

Understanding and Aiding Code Evolution by Inferring Change Patterns

Miryung Kim
Computer Science & Engineering
University of Washington
Seattle, WA USA
miryung@cs.washington.edu

1. Research Questions

Evolution continues to play an ever-increasing role in software engineering. Although changing a program is the core of software evolution, program change patterns have not been considered as a first class entity in most classic studies of software evolution. Past empirical studies of software evolution primarily relied on quantitative and statistical analyses of a program over time [1], but did not focus on semantic and qualitative change patterns of a program. We hypothesize that by treating change patterns as first class entities we can better understand software evolution and also aid programmers in changing software. As a first step, we plan to answer three specific research questions:

- What is an effective, explicit representation of software change?
- How do we effectively identify common change patterns and from which sources?
- Can we use inferred change patterns to better understand software evolution and ultimately aid programmers in changing software?

In an effort to answer these questions, we did two empirical studies for understanding why and how programmers create and maintain duplicated code. In the first study, we captured and replayed editing operations in Eclipse IDE to understand copy and paste programming patterns (Section 2). In the second study, we built a tool that automatically extracts the genealogy of code clones from a set of program versions and studied the extracted genealogies (Section 3). Based on these two studies, we developed a representation of structural changes—refactorings and API changes—and built a tool that automatically infer changes with respect to the representation (Section 4). We intend to show that inferred structural changes can enhance software engineering research and tools. Related work in the areas of automatic clone detection, empirical studies of code clones, code matching techniques, refactoring reconstruc-

tion tools, and mining software repositories is detailed elsewhere [3, 4, 5].

2. Change Patterns from Captured Edit Logs

Although programmers frequently copy and paste code when they develop software, implications of common copy and paste (C&P) usage patterns have not been studied previously. To understand common C&P usage patterns, we conducted an ethnographic study at IBM T.J. Watson Research Center [2]. We developed a logger in Eclipse IDE that records key strokes and editing operations such as copy, cut, paste, redo, undo, and delete. And we built a replayer that can play back the edit logs captured by the logger.

The change granularity of edit logs was too low level to find any meaningful change patterns. To infer high-level change patterns and associated intentions of a programmer, we resorted to manual analysis and semi-structured interviews in addition to replaying the edit logs. Our analysis method is detailed in [2]. As a result of this study, we gathered insights about copy and paste usage patterns. Here, we discuss some of key findings that are relevant to our genealogy study in the next section: (1) Limitations of programming languages may result in unavoidable duplicates in a code base. (2) Copied text is often reused as a structural template and is customized in the pasted context. (3) Programmers often update a similar change to clones from the same origin. In other words, after they create clones, they tend to modify the structural template embedded in the clones *consistently*. Our experience of capturing, replaying, and analyzing edit logs taught us that this approach is a good way to form insights about common change patterns.

3. Extraction of Code Clone Genealogies

Through our copy and paste study, we explored an approach of capturing and replaying edits in an IDE to identify change patterns. This approach is limited in a number of

ways. First, most projects do not retain archives of editing logs, but rather they have version control systems or software release archives. Second, while most projects are developed by more than one developer in a collaborative work environment, capturing edits in an IDE is limited to a single programmer workspace. Third, a longitudinal analysis is not feasible due to high cost of analyzing edits.

Our goal is to infer clone evolution patterns *from a set of program versions* stored in a source code repository. We defined a set of common clone evolution patterns based on our insights from the copy and paste study. Then we built a tool that automatically extract the genealogy of code clones from a set of program versions [5]. In a clone's genealogy, a group to which the clone belongs is traced to its origin clone group in the previous versions. In addition, clone groups that have originated from the same ancestor clone group are connected by a clone evolution pattern.

Using our tool, we studied (1) how long clones stay in a system and (2) how often and in which way clones change in open source projects [5]. It has been broadly assumed that clones are inherently bad and eliminating clones by refactoring would solve the problems of code clones. However, our study results indicate that the problem of code clones is not so black and white and that there are several types of clones that refactoring may not be the best solution: more specifically, in our study, 49% to 64% of clone genealogies consist of clones that cannot be easily removed using standard refactoring techniques, 48% to 72% of genealogies disappeared in a very relatively short amount of a time (within an average of eight check-ins out of over at least 160 check-ins), and 26% to 34% of them disappeared because they changed differently from one another. This finding imply that aggressive, immediate refactoring may not be necessary or beneficial for all types of clones.

Instead of measuring the extent of code clones or the changes in clone coverage, we focused on how individual clones *change* over the lifetime of software. By doing so, we believe we advanced our understanding of code clones.

4. Automatic Inference of Structural Changes

For the past few years, there has been growing interest in mining source code repositories; open source projects became popular, and the use of configuration management systems has been adopted as a standard practice of software development. Advancement in data mining made it possible to analyze a large amount of information and extract patterns in the data.

Many software engineering researchers have started mining software artifacts to discover patterns of software evolution. To infer change patterns, all mining software repository projects require a technique to match code entities such as files and functions in one version of a pro-

gram to corresponding code entities in another version of a program. The accuracy of matching is crucial for constructing a continuous history of code evolution. Our goal is to solve this matching problem by inferring changes from one version to another version. As a first step, we solve the matching problem at or above the level of method headers by inferring structural changes—refactorings or API changes—from one version to another version [4].

We have evaluated the accuracy (precision and recall) of method-level matches across versions in several open source projects and compared our approach with one existing method-level matching tool and two refactoring reconstruction tools. We plan to demonstrate how inferred structural changes can be used for other applications, such as semi-automatic update of outdated patches, bug detection, documentation assistance, etc.

5. Expected Contributions

1. A demonstration of an edit capture and replay approach toward gathering insights about software change patterns, in particular, in the context of understanding copy and paste programming patterns
2. A development of a genealogy representation that describes clone evolution patterns in a series of program versions, a tool that automatically extracts clone genealogies from a source code repository, and an analysis of genealogies
3. A development of a change vocabulary that concisely describes a set of related structural changes —refactorings or API changes, and a tool that automatically infers changes with respect to the change vocabulary
4. Initial evidence that our inference technique and its results enable software engineering tools that can utilize explicit, high-level change patterns

Acknowledgment I thank Vibha Sazawal and David Notkin for comments on my draft.

References

- [1] L. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.
- [2] 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. IEEE Computer Society, 2004.
- [3] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06*.
- [4] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, 2007.
- [5] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *ESEC/SIGSOFT FSE*, pages 187–196, 2005.