

# An Empirical Study of Code Clone Genealogies

Miryung Kim, Vibha Sazawal,  
David Notkin  
Computer Science & Engineering  
University of Washington  
Seattle, Washington, USA

{miryung,vibha,notkin}@cs.washington.edu

Gail C. Murphy  
Department of Computer Science  
University of British Columbia  
Vancouver, B.C., Canada  
murphy@cs.ubc.ca

*Note to Reviewers: This paper extends and expands upon the paper appearing in the ICSE 2005 Mining Software Repositories Workshop [22]. In particular, this paper presents the formal model of clone genealogy, provides complete analysis at a finer granularity (analyzing check-ins instead of releases), and assesses the analysis in many additional dimensions.*

## ABSTRACT

There has been a broad assumption that code clones are inherently bad and that eliminating clones by refactoring would solve the problems of code clones. To investigate whether this assumption is valid, we developed a formal definition of clone evolution and built a clone genealogy tool that automatically extracts the history of code clones from a source code repository. Using our clone genealogy extractor, we studied the evolution of code clones in two Java open source projects.

Our study of clone evolution contradicts some conventional wisdom about clones; refactoring may not benefit many clones for two reasons. First, many code clones exist in the system for only a short time, disappearing soon after; extensive refactoring of such short-lived clones may not be worthwhile if they are to diverge from one another very soon. Second, many clones, especially long-lived clones that have changed consistently with other elements in the same group, are not locally refactorable due to the programming language limitations. Our study discovers that there are types of clones that refactoring would not help, and it opens up opportunities for clone maintenance tools that target unaddressed classes of clones using clone genealogy information.

## 1. INTRODUCTION

The presence of code clones—code snippets that are similar in syntax and semantics—is generally considered to be an indication of poor software quality. The primary concern is that programmers may introduce bugs when changing code

if they inadvertently neglect to change related clones.

This view has directed the goals of previous research efforts—finding code clones and aggressively removing them. Several efforts have focused on automatically identifying code clones [6, 10, 13, 19, 20, 23, 25, 27, 31] and using the output of clone detection as a basis for refactoring [8, 18, 24]. Some methodologists have educated programmers about how to avoid or remove code clones. For example, Fowler [15] argues that code duplicates are bad smells of poor design and programmers should aggressively use refactoring techniques. As another example, the Extreme Programming (XP) [11] community has integrated frequent refactoring as a part of the development process and has argued that fewer clones are found in XP process software [31].

We became skeptical about the universal validity of this conventional wisdom after studying copy and paste programming practices in industry [21]. In the study, we found that skilled programmers often created and managed code clones with clear intent. Subjects copied and pasted code snippets to reuse logic that could not be abstracted due to the limitations of the Java programming language. In addition, our subjects often appeared to discover a shared abstraction of similar code through the process of copying, pasting, and modifying code; they kept and maintained clones for some period of time before they realized how to abstract the common part of the clones.

Our previous study suggested that the practice of creating and managing clones is not necessarily bad and motivated us to investigate how clones evolve over time. So we performed a longitudinal analysis of how clones change over the lifetime of software. To enable this analysis, we defined a formal model of clone evolution and built a tool that automatically extracts clone genealogies—the history of how each element in a group of clones has changed with respect to other elements in the same group—from a set of program versions. Using this tool, we studied, in two Java open source projects, (1) how long clones survive in a system and (2) how often and in which way clones change.

In summary, we found the following:

- Consistent with conventional wisdom, clones impose obstacles during software evolution because they often change consistently with their counterparts in the same group. In our study, 27% to 38% of clone ge-

nealogies consist of clones that have changed consistently with other elements in the same group.

- Contrary to conventional wisdom, popular refactoring techniques [15] cannot easily remove long-lived clones that change consistently with other clones. In the systems we studied, we found that 64% to 69% of clone genealogies consist of clones that cannot be easily removed using standard refactoring techniques. In fact, the longer code clones survive in the system, the more they represent this type of unfactorable clones that have changed consistently with other clones.
- In the systems that we studied, we found that many clones were volatile; among the clone genealogies that disappeared during evolution, 54% to 72% disappeared within an average of eight check-ins out of over 160 check-ins. If clones in the same group are to change differently from one another in a short time, aggressive, immediate refactoring may not be necessary or beneficial.

The results suggest understanding the characteristics of how clones actually evolve may open new approaches to clone management. We describe some of these possible approaches near the end of this paper.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 formally defines the model of clone evolution, which serves as the basis of our clone genealogy extractor described in Section 4. Section 5 describes our study procedure, and Section 6 presents analysis of clone evolution patterns and discusses implications of our results. Section 7 discusses our study limitations. Section 8 proposes clone maintenance tools that use clone genealogy information, and Section 9 summarizes our study.

## 2. RELATED WORK

### Automatic Detection of Code Clones

Although most consider code clones as identical or near identical fragments of source code [9, 20], code clones have no consistent definition in the literature. Indeed, a “clone” has been defined operationally based on the computation of individual clone detectors.

Clone detectors can be grouped into four basic approaches, each of which uses a different representation of source code and a different algorithm for comparing the representation of potential clones.

First, some detectors are based on lexical analysis. For instance, Baker’s [6] *Dup* uses a lexer and a line-based string matching algorithm. *Dup* removes white spaces and comments; replaces identifiers of functions, variables, and types with a special parameter; concatenates all files to be analyzed into a single text file; hashes each line for comparison; and extracts a set of pairs of longest matches using a suffix tree algorithm. Kamiya *et al.*, improved *Dup*’s algorithm and developed CCFinder [20], which transforms tokens of a program according to a language-specific rule and performs a token-by-token comparison. CCFinder is recognized as a state of the art clone detector that handles industrial size programs; it is reported to produce higher recall although its

precision is lower than some other tools [12]. *CP-Miner* [27] identifies a similar sequence of tokenized statements using a frequent subsequence mining technique.

Second, Baxter *et al.*, developed a *CloneDr* [10], which parses source code to build an abstract syntax tree and compares its subtrees by characterization metrics (hash functions).

