub-sampling, k-fold cross validation and bootstrap [67].

We used 10-fold cross-validation to evaluate the accuracy of the predictor which is a special case of k-fold cross validation. Here, the same data set is used for training and to validate the accuracy of the predictor. Data are randomly partitioned into k subsets or folds that are mutually exclusive to each other and of equal in size (approximately). The test is conducted k times and in each iteration one fold is used as a test data while the rest is used for training the classifier. For example, in the first iteration subset $S_1$ is used for evaluation while $S_2, S_3, S_4, ..., S_k$ are used for training. In the next iteration, subset $S_2$ is used for evaluation and the rest are used for training and so on. Although the computation power of the current machines allows us to work with more folds, we avoid to do so because of the possibility of introducing more bias and variance. Since a large number of clone classes are evolved without any change in the next version, we report overall weighted average precision and recall of the prediction model, and the weighted average precision and recall of the model in detecting classes other than safe. The weighted average precision and recall can be defined by equations 5.3 and 5.4 respectively.

$$Weighted\ Average\ Precision = \frac{\sum_{i=1}^{n} \frac{TP_i}{TP_i+FP_i} \times t_i}{\sum_{i=1}^{n} t_i} \quad (5.3)$$

$$Weighted\ Average\ Recall = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} t_i} \quad (5.4)$$

Here, $TP_i$ is the number of correct predictions and $FP_i$ is the number of false predictions for each class label $i$, and $t_i$ is the total number of elements with class label $i$ in the training data set.

Table 5.4 summarizes the weighted average precision and recall for all subject systems using different classifiers. According to the result, we can say that the overall weighted average precision and recall values

are good for all classifiers. However, since the number of safe change classes are very large compare to the other classes, the classifiers did not perform that much in detecting classes other than the safe, possibly due to the insufficient number of training data. However, compare to the other classifiers, Random Forest shows promise in both cases. For the second case, the precision ranges from 29% to 53% and the recall ranges from 16% to 43%.

### 5.6.3 Discussion

In this section, we first explain the way the predictor can be built and used. Later, we propose some hypothesis related with the prediction model and try to validate them.

#### How to use the predictor?

There are three major steps associated with the development of the predictor. The first step is to generate the training data set which involves collection of both spatial and temporal attributes of clone classes from various sources of information. Given a set of versions as input, the clone detection tool provides us the results of the detection. The detection information is then fed into a clone genealogy extractor to determine the mapping and classify the change patterns. Both spatial and temporal attributes of clone classes are then determined. Since the developer's information is not available through clone detection results, we extract the information (such as which line is edited by which developer) from the SVN version control repository and then map that information to the clone lines to determine which clones are created by whom. The next step is to reduce the list of dependent attributes because there may exist a number of redundant variables that may reduce model efficiency, increase time to interpret the result and slow down the model development time. The third step is to use the training data set to train the model and evaluate its performance in terms of weighted average precision and recall. Figure 5.1 shows the required steps in developing and applying the predictor. The model need to be attached with a clone analysis or clone management tool that can use the model to determine which clone classes require special attention and need careful observation. In our case, we attached the model with VisCad.

#### Can we improve the performance of the predictor using more attributes?

To build the prediction model, we used the top ten attributes selected by the attribute selection model which are only a fraction of the total number of attributes we considered. We hypothesize that use of more attributes can improve the performance of the prediction model. To validate our hypothesis, we built the same models again with the top ranked 20 attributes. We only report here the results obtained for the 48, Random Forest, Random Tree classifiers since the result obtained from the other classifiers are not that much significant. Table 5.5 shows the new results for all the subject systems. The results contradict our hypothesis. Although, the precision and recall values for all three classifiers affected, we did not find any significant improvements in the result. In many cases the performance of the predictor has decreased. For

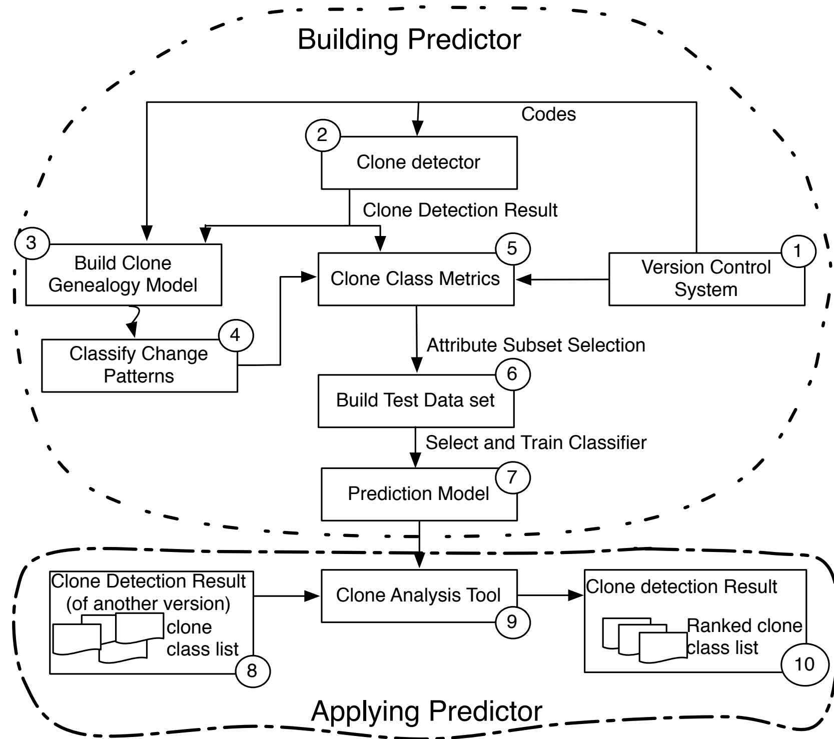



**Figure 5.1:** Building and using the predictor

example, the weighted average precision value for the Random Forest classifier has decreased for the three subject systems (ArgoUML, dnsjava and PMD).

**Are temporal attributes superior than the spatial attributes?**

We again hypothesize that the temporal attributes are more capable of predicting the changes in the clone classes than the spatial attributes. We use the top ranked 10 attributes returned by the information gain attribute selection model to validate our hypothesis. We classify them into spatial and temporal categories, and also determine the number of times they were selected. Table 5.6 shows the categorization of the selected attributes into spatial and temporal categories. The categorized data reveal that evolutionary attributes are important for the classification but they alone are not sufficient. For all subject systems, both spatial and temporal attributes were selected although we found that few attributes are very popular. The number of internal ($m_8$) and external clones ($M_9$) in the spatial category, and the survival time of the clone class ($m_{51}$) and the timing of the last unsafe pattern ($m_{50}$) in the temporal category were frequently selected by the attribute selection model than the rest of the attributes. However, the other attributes returned by the selector varies from one system to another and we cannot generalize the results. We tried to build decision tree models with only temporal or spatial attributes but we found that the performance of the predictors decrease a lot. These invalidate our hypothesis and suggest that we need to carefully evaluate the contribution of both the categories of attributes to build the classifier.

**Table 5.5:** Weighted average precision and recall values while considering top 20 ranked attributes. While the term "Over." refers to the result considering all change patterns, the term "E.S." refers to the result considering all change patterns except the same.

| Subject System | J48 | | | | Random Forest | | | | Random Tree | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Precision | | Recall | | Precision | | Recall | | Precision | | Recall | |
| | Over. | E.S. | Over. | E.S. | Over. | E.S. | Over. | E.S. | Over. | E.S. | Over. | E.S. |
| ArgoUML | 0.915 | 0.52 | 0.935 | 0.24 | 0.904 | 0.33 | 0.929 | 0.27 | 0.894 | 0.16 | 0.903 | 0.32 |
| dnsjava | 0.975 | 0.43 | 0.984 | 0.145 | 0.974 | 0.36 | 0.983 | 0.18 | 0.974 | 0.22 | 0.975 | 0.21 |
| JabRef | 0.92 | 0.523 | 0.942 | 0.14 | 0.91 | 0.32 | 0.931 | 0.16 | 0.906 | 0.21 | 0.908 | 0.20 |
| iText | 0.841 | 0.59 | 0.863 | 0.42 | 0.838 | 0.54 | 0.853 | 0.44 | 0.826 | 0.47 | 0.835 | 0.43 |
| PMD | 0.877 | 0.56 | 0.903 | 0.23 | 0.867 | 0.40 | 0.89 | 0.26 | 0.858 | 0.30 | 0.864 | 0.28 |

**Table 5.6:** Attributes categorized into two groups

| Subject System | Spatial Attribute | Temporal Attribute |
|---|---|---|
| ArgoUml | $m8, m9, m3, m_{19}$ | $m_{49}, m_{46}, m_{21}, m_{47}, m_{50}, m_{26}$ |
| DnsJava | $m8, m_{15}, m_{20}, m_{18}, m9, m_{19}, m7$ | $m_{23}, m_{49}, m_{51}$ |
| JabRef | $m8, m_{22}, m9, m7$ | $m_{51}, m_{23}, m_{49}, m_{51}, m_{47}, m_{50}, m_{38}$ |
| iText | $m8, m4, m3, m_{21}, m_{13}$ | $m_{49}, m_{51}, m_{50}, m_{21}, m_{26}, m_{44}$ |
| PMD | $m8, m9, m_{19}, m_{18}, m_{16}$ | $m_{50}, m_{23}, m_{51}, m_{41}, m_{38}$ |

**Can we use a binary classifier?**

Since, a large number of clone classes evolve without any changes, the classifiers have very few data of other classes (all classes except the safe) for learning. An alternative approach could be using a binary classifier that predicts whether a clone class evolves without any changes or not. We hypothesize that such a binary prediction model can improve the performance of the predictor. We used the rules outlined in Table 5.7 to classify clone classes into safe and unsafe categories and built the classifiers again.

Table 5.8 shows the weighted average precision and recall values of the Random Forest classifier for the subject systems. We report the overall results along with results for the *Not Safe* class. The results reveal that using a binary classifier we can improve the precision and recall. Although we loose some fine grained information while using the binary classifier,but we can predict correctly more cases where the clone classes will evolve with changes.

**Table 5.7:** Rules used to create a binary classifier

| Class Label | Description |
|---|---|
| Safe | Nothing change |
| Not Safe | If any changes occurs |

**Table 5.8:** Weighted average precision and recall for a binary classifier

| Subject System | Random Forest | | | |
|---|---|---|---|---|
| | Precision | | Recall | |
| | Overall | Not Safe | Overall | Not Safe |
| ArgoUML | 0.913 | 0.53 | 0.93 | 0.231 |
| dnsjava | 0.981 | 0.594 | 0.984 | 0.319 |
| JabRef | 0.937 | 0.601 | 0.939 | 0.372 |
| iText | 0.88 | 0.7 | 0.882 | 0.67 |
| PMD | 0.89 | 0.731 | 0.904 | 0.234 |

## 5.7 Related Work

This section describes tool support and techniques available for clone analysis, evolutionary characteristics of clones, and clone evolution analysis.

To aid clone analysis, a number of techniques have been developed including data filtering, data reduction and large scale clone visualizations that can help to identify important patterns of cloning and to reduce the volume of clone data need to be analyzed [75, 76, 86, 87, 126, 150, 159]. However, none of these techniques consider the change patterns of the code clones. We anticipate that the change pattern of the code clones can be a valuable information that can guide us in clone management. Our technique differs from the above approaches is that we have tried to predict the change pattern of clone classes that can provide a ranking of the clone classes and such ranking can guide the clone management activities. The virtue of the approach is that the ranking allows the developers or the maintenance engineers to calibrate the volume of data need to be analyzed with the limited time and manpower available at hand. Since many clone classes are propagated either consistently or without any changes, we anticipate that the technique can significantly reduce the volume of clone data for the analysis purpose.

Our prediction model uses the evolutionary characteristics of clones. Also, previous studies of clone evolution and change pattern classification lay the foundation for building our genealogy model and data collection which is influenced by the work of Kim et al. [91]. They studied the evolution of two open source Java systems and found that aggressive refactoring is not a good choice for clone management. Many clones exist in the systems for only a short period of time and many of the long-lived clones are changed consistently during its evolution. Aversano et al. [25] also conducted a similar study with clone evolution using a clone genealogy model similar to that of Kim et al. However, their model classifies the inconsistent changes into independent evolution and late propagation. They performed co-change analysis to assess the impact of bug fixing or evolutionary activities on clone maintenance. Based on the empirical study on two open source Java systems they concluded that the majority of the clone classes are maintained consistently.

One of our previous study of code clone evolution of 17 open source software systems [138] also revealed that the many of the clone groups are propagated without any syntactic changes or changes consistently in the subsequent releases, which is consistent with the findings of Kim et al.

The conventional wisdom characterized inconsistent changes as a possible threat to software maintenance. Krinke [96] analyzed the changes to clones to determine whether the clones within the clone classes are changed consistently or not. Data were collected from CVS version repositories for five open source software systems on 200 different dates where two subsequent revisions were one week apart from each other. Their study revealed that clones within the groups are not always consistently changed. Moreover, an inconsistently changed clone group rarely become consistent and thus invalidate the idea that the missing changes will appear later in the inconsistently changed clone groups. In another experiment, Krinke [97] studied the stability of cloned code over non-cloned code based on the change measure in 200 weeks of evolution of five open source software systems. The stability was measured by changes (number of additions, deletions and changes) and he found that cloned code undergone a great many number of deletions during the evolution of a software system. However, the average percentage of changes are more to the non-cloned code than the cloned one, and thus concluded that cloned code is more stable than the non-cloned code. He gathered data from the CVS repository on 200 different dates such that there exist at least one week gap between two subsequent revisions. However, to ignore experimental changes make by the developers we have retrieved versions from the SVN repositories such that each version is at least one month earlier or later than the next or previous version.

Recently, Cai and Kim [37] studied the characteristics of the long lived clones and also predicted the survival time of clone genealogies based on the data extracted from seven open source software systems. They found that the length of the genealogies are highly correlated with the evolutionary characteristics (such as, the number of developers edited the clones) instead of the traditional clone metrics including the population, dispersion and the size of the clones in LOC. To predict clone survival time, they developed a predictor using decision tree classifier. While they tried to identify the length of clone genealogies, we have tried to predict the changeability of the clones at the class level. Our clone genealogy extractor can capture change patterns at more fine grained level and the predictor uses that information to forecast potential unsafe changes.

## 5.8 Threats to Validity

One of the possible threats to the presented study lies on the clone detection tool used in the study since the clones detected by the tool might not be accepted by the human judge (false positive) or certain clones might have missed (false negative). We used CCFinderX, an improver version of CCFinder, to detect clones which has higher recall although the precision is lower than some other tools [36]. It is recognized as one of the state of the art clone detection tools and has been used in many empirical study and clone evolution

analysis.

Another possible threat to the study related with the clone genealogy extractor that maps clones in the subsequent versions. To avoid false mapping due to the reordering of code fragments, we have used both content and context to map the clones. Although, we cannot guarantee that the modified approach eliminates false positives completely, manual evaluation on a random selection of clone genealogies did not report any false mapping.

CCFinder cannot detect non-contiguous clones. If a line is inserted in a cloned fragment, it detects them as new cloned fragments in the next version, if the fragments have acceptable length. Otherwise, it reports some fragments as deleted. This might lead to a false warning. However, comparing to the large number of same change patterns, we found that such warning are few in numbers and because of their potential to yield buggy code, we believe that developers need to investigate them.

We collected evolutionary characteristics of the clones to predict their changeability. For this purpose, we extracted revisions that are at least thirty days apart from each other. We found that considering smaller time gap either overestimate (developer may change something for experiment only for a short time) or underestimate (developers may not change the code) the evolutionary characteristics of the system. Although our predictor showed promising result, maintenance engineers need to adjust the time gap depending on the characteristics of the project at hand.

## 5.9 Conclusion

In this chapter, we present an approach to build a decision tree based model that can predict the change patterns of a clone class. Although we considered a set of different classifiers, we found that the Random Forest performs comparatively better than others. We consider both spatial and temporal attributes and describe a possible way to collect the optimum number of attributes to improve the performance of the predictor. Our study reveals that the predictor can determine and classify the changes although the performance varies depending on the subject systems.

# Chapter 6

# Conclusion

This section concludes the thesis. While we summarize the contributions of the thesis in Section 6.1, directions for future research are outlined in Section 6.2.

## 6.1 Summary

Software clones are inevitable due to the copy-paste programming practices of the developers. Although we cannot ignore the maintenance implications of clones, it is also required to facilitate developers in copying and pasting code fragments. An ethnographic study on copy-paste practices [90] of developers reveals that copying and pasting not only saves time but also captures important design decisions. The improvements in clone detection techniques and the availability of the state-of-the-art clone detectors allow us to detect duplicated code fragments easily even when developers change identifiers, or add and delete lines in the copied code fragments. Although this solves the detection part, managing the large volume of raw clone data and identifying important insights from them are open problems in the area. In this thesis, we have extended support for clone analysis that can lead to better clone management.

It was believed that clones are harmful and eliminating clones through refactoring could be the solution. Such conventional wisdom is challenged by the findings of Kim et al. [91]. Their study reveals that many clones in software systems are volatile in nature and aggressive refactoring of these clones is not beneficial since they can be changed from one form to another. A large number of clones are propagated without any changes or with a similar repetitive set of operations. Since they studied only two small Java systems, there was a threat to the study that the findings may not be the general case, but rather a coincidence. We participated in a large-scale empirical study to validate their findings [138] and the lessons learned from that study lays the foundation for this thesis. First, due to the lack of a clone analysis support environment we faced great difficulties analyzing and identifying important patterns of cloning, and these challenges inspired us to develop a clone analysis support environment. Second, we realized that clone evolution analysis can provide an in-depth understanding about the changes, effects and reasons for cloning. We anticipated that the large volume of evolutionary clone data could guided us in deciding which clones in a system need special attention.

In this thesis, we have proposed a framework for code clone analysis, and based on the framework we

have developed VisCad, a support environment for code clone analysis (Chapter 3). VisCad allows users to analyze clone data from any clone detection tool that reports the source co-ordinates of the clones including the file name, starting and ending line numbers. It also allows developers, maintenance engineers or clone researchers to work with clone detection tools of choice. We believe that this is an important requirement for a generic clone analysis tool since every clone detection tool has its own strengths and weaknesses. VisCad provides a high level overview of cloning through visualizations, supports details on demand functionality, and also facilitates removal of clones that are either false positives or do not fall in the set of interesting clones. We not only explained how VisCad works, but also explained how it can be used in the real world through a case study that involved inter-project clone analysis between FreeBSD and NetBSD subject systems. The tool has also been available as an open source software and used by clone researchers. We are continuously working towards its improvement.

In the next part of the thesis (Chapter 4), we have proposed a set of visualization techniques for clone evolution analysis. Although a large number of visualizations have been proposed for software evolution analysis, there has been a marked lack in visualizing clone evolution. This is an extension to our previous work because we are extending support of clone analysis for multiple versions while the previous work focus on a single version. We have extended VisCad for this purpose and implemented the proposed visualizations. We also plan to release this extended version of VisCad as an open source software so that people can use and extend the rich set of libraries available in it.

In the last part of the thesis (Chapter 5), we have extended VisCad again to collect a large number of temporal and spatial attributes given a set of revisions of a subject system. Next, we have performed an empirical study on five open source software systems of different sizes and domains. The data collected from those systems is used to develop a number of classifiers to predict the changes to a clone class in a future version. The rationale of our study is that not all clone evolutionary change patterns have maintenance implications. Clone classes that have the possibility to evolve without any change do not require much attention. Instead, developers need to be careful about those clones that are likely to be changed other than the same evolution pattern. We have considered five different classification models and the results reveal that both temporal and spatial attributes have the potential to predict such changes, although the results vary depending on the subject systems. In our study, we have found that the Random Forest classifier performs comparatively better than the other models. Even when we consider weighted average precision and recall in detecting change patterns other than the same, we have found values up to 53% and 43% respectively. The study has shown promise in predicting future changes in clone classes based on their current status and previous histories, which can be an effective way to rank clone classes and warn developers about which clones require attention and careful editing.

## 6.2 Future Work

In this section, we explain directions for further research and our future goals.

### 6.2.1 Improvement of VisCad

Currently VisCad uses files both to store the results of various metrics calculation and the clone detection result in *VisCad input format*. As a result, query support is limited. We plan to use a relational database to store the data so that expert users can create a number of SQL-like queries that ease the findings of clones of interest. Moreover, current visualizations in VisCad are static in the sense that they do not support on the fly selection and removal of clones that are not interesting to the users. For example, a scatter plot can able to show the cloning relation among the subsystems. Users might be interested in those subsystems that contain a certain degree of cloning. Current workflow in VisCad requires that data be filtered first with such a requirement prior to visualization. We plan to incorporate such operations directly inside the visualizations.

### 6.2.2 Controlled Experiment

There is a marked lack of research in the evaluation and comparison of clone analysis support environments. We plan to conduct a controlled experiment to evaluate the effectiveness and the usability of the existing support environments including VisCad by having subjects perform tasks related to clone analysis. Such a study will help us to identify the usefulness and limitations of VisCad. We may also able to determine the complete feature space for a clone analysis tool from the study.

### 6.2.3 IDE Integration

VisCad is currently available as a standalone tool. However, we anticipate that integrating VisCad to the development environments can help developers to take advantage of its rich set of features more conveniently. Similarly, we can also increase the functionality of VisCad by tracking the copy-paste practices of the developers, managing clones as structural templates and making useful recommendations.

### 6.2.4 Clone Filtering

We have developed a prediction mechanism that collects the current and historical information about cloning, and predict the evolutionary change pattern of a clone class in the future version. Another application of the prediction mechanism we are particularly interested in is the automatic detection of false positive clones. We plan to collect different attributes of the clone fragments and also use a set of manually verified fragments to build a training data set. Our objective is to develop a classifier that can accurately determine false positives from the large volume of clone data since manual verification of all clones in a subject system is not feasible particularly when the size of the subject system is very large.

### 6.2.5 Extended Empirical Study

Our study has revealed that the evolutionary and the spatial attributes of clones can help us predicting the changes of clone classes in the future version. However, we have only used Java systems. Moreover, we work at the revision level. To ignore the short term changes made by the developers, we collected revisions that are at least one month apart from each other. However, we cannot guarantee that the data was not affected by such changes. We thus plan to carry out a study at the release level with a large number of subject systems covering different programming languages so that we can make a more general conclusion and can use those findings to improve the performance of the predictor. Furthermore, the reported study has considered Type 1 and Type 2 clones only and we plan to include Type 3 clones as well.

### 6.2.6 Clone Genealogy Extractor

To collect the evolutionary clone data and to visualize clone evolution, we have developed a clone genealogy extractor. We have manually verified a large number of genealogies to evaluate the performance of the extractor. Although a scenario-based comparison of clone detection techniques is available in literature [130], there is a marked lack in comparative study of clone genealogy extractors. We plan to perform a scenario-based comparison of existing clone genealogy extractors [139] including ours. As of today, only a few genealogy extractors are available that support detection of Type 3 clone evolution [139]. We plan to extend our genealogy extractor to support Type 3 clones also.

### 6.2.7 VisCad as a Generic Framework

VisCad is designed to be a generic framework that not only supports different clone detectors, languages, granularities of clones but also can be used for similar types of analyses for other software related problems. Examples can be, but not limited to, visualizing the bug reports in different ways, showing the effectiveness of a mutation testing [131, 132] in a visual manner and so on. One aspect of our future work is to adapt such applications in VisCad.

## 6.3 Conclusion

In this chapter, we have summarized the contributions of the thesis followed by our plans for extending current work. We briefly explained the three major issues we addressed in this thesis organized into three different chapters. We not only described the improvements that can be done to VisCad but also explained the opportunity of using VisCad in other problem domains also.

# References


[1] The AIST CCFinderX: http://www.ccfinder.net/ccfinderx.html (Dec 2011).

[2] The ArgoUML: http://argouml.tigris.org/ (Dec 2011).

[3] The CLICS: http://www.swag.uwaterloo.ca/clics/ (Dec 2011).

[4] The ConQAT: www.conqat.org/ (Dec 2011).

[5] The dnsjava: www.dnsjava.org/ (Dec 2011).

[6] The FreeBSD: http://www.freebsd.org/ (Dec 2011).

[7] The iText: itextpdf.com/ (Dec 2011).

[8] The JabRef: jabref.sourceforge.net/ (Dec 2011).

[9] The JHotDraw: http://www.jhotdraw.org/ (Dec 2011).

[10] The Linux Kernel: http://www.kernel.org/ (Dec 2011).

[11] The NetBSD: http://www.netbsd.org/ (Dec 2011).

[12] The PMD: pmd.sourceforge.net/ (Dec 2011).

[13] The Simian: http://www.redhillconsulting.com.au/products/simian/(Dec2011) .

[14] The VisCad: http://homepage.usask.ca/~mua237/viscad/viscad.html (Dec 2011).

[15] The WEKA: http://www.cs.waikato.ac.nz/ml/weka/ (Dec 2011).

[16] IEEE Standard for Software Maintenance. *IEEE Std 1219-1993*, pages 0–1, 1993.

[17] The CYCLONE: softwareclones.org/cyclone.php Dec 2011, December 2011.

[18] Eytan Adar. Guess: a language and interface for graph exploration. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 791–800, New York, NY, USA, 2006. ACM.

[19] Eytan Adar and Miryung Kim. Softguess: Visualization and exploration of code clones in context. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 762–766, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Christopher Ahlberg. Spotfire: an information exploration environment. *SIGMOD Rec.*, 25:25–29, December 1996.

[21] Christopher Ahlberg and Ben Shneiderman. Visual information seeking using the filmfinder. In *Conference companion on Human factors in computing systems*, CHI '94, pages 433–434, New York, NY, USA, 1994. ACM.

[22] Christopher Ahlberg, Christopher Williamson, and Ben Shneiderman. Dynamic queries for information exploration: an implementation and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 619–626, New York, NY, USA, 1992. ACM.

[23] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano D. Penta. Analyzing cloning evolution in the linux kernel. In *Information & Software Technology*, volume 44, 2002.

[24] Muhammad Asaduzzaman, Chanchal K. Roy, and Kevin A. Schneider. VisCad: flexible code clone analysis support for NiCad. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 77–78, New York, NY, USA, 2011. ACM.

[25] Lerina Aversano, Luigi Cerulo, and Massimiliano D. Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[26] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.

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

[28] Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy. Clone smells in software evolution. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, pages 24 –33, October 2007.

[29] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, WCRE '99, pages 326–336, Washington, DC, USA, 1999. IEEE Computer Society.

[30] Hamid A. Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Trans. Softw. Eng.*, 35:497–514, July 2009.

[31] Hamid A. Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In *Proceedings of the International Conference on Software Engineering*, pages 451–459, 2005.

[32] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society.

[33] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proceedings of the 16th Working Conference on Reverse Engineering*, WCRE '09, pages 85–94. IEEE Computer Society, 2009.

[34] Dirk Beyer and Ahmed E. Hassan. Evolution storyboards: Visualization of software structure dynamics. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 248–251, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Professional, Boston, USA, 20th anniversary edition, August 1995.

[36] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '02, pages 36–43, Washington, DC, USA, 2002. IEEE Computer Society.

[37] Dongxiang Cai and Miryung Kim. An empirical study of long-lived code clones. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software*, FASE'11/ETAPS'11, pages 432–446, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] Fabio Calefato, Filippo Lanubile, and Teresa Mallardo. Function clone detection in web applications: a semiautomated approach. *Journal of Web Engineering*, 3:3–21, May 2004.

[39] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[40] Myoungsu Cho, Bohyoung Kim, Dong K. Jeong, Yeong-Gil Shin, and Jinwook Seo. Dynamic query interface for spatial proximity query with degree-of-interest varied by distance to query point. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 693–702, New York, NY, USA, 2010. ACM.

[41] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization*, SoftVis '03, pages 77–86, New York, NY, USA, 2003. ACM.

[42] James R. Cordy. Comprehending reality: Practical challenges to software maintenance automation. In *Proceedings of the 11th International Workshop on Program Comprehension*, IWPC '03, pages 196–206. IEEE Computer Society Press, 2003.

[43] James R. Cordy. Exploring large-scale system similarity using incremental clone detection and live scatterplots. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC '11, pages 151–160, Kingston, ON, Canada, June 2011.

[44] James R. Cordy. Live scatterplots. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 79–80, New York, NY, USA, 2011. ACM.

[45] James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 1–12. IBM Press, 2004.

[46] James R. Cordy and Chanchal K. Roy. Debcheck: Efficient checking for open source code clones in software systems. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC '11, pages 217–218, Kingston, ON, Canada, June 2011.

[47] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC '11, pages 219 –220, June 2011.

[48] Thomas M. Cover and Joy A. Thomas. *Elements of information theory.* Wiley-Interscience, New York, NY, USA, 1991.

[49] Marco D'Ambros, Michele Lanza, and Harald Gall. Fractal figures: Visualizing development effort for cvs entities. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005*, pages 1–6, 2005.

