Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones

Miryung Kim and David Notkin
Computer Science & Engineering
University of Washington
Seattle, USA.
{miryung,notkin}@cs.washington.edu

ABSTRACT

Programmers often create similar code snippets or reuse existing code snippets by copying and pasting. Code clones —syntactically and semantically similar code snippets—can cause problems during software maintenance because programmers may need to locate code clones and change them consistently. In this work, we investigate (1) how code clones evolve, (2) how many code clones impose maintenance challenges, and (3) what kind of tool or engineering process would be useful for maintaining code clones.

Based on a formal definition of clone evolution, we built a *clone genealogy tool* that automatically extracts the history of code clones from a source code repository (CVS). Our clone genealogy tool enables several analyses that reveal evolutionary characteristics of code clones. Our initial results suggest that aggressive refactoring may not be the best solution for all code clones; thus, we propose alternative tool solutions that assist in maintaining code clones using clone genealogy information.

1. INTRODUCTION

We define code clones as syntactically similar code snippets that resemble one another semantically, which are often created by copy and paste[1]. Code clones may induce problems during software evolution. In particular, when a change is made to one element in a group of clones, a programmer must generally make consistent changes to the other elements in the group. Forgetting to update one or more elements may leave outdated code, a potential bug. In other words, code clones impose cognitive overhead because programmers must remember cloning dependencies to apply the same change consistently.

[1]Code clones have no consistent definition in the literature, but most consider them to be identical or near identical fragments of source code [5, 11].

Software engineering researchers have addressed problems surrounding code clones in many ways. First, several kinds of clone detectors have been built. Clone detectors [3, 4, 6, 7, 10, 11, 13, 14, 15, 17] identify similar code snippets automatically by comparing the internal representation of source code (e.g., a parametrized token string [3, 11], AST [6, 17], or PDG [13, 14]). Second, a few programming methodologists have educated programmers about how to avoid or remove code clones. Fowler [8] argues that code duplicates are bad smells of poor design and programmers should aggressively use refactoring techniques. The Extreme Programming (XP) community has integrated frequent refactoring as a part of development process. Nickell and Smith [17] argue that fewer code clones are found in XP process software, claiming that the XP process improves software quality. We believe that these previous research efforts are based on the following assumptions: (1) code clones indicate poor software quality, (2) aggressive refactoring would solve the problem of code clones, and (3) if programmers can locate code clones, they can improve the quality of the code base.

Based on our study of copy and paste programming practices [12], we became skeptical about the validity of some of these assumptions. We found that even skilled programmers sometimes had no choice but to create and manage code clones. Subjects copied and pasted code snippets to reuse the logic that is often not separable given the limitations of Java programming language. Our subjects often discovered an appropriate level of abstraction as they copied, pasted, and modified code; some subjects postponed refactoring until their design decisions become stable.

We hypothesize that programmers create and maintain code clones for two major reasons: (1) as programmers deal with volatile design decisions while they add new features or extend existing features, they prefer not to commit to a particular level of abstraction too quickly, and (2) programmers cannot refactor many code clones because of the primary design decisions in the software and the limitations of programming languages. To test our hypothesis, analyzed how code clones have evolved in two Java open source projects. We formally defined a model of clone evolution and then built an analysis tool that automatically extracts the history of code clones from a set of program versions. Using this tool, we investigated frequent clone evolution patterns.

Our initial result confirms some conventional wisdom about

code clones and also suggests that aggressive refactoring may not benefit many, perhaps not most, clones:

- Clones are not dormant and programmers often face the challenge of updating clones consistently. In fact, 32% $\sim$ 38% of code clones changed consistently with their counterparts at least once in their history.
- Aggressive refactoring may not be the best solution; 64% $\sim$ 68% of code clones were not factorable unless programmers sacrifice primary design decisions or make non-local changes.

Because programmers may not be able to remove or avoid all code clones, we propose clone maintenance tools as effective alternatives and supplements to refactoring. The proposed software engineering tools employ clone genealogy information—the history of code clones—to assist in maintaining clones.

The rest of this paper is organized as follows. Section 2 formally defines the model of clone evolution, which serves the basis of a clone genealogy extractor described in Section 3. Section 4 presents analysis of clone evolution patterns and discusses implications of our initial result. Section 5 proposes software engineering tools that use clone genealogy information. Section 6 summarizes and concludes our study.

2. MODEL OF CLONE EVOLUTION

We formally defined the model of clone evolution to reason how clones change regardless of underlying clone detection technologies.