Third, some detectors find clones by identifying an isomorphic program dependence graph. Komondoor and Horwitz’s clone detector finds isomorphic PDG subgraphs using program slicing [23]. Krinke uses a  $k$ -length patch matching to find similar PDG subgraphs [25]. PDG-based clone detection is robust to reordered statements, insertion and deletion of code, intertwined code, and non-contiguous code, but it is not scalable to large size programs.

Finally, metric-based clone detectors [19, 14, 28, 29] compare various software metrics called fingerprinting functions. These clone detectors find clones in a particular syntactic granularity such as a class, a function, or a method, because fingerprinting functions are often defined for a particular syntactic unit.

### Reengineering the Output of a Clone Detector

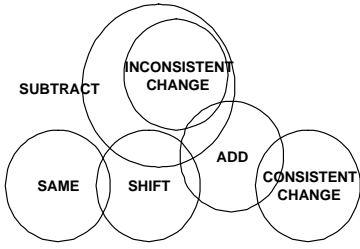
Researchers have also used the output of a clone detector as a basis for refactoring. For example, Balazinska *et al.*, developed a clone reengineering tool, called *CloRT* [7, 8]. *CloRT* finds clones using software metrics and a dynamic pattern matching algorithm, determines whether the *Strategy* or *Template* design pattern applies to these clones, factors out the common parts of methods, and parametrizes the differences with respect to the design patterns. As another example, Komondoor and Horwitz developed a semantics-preserving procedure extraction algorithm that runs on PDG-based clones [23, 24]. Finally, *CCShaper* [18] filters the output of CCFinder to find good candidates for the *Extract Method* and *Pull Up Method* refactoring patterns.

### Studies of Clone Coverage

Several studies have investigated the extent of code clones using a clone detector. Comparing the result of these studies is difficult because the definition of a clone depends on the computation of individual clone detectors and many detection algorithms take adjustable parameters. Nearly as much as 10% to 30% of the code in many large scale projects was identified as code clones (e.g., *gcc*-8.7% [13], *JDK*-29% [20], *Linux*-22.7% [27], etc). Antoniol *et al.*, [5] and Li *et al.*, [27] studied changes in clone coverage (the ratio of cloned code to the total lines of code) in Linux and found that clone coverage increased in the beginning but became fairly stable. They interpreted these data that the design of Linux is not deteriorating due to copy and paste practices. These quantitative studies of clones do not address how clones change over time and whether it is difficult for developers to manage code clones.

### A Study of Changes of Clones

Evolution of code clones was studied for the first time by Laguë *et al.*, [26]. They studied clones in six versions of a large telecommunication software and found that a significant number of clones were removed but the overall number of clones increased over time in the system. Their approach



**Figure 1: The relationship among evolution patterns**

traces code clones in consecutive versions using a metric-based clone detector and classifies clones into four categories: new clones, modified clones, never modified clones, and deleted clones. Their analysis does not address how elements in a group of code clones change with respect to other elements in the group. To the best of our knowledge, our clone genealogy extractor (detailed in Section 4) is the first tool that systematically analyzes clone evolution patterns by monitoring how a clone group evolves.

### Techniques for Analyzing Structural Changes

*Origin analysis* [16, 37] is similar to our genealogy analysis (described in detail in Section 3 and 4) because it employs a cloning relationship to trace code fragments across versions. The goal of origin analysis is to understand structural changes during evolution, and it has been applied to detect splitting and merging of code fragments. However it differs from our analysis that (1) it semi-automatically traces only code fragments specified by a user and (2) it does not monitor operational changes to a group of code clones, such as whether clones change consistently (or inconsistently) with other elements in the same group.

Antoniol *et al.*, proposed an automatic approach, based on vector space information retrieval, to identify several refactoring events, namely class renaming, replacement, merge, and split [4]. A similar approach was used to identify “move method” refactoring events [32]. These analyses do not focus on structural changes of code clones.

## 3. MODEL OF CLONE GENEALOGY

To study clone evolution structurally and semantically rather than quantitatively, we defined a model of clone genealogy. The genealogy of code clones describes how groups of code clones change over multiple versions of a program. In a clone’s genealogy, the origin of a group to which the clone belongs is traced to the previous version. The model associates related clone groups that have originated from the same ancestor clone group. In addition, the genealogy contains information about how each element in a group of clones has changed with respect to other elements in the same group.

We wrote our model in the Alloy modeling language [3] to check whether several evolution patterns can describe all possible changes to a clone group and to clarify the relationship among evolution patterns. (Our entire model is available on the web [1].)

The basic unit in our model is a Code Snippet, which has

two attributes, Text and Location. Text is an internal representation of code that a clone detector uses to compare code snippets. For example, when using CCFinder [20], text is a parametrized token sequence, whereas when using *CloneDr* [10], text is an isomorphic AST. A Location is used to trace code snippets across multiple versions of a program; thus, every code snippet in a particular version of a program has a unique location. To determine how much the text of a code snippet has changed across versions, we define a TextSimilarity function that measures the text similarity between two texts  $t1$  and  $t2$  ( $0 \leq \text{TextSimilarity}(t1, t2) \leq 1$ ). To trace a code snippet across versions, we define a LocationOverlapping function that measures how much two locations  $l1$  and  $l2$  overlap each other ( $0 \leq \text{LocationOverlapping}(l1, l2) \leq 1$ ). A Clone Group is a set of code snippets with identical text.  $CG.text$  is a syntactic sugar for the text of any code snippet in a clone group  $CG$ . A Cloning Relationship is defined between two clone groups  $CG_1$  and  $CG_2$  if and only if  $\text{TextSimilarity}(CG_1.text, CG_2.text) \geq sim_{th}$ , where  $sim_{th}$  is a constant between 0 and 1. An Evolution Pattern is defined between an old clone group  $OG$  in the  $k - 1^{th}$  version and a new clone group  $NG$  in the  $k^{th}$  version such that there exists a cloning relationship between  $NG$  and  $OG$ .

We defined several evolution patterns that describe all possible changes to a clone group. The relationship among evolution patterns is shown in the Venn diagram in Figure 1.