[50] Ian J. Davis and Michael W. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 77–78, New York, NY, USA, 2010. ACM.

[51] Ian J. Davis and Michael W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 242–246, Washington, DC, USA, 2010. IEEE Computer Society.

[52] Florian Deissenboeck, Benjamin Hummel, Elmar Juergens, Michael Pfaehler, and Bernhard Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 57–64, New York, NY, USA, 2010. ACM.

[53] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 603–612, New York, NY, USA, 2008. ACM.

[54] Christoph Domann, Elmar Juergens, and Jonathan Streit. The curse of copy&paste — Cloning in requirements specifications. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 443–446, October 2009.

[55] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 109–118, Washington, DC, USA, 1999. IEEE Computer Society.

[56] Stephen G. Eick, Paul Schuster, Audris Mockus, Todd L. Graves, and Alan F. Karr. Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):396–412, April 2002.

[57] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18:957–968, November 1992.

[58] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the International Joint Conference on Uncertainty in AI*, pages 1022–1029, 1993.

[59] Ken Fishkin and Maureen C. Stone. Enhanced dynamic queries via movable filters. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 415–420, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[60] Yoshihiko Fukushima, Raula Kula, Shinji Kawaguchi, Kyohei Fushida, Masataka Nagura, and Hajimu Iida. Code clone graph metrics for detecting diffused code clones. In *Proceedings of the 2009 16th Asia-Pacific Software Engineering Conference*, APSEC '09, pages 373–380, Washington, DC, USA, 2009. IEEE Computer Society.

[61] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[62] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In *Proceedings of the 30th SIGCSE technical symposium on Computer science education*, SIGCSE '99, pages 266–270. ACM Press, 1999.

[63] Nils Göde. Evolution of type-1 clones. In *Proceedings of the Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, Washington, DC, USA, 2009. IEEE Computer Society.

[64] Nils Göde and Rainer Koschke. Frequency and risks of changes to clones. In *Proceeding of the 33rd International Conference on Software Engineering*, ICSE '11, pages 311–320, New York, NY, USA, 2011. ACM.

[65] Antonio Gonzalez, Roberto Theron, Alexandru Telea, and Francisco J. Garcia. Combined visualization of structural and metric information for software evolution analysis. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, IWPSE-Evol '09, pages 25–30, New York, NY, USA, 2009. ACM.

[66] Mark A. Hall and Geoffrey Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Trans. on Knowl. and Data Eng.*, 15:1437–1447, November 2003.

[67] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[68] Jan Harder and Nils Göde. Efficiently handling clone data: RCF and cyclone. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 81–82. ACM, May 2011.

[69] Jonathan Helfman. Dotplot patterns: A literal look at pattern languages. *Theory and Practice of Object Systems*, 2(1):31–41, 1996.

[70] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Code clone analysis methods for efficient software maintenance. *Tech. Report of Software Engineering Lab in Osaka University*, (SEL-Mar-29-2004), 2004.

[71] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. ARIES: refactoring support tool for code clone. In *Proceedings of the third workshop on Software quality*, 3-WoSQ, pages 1–4, New York, NY, USA, 2005. ACM.

[72] Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, PROFES '02, pages 185–197. Springer, 2002.

[73] Patricia Jablonski and Daqing Hou. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 16–20, New York, NY, USA, 2007. ACM.

[74] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[75] Zhen M. Jiang and Ahmed E. Hassan. A framework for studying clones in large software systems. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 203–212, Washington, DC, USA, 2007. IEEE Computer Society.

[76] Zhen M. Jiang, Ahmed E. Hassan, and Richard C. Holt. Visualizing clone cohesion and coupling. In *Proceedings of the 13th Asia Pacific Software Engineering Conference*, pages 467–476, Washington, DC, USA, 2006. IEEE Computer Society.

[77] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[78] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 120–126, Washington, DC, USA, 1994. IEEE Computer Society.

[79] J. H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '94, pages 32–41. IBM Press, 1994.

[80] J. H. Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pages 7–16. IBM Press, 1996.

[81] Elmar Juergens. Research in cloning beyond code: a first roadmap. In *Proceedings of the 5th International Workshop on Software Clones*, IWSC '11, pages 67–68, New York, NY, USA, 2011. ACM.

[82] Elmar Juergens, Florian Deissenboeck, Martin Feilkas, Benjamin Hummel, Bernhard Schaetz, Stefan Wagner, Christoph Domann, and Jonathan Streit. Can clone detection support quality assessments of requirements specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 79–88, New York, NY, USA, 2010. ACM.

[83] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[84] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28:654–670, July 2002.

[85] Hyunmo Kang and Ben Shneiderman. Visualization methods for personal photo collections: browsing and searching in the photofinder. In *Proceedings of the 2000 IEEE International Conference on Multimedia and Expo*, volume 3, pages 1539–1542. IEEE, 2000.

[86] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of the Seventh International Workshop on Principles of Software Evolution*, pages 85–94, Washington, DC, USA, 2004. IEEE Computer Society.

[87] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 305 – 314, September 2005.

[88] Cory Kapser and Michael W. Godfrey. "Cloning Considered Harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 19 –28, October 2006.

[89] Daniel A. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8:1–8, January 2002.

[90] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[91] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[92] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, 2001. Springer-Verlag.

[93] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and Morris Bernstein. *Pattern matching for clone and concept detection*, pages 77–108. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[94] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering*, WCRE '06, pages 253 –262, October 2006.

[95] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, WCRE '01, pages 301–309, Washington, DC, USA, 2001. IEEE Computer Society.

[96] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceesings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 170 –178, October 2007.

[97] Jens Krinke. Is cloned code more stable than non-cloned code? In *Proceesings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '08, pages 57 –66, September 2008.

[98] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of the International Conference on Software Maintenance*, ICSM '97, pages 314–321, Washington, DC, USA, 1997. IEEE Computer Society.

[99] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *Proceedings of the Fourth International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, New York, NY, USA, 2001. ACM.

[100] Seunghak Lee and Iryoung Jeong. SDD: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 140–141, New York, NY, USA, 2005. ACM.

[101] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[102] Leitão and António Menezes. Detection of redundant code using $R^2D^2$. *Software Quality Control*, 12:361–382, December 2004.

[103] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176 – 192, March 2006.

[104] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proceedings of the 13th Asia Pacific Software Engineering Conference*, pages 269–276, Washington, DC, USA, 2006. IEEE Computer Society.

[105] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-CCFinder. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 106–115, Washington, DC, USA, 2007. IEEE Computer Society.

[106] Angela Lozano and Michel Wermelinger. Assessing the effect of clones on changeability. In *Proceedings of the International Conference on Software Maintenance*, ICSM '08, pages 227–236, 2008.

[107] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 18–21, Washington, DC, USA, 2007. IEEE Computer Society.

[108] Giuseppe A. Di Lucca, Massimiliano D. Penta, and Anna R. Fasolino. An approach to identify duplicated web pages. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 481–486, Washington, DC, USA, 2002. IEEE Computer Society.

[109] Udi Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, Berkeley, CA, USA, 1994. USENIX Association.

[110] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 107–114, Washington, DC, USA, 2001. IEEE Computer Society.

[111] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–253, Washington, DC, USA, 1996. IEEE Computer Society.

[112] Ettore Merlo, Giuliano Antoniol, Massimiliano D. Penta, and Vincenzo F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 412–416, Washington, DC, USA, 2004. IEEE Computer Society.

[113] Muhammad Asaduzzaman Minhaz F. Zibran, Ripon K. Saha and Chanchal K. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '11, pages 295 –304, April 2011.

[114] Manishankar Mondal, Md. S. Rahman, Ripon. K. Saha, Chanchal K. Roy, Jens Krinke, and Kevin A. Schneider. An empirical study of the impacts of clones in software maintenance. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, ICPC '11, pages 242–245, Washington, DC, USA, June 2011. IEEE Computer Society.

[115] Manishankar Mondal, Chanchal K. Roy, Md. S. Rahman, Ripon K. Saha, Jens Krinke, and Kevin A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the Software Engineering Track of the 27th ACM Symposium on Applied Computing*, Riva del Garda, Trento, Italy, March 2012 (to appear).

[116] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, WCRE '01, pages 13–22. IEEE Computer Society Press, 2001.

[117] Hoan A. Nguyen, Tung T. Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FASE '09, pages 440–455, Berlin, Heidelberg, 2009. Springer-Verlag.

[118] Jean-Francois Patenaude, Ettore Merlo, Michel Dagenais, and Bruno Laguë. Extending software quality assessment techniques to java systems. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, pages 49–56, Washington, DC, USA, 1999. IEEE Computer Society.

[119] Nam H. Pham, Hoan A. Nguyen, Tung T. Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 276–286, Washington, DC, USA, 2009. IEEE Computer Society.

[120] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 67–75, New York, NY, USA, 2005. ACM.

[121] Catherine Plaisant and Vinit Jain. Dynamaps: dynamic queries on a health statistics atlas. In *Conference companion on Human factors in computing systems*, CHI '94, pages 439–440, New York, NY, USA, 1994. ACM.

[122] John R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[123] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: What *is* that Smell? In *Proceedings of the 7th Working Conference on Mining Software Repositories*, pages 72–81. IEEE Computer Society, 2010.

[124] Steven P. Reiss. Tracking source locations. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 11–20, New York, NY, USA, 2008. ACM.

[125] Matthias Rieger. *Effective Clone Detection Without Language Barriers*. Dissertation, University of Bern, Switzerland, 2005.

[126] Matthias Rieger, Stéphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 100 – 109, November 2004.

[127] Joseph L. Rodgers and Alan W. Nicewander. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician*, 42(1):59–66, 1988.

[128] Chanchal K. Roy and James R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE '08, pages 81–90, Antwerp, Belgium, October 2008.

[129] Chanchal K. Roy and James R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[130] Chanchal K. Roy and James R. Cordy. Scenario-based comparison of clone detection techniques. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 153–162, Amsterdam, The Netherlands, June 2008.

[131] Chanchal K. Roy and James R. Cordy. Towards a mutation-based automatic framework for evaluating code clone detection tools. In *Proceedings of the Canadian Conference on Computer Science & Software Engineering*, C3S2E '08, pages 137–140, Montreal, Quebec, Canada, May 2008.

[132] Chanchal K. Roy and James R. Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops, 2009*, ICSTW '09, pages 157 –166, April 2009.

[133] Chanchal K. Roy and James R. Cordy. Are scripting languages really different? In *Proceeding of the 4th International Workshop on Software Clones*, IWSC '10, pages 17–24, Cape Town, South Africa, May 2010.

[134] Chanchal K. Roy and James R. Cordy. Near-miss function clones in open source software: an empirical study. *Journal of Software Maintenance*, 22(3):165–189, 2010.

[135] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. *Tech. Report 541*, Queens School of Computing, Kingston, ON, Canada, 2007.

[136] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74:470–495, May 2009.

[137] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the 18th international symposium on Software testing and analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[138] Ripon K. Saha, Muhammad Asaduzzaman, Minhaz F. Zibran, Chanchal K. Roy, and Kevin A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 87–96, Washington, DC, USA, 2010. IEEE Computer Society.

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

[140] Ripon K. Saha, Chanchal K. Roy, and Kevin A. Schneider. Visualizing the evolution of code clones. In *Proceeding of the 5th international workshop on Software clones*, IWSC '11, pages 71–72, New York, NY, USA, 2011. ACM.

[141] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11:92–99, January 1992.

[142] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Softw.*, 11:70–77, November 1994.

[143] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343, Washington, DC, USA, 1996. IEEE Computer Society.

[144] Harald Störrle. Towards clone detection in uml domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 285–293, New York, NY, USA, 2010. ACM.

[145] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, pages 679–684, New York, NY, USA, 2006. ACM.

[146] Robert Tairas, Jeff Gray, and Ira Baxter. Visualization of clone detection results. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, eclipse '06, pages 50–54, New York, NY, USA, 2006. ACM.

[147] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano D. Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15:1–34, February 2010.

[148] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.

[149] Md. Sharif Uddin, Chanchal K. Roy, Kevin A. Schneider, and Abram Hindle. On the effectiveness of Simhash for detecting near-miss clones in large scale software systems. In *Proceedings of the 18th Working Conference on Reverse Engineering*, WCRE '11, pages 13 –22, October 2011.

[150] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, pages 67–76, Washington, DC, USA, 2002. IEEE Computer Society.

[151] Filip V. Rysselberghe and Serge Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 328 – 337, September 2004.

[152] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. CVSscan: visualization of code evolution. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 47–56, New York, NY, USA, 2005. ACM.

[153] Lucian Voinea and Alexandru Telea. CVSgrab: Mining the History of Large Software Projects. In *EuroVis*, pages 187–194, 2006.

[154] Lucian Voinea and Alexandru Telea. An open framework for cvs repository querying, analysis and visualization. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 33–39, New York, NY, USA, 2006. ACM.

[155] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE '08, pages 219–228, Washington, DC, USA, 2008. IEEE Computer Society.

[156] Richard Wettel and Radu Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 63–70, Washington, DC, USA, 2005. IEEE Computer Society.

[157] Christopher Williamson and Ben Shneiderman. The dynamic homefinder: evaluating dynamic queries in a real-estate information exploration system. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '92, pages 338–346, New York, NY, USA, 1992. ACM.

[158] Jingwei Wu, Claus W. Spitzer, Ahmed E. Hassan, and Richard C. Holt. Evolution spectrographs: visualizing punctuated change in software evolution. In *Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 57 – 66, September 2004.

[159] Yali Zhang, Hamid A. Basit, Stan Jarzabek, Dang Anh, and Melvin Low. Query-based filtering and graphical view generation for clone analysis. In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, ICSM '08, pages 376–385, 2008.

[160] Minhaz F. Zibran and Chanchal K. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *Proceedings of the 19th International Conference on Program Comprehension*, ICPC '11, pages 266 –269. IEEE, June 2011.

[161] Minhaz F. Zibran and Chanchal K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 105 –114, September 2011.

[162] Minhaz F. Zibran and Chanchal K. Roy. IDE-based Real-time Focused Search for Near-miss Clones. In *Proceedings of the Software Engineering Track of the 27th ACM Symposium on Applied Computing*, Riva del Garda, Trento, Italy, March 2012 (to appear).

# Appendix A

# VisCad User Guide

# VisCad User Guide

**1 Introducing VisCad**

1.1 Introduction 4

1.2 System Requirement 5

1.3 Obtaining and Running VisCad 6

1.4 Importing clone detection result 7

1.5 User Interface Components 18

1.6 Analyze Clone Fragments 23

1.7 Analyze Clone Files 27

**2 Visualization**

2.1 Introduction 31

2.2 Scatter Plot 32

2.3 Treemap 36

2.4 Hierarchical Dependency Graph 41

**3 Code Clone Metrics**

3.1 Introduction 48

3.2 Obtaining Metrics For Files 49

3.3 Obtaining Metrics For Directories 55

3.4 Obtaining Metrics For Clone Classes 58

**4 Filtering**

4.1 Introduction 62

4.2 Overlapping Clone filterng 63

4.3 Textual filtering 68

# Introducing VisCad

# Introduction

Detection and analysis of similar code fragments (“code clones”) has become an integral part of software maintenance. In response, over the last decade a great many clone detection techniques and tools have been proposed. However, identifying useful cloning information from the large volume of textual data produced by these detectors is challenging. VisCad is a tool with which a user can visualize and analyze large volumes of raw cloning data in an interactive fashion. Users can analyze and identify distinctive code clones through a set of visualization techniques, metrics and data filtering operations. The loosely coupled architecture of VisCad allows users to work with the clones of any clone detection tools that report source co-ordinates of the found clones. This yields the opportunity to work with the clone detectors of choice, which is important for clone analysis since clone detectors have their own strengths and weaknesses.

## System Requirement

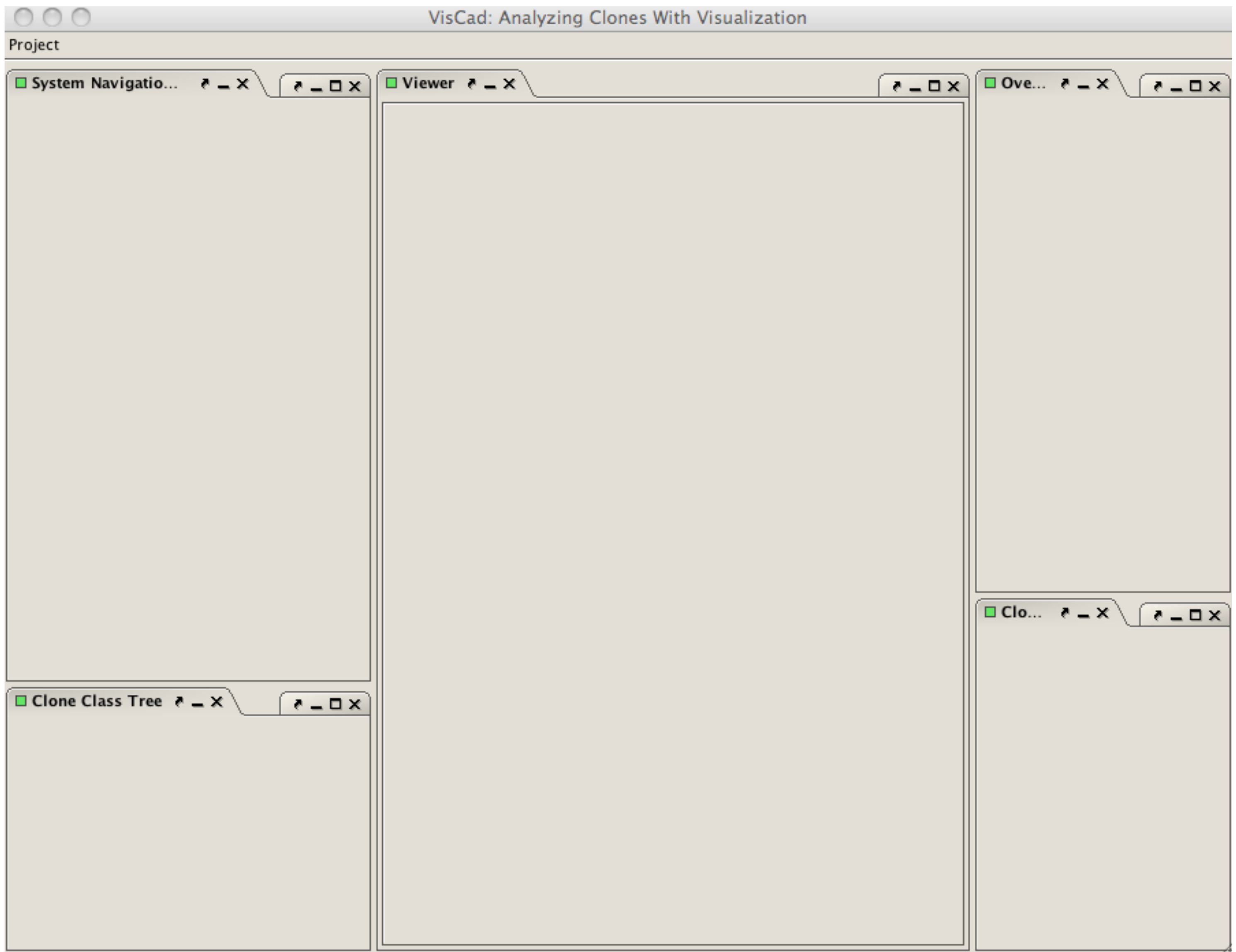
1. VisCad requires Java Runtime Environment (JRE) 6 or later. You can download the recent JRE from here .
2. We have successfully tested VisCad on Windows XP, Windows 7, Max OS X( version 10.6) and on some Linux distribution (such as Ubuntu).
4. The more RAM your computer has, the better performance you gain from VisCad. However, 2.0 GB or more is recommended.
5. Display dimension of 1024 x 768 or greater is recommended.

## Obtaining and Running VisCad

Follow the steps listed below:

1. Download the VisCad_beta.zip file from here . You can obtain the most recent version of VisCad, documentation, source code from this location.
2. Extract the contents of the archive.
3. Double click on the VisCad.jar file to run the program.

## VisCad user interface

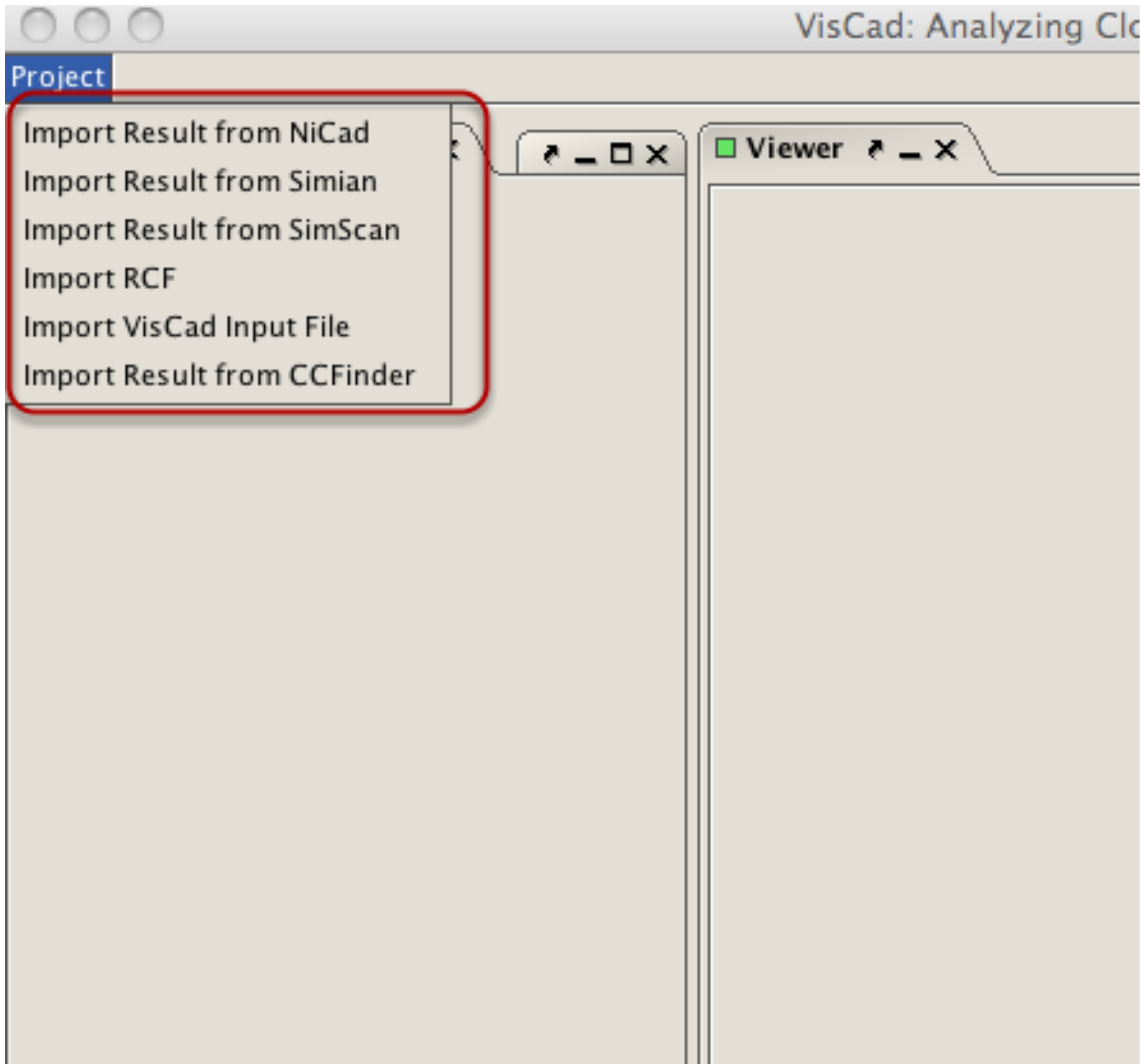



Running the viscad.jar file opens the above window.

## Importing clone detection result

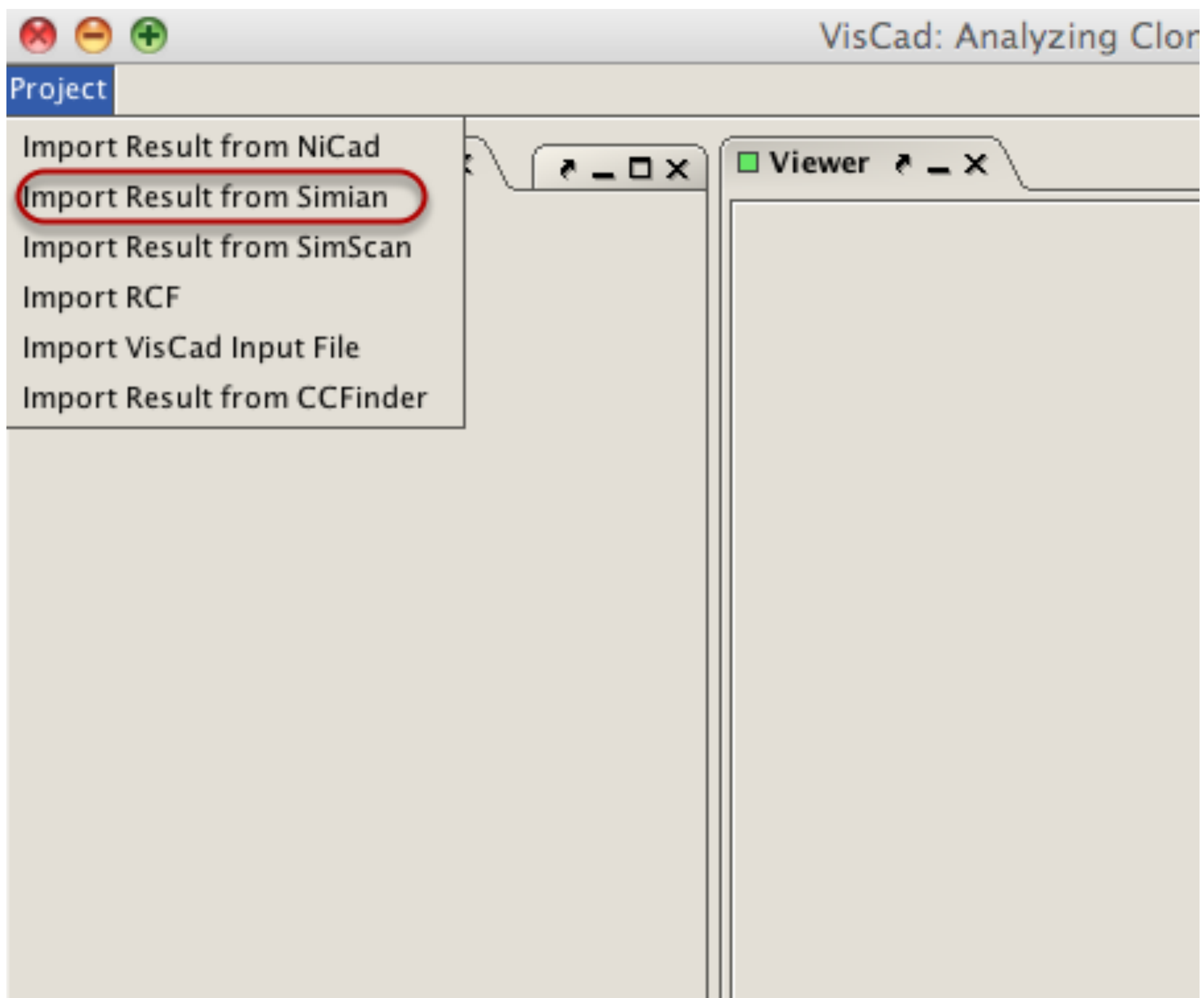
VisCad requires the subject system and the clone detection result you obtained by running clone detection on the subject system using the supported clone detectors.
Current you can directly import clone detection result of CCFinder, Simian, SimScan, NiCad. If you have clone detection result in RCF format, you can also import and anlyze the data in VisCad. For other clone detection tools, you need to convert the result into VisCad input file format. VisCadBeta.zip file contains an example of VisCad input file.