The basic unit of our analysis 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 [11], a parametrized token sequence is *Text*, whereas when using CloneDr [6], *Text* is an isomorphic AST. A *Location* is used to track code snippets across multiple versions of a program; thus, every code snippet in a particular version of a program has a unique *Location*. A *Clone Group* is a set of code snippets with identical *Text*.
A *Cloning Relationship* exists between an old clone group and a new clone group in two consecutive versions if and only if the similarity between the clone groups is over a similarity threshold sim_{th}. An *Evolution Pattern* is defined between an old clone group OG in the version k and a new clone group NG in the version $k+1$, where NG and OG have a *Cloning Relationship*.

- *Same*: all code snippets in NG did not change from OG.
- *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.
- *Subtract*: at least one code snippet in OG does not appear in NG. For example, programmers removed one clone snippet.

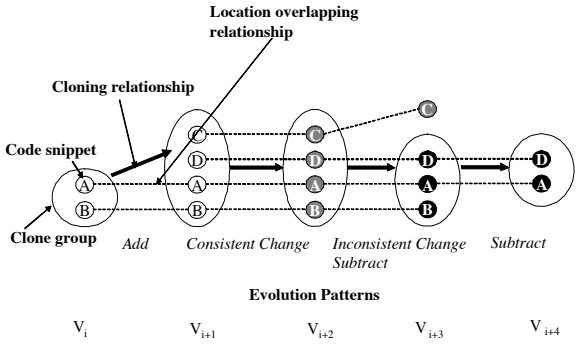

Figure 1: Example Clone Lineage

- *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.
- *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.

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 (node) in the version k is connected by an *Evolution Pattern* (directed edge) from a clone group in the version $k-1$. For example, Figure 1 shows a clone lineage including *Add*, *Subtract*, *Consistent Change*, and *Inconsistent Change*.

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 (node) is connected by at least one evolution pattern (edge). A clone genealogy approximates how programmers create, propagate, and evolve code clones by copying, pasting, and modifying code. Our model is written in the *Alloy* modeling language [2] and is available at [1].

3. CLONE GENEALOGY EXTRACTOR

Based on the clone evolution model in Section 2, we built a tool that automatically extracts clone genealogies over a project's lifetime.

Given the source code repository (CVS) of a project, our tool prepares versions of the project in chronological order. We used Kenyon's front-end to identify CVS transactions and check out the source code that corresponds to each transaction time [9].

Given multiple versions of a program, our tool identifies clone groups in each version using a clone detector. Our tool is designed to plug in different types of a clone detector. Currently we use CCFinder [11], a state-of-the-art clone detector, which compares a parametrized token string of code to detect code clones. Next, it finds cloning relationships between all consecutive versions using the same clone detector. Then, it separates each connected component of

Table 1: Clone Genealogies in *carol* and *dnsjava*

Number of Genealogies	*carol*	*dnsjava*
Total	122	95
False Positive	13	19
Locally Unfactorable	70 (64%)	52 (68%)
Consistent Changed	41 (38%)	24 (32%)

cloning relationships found over the project's life time and labels evolution patterns in each connected component. This connected component is called a clone genealogy.

4. CLONE EVOLUTION ANALYSIS

To understand how clones evolve, we extracted clone genealogies from two Java open source projects, *carol* and *dnsjava*, and studied evolution patterns shown in the genealogies. *Carol* is a library that allows clients to use different RMI implementations and it has grown from 7878 lines of code (LOC) to 23731 LOC from August 2002 to October 2004 (carol.objectweb.org). *Dnsjava* is a implementation of DNS in Java, and it has grown from 5038 LOC to 20752 LOC from March 1999 to June 2004 (www.dnsjava.org).

In our analysis, we chose 37 versions out of 164 check-ins of carol and 39 versions out of 47 releases of dnsjava that resulted in changes of LOCC (the total number of lines of code clones).

We set the minimum token length of CCFinder to be 30 tokens because many programmers do not consider short clones as real clones. We set the similarity threshold sim_{th} for cloning relationships to be 0.3 because empirically we found that sim_{th} 0.3 does not underestimate or overestimate the size or the length of genealogies.

CCFinder occasionally detects false positive clones that are similar only in a token sequence, although common sense says that they are not clones. If clones comprise only a syntactic template, we consider the clones as false positives. In our previous study of copy and paste programming practices [12], we defined "a syntactic template" as 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 to write a series of case statements in a switch-case block. We manually removed 13 out of 122 genealogies in carol and 19 out of 95 genealogies in dnsjava because they comprise only a syntactic template.

Using the clone genealogy information, we intend to examine two research questions: (1) how serious is the problem of code clones? and (2) whether would refactoring benefit most code clones? For each research question, we describe our analysis approach, initial result, and implication of our result.

Q: How many code clones impose maintenance challenges?