- Same: all code snippets in  $NG$  did not change from  $OG$ .  
 $\text{TextSimilarity}(NG.text, OG.text) = 1$   
 $\text{all } cn:\text{CodeSnippet} \mid \text{some } co:\text{CodeSnippet} \mid cn \text{ in } NG \Rightarrow$   
 $co \text{ in } OG \ \&\& \ \text{LocationOverlapping}(cn, co) = 1$   
 $\text{all } co:\text{CodeSnippet} \mid \text{some } cn:\text{CodeSnippet} \mid co \text{ in } OG \Rightarrow$   
 $cn \text{ in } NG \ \&\& \ \text{LocationOverlapping}(cn, co) = 1$
- Add: at least one code snippet in  $NG$  is a newly added one. For example, programmers added a new code snippet to  $NG$  by copying an old code snippet in  $OG$ .  
 $\text{TextSimilarity}(NG.text, OG.text) \geq sim_{th}$   
 $\text{some } cn:\text{CodeSnippet} \mid \text{all } co:\text{CodeSnippet} \mid co \text{ in } OG \Rightarrow$   
 $cn \text{ in } NG \ \&\& \ \text{LocationOverlapping}(cn, co) = 0$
- Subtract: at least one code snippet in  $OG$  does not appear in  $NG$ . For example, programmers refactored or removed a code clone.  
 $\text{TextSimilarity}(NG.text, OG.text) \geq sim_{th}$   
 $\text{some } co:\text{CodeSnippet} \mid \text{all } cn:\text{CodeSnippet} \mid cn \text{ in } NG \Rightarrow$   
 $co \text{ in } OG \ \&\& \ \text{LocationOverlapping}(cn, co) = 0$
- Consistent Change: all code snippets in  $OG$  have changed consistently; thus they belong to  $NG$  together. For example, programmers applied the same change consistently to all code clones in  $OG$ .  
 $sim_{th} \leq \text{TextSimilarity}(NG.text, OG.text) < 1$   
 $\text{all } co:\text{CodeSnippet} \mid \text{some } cn:\text{CodeSnippet} \mid co \text{ in } OG \Rightarrow$   
 $cn \text{ in } NG \ \&\& \ \text{LocationOverlapping}(cn, co) > 0$
- Inconsistent Change: at least one code snippet in  $OG$  changed inconsistently; thus it does not belong to  $NG$  anymore. For example, a programmer forgot to change one code snippet in  $OG$ .  
 $sim_{th} \leq \text{TextSimilarity}(NG.text, OG.text) < 1$

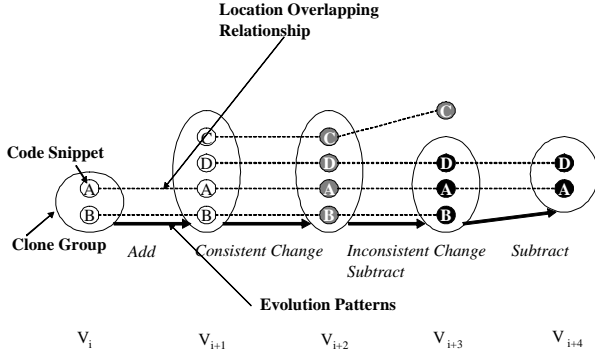


Figure 2: An example clone lineage

some  $co:CodeSnippet \mid all\ cn:CodeSnippet \mid cn\ in\ NG \Rightarrow co\ in\ OG \ \&\& \ LocationOverlapping(cn,co) = 0$

- Shift: at least one code snippet in  $NG$  partially overlaps with at least one code snippet in  $OG$ .<sup>1</sup>  
 $TextSimilarity(NG.text,OG.text) = 1$   
 some  $cn:CodeSnippet \mid some\ co:CodeSnippet \mid cn\ in\ NG \ \&\& \ co\ in\ OG \ \&\& \ (1 > LocationOverlapping(cn,co) > 0)$

A Clone Lineage is a directed acyclic graph that describes the evolution history of a sink node (clone group). In a clone lineage, a clone group in the  $k^{th}$  version is connected by an evolution pattern from a clone group in the  $k - 1^{th}$  version. For example, Figure 2 shows a clone lineage including Add, Subtract, Consistent Change, and Inconsistent Change. In the figure, code snippets with the same text are filled with the same color.

A Clone Genealogy is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group is connected by at least one evolution pattern.<sup>2</sup> A clone genealogy approximates how programmers create, propagate, and evolve code clones. For example, Figure 3 shows a clone genealogy that comprises two clone lineages.

#### 4. CLONE GENEALOGY EXTRACTOR

Based on the clone genealogy model in Section 3, we built a tool that automatically extracts clone genealogies over a project’s lifetime. Our clone genealogy extractor (CGE) requires three inputs: (1) multiple versions of a program in a chronological order,  $\{V_k \mid 1 < k \leq n\}$ , (2) a clone detector, and (3) a location tracker that traces a code snippet’s location across versions.

To assist a user of CGE to prepare multiple versions of a pro-

<sup>1</sup>This unintuitive pattern was found when we used Alloy to check whether the combination of patterns can describe all possible changes to a clone group.

<sup>2</sup>A clone genealogy is a connected component in the sense that there exists an undirected path for every pair of clone groups. Although a clone genealogy is often an inverted tree in practice, it is a connected component in theory because the in-degree of a new clone group can be greater than one when it is ambiguous to determine the most likely origin of a new clone group.

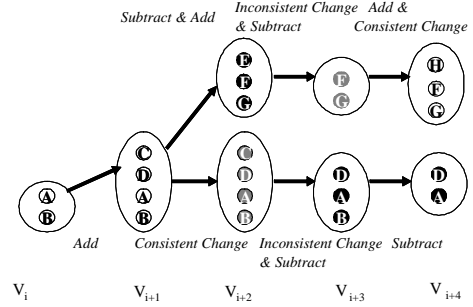


Figure 3: An example clone genealogy

gram, CGE automatically extracts all versions of a project in a chronological order from its source code repository (CVS). Because CVS records individual file versions but not which files were changed together, CGE uses *Kenyon’s* [17] frontend to identify CVS check-in transactions and to check out the source code that corresponds to each check-in. Depending on the granularity of evolution analysis, a user can select a subset of versions. For example, a user can select all versions corresponding to all check-ins or only the versions that increased (or decreased) the total number of lines of code clones (LOCC).