### Make the selection



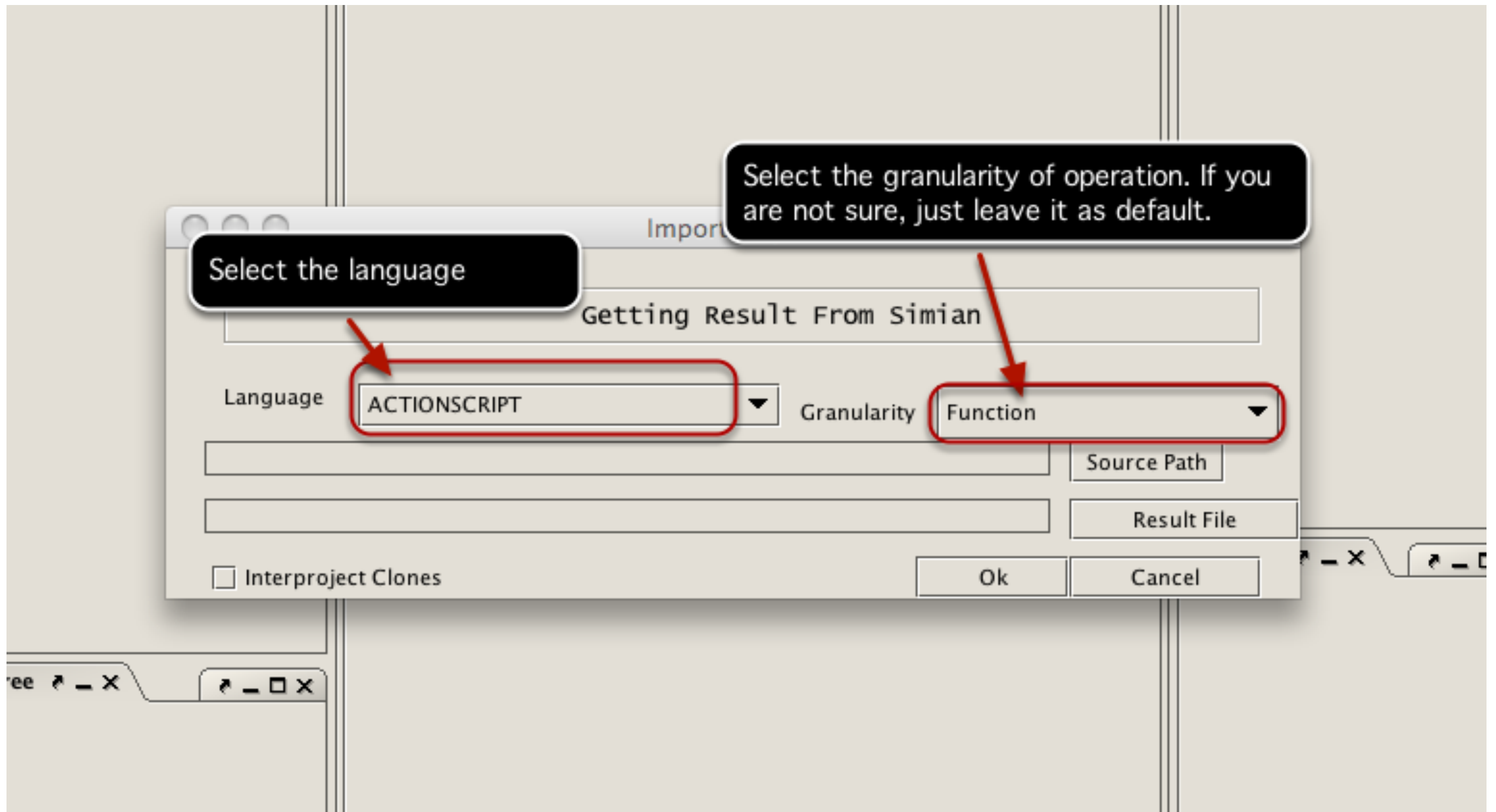

To import clone detection result or RCF data file, click on the *Project* menu. This opens a list of importing operations supported by VisCad.

## Import clone detection result of Simian



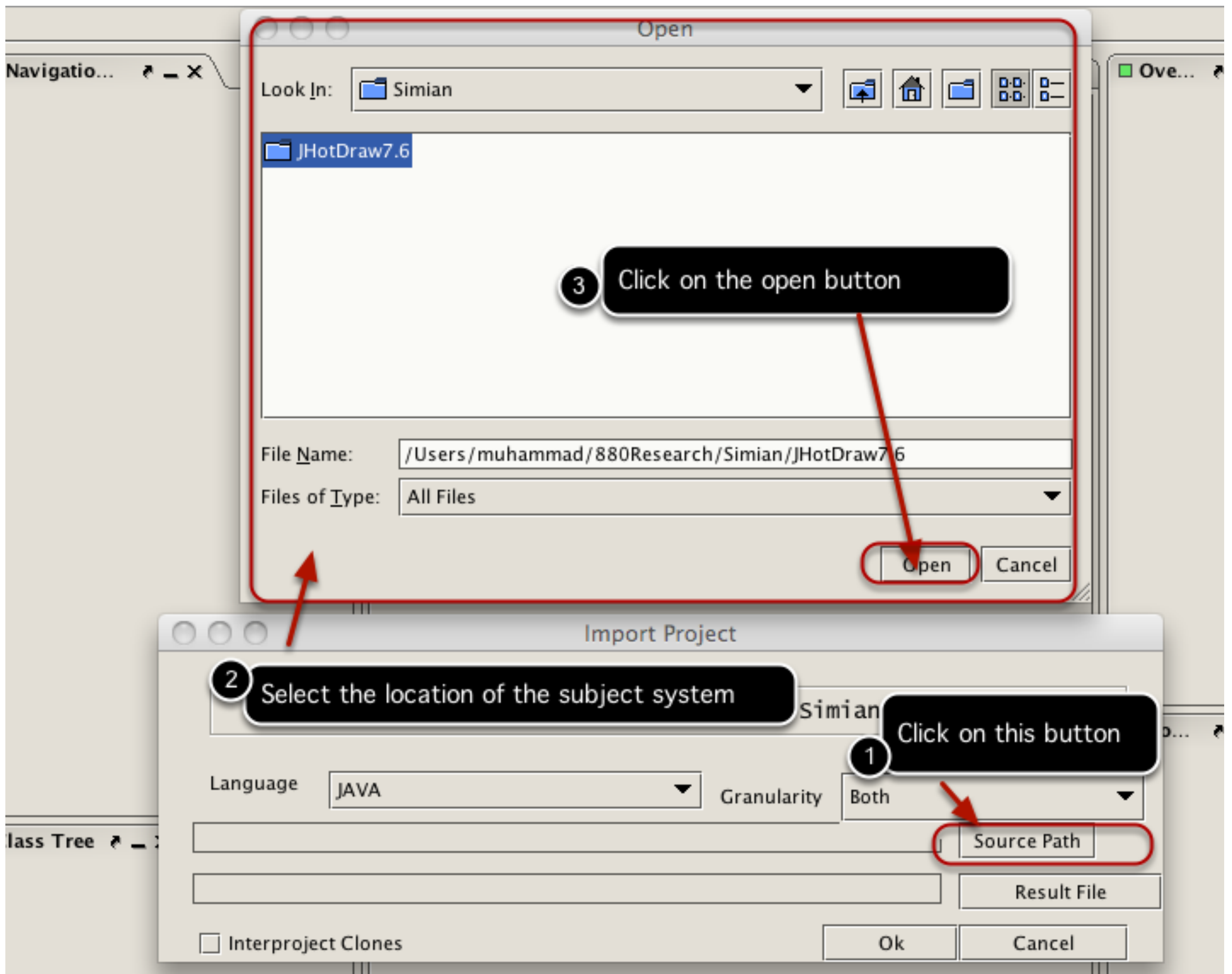

Suppose your computer contians the source code of JHotDraw(version 7.6) and located in the JHotDraw7.6 directory. You detect the clones using Simian on that directory and stores the result in jhotdrawResultSimian.txt file. To import the result, you need to click on the Project menu and select the *Import Result from Simian* menu item.

## Importing clone detection result of Simian( Continued)




This opens the above dialog box.

## Select subject system



## Select the clone detection result file

Open

Look In: Simian

JHotDraw7.6

jhotdrawResultSimian.txt

3 Click on the open button

File Name: jhotdrawResultSimian.txt

Files of Type: All Files

Open Cancel

locate the result file

2

Import Project

Getting Result From Simian

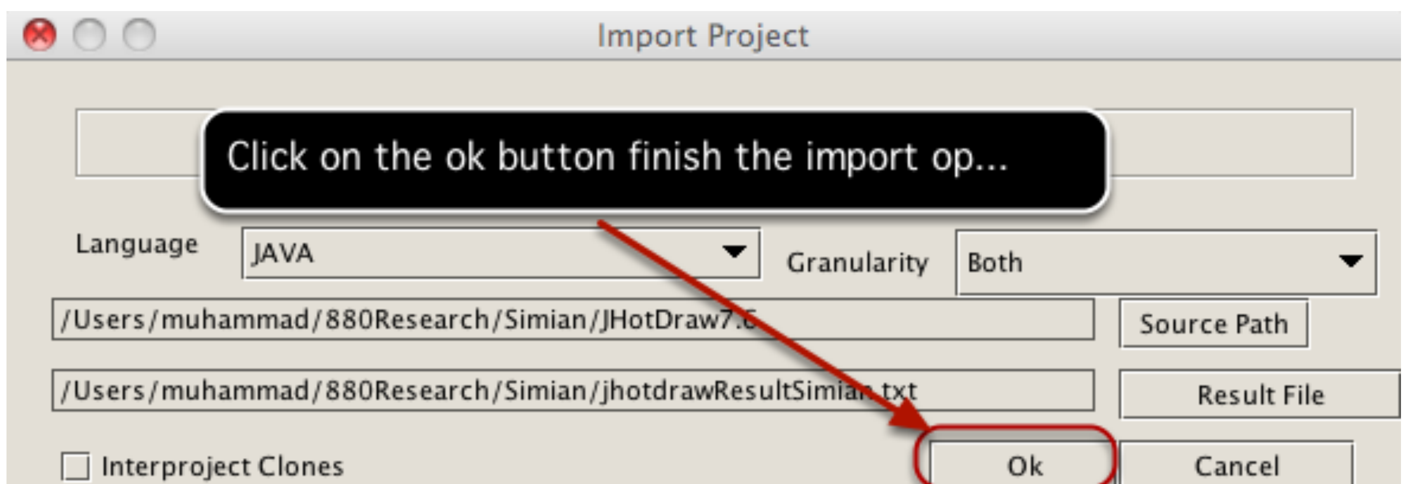
Language JAVA Granularity Both

/Users/muhammad/880Research/Simian/JHotDraw7.6 Source Path

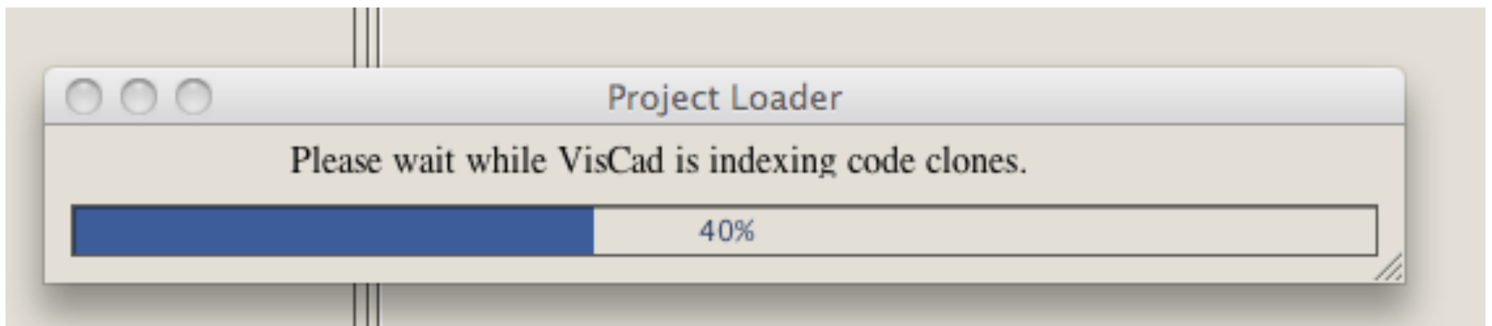
Result File

Interproject Clones 1 Click on this button Ok Cancel

## Complete the import opeation



## Loading clone detection result




VisCad is now loading the clone detection result.

## Complete the loading operation

VisCad: Analyzing Clones With Visualization

Project

System Navigatio...

Systems
JHotDraw7.6 (Clone: 1758 Cloned File: 342)
Documentation (Clone: 0 Cloned File: 0)
JavaDoc (Clone: 0 Cloned File: 0)
Samples (Clone: 0 Cloned File: 0)
Source (Clone: 1758 Cloned File: 342)

Clone Class Tree

JHotDraw7.6
Clone Classes:737

Viewer

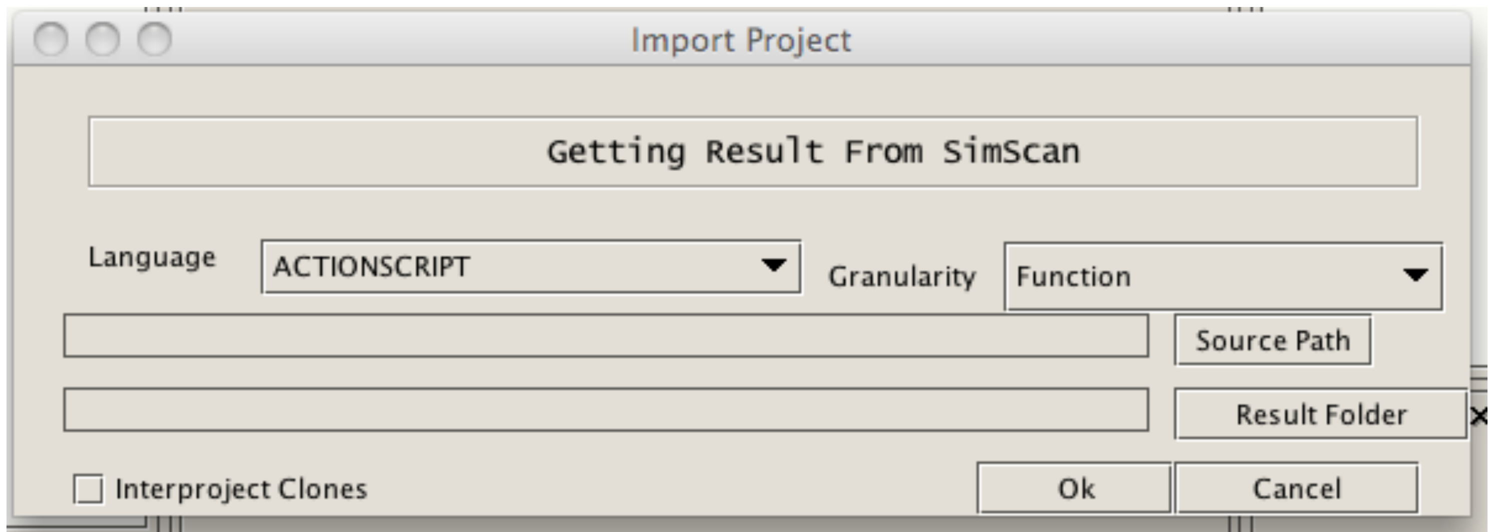
Ove...

| Category | Details |
|---|---|
| processedFiles | 680 |
| duplicatedLines | 19429 |
| tool | Similarity Ana... |
| duplicatedFiles | 1414 |
| processedLines | 61258 |
| duplicatedBlo... | 342 |
| processingTi... | 3.150sec |

Clo...

The result is loaded into VisCad.

## Import result from SimScan

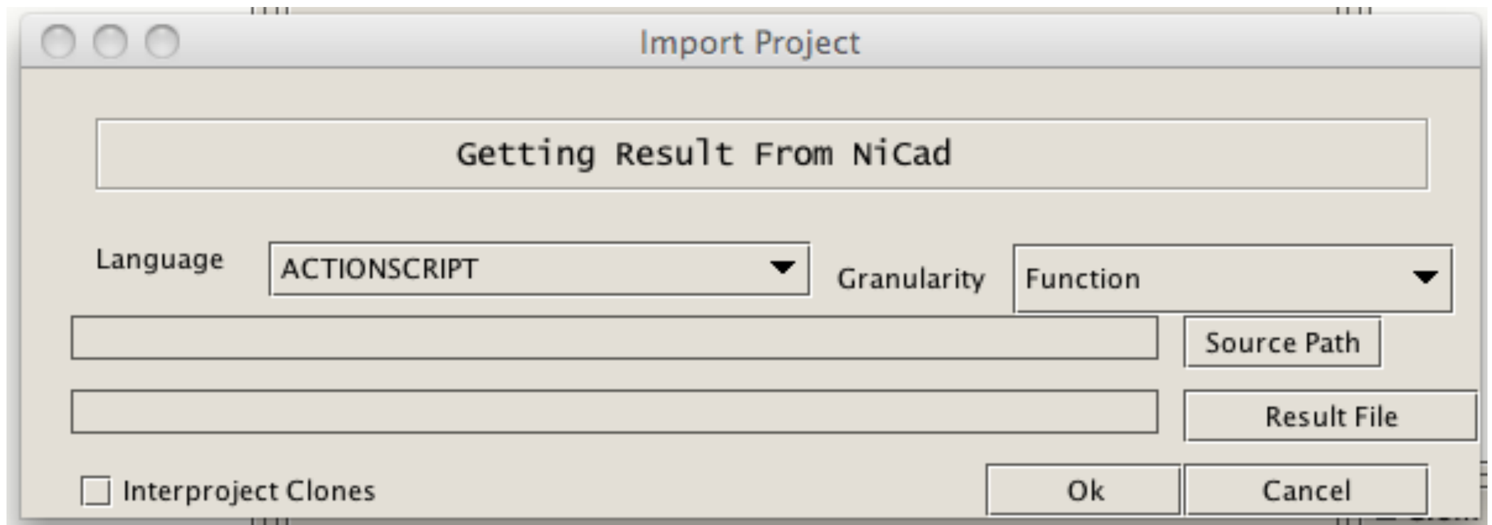



SimScan creates a folder that contains the output of the clone detection. A sample clone detection result (jhotdrawReportSimScan.zip) for JHotDraw (Version 7.6) is included in the VisCadBeta.zip file. You can also find the source code of the subject system in a zip file (jhotdraw-7.6.nested.zip).

Follow the steps listed below:

1. Unzip the jhotdraw-7.6.nested.zip file and rename the folder to JHotDraw7.6.
2. Unzip the jhotdrawReportSimScan.zip file. This will create a folder (jhotdrawReportSimScan) that contains the clone detection results. Put the folder in the same place where JHotDraw7.6 folder resides.
3. Select the ‘Import Result from SimScan’ menu item from the ‘Project’ menu (located on the top left of the VisCad interface). This opens a dialog to import the result.
4. Click on the Source Path button and select the source code directory (In this case, JHotDraw7.6 folder). This should be the folder on which you apply clone detection.
5. Now, click on the Result Folder button and select the clone detection result folder (In this case, jhotdrawReportSimScan).
6. Select the language of the subject system and also the granularity of clone detection. If you are not sure, you can leave them as default.
7. Now, click on the Ok button to load the subject system by VisCad. The loading time may vary depending on the size of the system.
8. You can click on the Cancel button to cancel the import operation.

## Import result from NiCad

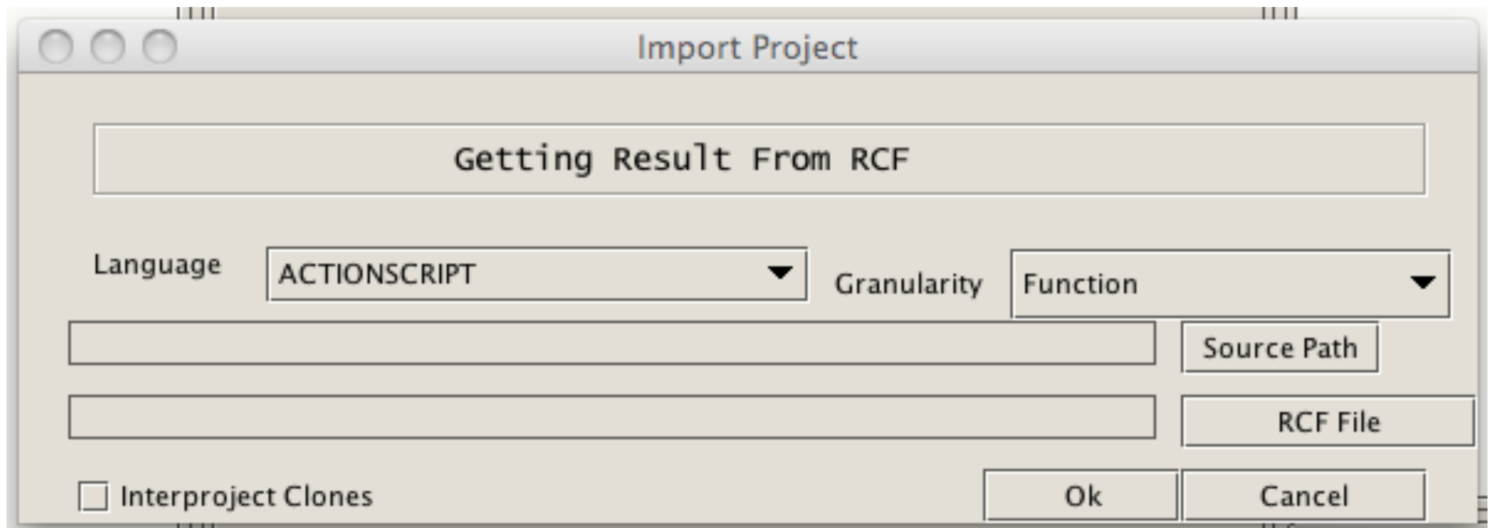



A sample clone detection result (JHotDraw7.6_functions-clones-0.3.xml) for JHotDraw (Version 7.6) is included in the zip file. You can also find the source code of the subject system in a zip file (jhotdraw-7.6.nested.zip). Clones were detected using NiCad.

Follow the steps listed below:

1. Unzip the jhotdraw-7.6.nested.zip file and rename the folder to JHotDraw7.6.
2. Put the clone detection result file in the same directory where JHotDraw7.6 folder resides.
3. Select the ‘Import Result from NiCad’ menu item from the ‘Project’ menu (located on the top left of the VisCad interface). This opens a dialog to import the result.
4. Click on the Source Path button and select the source code directory (In this case, JHotDraw7.6 folder). This should be the folder on which you apply clone detection.
5. Now, Click on the Result File button and select the clone detection result file (In this case, JHotDraw7.6_functions-clones-0.3.xml).
6. Select the language of the subject system and also the granularity of clone detection. If you are not sure, you can leave them as default.
7. Now, click on the Ok button to load the subject system by VisCad. The loading time may vary depending on the size of the system.
8. You can click on the Cancel button to cancel the import operation.

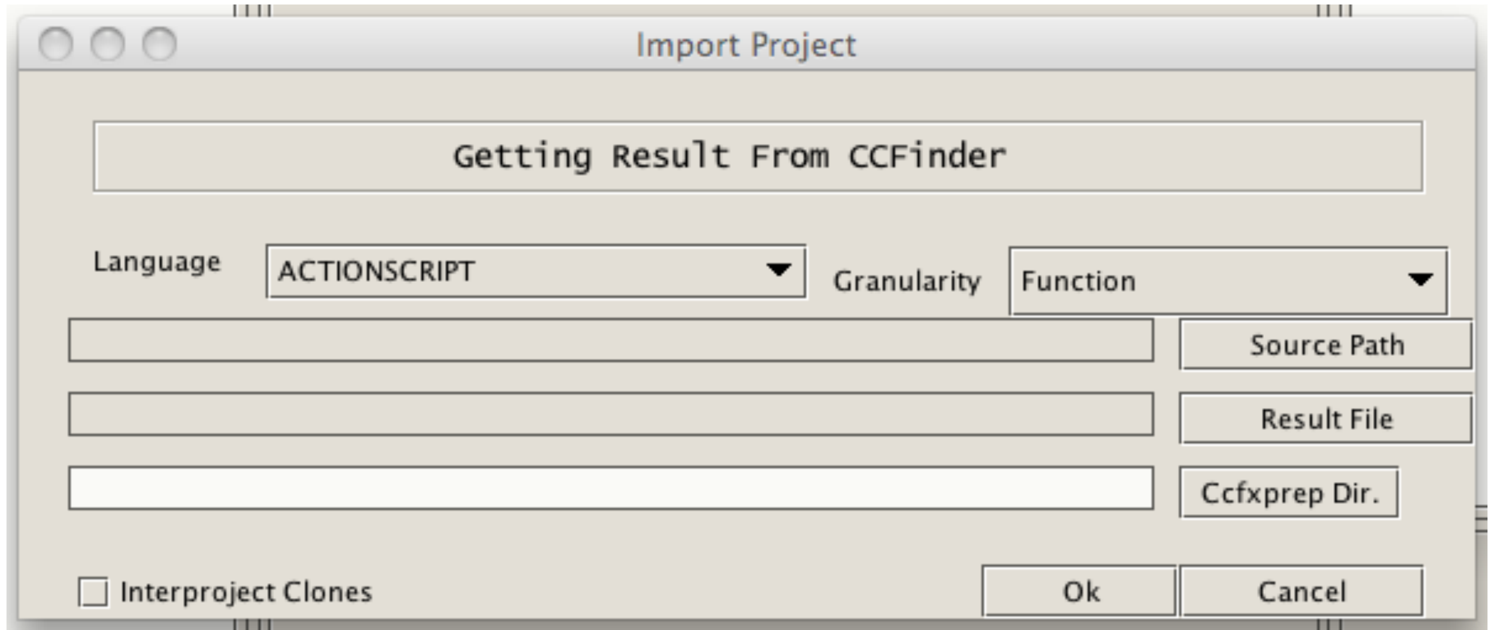
## Import RCF file




RCF is a data format that can store clone data for a number of versions of a subject system.
For this example, we have used GNU Wget. Download both the subject system and the RCF file from here. The RCF file is packaged with the source code of the subject system.
Follow the steps listed below:

1. Extract the content of the archive and rename the folder to rcfResult. You now have two folders inside the rcfResult folder. One is the wget that contains the source code of seven different versions of the system. Another one is the wget.rcf that contains the clone detection results of those systems in the RCF format.
2. Select the 'Import RCF' menu item from the 'Project' menu (located on the top left of the VisCad interface). This opens a dialog to import the result.
3. Click on the Source Path button and select the wget folder located inside the rcfResult folder.
4. Now, click on the RCF File button and select the wget.rcf file.
5. Select the language of the subject system and also the granularity of clone detection. If you are not sure, you can leave them as default.
6. Now, click on the Ok button to load all seven-subject subjects by VisCad. The loading time may vary depending on the number of subject systems.
7. You can click on the Cancel button to cancel the import operation.

## Import result from CCFinder

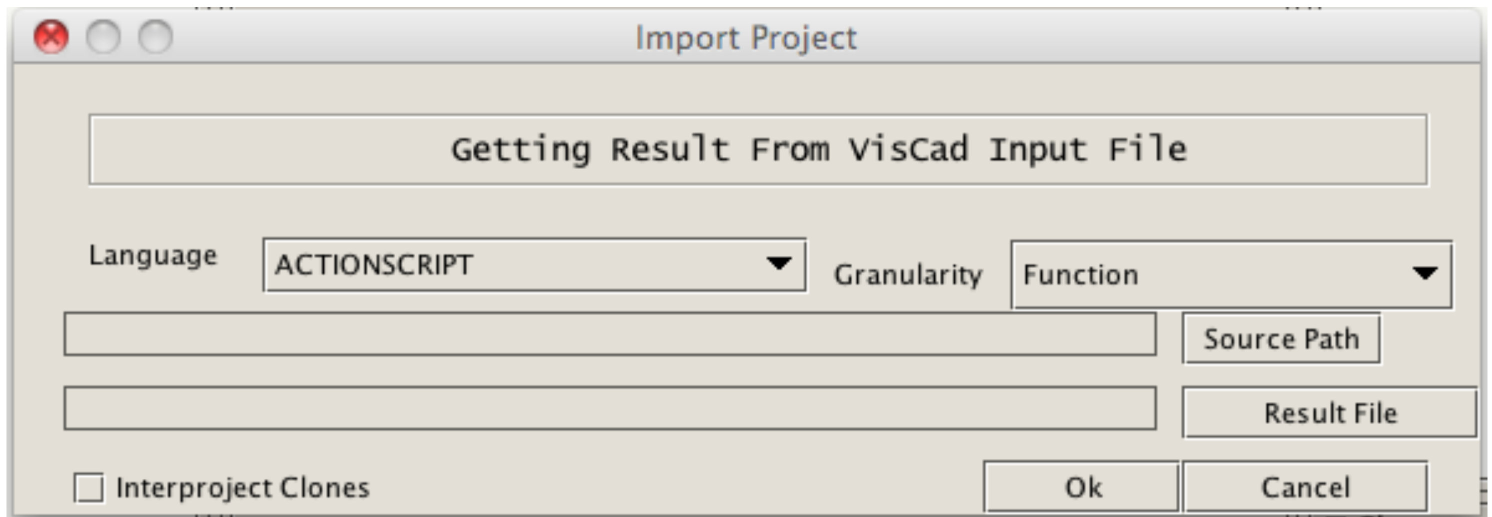



During the clone detection, CCFinder creates a directory (.ccfxprepdir). If you apply clone detection on JHotDraw7.6 directory, then the .ccfxprepdir will be created right inside that directory.