If code clones stay dormant, these unchanging clones might not pose challenges during software evolution. But consistently changing clones would reduce productivity because programmers often need to locate code clones and apply the equivalent change to the code clones.

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 very 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. We measured the number of genealogies with a consistent change pattern. Out of 109 genealogies in carol, 41 genealogies (38%) include a consistently changing pattern. Out of 76 genealogies in dnsjava, 24 genealogies (32%) include a consistently changing pasttern (see Table 1). This result implies that programmers had faced the challenge of updating clones consistently with other elements in the same clone group.

Q: Would aggressive refactoring be the best solution for maintaining code clones?

Finding a new abstraction to remove code duplication has been a core approach for effective programming. There has been a broad assumption that code clones are inherently bad because code clones defy the principle of abstraction. To examine the validity of this assumption, we set up two hypotheses.

Hypothesis 1: Many code clones are not locally factorable given the primary design decisions of software and the limitations of programming languages.

In our analysis, we define that a clone group is "locally factorable" 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 [8]. On the other hand, if a programmer must make non-local changes in the design or modify publicized interfaces to remove duplication of if a programmer cannot remove duplication due to programming language limitations, we consider that the clone group is not locally factorable. Our previous work describes a taxonomy of locally unfactorable code clones that are often created by copy and paste [12]. 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 factorable" or "locally unfactorable." Then, we measured how many clone genealogies are locally unfactorable. 70 genealogies (64%) in carol and 52 genealogies (68%) in dnsjava comprise locally unfactorable clone groups; this result indicates that popu-

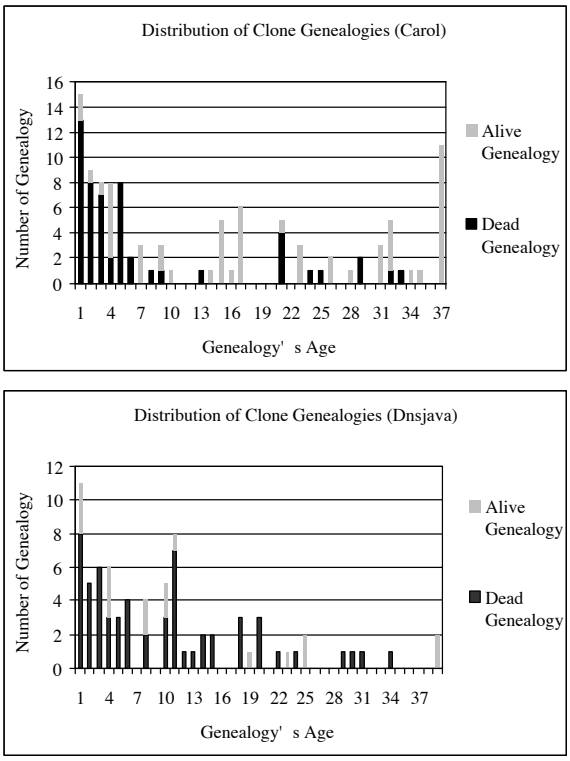

Figure 2: Many clone genealogies disappear after a relatively short time.

lar refactoring techniques would not benefit most clones. In fact, we found that many long-lived, consistently changing clones are locally unfactorable. Out of 37 genealogies that lasted more than 20 versions in carol, 19 of them include both consistent change patterns and locally unfactorable clones. Out of 11 genealogies that lasted more than 20 versions in dnsjava, 3 of them include both consistent change patterns and locally unfactorable clones.

Hypothesis 2: Programmers prefer not to commit to a particular abstraction immediately when dealing with volatile design decisions.

A dead genealogy means that all of its clone lineages were discontinued because the code clones disappeared, diverged, or they were refactored. An alive genealogy means that at least one of its clone lineage is still evolving and the clones have not disappeared yet. Figure 2 shows distribution of dead and alive clone genealogies over their age. The age of a clone genealogy is the number of versions that the genealogy spans. In carol, out of 53 dead genealogies, 42 genealogies disappeared less than 10 versions. In dnsjava, out of 59 dead genealogies, 41 genealogies disappeared less than 10 versions. We believe that programmers created and maintained code clones while they explored new design space, and then later, they removed, diverged, or refactored the code clones as the relevant design decisions became stable. When we manually inspected all dead lineages, we found that 25% (carol) $\sim$ 48% (dnsjava) of them were discontinued because of divergent changes in the clone group. Programmers would not get the best return on their refactoring investment if the clones are to diverge.

5. CLONE MAINTENANCE TOOLS

Our study result indicates that popular refactoring techniques may not remove most code clones, especially clones that are difficult to maintain. Thus, we propose clone maintenance tools as alternatives and supplements to refactoring. This section lists possible software engineering tools that can be built on top of our clone genealogy extractor.