CGE identifies clone groups in each version  $V_k$  using a clone detector. Currently we use CCFinder [20] described in Section 2, but any clone detector can be used.

CGE also implements the  $TextSimilarity$  function using a clone detector. CGE currently identifies the common part between two texts  $t1$  and  $t2$  using CCFinder and calculates the common part’s relative proportion to the size of  $t1$  and  $t2$ .

$$TextSimilarity(t1,t2) = \frac{2|t1 \cap t2|}{|t1| + |t2|} \quad (1)$$

where  $|t|$  is the size of text  $t$  and  $t1 \cap t2$  is the common part of  $t1$  and  $t2$ .

CGE uses a location tracker to implement the  $LocationOverlapping$  function, which computes an overlapping score between a location  $L_i$  in  $V_{k-1}$  and a location  $L_j$  in  $V_k$ . CGE currently uses a file and line based location tracker that we built on top of *diff*. The *diff*-based location tracker maps the line numbers of  $L_j$  to old line numbers in the same file  $f$  in  $V_{k-1}$  based on the differences caused by insertion or deletion of code. Then it computes the relative proportion of an overlapped region between  $L_i$  and the calibrated  $L_j$ .

$$LocationOverlapping(L_i, L_j) = \frac{\min(n_e, o_e) - \max(n_s, o_s)}{n_e - n_s} \quad (2)$$

where  $L_i$  spans from the line  $o_s$  to the line  $o_e$ , and the calibrated location of  $L_j$  in  $V_{k-1}$  spans from the line  $n_s$  to the line  $n_e$ .

Using the same clone detector, CGE finds cloning relationships between the clone groups in  $V_{k-1}$  and the clone groups in  $V_k$  for  $1 < k \leq n$ . Given a clone group in  $V_k$ , a clone detector may find several cloning relationships to clone groups in  $V_{k-1}$ . CGE applies a heuristic to determine the most likely

origin (a clone group in  $V_{k-1}$ ) and to remove less interesting cloning relationships. Our heuristic first selects a cloning relationship with the best location overlapping score. If a cloning relationship with the best similarity score is different from the one with the best location overlapping score, our heuristic selects both because of ambiguity.

After applying the heuristic, CGE separates each connected component of cloning relationships. Then, it labels evolution patterns in each connected component. Each connected component is called a clone genealogy. CGE visualizes a genealogy graph using the *Graphviz* package [2] and allows a user to browse code relevant to a selected genealogy.

## 5. STUDY PROCEDURE

Our CGE captures various kinds of clone evolution patterns and thus allows us to explore a wide variety of research questions about clone evolution such as, “how often do programmers copy an existing clone and modify it later?” and “how often do programmers forget to update clones consistently?” In this study, we focused on the following questions: (1) how often do programmers update clones consistently? (2) how long do clones live in the system? and (3) what are evolutionary characteristics of clones that cannot be easily removed with refactoring techniques in Java?

To determine these characteristics, we chose two subject programs with a long evolutionary history and applied our tool to extract clone genealogies from those programs. Because the clone detector can introduce false positives, we manually removed some of the extracted genealogies. We then analyzed the remaining genealogies and computed how old genealogies are and what kind of evolution patterns they include.

Since our previous copy and paste study [21] had focused on Java programs, we decided to focus on subject programs written in Java. We selected *carol* and *dnsjava* because they met this condition and both had CVS history for over a year and a half. In addition, their code size allowed us to manually inspect genealogies if necessary. *Carol* is a library that allows clients to use different RMI implementations and has evolved over 26 months from August 2002 to October 2004. *Dnsjava* is an implementation of DNS in Java that has evolved over 68 months from March 1999 to November 2004. Table 1 describes the programs’ size in lines of code (LOC), the period that we studied, and the number of CVS check-ins during the period.<sup>3</sup> We studied the history of *dnsjava* between April 2003 to November 2004 because its CVS history is available only from April 2003.

For our analysis, we focused on versions of the programs in which the LOCC (the total number of lines of code clones) increased or decreased from the preceding version. We chose this set of program versions because it approximates a set of program versions that added or deleted code clones or made changes to code clones. Versions of a program that do not touch code clones do not directly impact evolution patterns in clone genealogies. For our target programs, this resulted

<sup>3</sup>A *check-in* in our analysis corresponds to a single logical CVS transaction that commits a set of revisions together within a time window of 200 seconds [36].

**Table 1: Description of Two Java Subject Programs**

Program	<i>carol</i>	<i>dnsjava</i>
URL	carol.objectweb.org	www.dnsjava.org
LOC	7878 ~ 23731	16754 ~ 21188
duration	26 months	19 months
# of check-ins	164	248

**Table 2: Clone Genealogies in *carol* and *dnsjava***  
( $min_{token} = 30$ ,  $sim_{th} = 0.3$ )

# of genealogies	<i>carol</i>	<i>dnsjava</i>
total	122	71
false positive	13	22
true positive	109	49
locally unfactorable	70 (64%)	34 (69%)
consistently changed	41 (38%)	13 (27%)

in studying 37 versions out of 164 versions of *carol* and 48 versions out of 248 versions of *dnsjava*.

We used 30 tokens as a minimum token length for CCFinder because programmers do not generally consider very short clones as code clones. Setting the minimum at 30 tokens resulted in an average clone size of four lines in *carol* and *dnsjava*. With this setting, CCFinder found clone coverage ratio of 6% to 15% in *carol* and 5% to 8% in *dnsjava*.

We set the threshold  $sim_{th}$  of TextSimilarity function to be 0.3 because we empirically found that 0.3 does not underestimate nor overestimate the size and the length of genealogies. We discuss how  $sim_{th}$  affects our results in detail in Section 7.1.