Follow the steps listed below:

1. Unzip the jhotdraw-7.6.nested.zip file and rename the folder to JHotDraw7.6. Detect clones with CCFinder and then rename the .ccfxprepdir directory to ccfxprepdir.
2. The output file is a binary one (such as a.ccfxd). You need to convert that into a text file (see here). From now, we will refer the text file as the result file.
3. Put the clone detection result file in the same directory where JHotDraw7.6 folder resides.
4. Select the 'Import CCFinder Project' menu item from the 'Project' menu (located on the top left of the VisCad interface). This opens a dialog to import the result.
5. Click on the Source Path button and select the source code directory (In this case, JHotDraw7.6 folder). This should be the folder on which you apply clone detection.
6. Now, click on the Result File button and select the clone detection result file.
7. Click on the Ccfxprep Dir. button and select the ccfxprepdir directory.
8. Select the language of the subject system and also the granularity of clone detection. If you are not sure, you can leave them as default.
9. Now, click on the Ok button to load the subject system by VisCad. The loading time may vary depending on the size of the system.
10. You can click on the Cancel button to cancel the import operation.

## Import VisCad input file




For any other clone detection tools, the result file needs to be converted to the VisCad input file format. A sample clone detection result (resultViscadInputFile.txt) for JHotDraw (Version 7.6) is included in the VisCadBeta.zip file. Clones were detected using Simian.

Follow the steps listed below:

1. Unzip the jhotdraw-7.6.nested.zip file and rename the folder to JHotDraw7.6.
2. Put the clone detection result file in the same directory where JHotDraw7.6 folder resides.
3. Select the 'Import VisCad Input File' menu item from the 'Project' menu (located on the top left of the VisCad interface). This opens a dialog to import the result.
4. Click on the Source Path button and select the source code directory (In this case, JHotDraw7.6 folder). This should be the folder on which you apply clone detection.
5. Now, click on the Result File button and select the clone detection result file (In this case, resultViscadInputFile.txt).
6. Select the language of the subject system and also the granularity of clone detection. If you are not sure, you can leave them as default.
7. Now, click on the Ok button to load the subject system by VisCad. The loading time may vary depending on the size of the system.
8. You can click on the Cancel button to cancel the import operation.

## User Interface Components

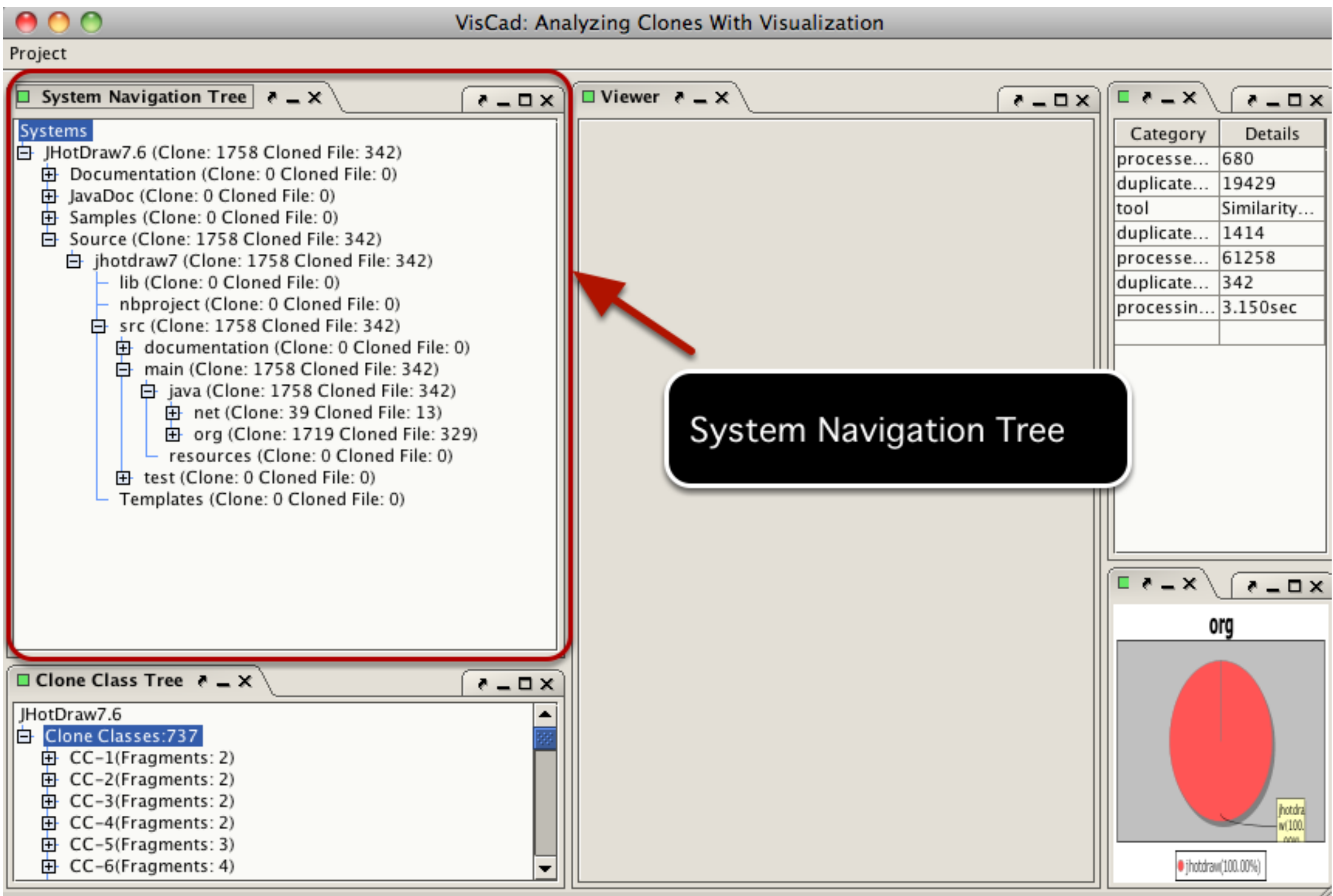
The main user interface of VisCad can be divided into three parts.

1. Left Part: The left part accommodates the *clone browser*. The clone browser has two parts, one of which displays the distribution of clones over the directories and sub- directories in the subject system, known as *System Navigation Tree*. The other part, located on the bottom of the clone browser, lists all *clone classes* and the number of clone snippets in each class, called *Clone Class Tree*.

2. Middle Part: The middle part of VisCad accommodates different views in separate tabs. We will refers this part as *Viewer*.

3. Right Part: The top- right window shows the clone detection specific information VisCad obtained while parsing the result file for the selected
subject system. For any selected directory in the *system navigation tree*, the bottom-right window shows the distribution of clones in its sub-directories through a pie chart.

### System Navigation Tree




It maps clones to files and directories

## Clone Class Tree

Clone class tree groups the detected clones into different classes.

# Viewer

Depending on the user's selection, it accomodates different views(such as the scatter plot, treemap, hierarchical dependency graph, source code browser etc.)

## Obtaining information specific to clone detection

Tool name

Provide information that is produced by the clone detectors as part of clone detection result

| Category | Details |
| --- | --- |
| processedFiles | 680 |
| duplicatedLines | 19429 |
| tool | Similarity Analyser 2.3.32... |
| duplicatedFiles | 1414 |
| processedLines | 61258 |
| duplicatedBlocks | 342 |
| processingTime | 3.150sec |

This part contains information such as tool name, overview of clone detection result.

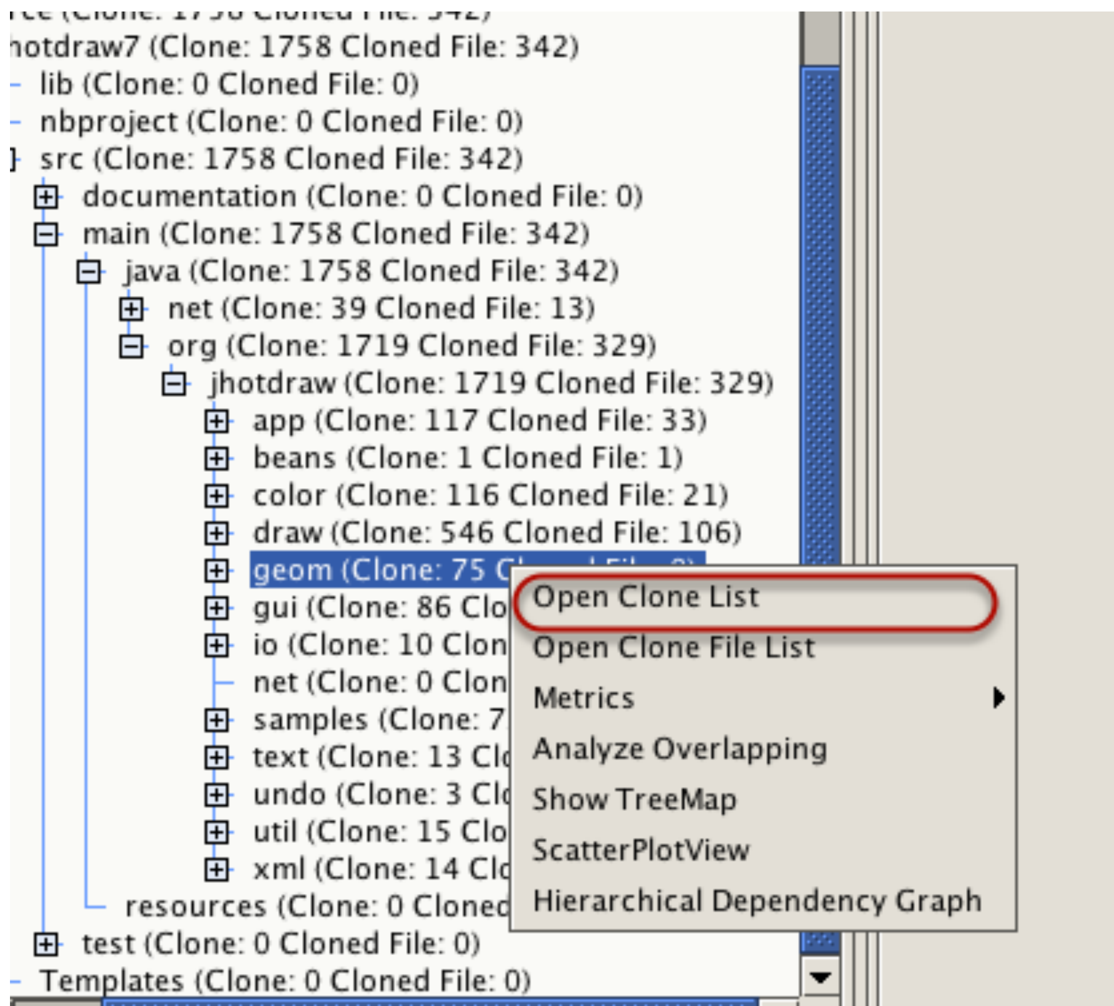
## Clone distribution window

Click on a directory in the system navigation tree to select it. Based on the selection, the clone distribution window updates and shows the distribution of clones in its subdirectories through a pie chart.

## Analyze Clone Fragments

You can analyze the source code of the clone fragments using the *code browser*. The same component is used for analyzing clone code fragments in other places also.

### Open the popup menu and make selection




If we want to analyze the clone fragments on the *geom directory*, we need to select the directory, right click on the mouse button to bring up the popup menu and select the *Open clone list* menu item.

## The code browser

Viewer

Clone Code Browser

1 First tab in the code browser shows the clone files and the fragments within the files

Clone Files and Fragments | Clone Pairs

| Status | Name | Path | LOC | CLOC | Clones |
|---|---|---|---|---|---|
| ☑ | BezierPath.j... | JHotDraw7.6/Source/jhotdraw... | 1,248 | 248 | 16 |
| ☑ | BezierPathIt... | JHotDraw7.6/Source/jhotdraw... | 335 | 172 | 8 |
| ☑ | ConvexHull... | JHotDraw7.6/Source/jhotdraw... | 261 | 20 | 2 |
| ☑ | DoubleStro... | JHotDraw7.6/Source/jhotdraw... | 283 | 120 | 10 |
| ☑ | Geom.java | JHotDraw7.6/Source/jhotdraw... | 859 | 206 | 16 |
| ☑ | GrowStroke... | JHotDraw7.6/Source/jhotdraw... | 91 | 28 | 2 |
| ☑ | Insets2D.java | JHotDraw7.6/Source/jhotdraw... | 352 | 56 | 2 |
| ☑ | Polygon2D.j... | JHotDraw7.6/Source/jhotdraw... | 1,124 | 480 | 20 |

| Status | PCID | CCID | Path | SL | EL | CLOC |
|---|---|---|---|---|---|---|
| ☑ | 190 | 80 | JHotDraw7.... | 188 | 196 | 9 |
| ☑ | 542 | 219 | JHotDraw7.... | 835 | 841 | 7 |
| ☑ | 543 | 219 | JHotDraw7.... | 855 | 861 | 7 |
| ☑ | 598 | 243 | JHotDraw7.... | 1,233 | 1,246 | 14 |
| ☑ | 799 | 325 | JHotDraw7.... | 819 | 828 | 10 |
| ☑ | 800 | 325 | JHotDraw7.... | 875 | 884 | 10 |
| ☑ | 1,070 | 439 | JHotDraw7.... | 492 | 504 | 13 |
| ☑ | 1,071 | 439 | JHotDraw7.... | 590 | 602 | 13 |
| ☑ | 1,072 | 439 | JHotDraw7.... | 540 | 552 | 13 |
| ☑ | 1,466 | 612 | JHotDraw7.... | 386 | 404 | 19 |
| ☑ | 1,467 | 612 | JHotDraw7.... | 359 | 377 | 19 |
| ☑ | 1,517 | 635 | JHotDraw7.... | 509 | 536 | 28 |
| ☑ | 1,518 | 635 | JHotDraw7.... | 559 | 586 | 28 |

Differencer

2 Clone Files

4 Shows the clone fragments located in the selected files( on the left side)

3 We can change the file selection. This also change the displayed clone fragments in the right side.

The previous selection open the code browser in the *viewer* window in a new tab.

# An Example

Viewer

Clone Code Browser

Click on the checkbox to select or deselect a file

Clone Files and Fragments | Clone Pairs

| Status | Name | Path | LOC | CLOC | Clo... |
|---|---|---|---|---|---|
| ☐ | BezierPath.java | JHotDraw7.6/Source... | 1,248 | 248 | 15 |
| ☐ | BezierPathIterator.java | JHotDraw7.6/Source... | 335 | 172 | 8 |
| ☐ | ConvexHull.java | JHotDraw7.6/Source... | 261 | 20 | 2 |
| ☐ | DoubleStroke.java | JHotDraw7.6/Source... | 283 | 120 | 10 |
| ☐ | Geom.java | JHotDraw7.6/Source... | 859 | 206 | 16 |
| ☐ | GrowStroke.java | JHotDraw7.6/Source... | 91 | 28 | 2 |
| ☑ | Insets2D.java | JHotDraw7.6/Source... | 352 | 56 | [illegible] |
| ☐ | ...lygon2D.java | JHotDraw7.6/Source... | 1,124 | 480 | 20 |

| Status | PCID | CCID | Path | SL | EL | CLOC |
|---|---|---|---|---|---|---|
| ☑ | 1,468 | 613 | JHotDraw7.... | 317 | 344 | |
| ☑ | 1,469 | 613 | JHotDraw7.... | 249 | 276 | |

Selected file

Only two clone fragments are located in the selected Insets2D.java file

We can also change the selection. Clicking on the checkbox select/deselect a clone fragment.

Suppose, we want to analyze the clone fragments located only in the Insets2D. java file.
We need to deselect all files except Insets2D.java file by clicking on the checkboxes. From the right side, we can see that only two code fragments are located in this file.

## Examining clone pairs

For the selected clone fragments, the next tab in the code browser shows the clone pairs. Selecting a clone pair also displays the code fragments in the bottom panel.

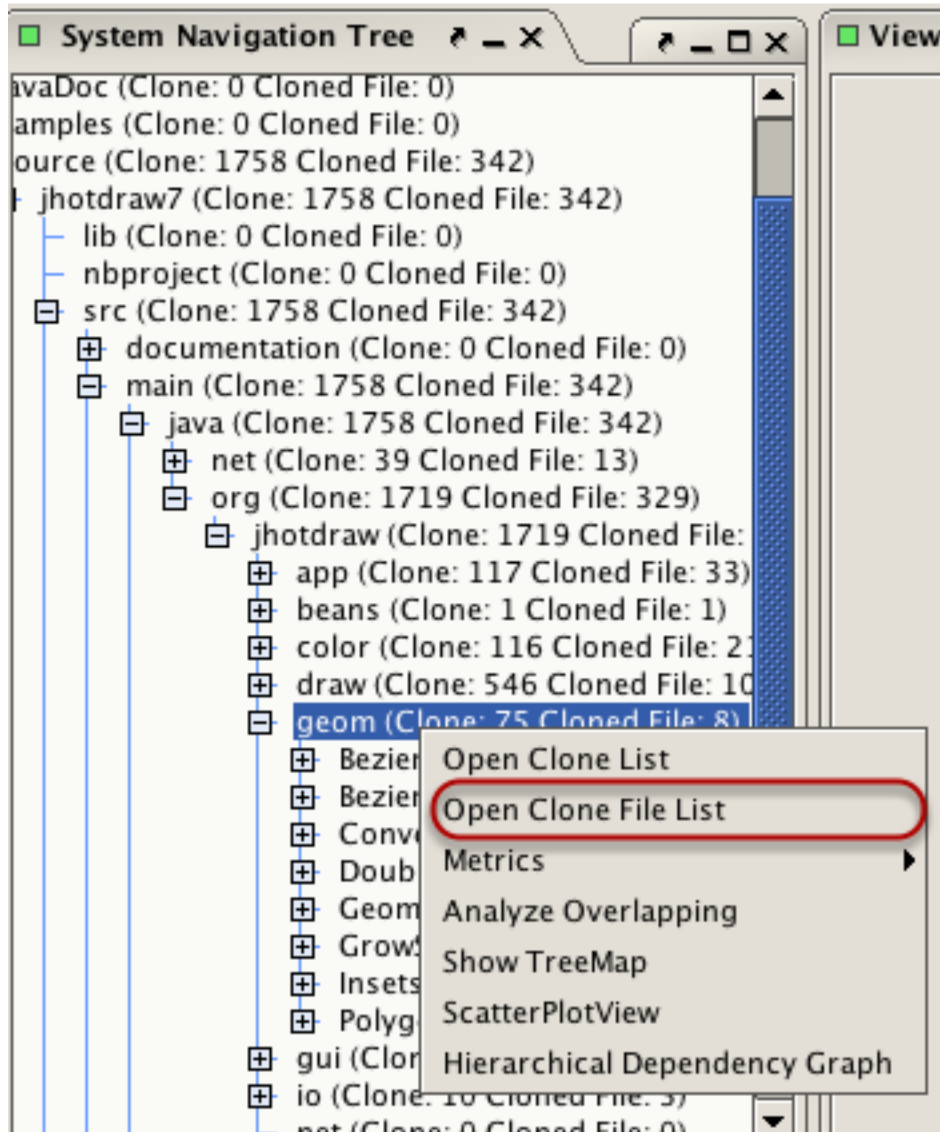
## Source code difference analysis

We can understand the source code differences using the diff viewer. In the above figure, the two clone fragments are exact copy of each other.
Similar to PCID, another number(CCID) is used to uniquely identify each clone class.

## Analyze Clone Files

You can use this view to analyze and compare clone files with grouping and selection features.

### Make selection



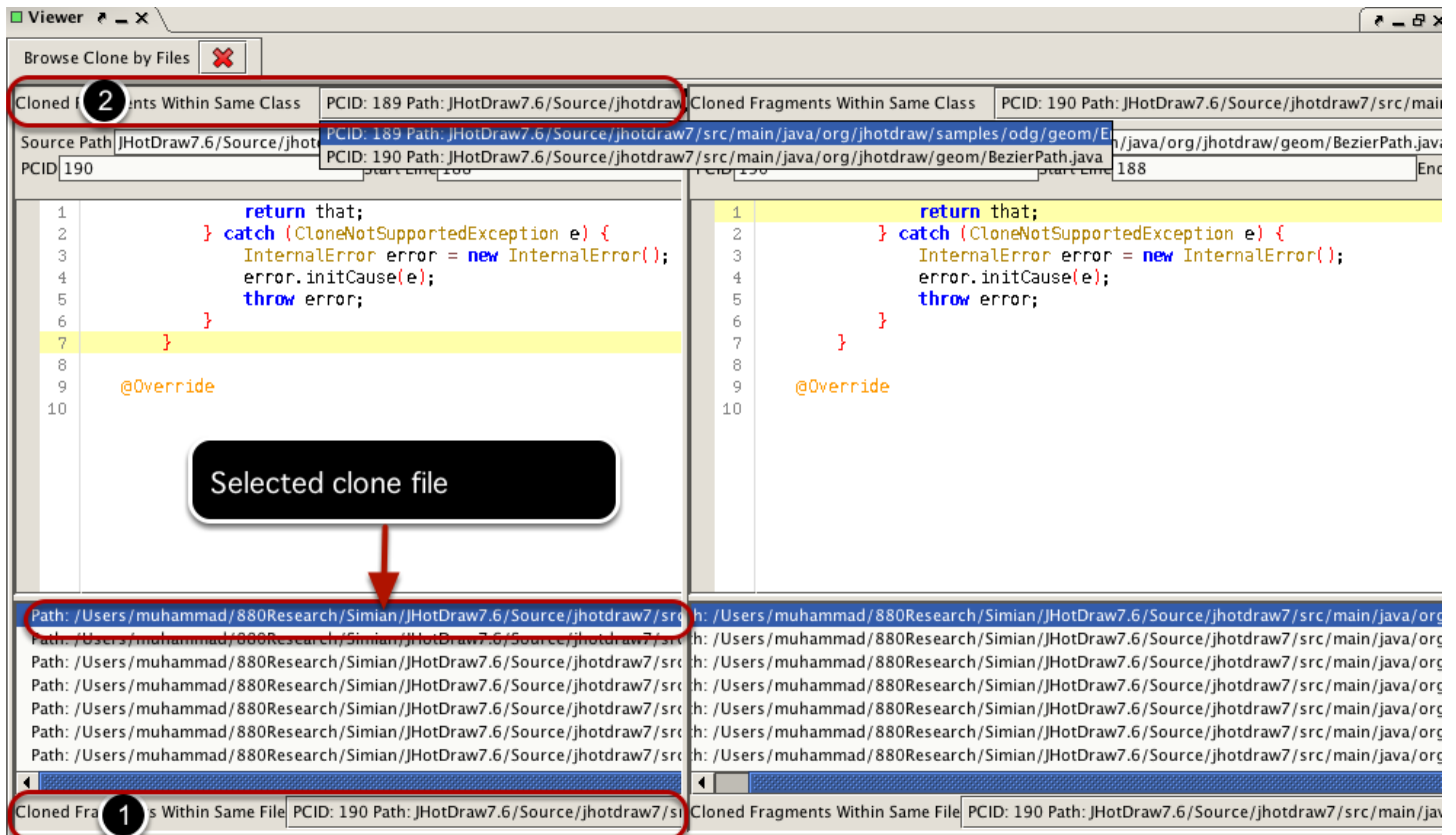

Select a target directory. Right click on the selected directory to open the popup menu and select *Open Clone File List* menu item.

# List of clone files

List of clone files located within the selected directory

We can select a clone file from the list of files.

## Features




This view has some advantages. It groups clone fragments located within the selected file( labelled with 1). For the selected clone fragment, it also groups all clone fragments that falls within the same clone class( labelled with 2).

The left and right part shows the same list of clone files. We can change the selection and compare the related clone fragments side by side.

# Visualization

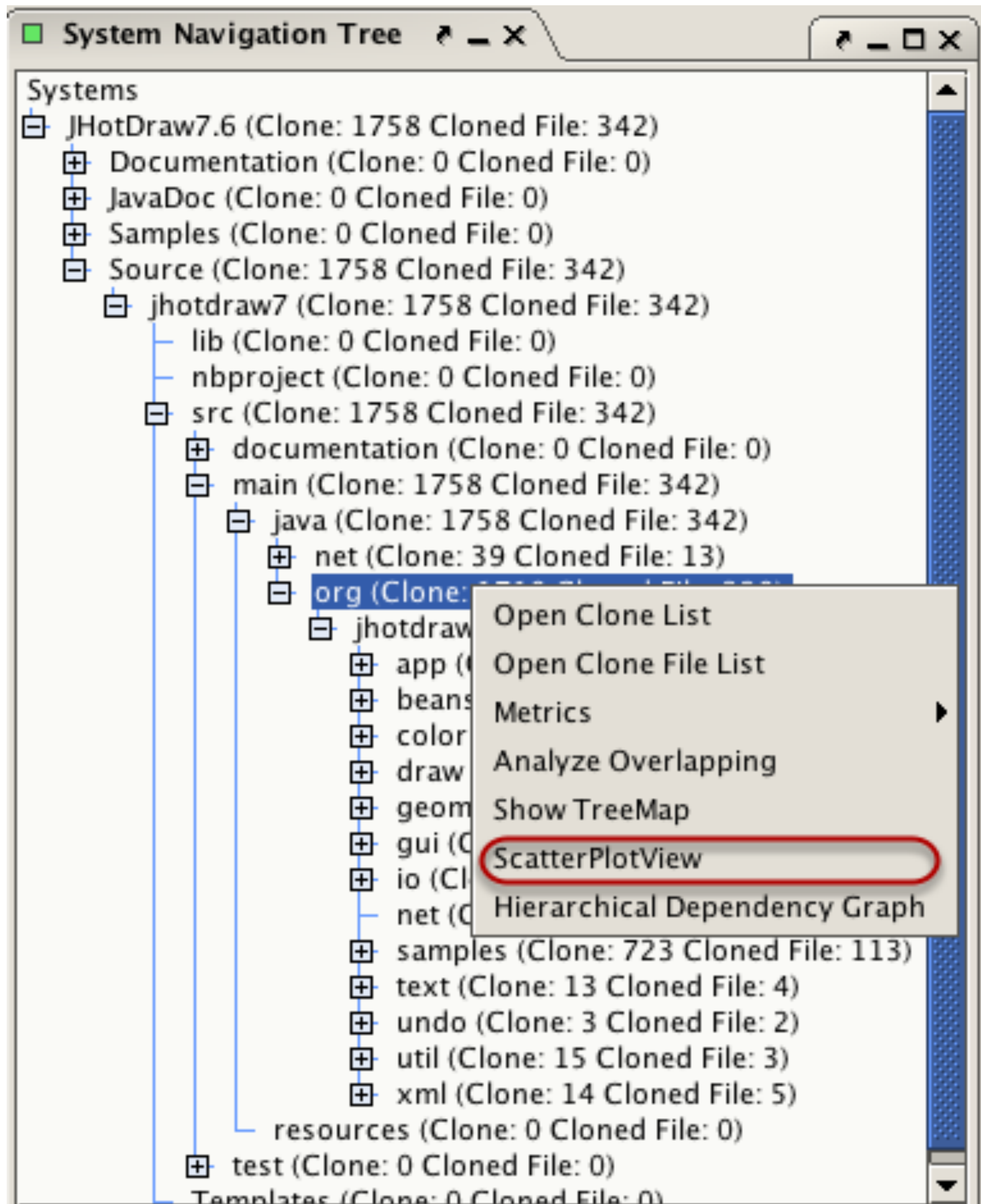
## Introduction

Visualization plays an important role in code clone analysis since it can provide high level overview of cloning in a system. At present, VisCad supports three different visualizations which are scatter plot, treemap and hierarchical dependency graph.

