

Supporting Uses of Editing Process Patterns

Miryung Kim, Vibha Sazawal, and David Notkin

Department of Computer Science and Engineering

University of Washington

Seattle WA 98195 USA

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

ABSTRACT

Programmers often use information about how code was edited as they maintain software. We believe that editing process information is worthwhile to observe and maintain because it is supplemental to a programmer's static understanding of a code base. In addition, by identifying editing patterns, we can provide customized assistance for commonly used patterns.

We propose the automatic capture of programmers' editing activities. We discuss an instrumentation infrastructure that records editing practices and provides an API to software engineering applications. Software engineering researchers could use the infrastructure to study how programmers perform tasks, and tool-builders could build on the infrastructure to provide customized support for editing patterns. We describe several software engineering tools that could employ editing process information, including several applications informed by a case study of copy and paste, a popular editing practice in programming.

Keywords

Editing patterns, Software development environment

INTRODUCTION

A single keystroke offers little information. But from a sequence of keystrokes and editing operations, millions of lines of computer software are written. Programmers often use their memory of how they edited code as software evolves. They remember that "the code looks a certain way because it was copied from another place," or "the last time I edited that file, I also edited another file," or "the last time I viewed this module, I searched for a variable defined in the first function." These recollections are then used as input when completing tasks or making decisions.

We believe that programmers' editing processes contain useful information for program understanding, which is why programmers themselves make use of their own editing

process recollections. But human recall of editing operations can be short-lived, somewhat inaccurate, and difficult to transfer from person to person. We believe that editing logs can be used to identify common editing patterns and that tools can provide customized support for such commonly used patterns.

We propose the automatic capture of editing process information. Such a system would serve as an external memory for programmers and also enable a new family of tools that provide support to programmers as they maintain software. To build confidence in our belief that editing process information is interesting and that opportunities for customization exist, we conducted an ethnographic study focused on one common editing practice -- copy and paste (C&P). We found that programmers follow a small number of patterns when they copy and paste, and that some of these C&P usage patterns reflect important design decisions in software. Our results suggest that tools customized to support these patterns could ease software evolution and provide better design decision-making support. We believe that other kinds of editing patterns are similarly promising.

We used our experiences studying C&P usage patterns to inform our design of a general-purpose infrastructure for tools that employ editing history. The infrastructure will capture editing practices, support the discovery of patterns, and provide relevant information to applications. Software engineering researchers can use the infrastructure to gather information on how programmers perform tasks, and tool-builders can use the infrastructure to provide customized support for editing patterns. The infrastructure we propose consists of two parts: a Recorder and an Information Extractor. The Recorder will efficiently log the minimal information required to reconstruct document changes performed by a programmer. The Information Extractor will regenerate the programming context associated with the low level events logged by the Recorder. For example, the Information Extractor can identify in which method of which type an editing operation took place.

We also intend to assess the suitability of our infrastructure by building software engineering applications that make use of the Recorder and the Information Extractor's APIs. The software engineering tools mentioned below are motivated

by our insights on C&P programming practices and our experiences of conducting an observational study:

- a replayer that allows researchers to study and review classes of software engineering tasks in a non-intrusive way
- a tool that logs copy-and-paste operations and then tracks the dependences induced by these operations
- a tool that extracts templates from code fragments that are frequently copied and customized

We also envision tools that employ other types of editing process information:

- a tool that supplements a programmer's understanding of dependencies using information extracted from the editing process
- a programming by demonstration tool [3] in the domain of program editing

The rest of this paper is organized as follows. First, we summarize results from our study of copy and paste programming practices. Second, we describe how insights from the C&P study support the hypothesis that editing process information is significant and should be preserved.

Next, we present the functionality of the proposed infrastructure in more detail. Finally, we present several software engineering tools that can be implemented using the API provided by our proposed infrastructure.

ETHNOGRAPHIC STUDY OF COPY AND PASTE PROGRAMMING PRACTICES

Although copy and paste (C&P) is commonly used, the implications and usage patterns of C&P practices while programming have not been studied previously. We conducted an ethnographic study to identify common C&P usage patterns and explore how such patterns could be supported by tools.

We first observed programmers directly by watching them program "over-the-shoulder." Because it was difficult to manually log editing operations performed by the subjects, we frequently interrupted their programming flow to ask them to explain what they were copying and pasting and why. To avoid such interruptions, we then developed a logger to record programmers' editing operations in a non-intrusive way. We also developed a replayer that plays the editing logs captured by the logger. By replaying logs of editing operations captured by the logger, we were able to quickly document and analyze each copy and paste task. Each subject then met with us to confirm our interpretation of their tasks. Details about the observation setup are summarized in Table 1. To analyze our results, we used techniques from Contextual Inquiry [1]. Patterns of copy and paste tasks were induced from the detailed notes produced about each copy and paste operation. Our

complete taxonomy of copy and paste usage patterns is presented in [2].