CCFinder occasionally detects false positive clones that are similar in a token sequence, although common sense says that they are not clones. We used our previously defined concept of a “syntactic template” to identify clones as false positives. The idea of a syntactic template comes from our study of copy and paste programming [21]. A syntactic template is a template of repeated code appearing in a row because a programmer often copies and pastes a code fragment when writing a series of syntactically similar code fragments. For example, a programmer often copies a field declaration statement when writing a block of field declaration, an invocation statement when writing a static initializer, or a case statement when writing a series of case statements in a switch-case block. We manually removed 13 out of 122 genealogies in *carol* and 22 out of 71 genealogies in *dnsjava* because they consist of only syntactic templates (see Table 2). Although there could be false negative clones that CCFinder cannot find, we do not think there are many false negatives because a previous comparison of clone detectors [12] suggests that CCFinder has much higher recall than *CloneDr* (AST-based) [10] or *Covet* (metric-based) [28], although its precision is lower than the *CloneDr*.

## 6. STUDY RESULTS

This section presents the evolution patterns of code clones in *carol* and *dnsjava* and answers the questions raised in Section 5.

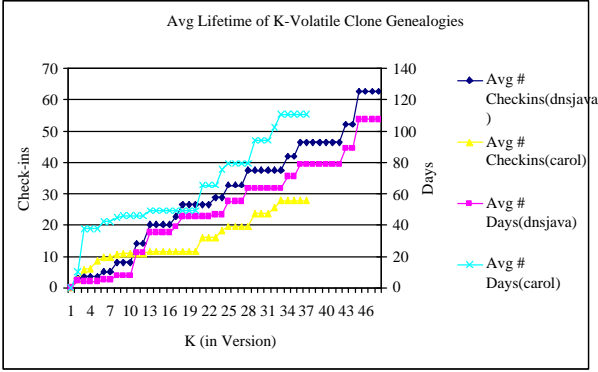


Figure 4: The average lifetime of  $k$ -volatile clone genealogies

## 6.1 Consistently Changing Clones

To determine how often code clones change with other clones in the same clone group consistently, we measured the number of genealogies with a consistent change pattern. We define that a clone genealogy includes a consistently changing pattern if and only if all lineages in the clone genealogy include at least one “consistent change” pattern. Our definition is conservative because, if one lineage in the genealogy does not include a consistent change pattern, the genealogy is considered not to have a consistent change pattern.

Out of 109 genealogies in *carol*, 41 genealogies (38%) include a consistently changing pattern. Out of 49 genealogies in *dnsjava*, 13 genealogies (27%) include a consistently changing pattern. Consistent with conventional wisdom, many of the clones in the study impose the challenge of consistent update.

## 6.2 Volatile Clones

To understand how long clones survive in the systems, we measured the age of a clone genealogy—how many versions (generations) a genealogy spans. In our analysis, we classified genealogies in two groups, dead genealogies that do not include clone groups of the last version and alive genealogies that include clone groups of the last version. We differentiate a dead genealogy from an alive genealogy because the age of a dead genealogy provides information about how long clones stayed in the system before they disappeared. On the other hand, for an alive genealogy, we cannot decide how long its clones would survive because they are still evolving. At the end point of our analysis, in *carol*, out of 109 clone genealogies, 53 of them are dead and 56 of them are alive. In *dnsjava*, out of 49 clone genealogies, 31 of them are dead and 18 of them are alive.

To reason about how long genealogies survived in the unit of an absolute time rather than in the number of versions used in our analysis, we define  $k$ -volatile genealogies and measure the average lifetime of  $k$ -volatile genealogies.  $K$ -volatile genealogies are clone genealogies that have disappeared within  $k$  versions, i.e.,  $k$ -volatile genealogies =  $\{g | g \text{ is a dead genealogy and } 0 \leq g.age \leq k\}$ . Figure 4 shows the average lifetime of  $k$ -volatile genealogies in the number of check-ins (left axis) and the number of days (right axis). Let  $f(k)$  be

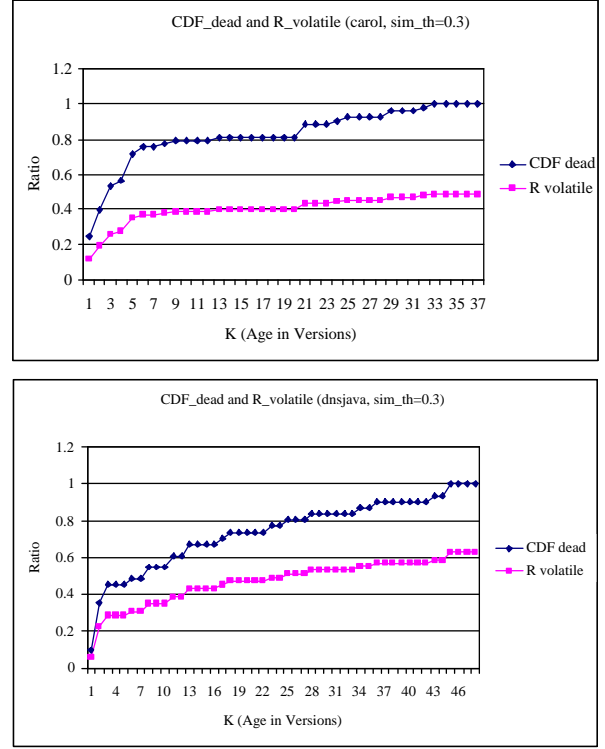


Figure 5:  $CDF_{dead}(k)$  and  $R_{volatile}(k)$  of *carol* and *dnsjava*

the number of genealogies with the age  $k$  and  $f_{dead}(k)$  be the number of dead genealogies with the age  $k$ .  $CDF_{dead}(k)$  is a cumulative distribution function of  $f_{dead}(k)$  and it means the ratio of  $k$ -volatile genealogies among all dead genealogies. Let  $R_{volatile}(k)$  be the ratio of  $k$ -volatile genealogies among all genealogies in the system.

$$CDF_{dead}(k) = \frac{\sum_{i=0}^k f_{dead}(i)}{\sum_{i=0}^n f_{dead}(i)} \quad (3)$$

$$R_{volatile}(k) = \frac{\sum_{i=0}^k f_{dead}(i)}{\sum_{i=0}^n f(i)} \quad (4)$$

Figure 5 presents  $CDF_{dead}(k)$  and  $R_{volatile}(k)$  for *carol* and *dnsjava*. In *carol*, 75% of dead genealogies (37% of all genealogies) have disappeared within 5 versions, and 79% of dead genealogies (39% of all genealogies) have disappeared within 10 versions. When we interpret these data in the number of check-ins or in the number of days by referring to Figure 4, they mean that 75% of dead genealogies lasted an average of 9.6 check-ins and 41.7 days and 79% lasted an average of 10.8 check-ins and 45.6 days during the evolution period of 164 check-ins and 800 days in *carol*.