## Scatter Plot

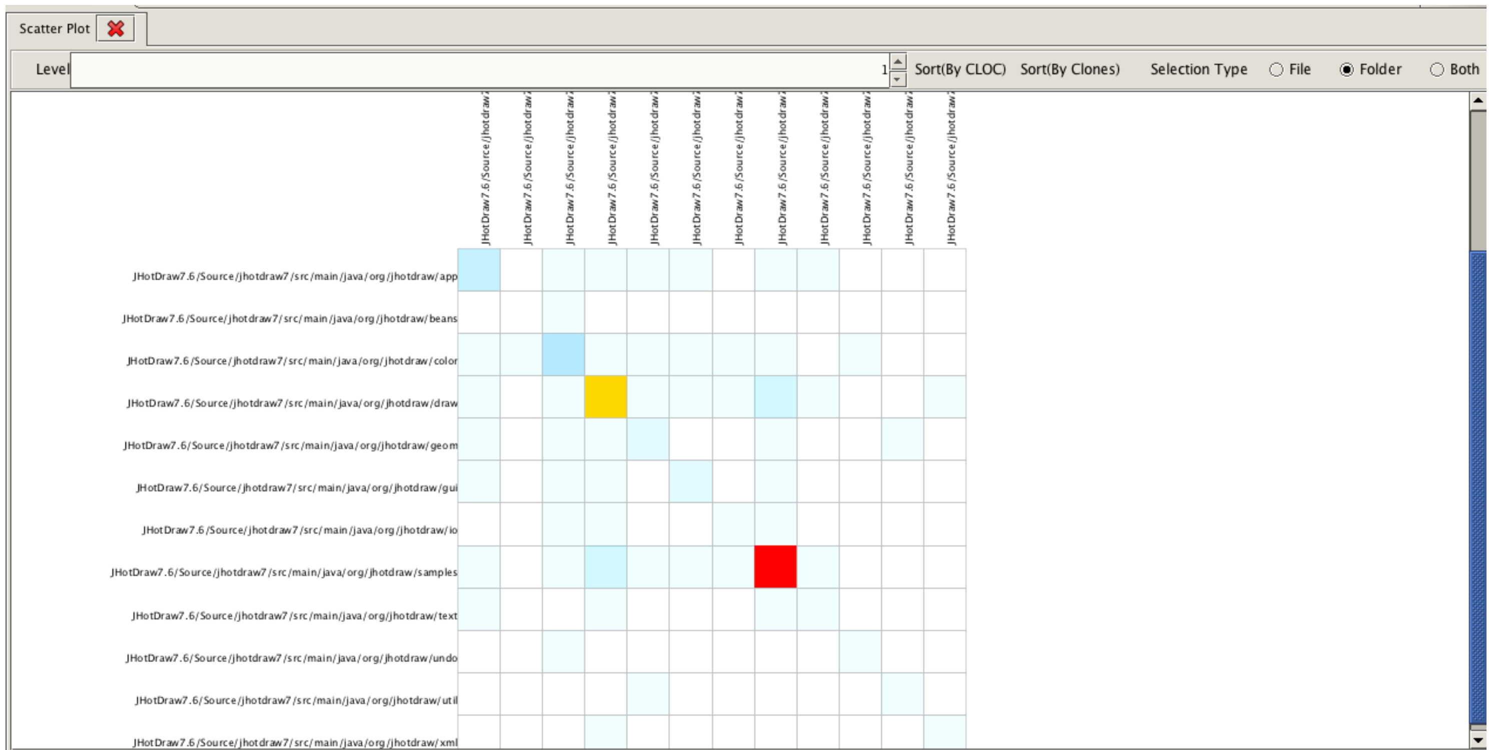
A scatter plot can be viewed as a two dimensional matrix where each cell represents the cloning status between a pair of files or directories. In VisCad, cells render the clone pairs distributed between a pair of files or directories using a color heatmap. Cells are also labelled in the horizontal and vertical axes.

### Opening a scatter plot



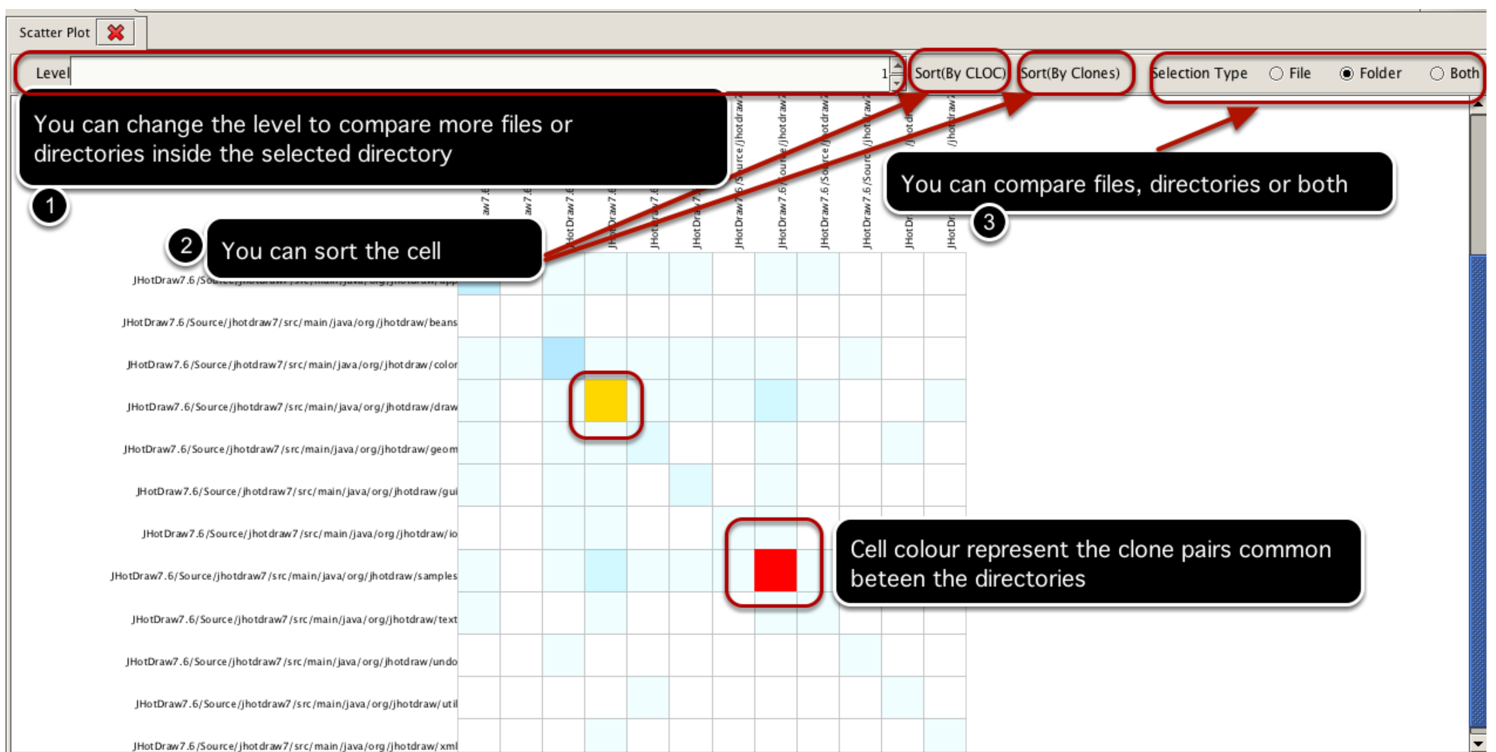

Select a directory from the system navigation tree and right click on it to open a popup menu. Click on the *Scatter Plot View* menu item.

## An example

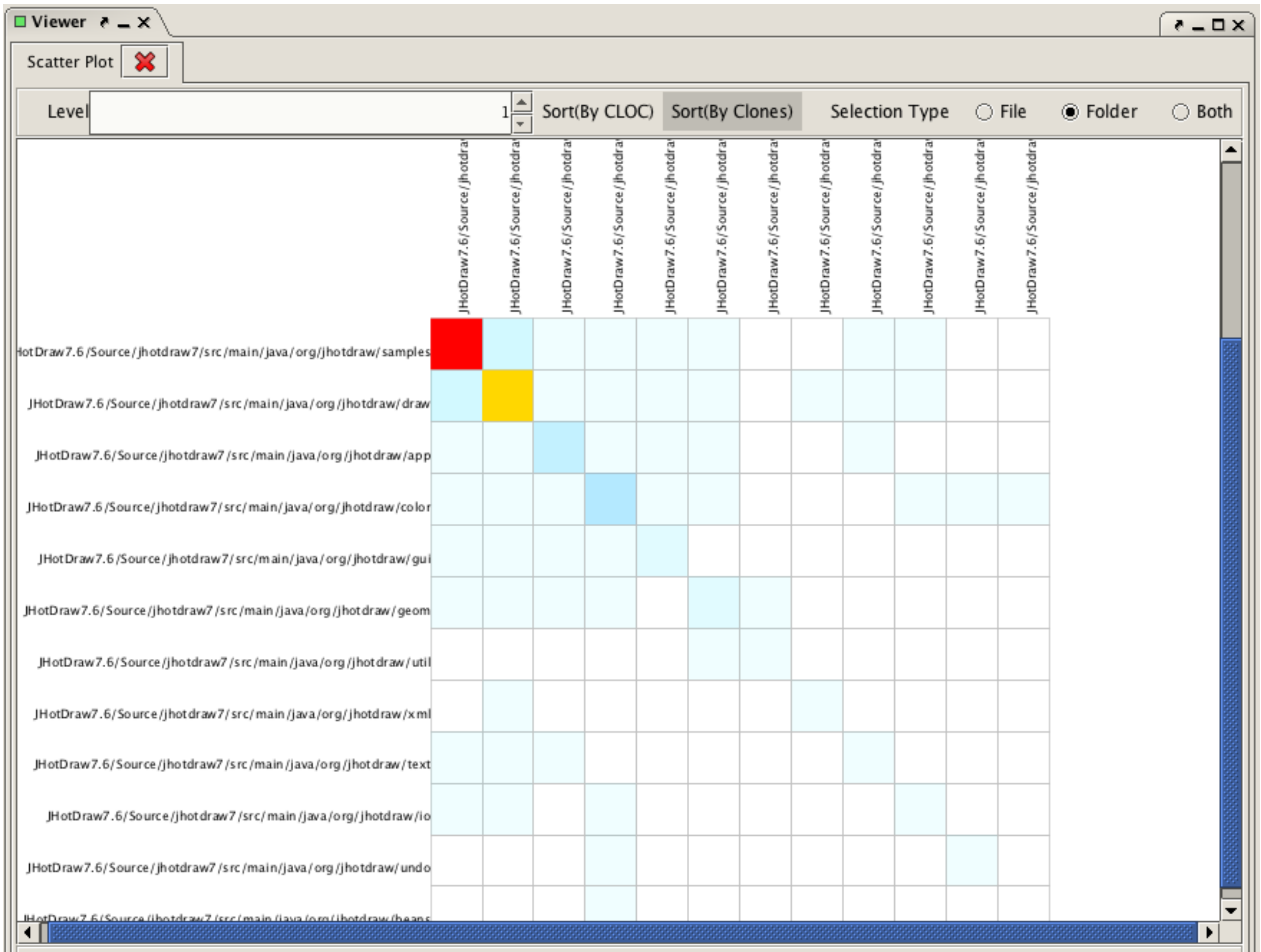



The previous selection opens the above scatter plot on a new tab in the *viewer* view panel (in the middle of the VisCad user interface).

## Options

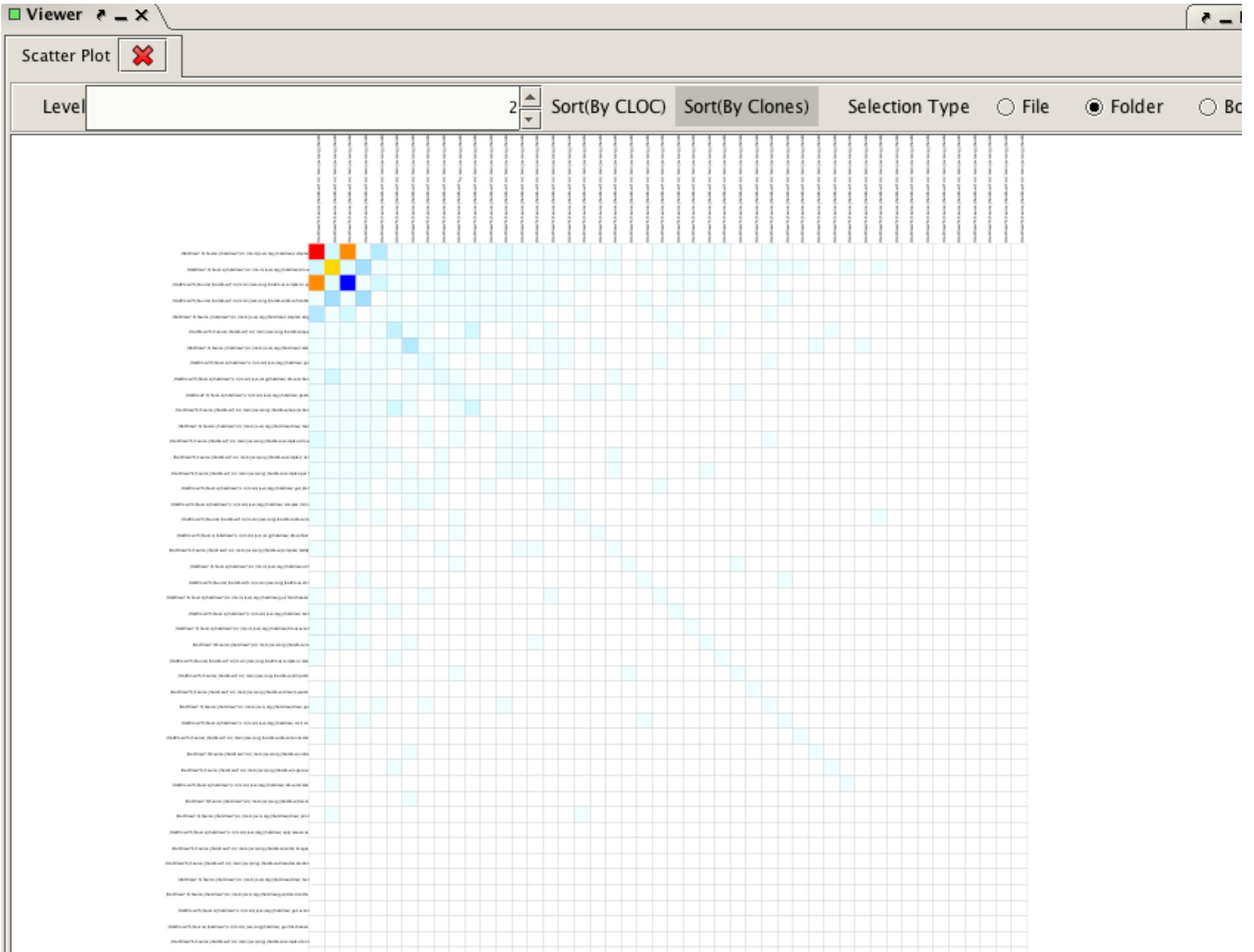


## Example of Sorting




Sorting allows to identify cloning patterns easily.

## Zoom in or zoom out

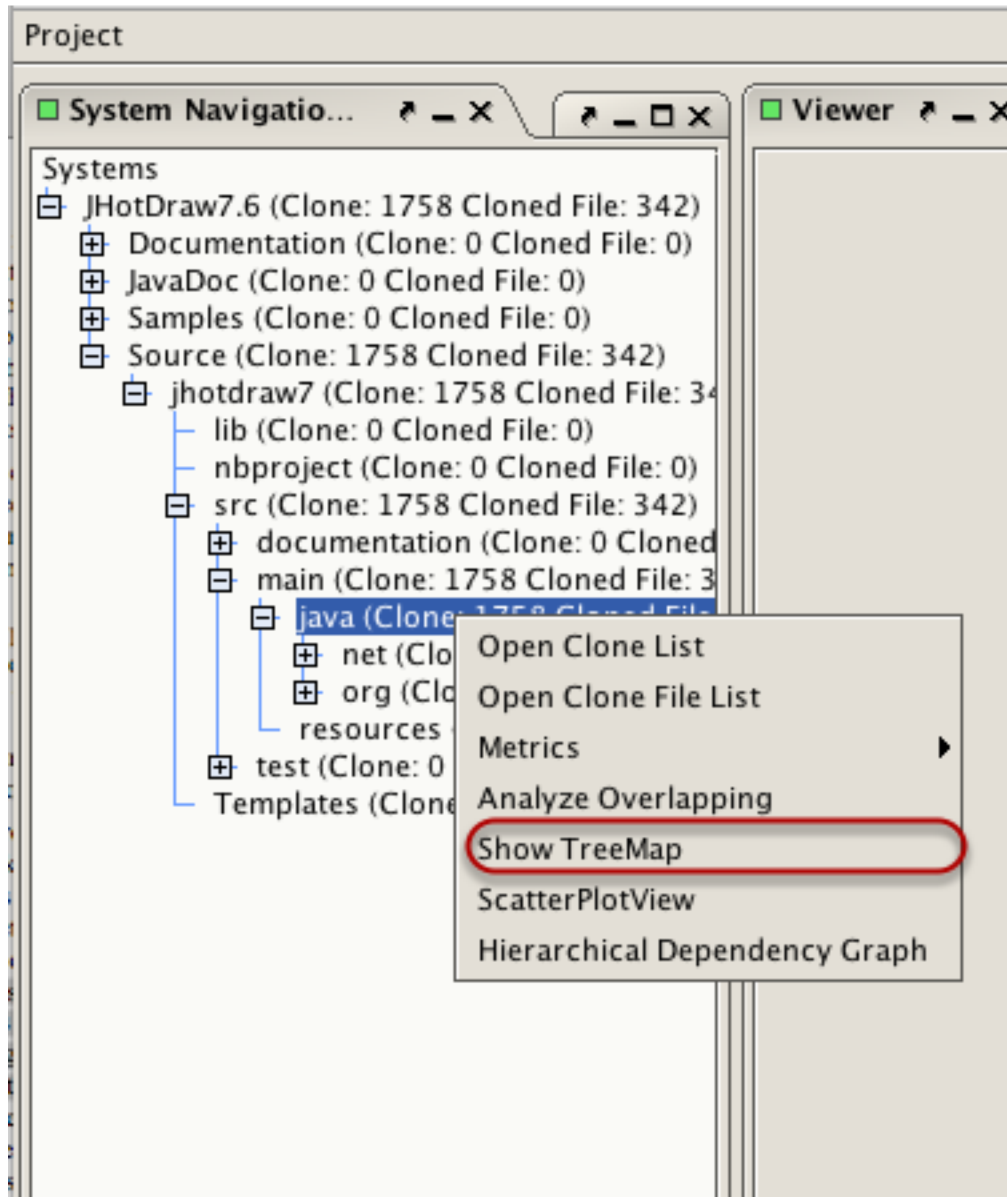



You can hold the right mouse button and move it in the inward or outward direction to perform zoom in or zoom out operations. The above figure shows an example of zoom out operation. You can also identify the files or directories involved with a cell with tooltip by holding the mouse pointer on the cell.

## Treemap

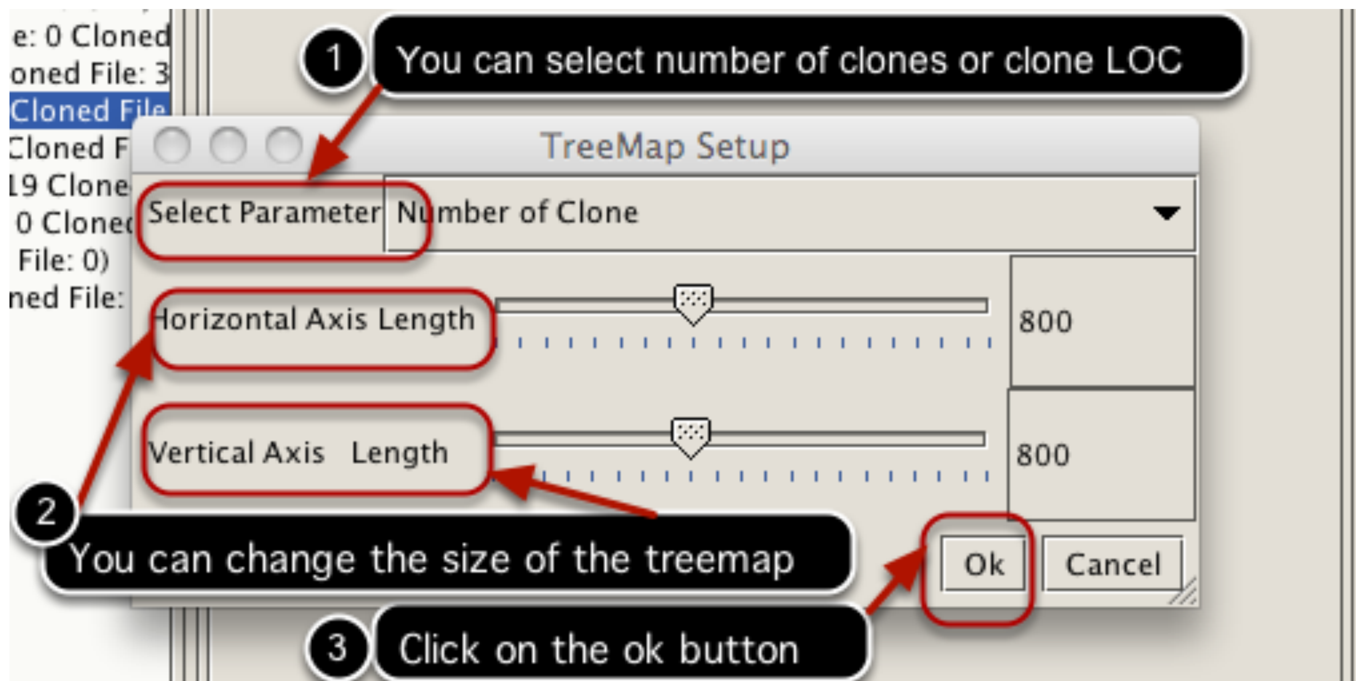
Treemap preserves the hierarchical structure of subject systems where each rectangle represents a file or directory. The rectangles representing the files are aggregated to indicate the cloning status of a directory in the system hierarchy.

### Opening treemap



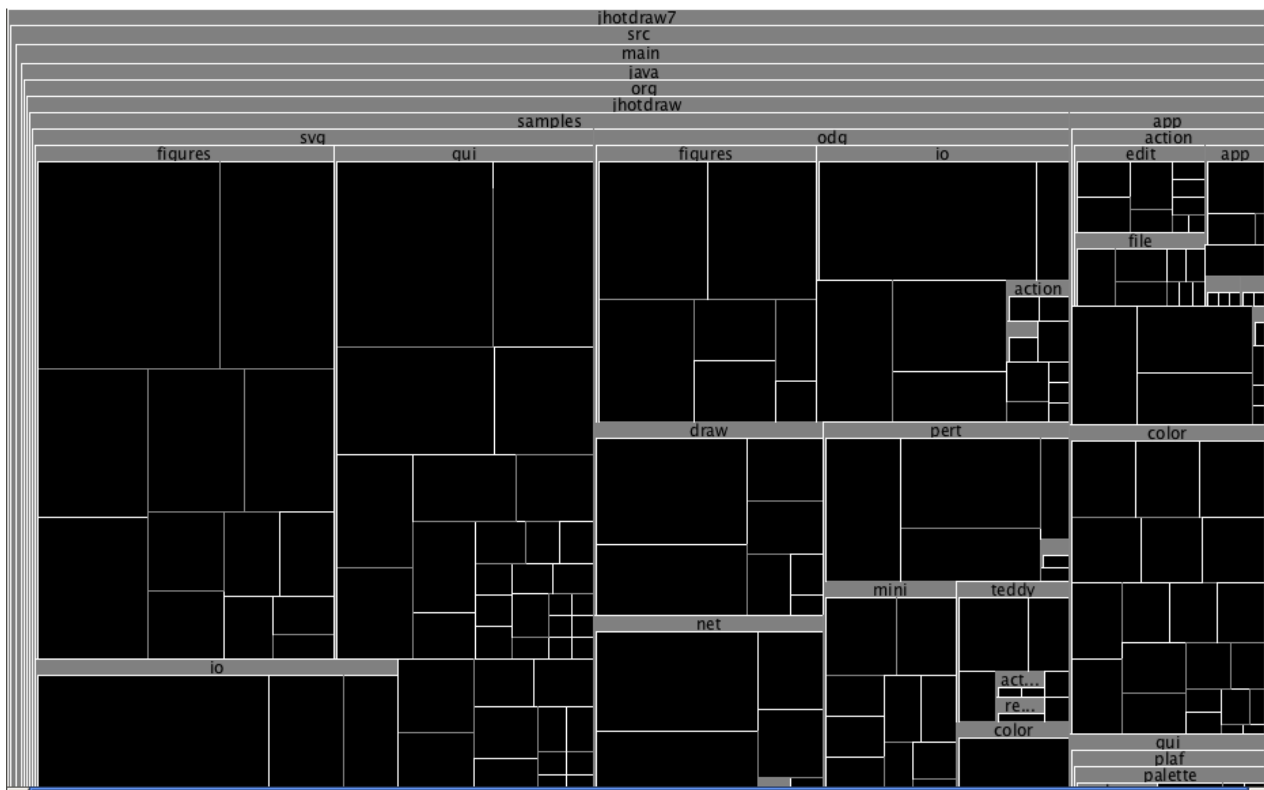

Select a directory and right clink on it to open the popup menu. Click on the *Show TreeMap* menu item.

## Configure the treemap




A dialog box appears that can be used to configure thre treemap. Click on the ok button to open the treemap.

## An Example




An example of a treemap for the JHotDraw system(Version 7.6). Putting the mouse over any rectangle will provide details information about that using a tooltip.

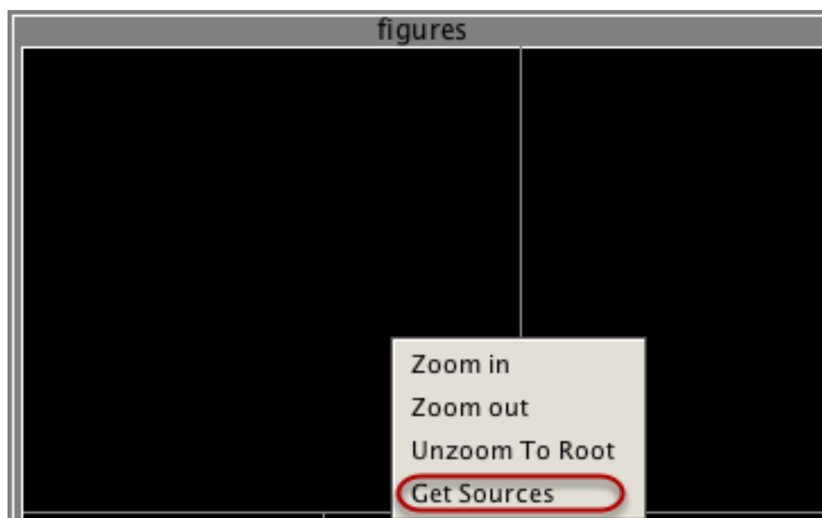
## Zoom in or zoom out

Right click on rectangle representing a directory opens a popup menu. You can then perform zoom in or zoom out operation.

## An example of zoom in operation

An example of zoom in operation on the *samples* directory. We can select the zoom out operation to move one step back or can select the *unzoom to root* option to move back to the beginning state.

## Obtaining source code of the clone fragment




Right click on a rectangle and select the *Get Source* menu item. This will open the clone code fragments of the file(s) in a new tab using the clone code browser. For details about the code browser, see Analyze Clone Fragments.

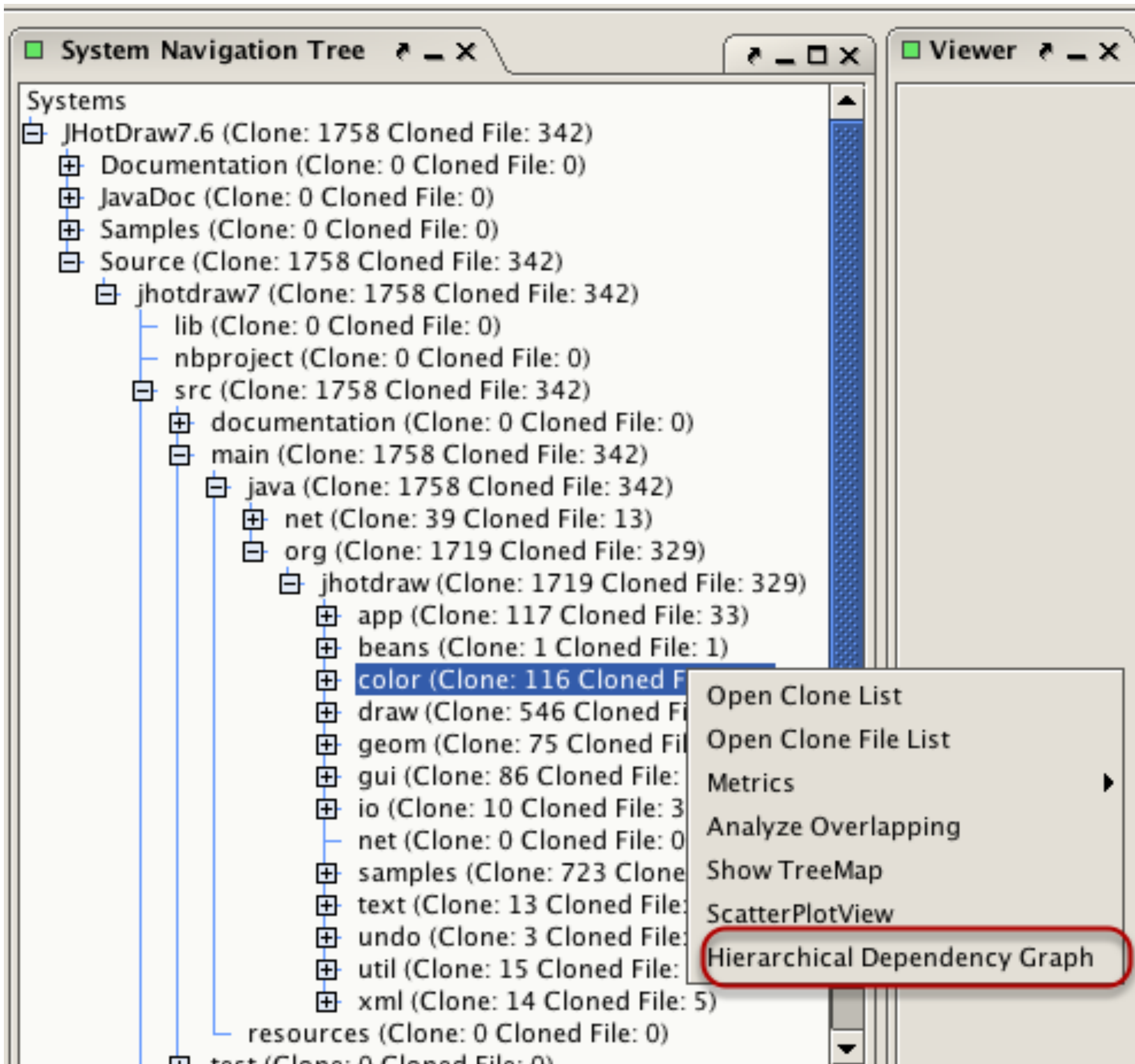
## Obtaining clone distribution information

You can understand how the clone fragments of a clone class are distributed with the help of a treemap. Select a clone class from the bottom-left panel and the files containing those clones will be marked with red color in the treemap.