5.1 Simultaneous Text Editing

Abstraction, isolating code duplication in a programming language unit, provides two advantages during software evolution. First, programmers can locate the duplicated logic in one place. Second, programmers can apply the change only once in the refactored code. Clone detectors automatically locate code clones, resolving the first issue. However, programmers still need to update code clones manually one by one when the same change is required, leaving the second issue unresolved. Simultaneous text editing [16] is a new method for automating repetitive text editing. After describing a set of regions to edit, the user can edit any one record and see equivalent edits applied simultaneously to all other records. We propose simultaneous editing of consistently changing clones. The proposed editor uses clone genealogy information to automatically identify code snippets that are likely to change consistently in the future. Then, when a programmer edits one of the clones, upon request, the equivalent edit is made to other clones simultaneously. This proposed editor not only provides the same advantages as abstraction but also allows divergent changes flexibly.

5.2 Cloning Related Bug Detection

Many programming errors occur when programmers create and update code clones. For example, Li *et al.*, found that a few errors in Linux were created when a programmer copied code but failed to rename identifiers correctly in the pasted code [15]. As another example, Ying *et al.*, also reported a cloning related bug in Mozilla [18]; a web browser using gtk UI toolkit and the version using xlib UI toolkit were code clones. When a developer changed the version using gtk but did not update the version using xlib, this missed update led to a serious defect, called "huge font crashes X Windows." If a clone genealogy extractor finds clones that have changed similarly before but change inconsistently later, this information may strongly suggest a bug.

Programmers often copy and paste to reuse existing code snippets. If the copied code contains a bug, this bug can be propagated to many places via copy and paste. In Mozilla, we found that a buggy code snippet was copied for 12 times [12]. If the copied snippets did not change, a clone detector can locate the buggy snippets automatically. But if the copied code was modified very differently from its template, a clone detector may not be able to find it. Our clone genealogy tool infers how programmers copied, modified, and evolved existing code. By traversing a genealogy graph, we can locate code snippets that have originated from the same buggy code even if they have changed very much.

5.3 Decision Support for Maintaining Code Clones

Clone detectors assist programmers in locating code clones automatically. However, even if programmers can find all clones, they may not know which of them should be updated together when the clones change. The history of code clones may help programmers to make informed decisions about how to manage code clone. For example, if a set of clone snippets have changed consistently in the past, they might evolve similarly in the future as well. Programmers can decide what to change together based on the clone history.

We believe that there's a right timing to refactor code clones. If programmers refactor code clones too early, they might not get the best return on their investment because the code clones may diverge. On the other hand, if programmers wait too long before they restructure code, they would get only marginal benefit on their investment. Programmers can decide when to refactor code clones based on clone genealogy information: (1) how old clones are and (2) how clones have changed in the past.

5.4 Locating the Origin of Copied Code

Programmers often copy an example code snippet or a working component and then modify a small part of it. If programmers do not fully understand the logic of the copied code, they cannot adapt the copied code appropriately as the related design changes. Besides, programmers may have copied outdated example code and do not know how to make it up-to-date. In these cases, programmers may want to find the origin of copied code and consult the original author. However, CVS history retains only who checked in the copied code but does not provide who is the original author or when the original code was written. By overlaying authorship on a clone genealogy, programmers would be able to find the origin of frequently copied code.

6. CONCLUSIONS

There has been a broad assumption that code clones are inherently bad because they defy the principle of abstraction. Thus, previous research efforts focused on mainly two areas: automatically detecting code clones and educating programmers how to remove or avoid clones. However, the history of code clones indicates that this assumption may not be necessarily true and that the current refactoring solution may not work for many clones. We propose clone maintenance tools that use clone genealogy information—code clones' history that is automatically extracted from a source code repository.

7. 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.

8. REFERENCES

[1] http://www.cs.washington.edu/homes/miryung/cge.

[2] *Micromodels of Software: Lightweight Modelling and Analysis with Alloy.* http://alloy.mit.edu, 2004.

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

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pages 98–107, 2000.

[5] 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.

[6] 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.

[7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[9] GRASE-Lab. *User Manual: Kenyon.* http://dforge.cse.ucsc.edu/projects/kenyon, 2005.

[10] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*.

[11] 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.

[12] M. Kim, L. Bergman, T. A. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE*, pages 83–92, 2004.

[13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.

[14] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.

[15] 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.

[16] 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.

[17] E. Nickell and I. Smith. Extreme programming and software clones. In *the Proceedings of the International Workshop on Software Clones*, 2003.

[18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.