In *dnsjava*, 48% of dead genealogies (31% of all genealogies) have disappeared within 5 versions, and 61% of dead genealogies (39% of all genealogies) have disappeared within 10 versions. These data mean that 48% of dead genealogies lasted an average of 5.1 check-ins and 5.3 days and 61% lasted an average of 14.2 check-ins and 22.4 days during the evolution period of 248 check-ins and 575 days in *dnsjava*.

Table 3: Code Example of Locally Unfactorable Clones

<pre> public void exportObject(Remote obj) throws RemoteException{     if (TraceCarol.isDebugEnabledRmiCarol()) {         TraceCarol.debugRmiCarol(             "MultiPRODelegate.exportObject(" ... .     )     try {         if (init) {             for (Enumeration e = activePtcls.elements(); ...                 ((ObjDlgt)e.nextElement()).exportObject(obj);             }         } else {             initProtocols();             //iterate protocol elements and export obj         }     } } catch (Exception e) {     String msg = "exportObject(Remote obj) fail";     TraceCarol.error(msg,e);     throw new RemoteException(msg); } } </pre>	<pre> public void unexportObject(Remote obj) throws NoSuchObjectException {     if (TraceCarol.isDebugEnabledRmiCarol()) {         TraceCarol.debugRmiCarol(             "MultiPRODelegate.unexportObject(" ... .     )     try {         if (init) {             for (Enumeration e = activePtcls.elements(); ...                 ((ObjDlgt)e.nextElement()).unexportObject(obj);             }         } else {             initProtocols();             //iterate protocol elements and unexport obj         }     } } catch (Exception e) {     String msg = "unexportObject(Remote obj) fail";     TraceCarol.error(msg,e);     throw new NoSuchObjectException(msg); } } </pre>
--	---

In both systems, surprisingly, a large number of clones were *volatile*. The large extent of volatile clones suggests that a substantial amount of the work done by a developer applying a strategy of aggressive, immediate refactoring may not be cost-effective in some cases. When we manually inspected all dead lineages, we found that 26% (*carol*) to 56% (*dnsjava*) of them were discontinued because of divergent changes in the clone group, meaning that all code snippets in the group survived but they changed differently from one another. Refactoring of such volatile clones may not be necessary and can be counterproductive if a programmer has to undo refactoring in some cases.<sup>4</sup>

### 6.3 Locally Unfactorable Clones

We define that a clone group is “locally refactorable” if a programmer can remove duplication with standard refactoring techniques, such as *pull up a method*, *extract a method*, *remove a method*, *replace conditional with polymorphism*, etc [15]. On the other hand, (1) if a programmer cannot use standard refactoring techniques to remove clones, (2) if a programmer must deal with cascading non-local changes in the design to remove duplication (for example, modifications to publicized interfaces), or (3) if a programmer cannot remove duplication due to programming language limitations, we define that the clone group is locally unfactorable.<sup>5</sup>

Table 3 presents a code example of a locally unfactorable clone group that we found in *carol*. In this example, `exportObject` and `unexportObject` are paired operations that have identical control logic (if-else logic, iterator logic, and

<sup>4</sup>This observation is consistent with the general notion that delaying some design decisions in software development may at times add value [33].

<sup>5</sup>Our previous work [21] describes a taxonomy of locally unfactorable clones that are often created by copy and paste in Java. Basit *et al.*, also summarize the characteristic of locally unfactorable clones that are difficult to remove using abstractions in C++ [9].

exception handling logic) but throw different types of exceptions, pass different messages to the tracing module, and invoke different methods. In addition, they include code duplication that corresponds to crosscutting concerns [34], debugging and tracing. It is difficult to remove this duplication because Java 1.4 does not provide a unit of abstraction that encapsulates similar logic involving different types or different method invocations inside the logic.

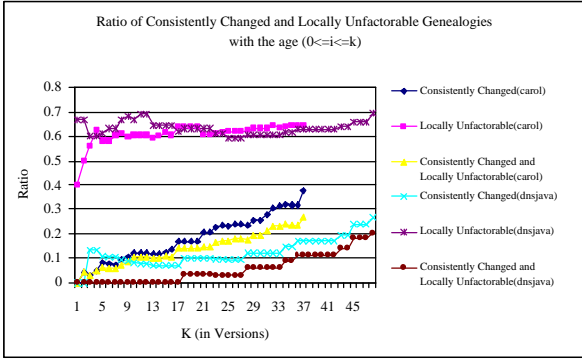
We define that a clone lineage is locally unfactorable if the latest clone group (a sink node of the lineage) is locally unfactorable. We define that a clone genealogy is locally unfactorable if and only if all clone lineages in the genealogy are locally unfactorable. A locally unfactorable genealogy means that a programmer cannot discontinue any of its clone lineages by refactoring.

In the two subject programs, we inspected all clone lineages and manually labeled them as “locally refactorable” or “locally unfactorable.” Then, we measured how many clone genealogies are locally unfactorable. 70 genealogies (64%) in *carol* and 34 genealogies (69%) in *dnsjava* comprise locally unfactorable clone groups; this result indicates that standard refactoring techniques would not remove most clones easily.

### 6.4 Long-Lived Clones

Programmers would get a good return on their refactoring investment if clones live for a long time and if they tend to change with other clones. But even if they are good refactoring candidates, if they are locally unfactorable, it may not be feasible to refactor them.

Out of 37 genealogies that lasted more than 20 versions in *carol*, 29 of them include consistent change patterns, 24 of them comprise locally unfactorable clones, and 19 of them include both consistent change patterns and locally unfactorable clones. Out of 19 genealogies that lasted more than



**Figure 6: Cumulative ratio of consistently changed and locally unfactorable genealogies**

20 versions in *dnsjava*, 10 of them include consistent change patterns, 15 of them comprise locally unfactorable clones, and 9 of them include both consistent change patterns and locally unfactorable clones.