## Hierarchical Dependency Graph

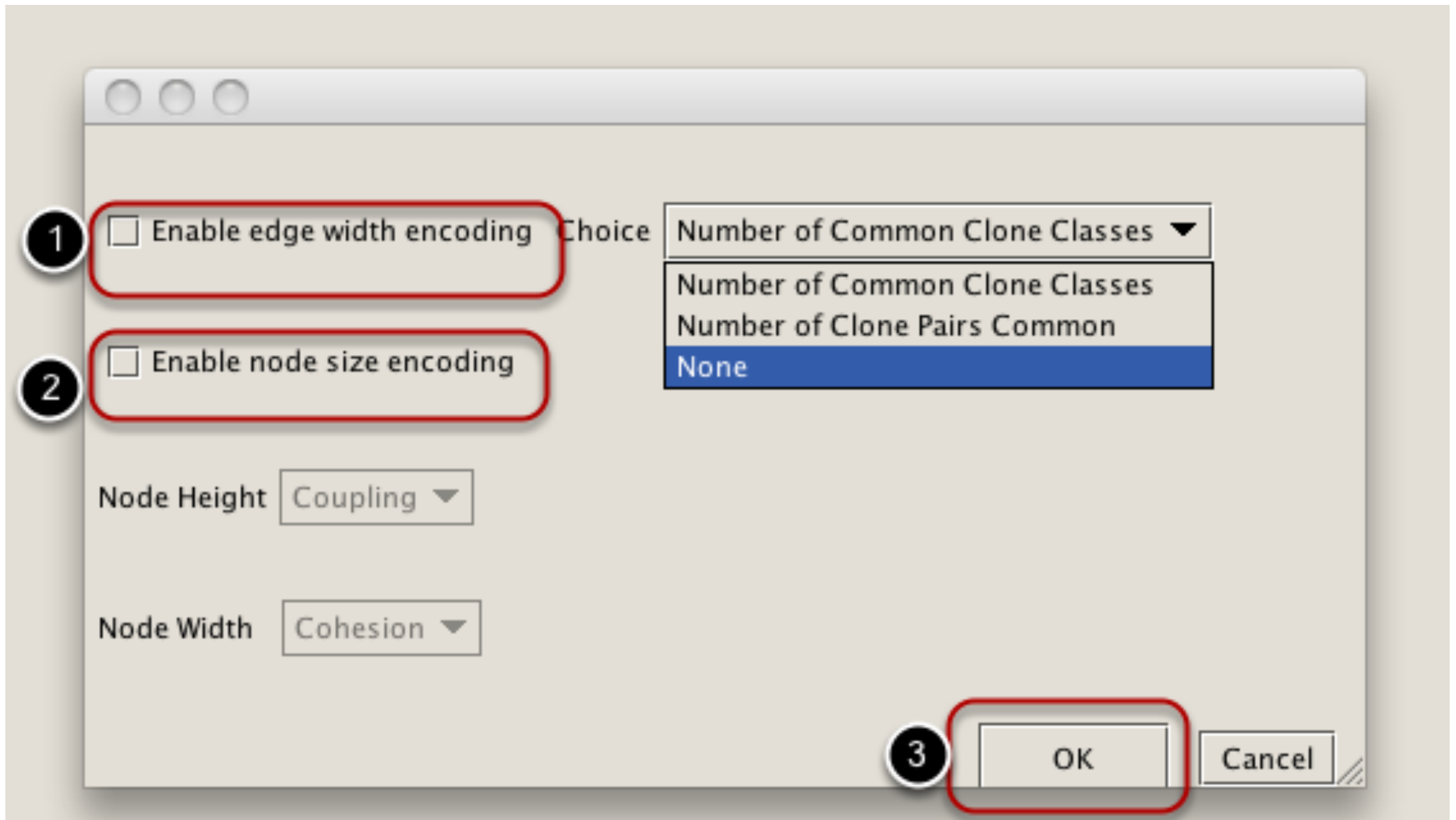
Clones are more problematic when members of a clone class scattered in different parts of a software system because this requires changes need to be made in different parts of the system. Thus, it is required to discover how clone fragments are distributed across subsystems/directories. Moreover, understanding cloning relationships among different subsystems can also reveal their dependencies. VisCad can render the hierarchical organization of a software system along with the distribution of clones using a hierarchical dependency graph.

### Opening the graph



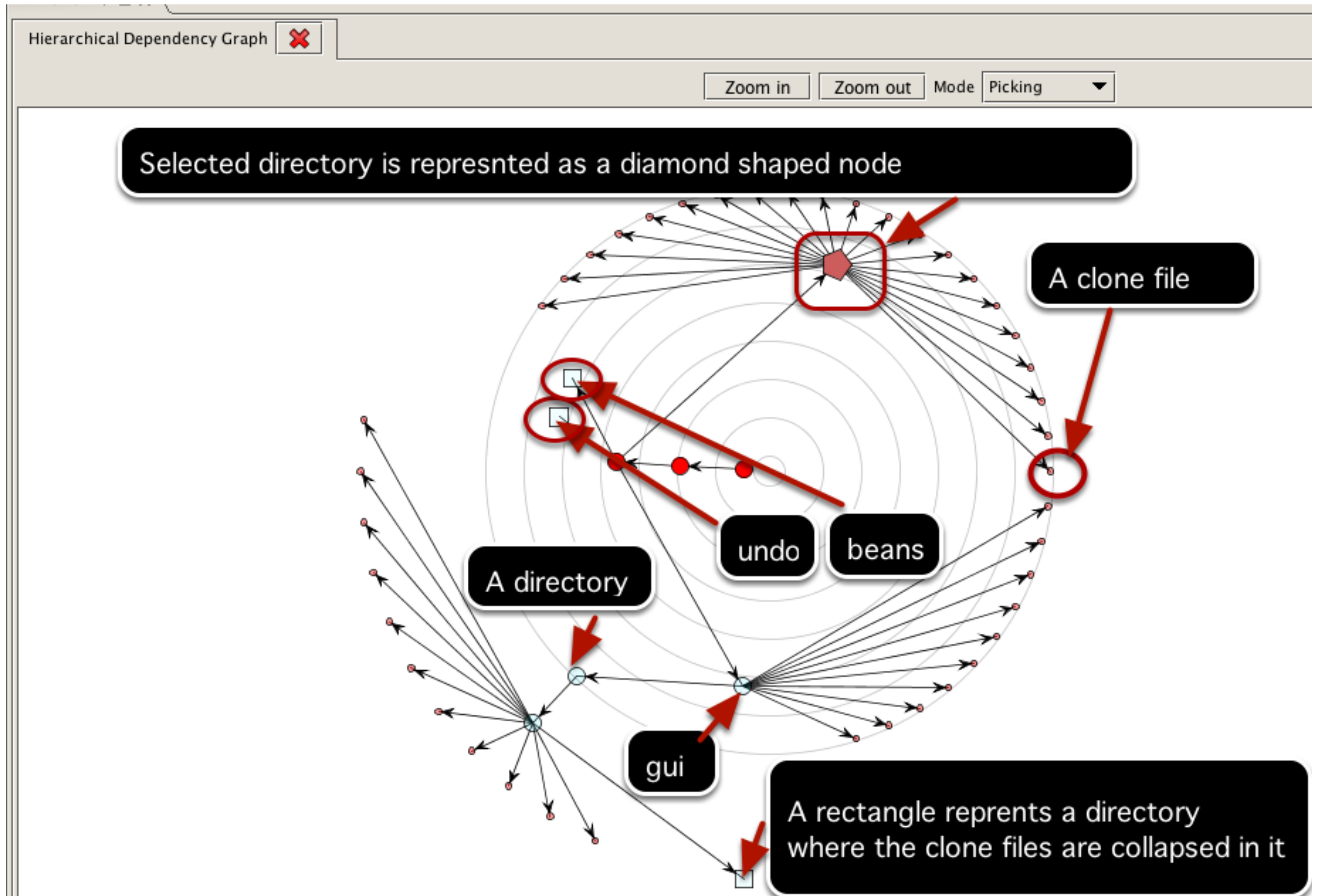

Select a directory from the *system navigation tree* and right click on it to open a popup menu. Click on the *Hierarchical Dependency Graph* menu item.

## Configure the graph




Edge width can represent number clone classes/clone pairs distribued between a pair of directories. Nodes width and height can represent cohesion and coupling value. Click on the ok button when you are done.

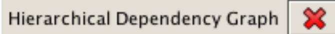
## An example

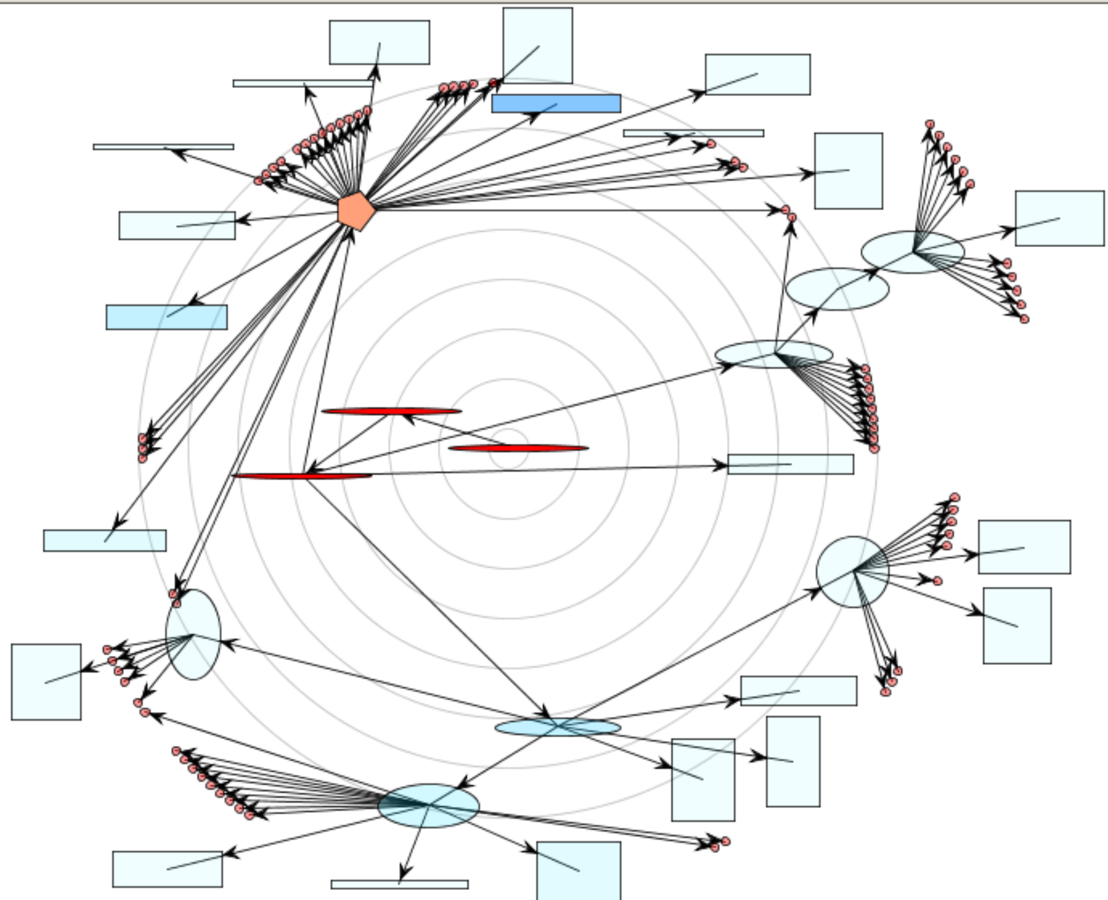



Suppose we want to understand which directories in the system contain the external clones of the color directory. We select the *color* directory from the system navigation tree, open the popup menu and select the *Hierarchical Dependency Graph.*
With the default settings, we get a graph as shown in the above figure. From the figure we can understand that the color directory has external cloning relationship with the *gui*, *undo* and *beans* directories.

VisCad visualizes the graph by filtering those nodes that do not have any cloning relation with the selected directory.

## Another example



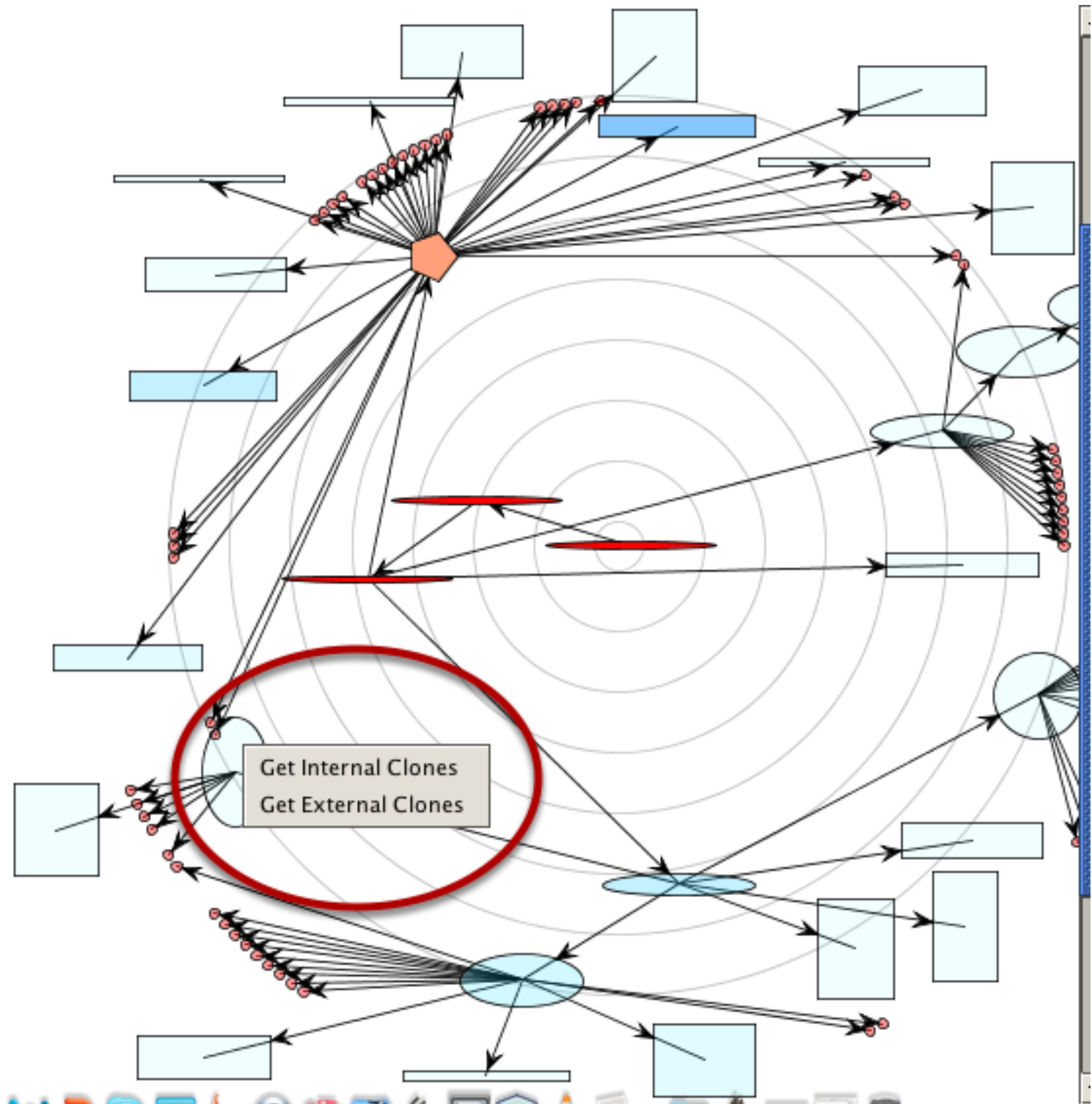




The above figure shows the hierarchical dependency graph for the *jhotdraw/draw* directory. In this case, node's (represnting the directory) width and height represent the clone cohesion and coupling value. Node color value represent the external cloning relation level with the target directory.

## Two mode of interaction

In the *picking* mode, you can select individual node and move them. In the *transforming* mode, you can move the entire graph.

## Obtaining external or internal clone pairs




Right click on a node( represnting a directory) opens a popup menu. You can then select the target operation. The  clone pairs( internal/external) will be viewed in a new tab using the *clone pair browser* component (see the next figure).

## Example of a clone pair browser

Viewer

Hierarchical Dependency Graph | external clone pairs

Clone Pair Browser

External clone pairs for the directory: /Users/muhammad/880Research/Simian/JHotDraw7.6/Source/jhotdraw7/src/main/java/org/jhotdr...

| No | PCID-1 | CCID-1 | Path-1 | SL-1 | EL-1 | CLOC-1 | PCID-2 | CCID-2 | Path-2 | SL-2 | EL-2 | CLOC-2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1,152 | 473 | JHotDra... | 104 | 118 | 15 | 1,154 | 473 | JHotDra... | 105 | 119 | 15 |
| 2 | 1,153 | 473 | JHotDra... | 109 | 123 | 15 | 1,154 | 473 | JHotDra... | 105 | 119 | 15 |
| 3 | 215 | 92 | JHotDra... | 195 | 200 | 6 | 216 | 92 | JHotDra... | 221 | 226 | 6 |
| 4 | 215 | 92 | JHotDra... | 195 | 200 | 6 | 217 | 92 | JHotDra... | 191 | 196 | 6 |
| 5 | 215 | 92 | JHotDra... | 195 | 200 | 6 | 218 | 92 | JHotDra... | 221 | 226 | 6 |
| 6 | 463 | 185 | JHotDra... | 54 | 61 | 8 | 465 | 185 | JHotDra... | 163 | 170 | 8 |
| 7 | 464 | 185 | JHotDra... | 156 | 163 | 8 | 465 | 185 | JHotDra... | 163 | 170 | 8 |
| 8 | 452 | 181 | JHotDra... | 45 | 54 | 10 | 454 | 181 | JHotDra... | 42 | 51 | 10 |
| 9 | 453 | 181 | JHotDra... | 46 | 55 | 10 | 454 | 181 | JHotDra... | 42 | 51 | 10 |
| 10 | 195 | 83 | JHotDra... | 72 | 77 | 6 | 196 | 83 | JHotDra... | 42 | 47 | 6 |

Left> JHotDraw7.6/Source/jhotdraw7/src/main/java/org/jhotdraw/samples/draw/DrawApplet.java

right> JHotDraw7.6/Source/jhotdraw7/src/main/java/org/jhotdraw/samples/pert/PertApplet.java

Differencer

Click on this buton to see the source code differences

```
100        InputStream in = url.openCo
101        try {
102            NanoXMLDOMInput domi =
103            result = (Drawing) domi
104        } finally {
105            in.close();
106        }
107    } else {
108        result = null;
109    }
110    return result;
111 }
```

```
101        InputStream in = url.o
102        try {
103            NanoXMLDOMInput do
104            result = (Drawing)
105        } finally {
106            in.close();
107        }
108    } else {
109        result = null;
110    }
111    return result;
112 }
```

# Code Clone Metrics

## Introduction

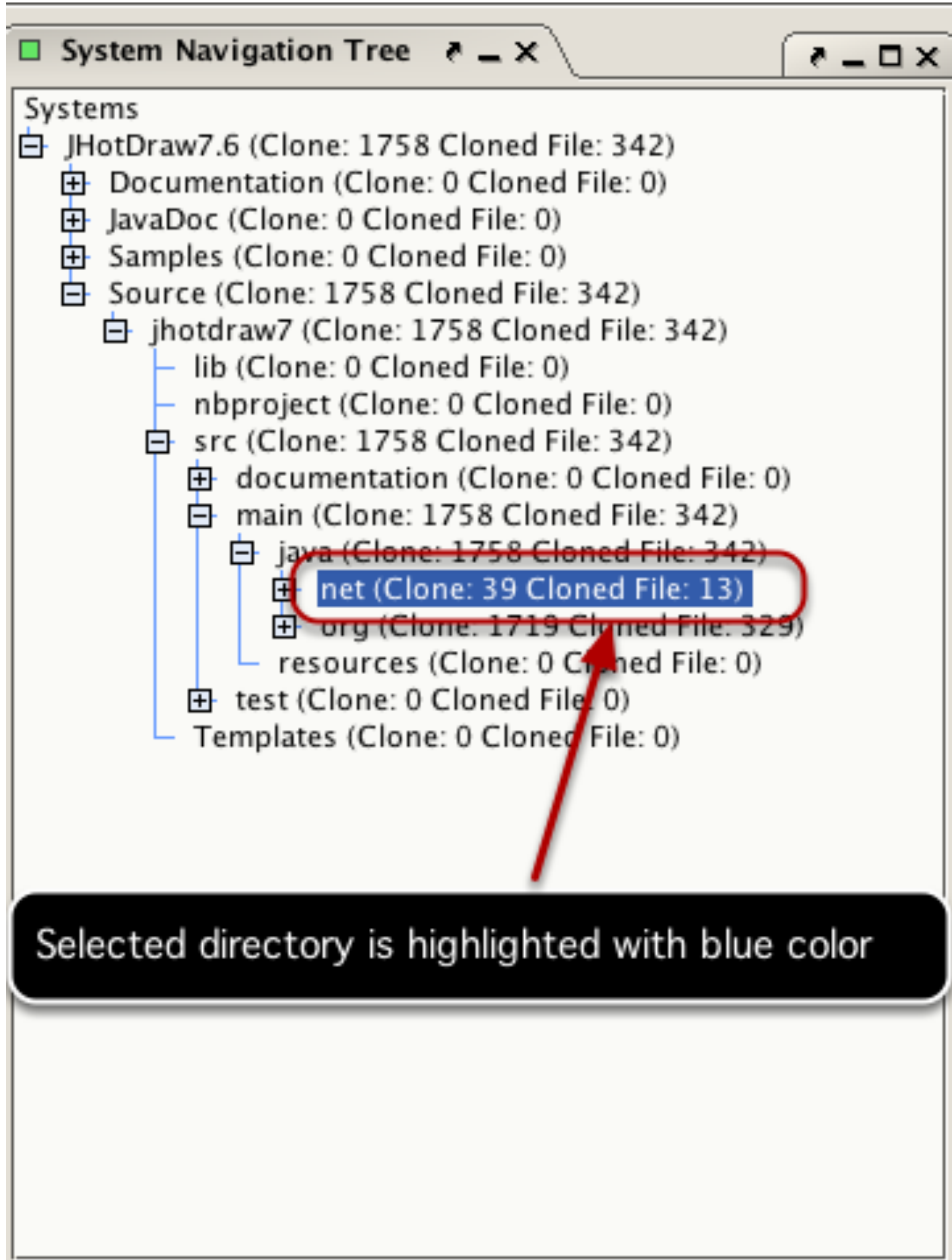
For supporting in-depth clone analysis, VisCad can compute a set of metrics. We can divide the metrics into two broard categories. The first set of metrics (*clone system metric set*) relate clones with the organizational structure of the subject system and can be computed for different system boundaries, such as for the entire system, for subsystems/directories or for source files, as per the user's choice. Depending on the granularity of operation, we can again subdivide them into two groups, the *file metric set* and the *directory metric set*. The next set of metrics (*clone class metrics set*) deals with the clone classes.

VisCad supports four operations for each metric set. These are:
1. Exporting : Results of metric computations can be exported in CSV (comma separated values)format.
2. Plotting : Although metrics are important for quantitative analysis, identifying important patterns from a large set of data is difficult. To avoid such difficulties, users can plot the metrics values with a bar chart which helps in identifying an anomaly within clone patterns easily.
3. Browsing Clone Code: Depending on the metrics values, user may be interested to explore the clone fragments located within a file or directory. VisCad also supports such operation.
4. Sorting : Values can be sorted to locate the maximum or the minimum value easily.

## Obtaining Metrics For Files

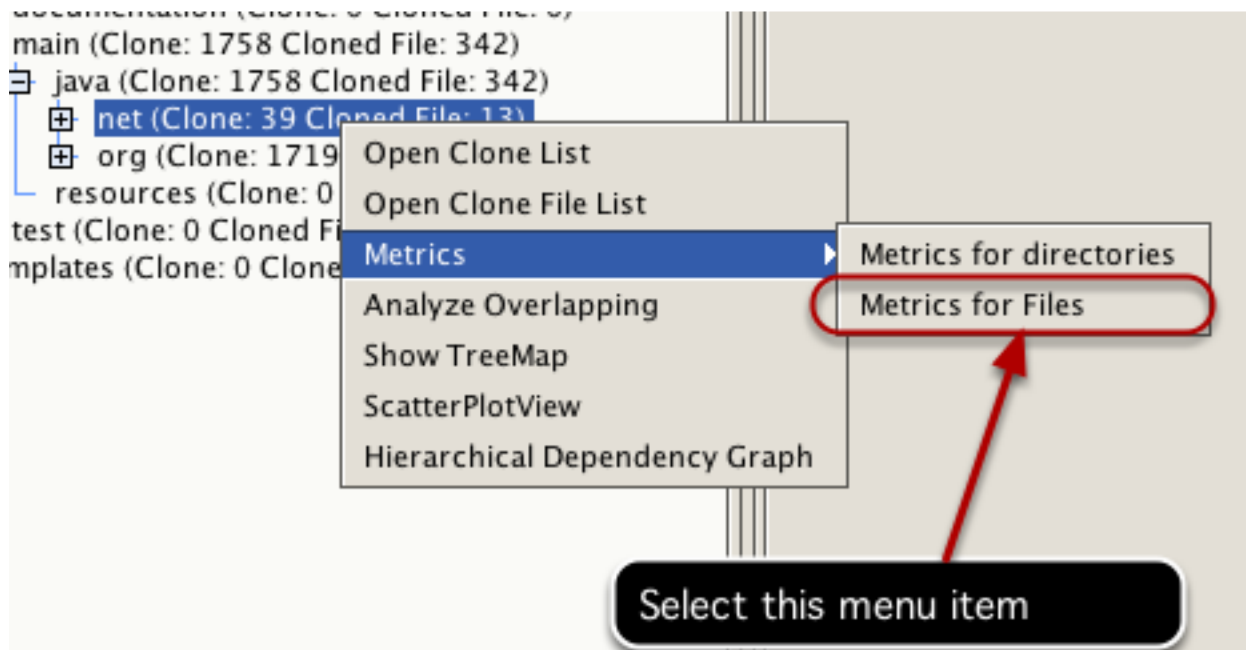
This section discusses the steps for obtaining various metrics values for the clone files located in a directory.

### Select a directory




Click on a target directory to select it from the system navigation tree. Metrics will be computed for all clone files located within this directory

## Open the popup menu and select the operation




Right click on the selected directory to open the popup menu. Select Metrics->Metrics for files from the menu.

## Result

Viewer

File Metric Values

New Tab Panel

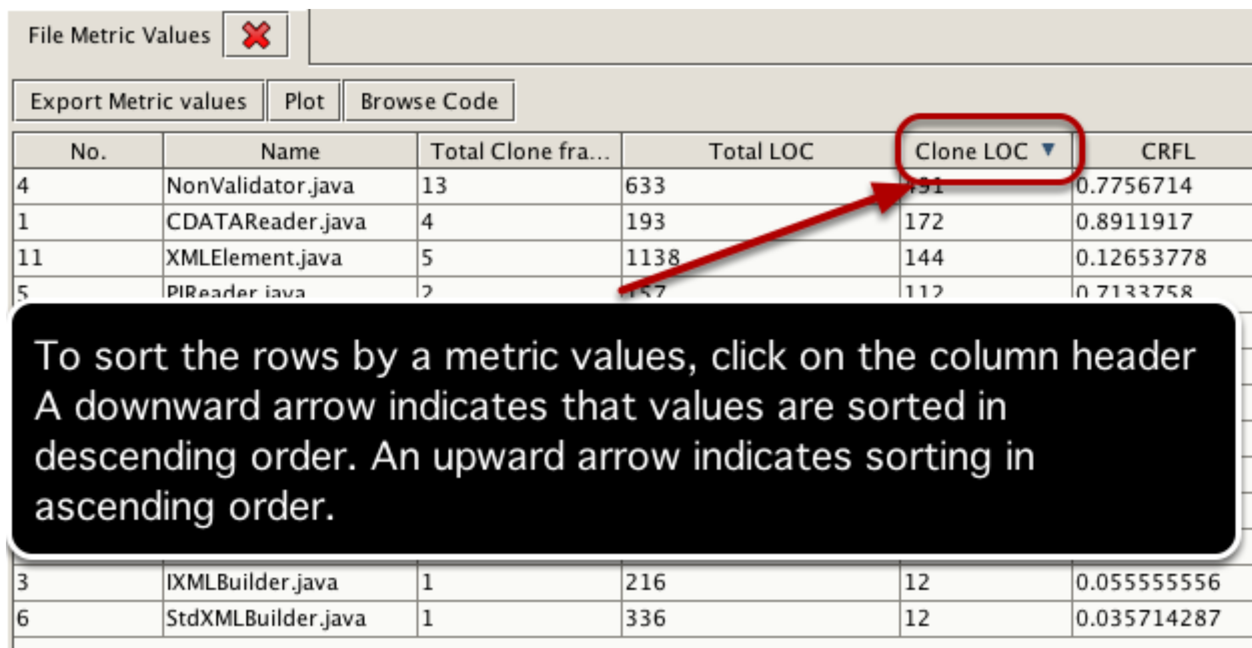
Export Metric values | Plot | Browse Code

Buttons for three operations
1. Exporting 2. Plotting 3. Browsing clone code fragment

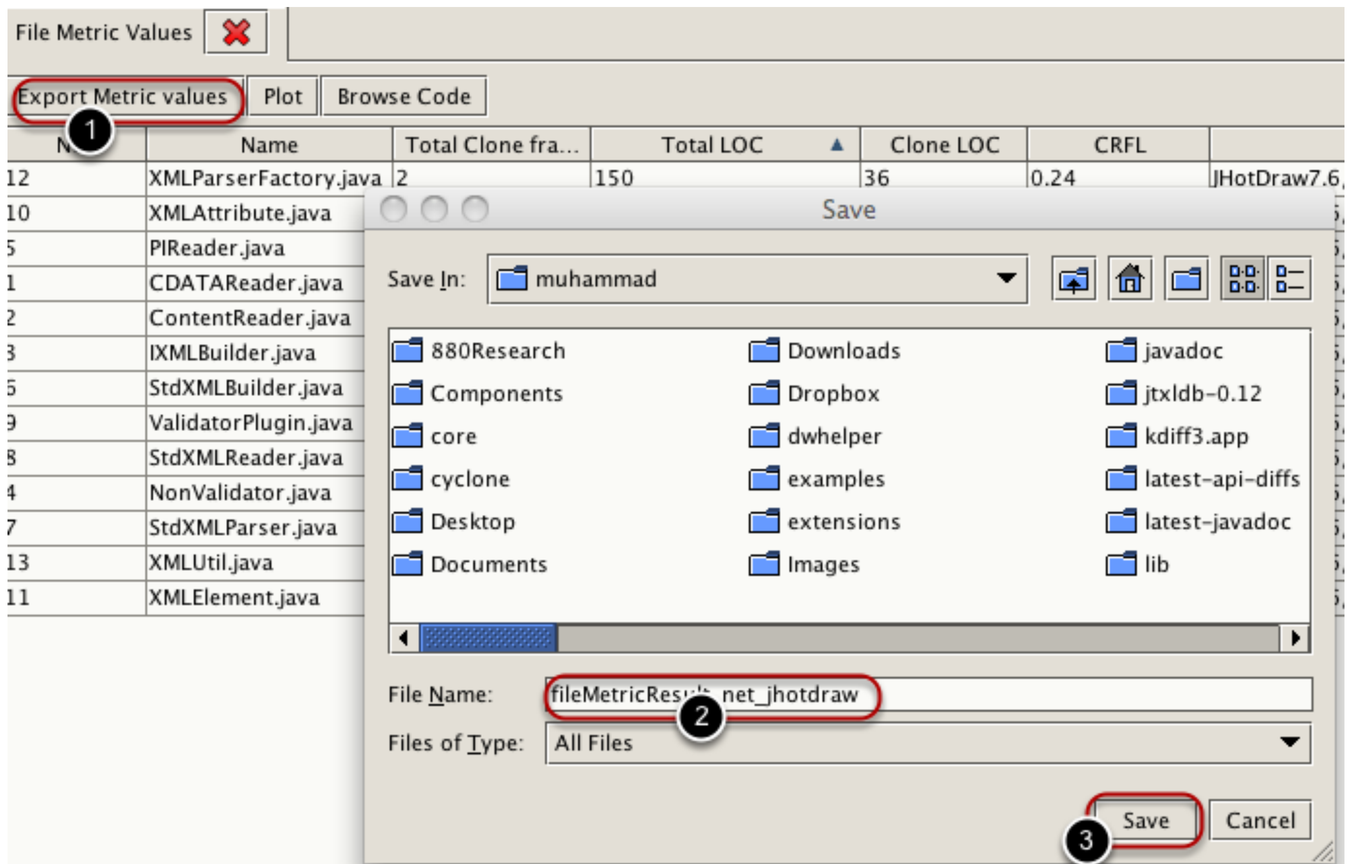
| | Name | Path | Total Clone fragments | Total LOC | Clone LOC | CRFL |
|---|---|---|---|---|---|---|
| 1 | CDATAReader.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/CDATAReader.java | 4 | 193 | 172 | 0.8911917 |
| 2 | ContentReader.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/ContentReader.java | 2 | 212 | 36 | 0.16981132 |
| 3 | [illegible] | [illegible] | | 216 | 12 | 0.055555556 |
| 4 | | | | 633 | 491 | 0.7756714 |
| 5 | | | | 157 | 112 | 0.7133758 |
| 6 | | | | 336 | 12 | 0.035714287 |
| 7 | | | | 696 | 78 | 0.112068966 |
| 8 | | | | 627 | 40 | 0.06379585 |
| 9 | | | | 422 | 14 | 0.033175357 |
| 10 | XMLAttribute.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/XMLAttribute.java | 1 | 151 | 44 | 0.29139072 |
| 11 | XMLElement.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/XMLElement.java | 5 | 1138 | 144 | 0.12653778 |
| 12 | XMLParserFactory.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/XMLParserFactory.java | 2 | 150 | 36 | 0.24 |
| 13 | XMLUtil.java | JHotDraw7.6/Source/jhotdraw7/src/main/java/net/n3/nanoxml/XMLUtil.java | 1 | 763 | 14 | 0.018348623 |

The computed metrics values for all clone files will be displayed as a new tab panel(titled *File metric Values*) in the viewer window.

## Sorting metric values




## Exporting/Saving metric values

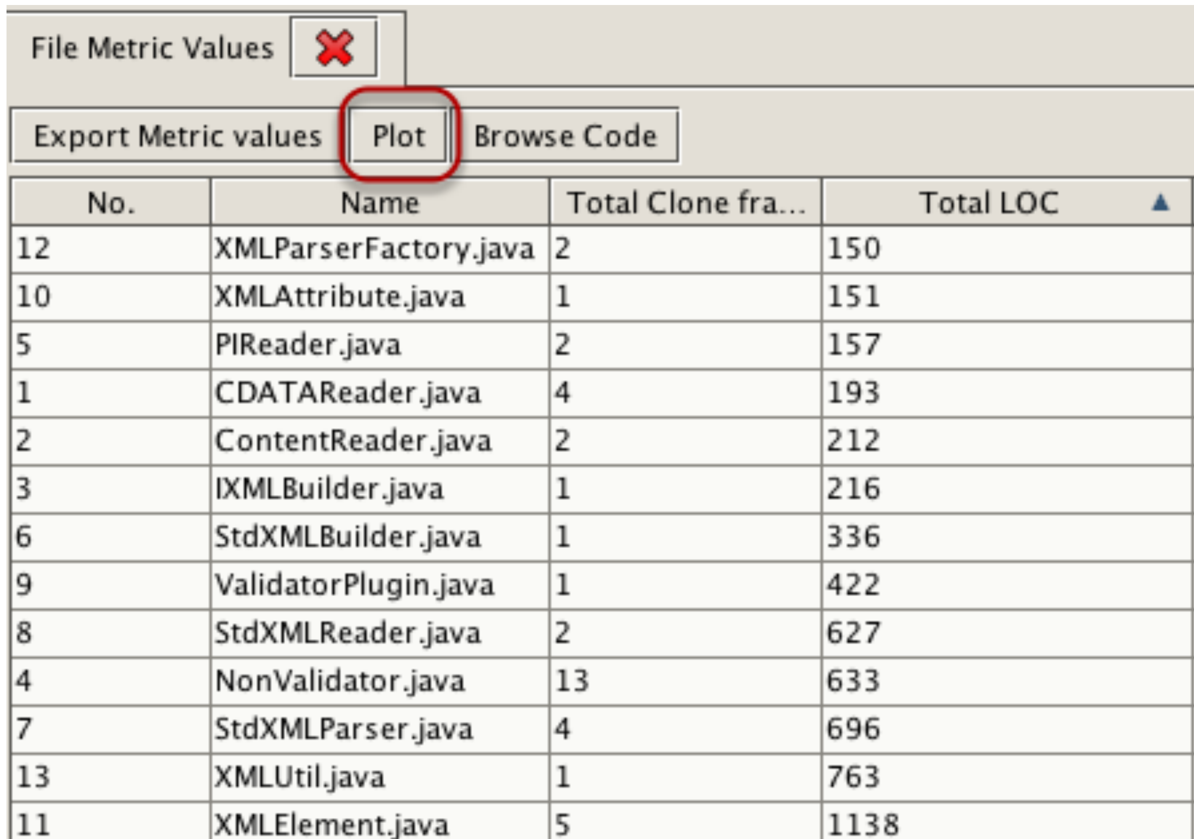



To save the result, follow the following steps:

Step-1: To save the result, click on the *Export Metric Values* button. This will open a dialog box. Select

the folder where you want to save the result.
Step-2: Provide the name of the file you want to save .
Step-3: Click on the Save button to complete the operation.

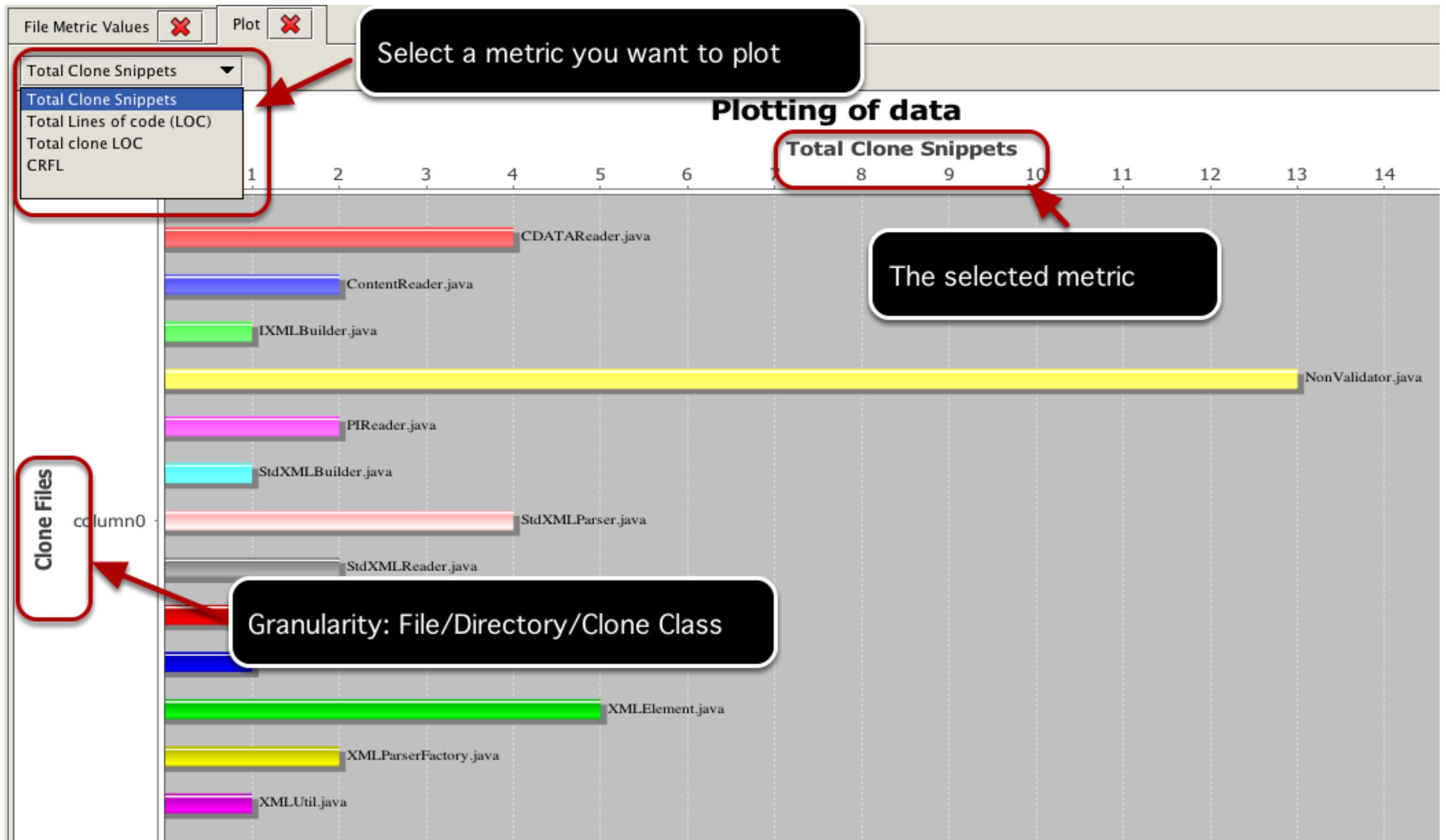
## Plotting metric values




| No. | Name | Total Clone fra... | Total LOC |
|---|---|---|---|
| 12 | XMLParserFactory.java | 2 | 150 |
| 10 | XMLAttribute.java | 1 | 151 |
| 5 | PIReader.java | 2 | 157 |
| 1 | CDATAReader.java | 4 | 193 |
| 2 | ContentReader.java | 2 | 212 |
| 3 | IXMLBuilder.java | 1 | 216 |
| 6 | StdXMLBuilder.java | 1 | 336 |
| 9 | ValidatorPlugin.java | 1 | 422 |
| 8 | StdXMLReader.java | 2 | 627 |
| 4 | NonValidator.java | 13 | 633 |
| 7 | StdXMLParser.java | 4 | 696 |
| 13 | XMLUtil.java | 1 | 763 |
| 11 | XMLElement.java | 5 | 1138 |

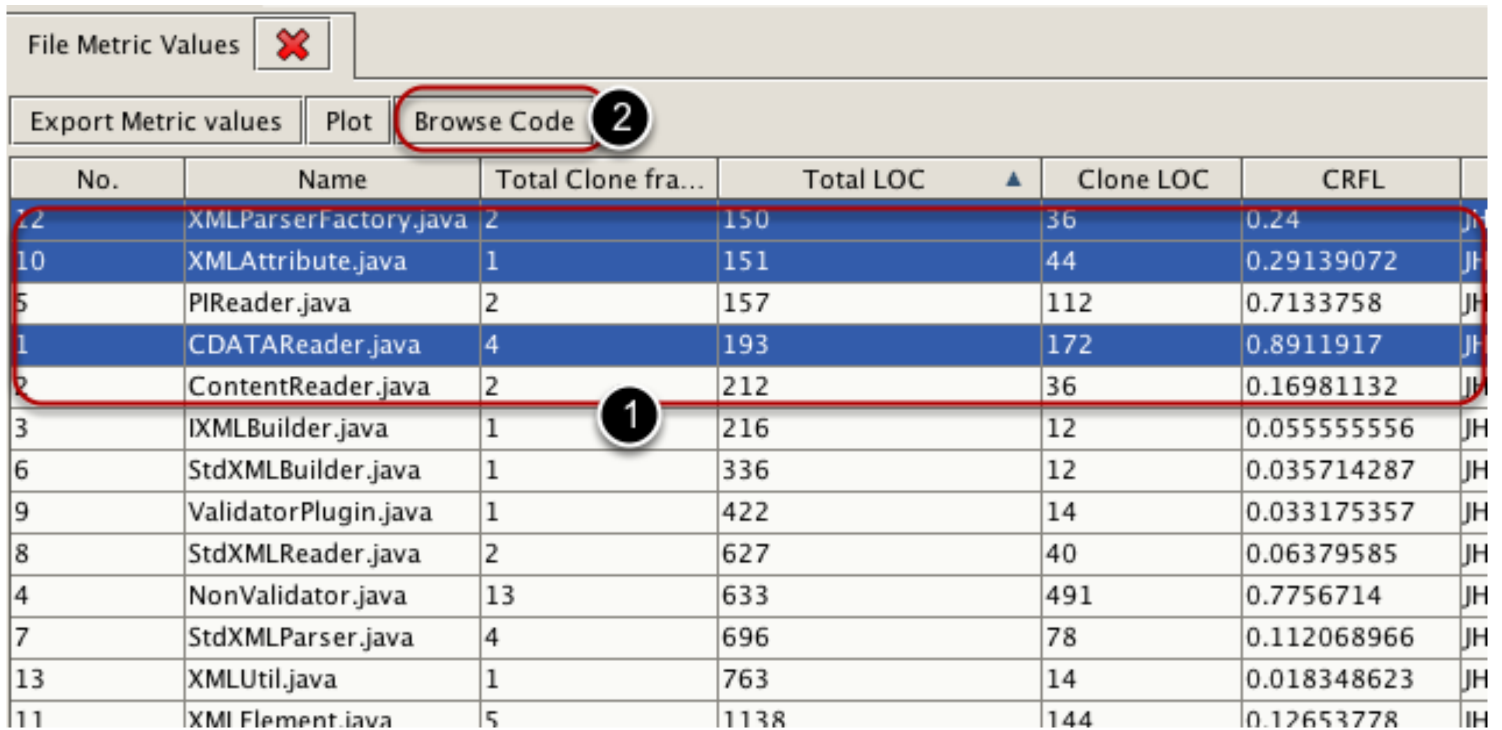
To plot the metric values, click the on the *Plot* button.

## Plotting metric values (Continued)




You can change the selection to plot different metric values. Putting the cursor over a bar provides the detail path information.

## Browsing clone code fragments

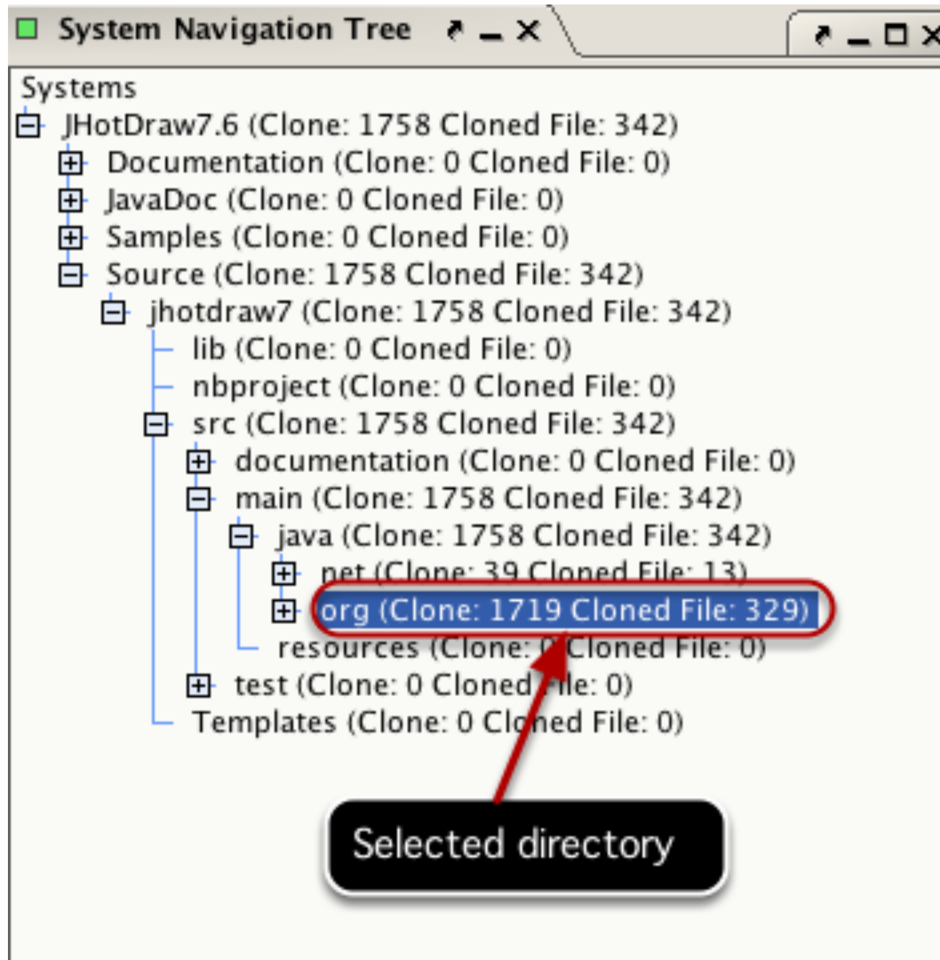



Step-1: Select one or more rows by holding the command key in MAC or control key in Windows.
Step-2: Now click on the *Browse Code* Button to analyze the clone code fragments in the selected file(s) with the *source code browser*.

## Obtaining Metrics For Directories

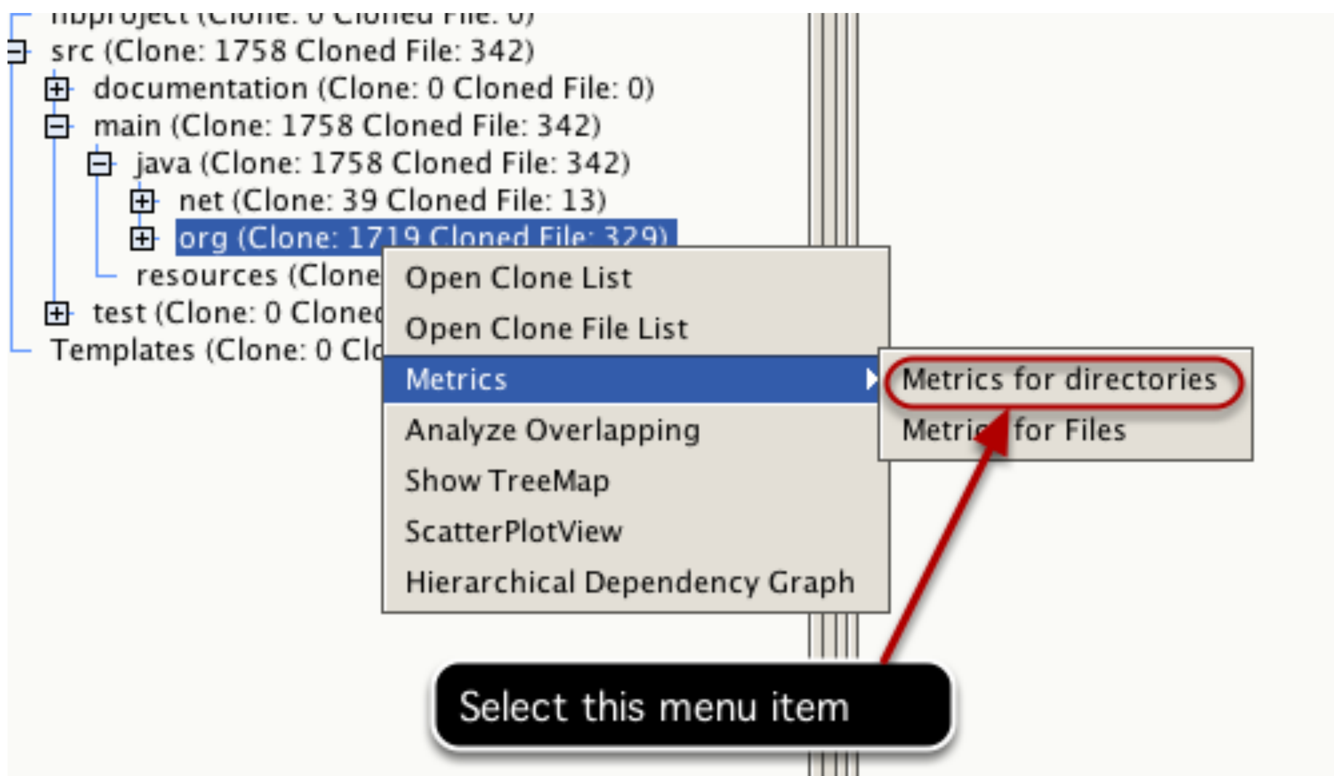
This section discusses the steps for obtaining various metrics values for all clone directories within a selected directory. A clone directory is a directory that contains at least one clone fragment.

### Select a directory




Click on a target directory to select it from the system navigation tree. Metrics will be computed for all clone directories within it.

## Open the popup menu and select the operation




Right click on the selected directory to open the popup menu. Select *Metrics->Metrics for directories* from the menu.

## Result

Viewer

Directory Metric Values

New tab panel

Click on any column header to sort the rows with t...

Export Metric Values | Plot | Browse Code

1. Exporting, 2. plotting metric values and 3. browsing source code of the clone fragments are suppoted by these buttons

| ame | | one | TLOC | TLOC(Clone... | TCLOC | PTCLOC | TF | FAWC | PFAWC | Avg. CRFL | CCH | CCP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| jh...raw | JHotDraw7.6/... | 719 | 126112 | 91693 | 152470 | 27.6754 | 655 | 329 | 50.229008 | 2.0089977 | 1.0 | 0.0 |
| app | JHotDraw7.6/... | 11 | 10705 | 6942 | 4478 | 16.777206 | 73 | 33 | 45.20548 | 0.78955793 | 1.0 | 0.0 |
| action | JHotDraw7.6/... | 70 | 4988 | 3405 | 2624 | 18.20369 | 52 | 26 | 50.0 | 0.8784873 | 0.98214287 | 0.017857144 |
| app | JHotDraw7.6/... | 16 | 843 | 573 | 500 | 25.622776 | 8 | 3 | 37.5 | 0.83936954 | 0.5694444 | 0.43055558 |
| edit | JHotDraw7.6/... | 24 | 1035 | 882 | 1303 | 31.304348 | 12 | 9 | 75.0 | 1.5976193 | 1.0 | 0.0 |
| file | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 49664423 | 0.6923077 | 0.30769232 |
| view | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 1186858 | 1.0 | 0.0 |
| window | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 3940363 | 0.75 | 0.25 |
| osx | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 15339233 | 1.0 | 0.0 |
| beans | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 112 | 0.5 | 0.5 |
| color | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 5891644 | 0.9066667 | 0.09333334 |
| draw | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 1725649 | 0.880657 | 0.119343 |
| action | JHotDraw7.6/... | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | .6736855 | 0.8648649 | 0.13513513 |
| connector | JHotDraw7.6/... | 3 | 970 | 182 | 56 | 2.8865979 | 11 | 2 | 18.181818 | 0.30147058 | 0.75 | 0.25 |
| event | JHotDraw7.6/... | 10 | 2118 | 392 | 284 | 6.704438 | 26 | 4 | 15.384615 | 0.7462152 | 0.8333333 | 0.16666667 |
| gui | JHotDraw7.6/... | 5 | 972 | 862 | 5648 | 68.6214 | 7 | 5 | 71.42857 | 5.462257 | 0.48333335 | 0.51666665 |
| handle | JHotDraw7.6/... | 153 | 5838 | 4843 | 5694 | 36.86194 | 26 | 19 | 73.07692 | 1.0247235 | 0.91292524 | 0.08707483 |
| io | JHotDraw7.6/... | 20 | 1449 | 1210 | 1034 | 24.292616 | 8 | 5 | 62.5 | 0.87536085 | 0.71833336 | 0.2816667 |
| layouter | JHotDraw7.6/... | 6 | 445 | 217 | 98 | 11.011236 | 6 | 2 | 33.333332 | 0.46978492 | 1.0 | 0.0 |
| liner | JHotDraw7.6/... | 20 | 713 | 636 | 853 | 53.997196 | 5 | 3 | 60.0 | 1.3474804 | 1.0 | 0.0 |
| locator | JHotDraw7.6/... | 2 | 871 | 239 | 62 | 3.5591273 | 8 | 1 | 12.5 | 0.25941423 | 1.0 | 0.0 |
| print | JHotDraw7.6/... | 1 | 170 | 149 | 34 | 10.0 | 2 | 1 | 50.0 | 0.22818792 | 0.5 | 0.5 |
| tool | JHotDraw7.6/... | 47 | 4450 | 3421 | 1937 | 21.146067 | 20 | 13 | 65.0 | 0.654375 | 0.87820506 | 0.12179487 |
| geom | JHotDraw7.6/... | 75 | 5943 | 4553 | 2799 | 22.37927 | 13 | 8 | 61.53846 | 0.6186949 | 0.9722222 | 0.027777778 |
| gui | JHotDraw7.6/... | 86 | 16080 | 8882 | 9229 | 12.978856 | 89 | 30 | 33.707867 | 1.4307154 | 0.8474638 | 0.15253624 |
| fontchooser | JHotDraw7.6/... | 13 | 1540 | 879 | 614 | 16.558441 | 7 | 4 | 57.142857 | 0.69577664 | 0.9166667 | 0.083333336 |

The computed metrics values for all clone directories will be displayed as a new tab panel(titled *Directory Metric Values*) in the viewer window.