Figure 6 shows the cumulative ratio of (1) consistently changed genealogies, (2) locally unfactorable genealogies, and (3) locally unfactorable genealogies that include consistent change patterns. As  $k$  increases (meaning that the older genealogies get), the more genealogies include a consistently changing pattern and comprise locally unfactorable clones. This result suggests that refactoring techniques cannot improve many trouble-making clones, which are long-lived and consistently changing clones.

## 7. DISCUSSION

Section 7.1 discusses how the text similarity threshold affects our analysis results and Section 7.2 describes limitations of our study.

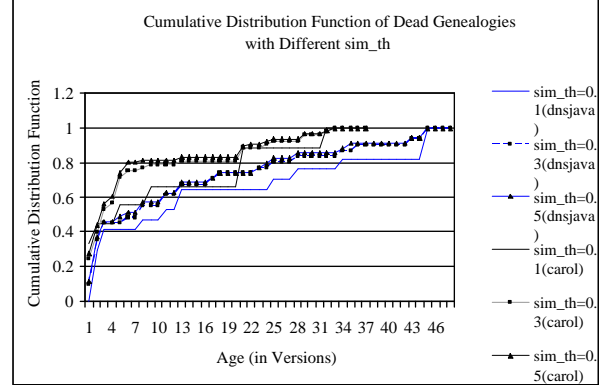
### 7.1 Similarity Threshold

The text similarity threshold,  $sim_{th}$ , sets the bar for defining a cloning relationship, so it could affect the size and length (age) of clone genealogies. Ultimately  $sim_{th}$  could affect the number of volatile clones and consistently changing clones because a low  $sim_{th}$  would find a consistent change pattern between OG and NG while a high  $sim_{th}$  would consider that OG’s lineage is discontinued.

Table 4 shows that, when 0.1 is used, CGE finds fewer genealogies with a larger size and a longer length because it finds more cloning relationships and thus combines several genealogies to one. When 0.5 is used, CGE finds more genealogies with a smaller size and a shorter length because it breaks a long clone genealogy into many short and small genealogies. When  $sim_{th}$  is 0.1, the ratio of consistently changed genealogies is 8% higher in *dnsjava* and 26% higher in *carol* than using 0.3. Figure 7 shows  $CDF(k)$  in *carol* and *dnsjava* when  $sim_{th}$  is 0.1, 0.3, and 0.5.  $CDF(k)$  of 0.3 and  $CDF(k)$  of 0.5 are not much different.  $CDF(k)$  of 0.1 shows a coarse-grained distribution because  $sim_{th}$  0.1 reduces the total number of genealogies. Figure 7 shows that our choice of  $sim_{th}$  0.3 generates a finer-grained distribution than using 0.1 and estimates the number of volatile

**Table 4: The Average Size and Length of Genealogies with Varying  $sim_{th}$**

		$sim_{th}$		
		0.1	0.3	0.5
# of genealogies including false positives	<i>carol</i>	27	122	153
	<i>dnsjava</i>	49	71	78
# of consistently changed genealogies	<i>carol</i>	16	41	53
	<i>dnsjava</i>	10	13	14
avg size (in # of clone groups)	<i>carol</i>	117.52	26.00	20.74
	<i>dnsjava</i>	60.39	41.68	37.94
avg age (length)	<i>carol</i>	25.19	12.57	12.56
	<i>dnsjava</i>	29.51	22.66	21.62



**Figure 7: Cumulative distribution function of dead genealogies with varying  $sim_{th}$**

genealogies more conservatively than using 0.5.

### 7.2 Study Limitations

**Clone Detection Technique.** CGE incorrectly counts the number of consistent change patterns in some cases because CCFinder detects only a contiguous token string as a clone. For example, when code is inserted in the middle of one clone in a clone group, the existing clone group is broken into two new clone groups with shorter contiguous text; thus the change pattern would be identified as two consistent patterns rather than one inconsistent change pattern. As another example, even if a programmer consistently modified *OG* to create *NG*, CCFinder does not find a cloning relationship between *OG* and *NG* if they do not share a contiguous token string greater than the size of  $sim_{th}(|OG.text| + |NG.text|)/2$ . In this case, the absence of a cloning relationship can be interpreted as a discontinuation of a lineage in our analysis. We speculate that this limitation can be overcome by plugging in clone detectors that find non-contiguous code clones, such as *CP-Miner* [27], PDG-based detectors [25, 23], and metric-based detectors [19, 14, 28, 29].

**Location Tracking Technique.** We implemented a file and line based location tracker based on the *diff* algorithm; thus our location tracking algorithm is limited in two ways. First, it depends on *diff* to resolve ambiguity in finding a corresponding line. For example, when a file *A* contains *abc* in the  $k-1^{th}$  version and contains *cba* in the  $k^{th}$  version, *diff*



considers that  $ab$  is deleted before  $c$  and  $ba$  is inserted after  $c$ , even if a programmer replaced  $a$  to  $c$  and  $c$  to  $a$ . Second, our algorithm considers that two files are not related when their file names do not match. For example, when a file  $A$  is renamed to  $B$  or  $A$  is split into two files  $B1$  and  $B2$ , the evolution patterns between  $A$  and  $B$  or  $A$  and  $B1(B2)$  would be identified as *add* and *subtract* patterns. We speculate that code entity tracing techniques in Section 2 can improve our location tracker by inferring how classes were renamed, split, or merged.

**Subject Programs.** Our two subject programs are *carol* and *dnsjava*; both programs' sizes are about 20,000 lines of code. The clone coverage ratio of these programs was smaller than many programs reported in the literature. We speculate that *carol* and *dnsjava* may have fewer locally unfactorable and consistently changing clones than larger programs whose duplication is difficult to remove without compromising many existing design decisions. Both *carol* and *dnsjava* have been maintained by a small number of people: two developers for *dnsjava* and six developers for *carol*. The small team size may have affected our study results.

The granularity of our analysis was a check-in; thus we could not observe the changes between each check-in. Based on our experience of observing programmers copy and paste, we suspect that programmers create more clones temporarily before finding an appropriate level of an abstraction.

Our definition of locally unfactorable clones is Java language dependent; thus our claims about the locally unfactorable clones may not apply to other languages. We speculate that, in other programming languages, different types of locally unfactorable clones would be found depending on the language constructs.