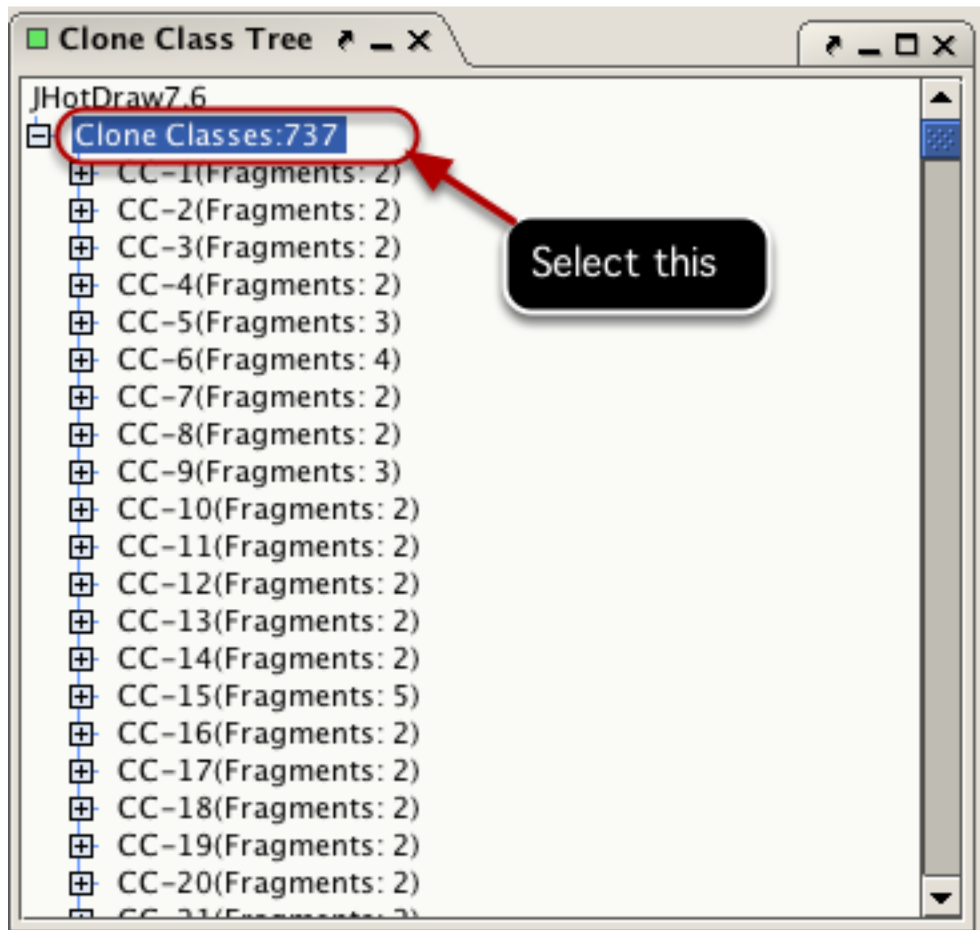
## Other operations

Various operations such as sorting, exporting and plotting computed metrics values, and browsing source code of the clone fragments are also supported. For details, see those operations in the discussion for files (Obtaining Metrics For Files).

## Obtaining Metrics For Clone Classes

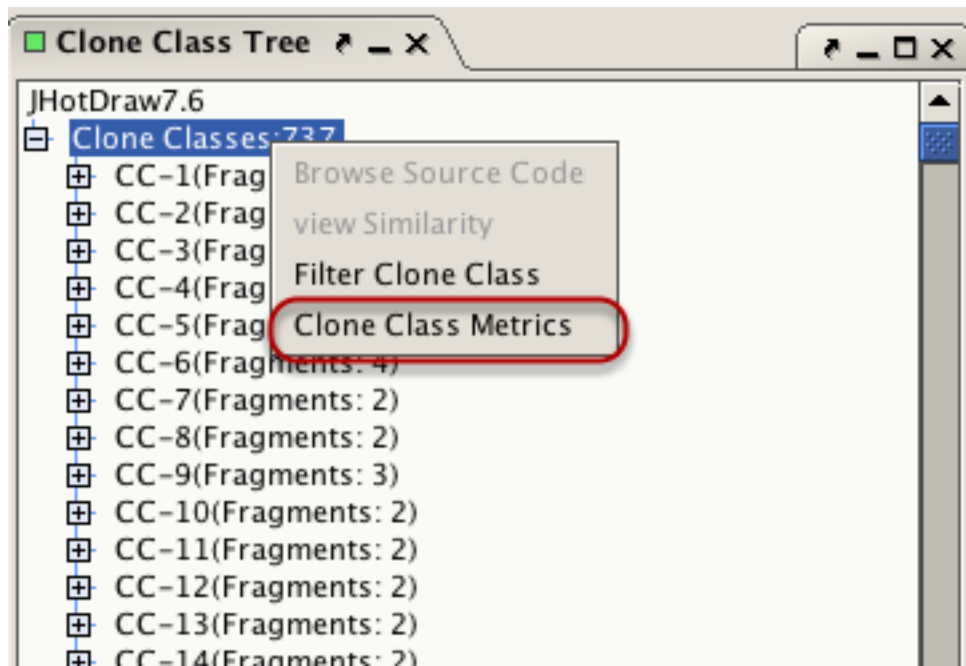
This section discusses the steps for obtaining various metrics values for the clone classes.

### Make the selection




Click on the root node of the clone classes to select it from the clone class tree. Metrics will be computed for all clone classes within it.

## Open the popup menu and select the operation




Right click on the selected node to open the popup menu. Select *Clone Class metrics* menu item from the popup menu.

## Result

Viewer

Clone Class Metric Details

New tab panel

To sort the rows, click on a column header

Export Values | Plot | Browse Code

1. Exporting, 2. Sorting metrics values and 3. browsing the source code of the clone fragments within the selected clone classes are also possible.

| CCID | CCS | CCL | Max Length | Min length | Avg. Length | Files Associated |
|---|---|---|---|---|---|---|
| 1 | 2 | 18 | 9 | 9 | 9.0 | 1 |
| 2 | 2 | 12 | 6 | 6 | 6.0 | 1 |
| 3 | [illegible] | 12 | 6 | 6 | 6.0 | 2 |
| 4 | 2 | 14 | 7 | 7 | 7.0 | 2 |
| 5 | 3 | 18 | 6 | 6 | 6.0 | 2 |
| 6 | 4 | 29 | 10 | 6 | 7.25 | 4 |
| 7 | | | | | | |
| 8 | | | | | | |
| 9 | | | | | | |
| 10 | | | | | | |
| 11 | | | | | | |
| 12 | | | | | | |
| 13 | | | | | | |
| 14 | 2 | 12 | 6 | 6 | 6.0 | 2 |
| 15 | 5 | 30 | 6 | 6 | 6.0 | 4 |
| 16 | 2 | 42 | 32 | 10 | 21.0 | 2 |
| 17 | 2 | 12 | 6 | 6 | 6.0 | 1 |
| 18 | 2 | 16 | 8 | 8 | 8.0 | 1 |
| 19 | 2 | 12 | 6 | 6 | 6.0 | 1 |
| 20 | 2 | 12 | 6 | 6 | 6.0 | 2 |
| 21 | 2 | 18 | 9 | 9 | 9.0 | 2 |
| 22 | 2 | 12 | 6 | 6 | 6.0 | 1 |
| 23 | 2 | 12 | 6 | 6 | 6.0 | 1 |

The computed metrics values for all clone classes will be displayed as a new tab panel(titled *Clone Class Metric Details*) in the viewer window.

## Other operations

Various operations such as sorting, exporting and plotting computed metrics values, and browsing source code of the clone fragments are also supported. For details, see those operations in the discussion for files (Obtaining Metrics For Files).

# Filtering

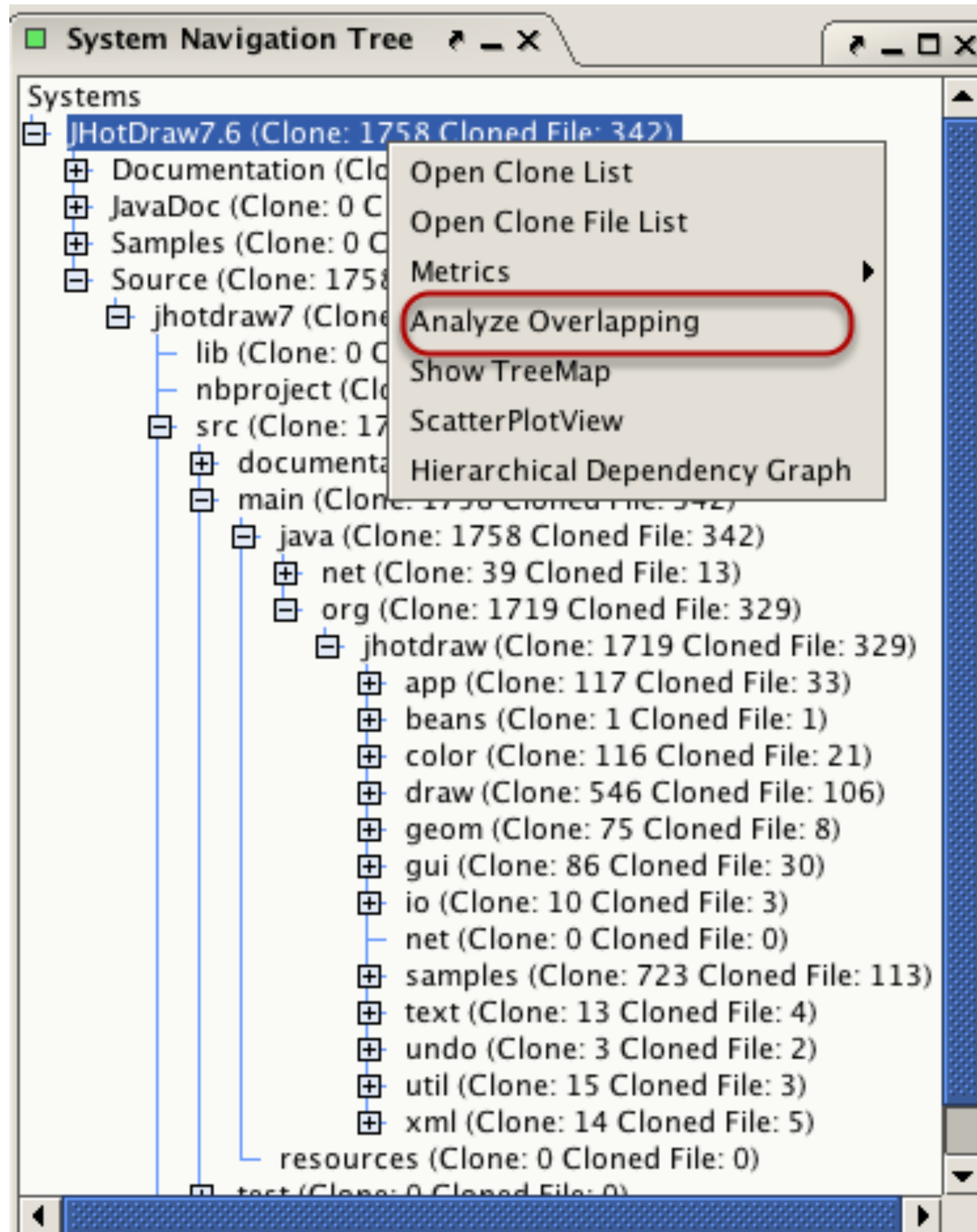
## Introduction

The first and foremost challenge in clone analysis is the large volume of clone detection results. Not all clones are useful to the user and the objective of the analysis at hand governs the set of the useful clones. Here, the term ‘useful clones’ refers to those clone fragments that the maintenance engineers are looking for or are interested in. For example, when the objective is to analyze the inter-project clones, users may be more interested in the clone classes whose fragments are distributed across different projects and these clone fragments form the set of useful clones. In that case, we can filter out the clone pairs that are not distributed across different projects. VisCad supports a set of filtering operations to remove clones that are not useful/interesting to the users.

## Overlapping Clone filterng

Clones may overlapped each other and removing those overlapping clones also reduces the size of the result set.

### Analyze overlapping




Click on a directory to selct it. Right click on it to open the popup menu and select *Analyze Overlapping* menu item.

## Overlapping Clone pairs list

Overlapping

Filter Overlapped Clones

```
141            bounds = getOwner().get(TRANSFORM).createTransformedShape(bounds);
142        }
143        bounds = view.getDrawingToViewTransform().createTransformedShape(bounds);
144        Stroke stroke1;
145        Color strokeColor1;
146        Stroke stroke2;
147
148
149
150            stroke1 = (Stroke) getEditor().getHandleAttribute(stroke1Enabled);
151            strokeColor1 = (Color) getEditor().getHandleAttribute(strokeColor1Enabled);
152            stroke2 = (Stroke) getEditor().getHandleAttribute(stroke2Enabled);
153            strokeC                                              olor2Enabled);
154        } else {
155            stroke                                               ed);
156            stroke                                               or1Disabled);
157            stroke2 = (Stroke) getEditor().getHandleAttribute(stroke2Disabled);
```

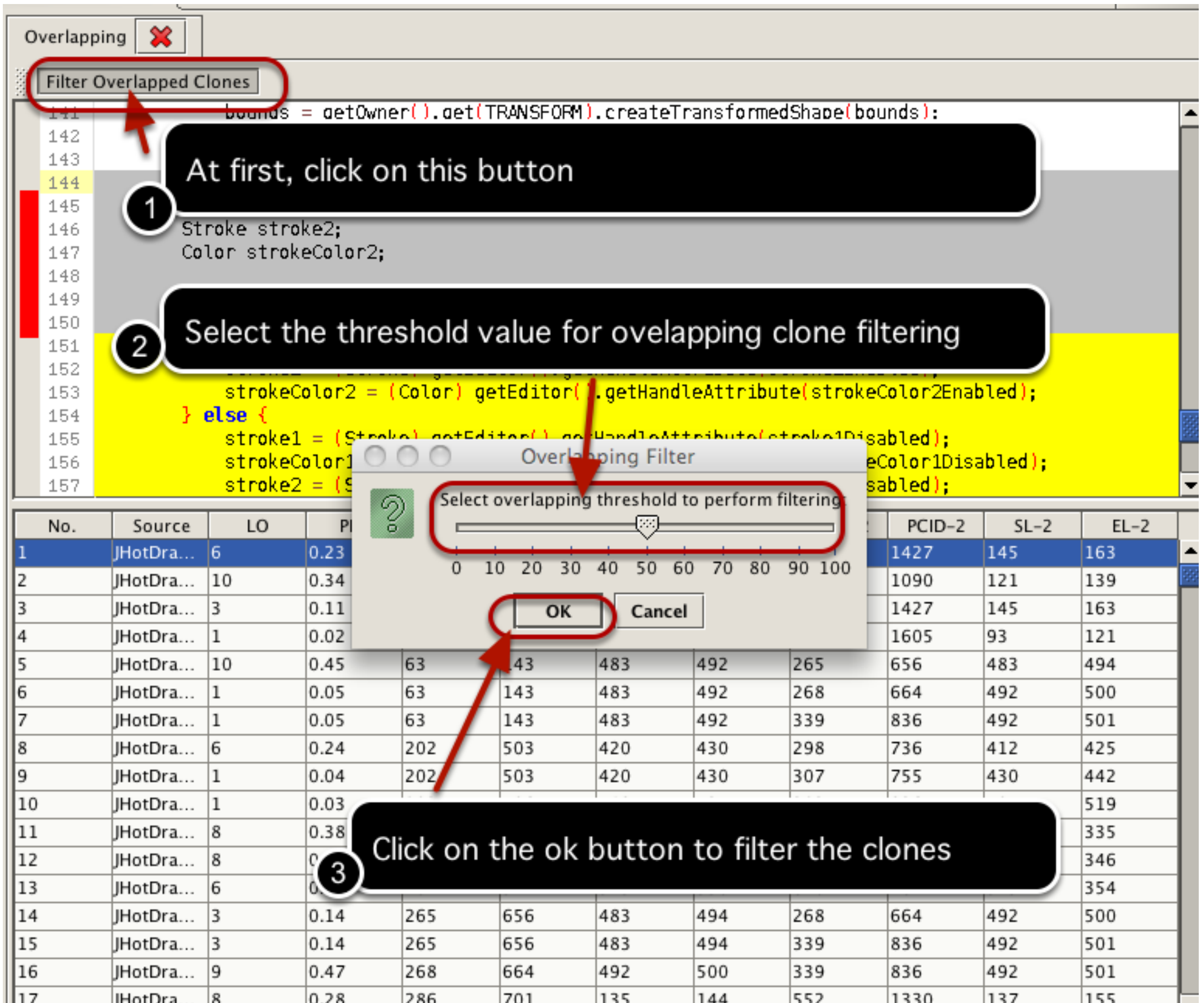
The length of the red bar indicates the level of overlapping

List of overlapping clone pairs

| No. | Source | LO | PLO | CCID-1 | PCID-1 | SL-1 | EL-1 | CCID-2 | PCID-2 | SL-2 | EL-2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | JHotDra... | 6 | 0.23 | 28 | 64 | 144 | 150 | 594 | 1427 | 145 | 163 |
| 2 | JHotDra... | 10 | 0.34 | 124 | 314 | 126 | 135 | 448 | 1090 | 121 | 139 |
| 3 | JHotDra... | 3 | 0.11 | 289 | 712 | 161 | 169 | 594 | 1427 | 145 | 163 |
| 4 | JHotDra... | 1 | 0.02 | 448 | 1090 | 121 | 139 | 674 | 1605 | 93 | 121 |
| 5 | JHotDra... | 10 | 0.45 | 63 | 143 | 483 | 492 | 265 | 656 | 483 | 494 |
| 6 | JHotDra... | 1 | 0.05 | 63 | 143 | 483 | 492 | 268 | 664 | 492 | 500 |
| 7 | JHotDra... | 1 | 0.05 | 63 | 143 | 483 | 492 | 339 | 836 | 492 | 501 |
| 8 | JHotDra... | 6 | 0.24 | 202 | 503 | 420 | 430 | 298 | 736 | 412 | 425 |
| 9 | JHotDra... | 1 | 0.04 | 202 | 503 | 420 | 430 | 307 | 755 | 430 | 442 |
| 10 | JHotDra... | 1 | 0.03 | 203 | 506 | 519 | 535 | 383 | 936 | 504 | 519 |
| 11 | JHotDra... | 8 | 0.38 | 207 | 514 | 323 | 330 | 495 | 1200 | 323 | 335 |
| 12 | JHotDra... | 8 | 0.25 | 207 | 514 | 323 | 330 | 658 | 1571 | 323 | 346 |
| 13 | JHotDra... | 6 | 0.16 | 207 | 514 | 323 | 330 | 678 | 1614 | 325 | 354 |
| 14 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 268 | 664 | 492 | 500 |
| 15 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 339 | 836 | 492 | 501 |
| 16 | JHotDra... | 9 | 0.47 | 268 | 664 | 492 | 500 | 339 | 836 | 492 | 501 |
| 17 | JHotDra... | 8 | 0.28 | 286 | 701 | 135 | 144 | 552 | 1330 | 137 | 155 |

VisCad analyzes and shows the list of overlapping clone pairs.

## Set threshold value for overlapping clone filtering



# Saving the result

Filter Overlapped Clones

```
141             bounds = getOwner().get(TRANSFORM).createTransformedShape(bounds);
142         }
143         bounds = view.getDrawingToViewTransform().createTransformedShape(bounds);
144         Stroke stroke1;
145         Color strokeColor1;
146         Stroke stroke2;
147         Color
148
149         if (
150             s
151             strokeColor1 = (Color) getEditor().getHandleAttribute(strokeColor1Enabled);
152             stroke2 = (Stroke) getEditor().getHandleAttribute(stroke2Enabled);
153             strokeColor2 = (Color) getEditor().getHandleAttribute(strokeColor2Enabled);
154         } else {
155             stroke1 = (Stroke) getEditor().getHandleAttribute(stroke1Disabled);
156             strokeColor1 = (C                                  strokeColor1Disabled);
157             stroke2 = (Stroke                                  ke2Disabled);
```

Click on the Yes button to open a save dialog box.

Result of filtering

Do you want to export the result?

Yes No

| No. | Source | LO | PLO | | | | | CCID-2 | PCID-2 | SL-2 | EL-2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | JHotDra... | 6 | 0.23 | | | | | 4 | 1427 | 145 | 163 |
| 2 | JHotDra... | 10 | 0.34 | | | | | 8 | 1090 | 121 | 139 |
| 3 | JHotDra... | 3 | 0.11 | | | | | 4 | 1427 | 145 | 163 |
| 4 | JHotDra... | 1 | 0.02 | 448 | 1090 | 121 | 139 | 674 | 1605 | 93 | 121 |
| 5 | JHotDra... | 10 | 0.45 | 63 | 143 | 483 | 492 | 265 | 656 | 483 | 494 |
| 6 | JHotDra... | 1 | 0.05 | 63 | 143 | 483 | 492 | 268 | 664 | 492 | 500 |
| 7 | JHotDra... | 1 | 0.05 | 63 | 143 | 483 | 492 | 339 | 836 | 492 | 501 |
| 8 | JHotDra... | 6 | 0.24 | 202 | 503 | 420 | 430 | 298 | 736 | 412 | 425 |
| 9 | JHotDra... | 1 | 0.04 | 202 | 503 | 420 | 430 | 307 | 755 | 430 | 442 |
| 10 | JHotDra... | 1 | 0.03 | 203 | 506 | 519 | 535 | 383 | 936 | 504 | 519 |
| 11 | JHotDra... | 8 | 0.38 | 207 | 514 | 323 | 330 | 495 | 1200 | 323 | 335 |
| 12 | JHotDra... | 8 | 0.25 | 207 | 514 | 323 | 330 | 658 | 1571 | 323 | 346 |
| 13 | JHotDra... | 6 | 0.16 | 207 | 514 | 323 | 330 | 678 | 1614 | 325 | 354 |
| 14 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 268 | 664 | 492 | 500 |
| 15 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 339 | 836 | 492 | 501 |
| 16 | JHotDra... | 9 | 0.47 | 268 | 664 | 492 | 500 | 339 | 836 | 492 | 501 |
| 17 | JHotDra... | 8 | 0.28 | 286 | 701 | 135 | 144 | 552 | 1330 | 137 | 155 |

## Saving the result(Continued)

Overlapping

Filter Overlapped Clones

```
141             bounds = getOwner().get(TRANSFORM).createTransformedShape(bounds);
142         }
143         bounds = view.getDrawingToViewTransform().createTransformedShape(bounds);
144         Stroke stroke1;
145         Color strokeColor1;
146         Stroke stroke2;
147         Color strokeColor2;
148
149         if (
150
151                                                                   ed);
152
153                                                                   ed);
154         } el
155
156                                                                   led);
157
```

1 Set the name and location of the file

880Research, Components, core, cyclone, Desktop, Documents, Downloads, Dropbox, dwhelper, examples, extensions, Images, javadoc, jtxldb-0.12, kdiff3.app, latest-api-diffs, latest-javadoc, lib

File Name:

Files of Type: All Files

2 Click on this button to save the result

Cancel

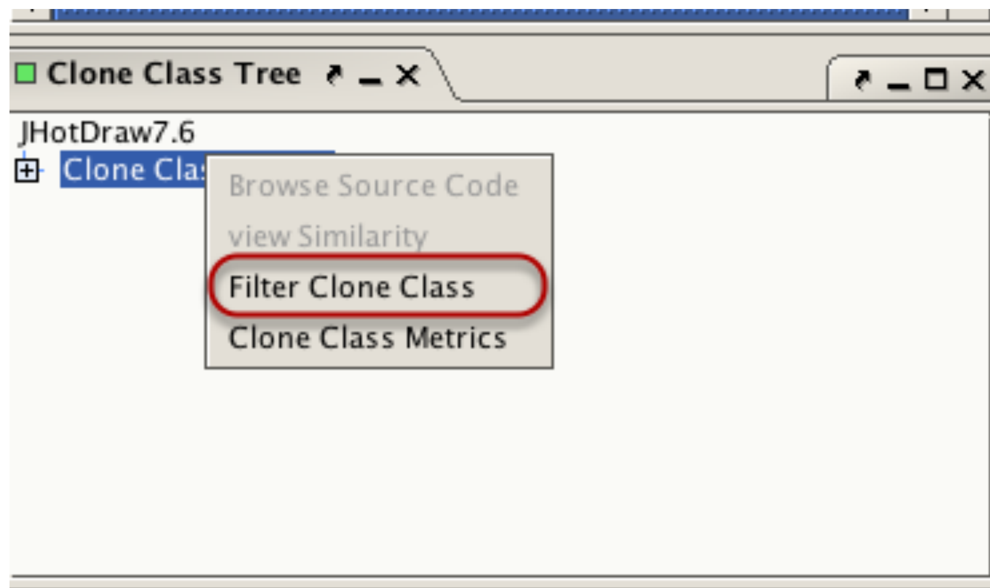
| No. | Source | | | | | | | | | SL-2 | EL-2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | JHotDra... | 6 | | | | | | | | 45 | 163 |
| 2 | JHotDra... | 1 | | | | | | | | 21 | 139 |
| 3 | JHotDra... | 3 | | | | | | | | 45 | 163 |
| 4 | JHotDra... | 1 | | | | | | | | 3 | 121 |
| 5 | JHotDra... | 1 | | | | | | | | 83 | 494 |
| 6 | JHotDra... | 1 | | | | | | | | 92 | 500 |
| 7 | JHotDra | 1 | | | | | | | | 92 | 501 |
| | | | | | | | | | | 12 | 425 |
| | | | | | | | | | | 30 | 442 |
| | | | | | | | | | | 04 | 519 |
| 11 | JHotDra... | 8 | 0.38 | 207 | 514 | 323 | 330 | 495 | 1200 | 323 | 335 |
| 12 | JHotDra... | 8 | 0.25 | 207 | 514 | 323 | 330 | 658 | 1571 | 323 | 346 |
| 13 | JHotDra... | 6 | 0.16 | 207 | 514 | 323 | 330 | 678 | 1614 | 325 | 354 |
| 14 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 268 | 664 | 492 | 500 |
| 15 | JHotDra... | 3 | 0.14 | 265 | 656 | 483 | 494 | 339 | 836 | 492 | 501 |
| 16 | JHotDra... | 9 | 0.47 | 268 | 664 | 492 | 500 | 339 | 836 | 492 | 501 |
| 17 | JHotDra... | 8 | 0.28 | 286 | 701 | 135 | 144 | 552 | 1330 | 137 | 155 |

VisCad saves the filtered clone detection result as a *VisCad input file format*.

## Textual filtering

Textual filtering allows to remove clones that are only structurally similar without having any semantic similariy. For each clone class, VisCad determines the clone fragment that maximizes the sum of the textual similarity to all other fragments of that class. We call this fragment as the ‘leading clone fragment’ for that class. If the textual similarity between the ‘leading clone fragment’ and any other clone fragment in the clone class falls below a given threshold value, we remove the fragment from the analysis. We discard an entire clone class from the analysis when the textual similarities between the leading clone fragment and all other non-leading clone fragments of that clone class fall below the threshold value.

### Make the selection




Select the *Clone Classes* node from the *Clone Class Tree.* Right click on it to bring the popup menu and select the *Filter Clone Class* option. This open the *similarity browser* view.

## Similarity Browser View

Selected clone class

Similarity values. First one is the leading clone fragment

Graphically shows the similarity

Circle with blue color indicates leading node for the selected clone
Circles with the red color represent other clones.

| PCID | Distance | CCID |
|---|---|---|
| 274 | 0.0 | 111 |
| 275 | 0 | 111 |
| 276 | 0 | 111 |
| 277 | 0 | 111 |
| 272 | 0.02 | 111 |
| 273 | 0.02 | 111 |
| 271 | 0.05 | 111 |
| 269 | 0.11 | 111 |
| 270 | 0.11 | 111 |

## Filter the result

Click on this button

Set the threshold value for filtering

Click on the ok button

## Filter the result(Continued)

VisCad saves the filtered clone detection result in *VisCad input file format*.