## 8. CLONE MAINTENANCE TOOLS

Our study results indicate that the problems of code clones are not so black and white as previous research has assumed and that there are several types of clones that refactoring would not help. We propose to use clone genealogy information to identify clones that may benefit from new clone maintenance approaches. In our previous workshop paper [22], we proposed several clone maintenance tools. We discuss which types of clones the proposed tools target.

**Suggesting When to Refactor.** For locally refactorable clones, we believe that there may be an optimal time to refactor them. If programmers refactor code clones too early, they might not get the best return on their investment because the code clones may diverge or disappear. On the other hand, if programmers wait too long before they restructure code, they would get only marginal benefit on their investment. To differentiate volatile clones from long-lived clones, programmers can employ clone genealogy information to determine the age of code clones.

**Simultaneous Text Editing.** Our result indicates that the longer clones live, the more they represent locally unfactorable and consistently changing clones. Refactoring techniques would not help maintaining this type of clones consistently. Thus, we propose simultaneous editing of locally unfactorable and consistently changing clones. Simultane-

ous text editing is proposed for automating repetitive text editing and prototyped by Miller and Myers [30]. After describing a set of regions to edit, a user can edit any one record and see equivalent edits applied simultaneously to all other records. A similar editing technique, called *linked editing*, applies the same editing change to a set of code clones specified by a user [35]. These editing techniques require a user to manually specify what must be edited simultaneously. We envision that clone genealogy information can be used to automatically select clones that would benefit most from simultaneous editing.

Furthermore, we believe that many software engineering tools could leverage a rich set of evolution patterns in our clone genealogy model. For example, information about clones that have changed consistently for a long time but then change inconsistently later could be used to automatically place concerns to watch for in a bug database. As another example, information about the propagation of a copied snippet through a codebase could provide visual links in an integrated software development environment.

## 9. CONCLUSIONS

There has been a broad assumption that code clones are inherently bad and refactoring would remove the problems of clones. Thus, previous research efforts focused on mainly two areas: automatically detecting code clones and educating programmers on how to remove or avoid clones. To investigate the validity of this assumption, we built a clone genealogy extractor and investigated clone evolution structurally and semantically rather than quantitatively.

Our study of clone genealogy contradicts some conventional wisdom about code clones: (1) aggressive, immediate refactoring may not be necessary for many volatile clones, and (2) refactoring techniques cannot assist in removing many long-lived, consistently changing clones. We conclude that there are classes of clones that require different types of maintenance support than conventional refactoring. We propose that clone genealogy information can be used to identify clones that may benefit from new approaches to clone management.

## 10. ACKNOWLEDGMENTS

We thank Software Engineering Laboratory at the Osaka University for providing CCFinder and GRASE lab at the University of California, Santa Cruz for providing Kenyon. We also thank Annie Ying and Andreas Zeller for their detailed comments on our draft and Robert DeLine, Dan Grossman, Philippe Kruchten, and Tessa Lau for discussions that helped us refine our ideas.

## 11. REFERENCES

- [1] <http://www.cs.washington.edu/homes/miryung/cge>.
- [2] <http://www.graphviz.org>.
- [3] *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. <http://alloy.mit.edu>, 2004.
- [4] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, pages 31–40, 2004.

- [5] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Analyzing cloning evolution in the Linux kernel. *Information & Software Technology*, 44(13):755–765, 2002.
- [6] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
- [7] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *IEEE METRICS*, pages 292–303, 1999.
- [8] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Partial redesign of Java software systems based on clone analysis. In *WCRE*, pages 326–336, 1999.
- [9] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *ICSE*, 2005.
- [10] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
- [11] K. Beck. *extreme Programming explained, embrace change*. Addison-Wesley, 2000.
- [12] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM*, pages 36–43, 2002.
- [13] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.
- [14] F. Fioravanti, G. Migliarese, and P. Nesi. Reengineering analysis of object-oriented systems via duplication. In *ICSE*, pages 577–586, 2001.
- [15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [16] M. Godfrey and Q. Tu. Tracking structural evolution using origin analysis. In *IWPSE*, pages 117–119, 2002.
- [17] GRASE-Lab. *User Manual: Kenyon*. <http://dforge.cse.ucsc.edu/projects/kenyon>, 2005.
- [18] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring support based on code clone analysis. In *PROFES*, pages 220–233, 2004.
- [19] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*.
- [20] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
- [21] M. Kim, L. Bergman, T. A. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOP. In *ISESE*, pages 83–92, 2004.
- [22] M. Kim and D. Notkin. Using a clone genealogy extractor for understanding and supporting evolution of code clones. In *the Workshop on Mining Software Repositories (http://msr.uwaterloo.ca/msr2005)*, 2005.
- [23] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.
- [24] R. Komondoor and S. Horwitz. Effective, automatic procedure extraction. In *IWPC*, pages 33–43, 2003.
- [25] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.
- [26] B. Laguë, D. Proulx, J. Mayrand, E. Merlo, and J. P. Hudépohl. Assessing the benefits of incorporating function clone detection in a development process. In *ICSM*, pages 314–321, 1997.
- [27] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.
- [28] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, page 244, 1996.
- [29] E. Merlo, G. Antoniol, M. D. Penta, and V. F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analyses. In *ICSM*, pages 412–416, 2004.
- [30] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.
- [31] E. Nickell and I. Smith. Extreme programming and software clones. In *the Proceedings of the International Workshop on Software Clones*, 2003.
- [32] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE*, pages 126–130, 2003.
- [33] K. Sullivan, P. Chalasani, S. Jha, and V. Sazawal. *Software Design as an Investment Activity: A Real Options Perspective in Real Options and Business Strategy: Applications to Decision Making*. Risk Books, November 1999.
- [34] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. *N degrees of separation: Multi-dimensional separation of concerns*. In *ICSE*, pages 107–119, 1999.
- [35] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *the Proceedings of the International Symposium on Visual Languages and Human-Centric Computing*, 2004.
- [36] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *the Proceedings of the Workshop on Mining Software Repositories*, pages 2–6, 2004.
- [37] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In *WCRE*, pages 146–154, 2003.