	Direct Observation	Observation using a logger and a replayer.
Subjects	Researchers at IBM TJ Watson	
No. of Subjects	4	5
Coding Hours	About 10 hours	About 50 hours
Interviews	Questions asked during observation	Twice after analysis
Programming Languages	Java, C++, Jython	Java

Table 1. Study Setting

SIGNIFICANCE OF EDITING PROCESS INFORMATION

Our study of copy and paste programming practices provides evidence that editing process details are used by programmers and worthwhile to observe and maintain. We also found that programmers follow a small number of editing patterns and that these patterns provide opportunities for customized tool support.

Does the existence of editing patterns provide opportunities for assistance?

Editing patterns that are potentially error-prone can be supported or automated.

In the C&P study, we observed multiple programmers copy an entire code fragment and then remove code that was irrelevant to the pasted context. In other words, programmers copy entire code fragments, but they intend to reuse only the structural template contained in the code fragment. These structural templates can be either reusable syntactic elements or reusable programming logic. Examples of reusable programming logic include design patterns, usage of an interface, implementation of an interface, and complicated control structures.

We noticed that cautious programmers modified the portion of pasted code that was specific to the current intended use immediately after they copied and pasted. Failing to modify irrelevant portions of copied code can incur identifier naming conflicts or logical errors. We believe that tools can assist this "copy-then-remove" pattern by identifying the structural template of a copied code fragment and then removing the irrelevant code automatically.

How do programmers use editing process information?

Programmers use editing process information as they make changes to existing code and decide when to restructure code.

In the C&P study, we noted that programmers used their memory of copy and paste dependencies when they applied

consistent changes to duplicated code. If code has been copied and pasted to reuse a structural template, all the structural clones must be modified when that template evolves. When programmers lose knowledge about where structural clones are located and what the template of a set of structural clones is, they may produce defects when making changes to the software.

Programmers also use their memory of copy and paste history to determine when to restructure code. Programmers restructure code after they copy and paste the same code fragment several times. A few programmers told us that they deliberately delay code restructuring until they copy and paste several times, because such reuse helps them discover the right level of abstraction. We suspect that large or frequently copied code fragments are good candidates for refactoring.

Why are editing patterns worthwhile to observe and maintain?

Tools can use editing process information to support error-prone processes or suggest refactorings as described above. Editing process information also reflects design decisions made by a programmer during the editing process, and thus can be used for understanding a program.

In the C&P study, we observed that C&P dependencies often reflect design decisions made by a programmer during the editing process. We observed that when a code fragment was copied from A and pasted to B, other code fragments were also copied from A and pasted to B. We believe that code fragments are often copied together because they belong to the same functionality or concern. Some examples of these dependencies include caller/called methods and grouped operations such as `writeToFile()`, `closeToFile()`, and `openToFile()`.

We investigated the dependencies between a copied code fragment and a pasted code fragment to learn why programmers choose particular code fragments as templates. A code fragment is chosen as a template for another code fragment when both code fragments access a similar data structure, or they crosscut a code base in the similar way, or they inherit from the same superclass. Based on our analysis, we conclude that C&P dependencies reflect important design decisions such as crosscutting concerns, feature extensions, paired operations, semantically parallel concerns, and type dependencies (such as inheritance). Preservation of editing operations can enable tools that "remember" previously undocumented design decisions, because the decisions can be inferred from editing process details that have been saved.

INFRASTRUCTURE TO SUPPORT USES OF EDITING PATTERNS

From our case study of copy and paste programming, we conjecture that various kinds of editing process information

other than C&P can also be used to provide customized assistance to programmers. We intend to build an infrastructure to support both the research community's investigation of editing process information and the use of editing process information by software engineering tools. The logger and replayer that we have developed for the C&P study will serve as a basis to implement this infrastructure.

Recorder

The Recorder will log all information needed to reconstruct document changes performed by a programmer. We plan to develop the Recorder by extending the Java Editor of the Eclipse IDE. The Recorder will capture the initial contents of all documents opened in the workbench and will log all operations that create incremental changes to the documents. The instrumented editor will record the type of editing operation, the file name of the document, the length and offset of relevant text, and a time stamp. When editing operations such as copy, cut, paste, delete, undo, and redo are performed, the selected text ranges will also be recorded. When the Eclipse IDE performs document changes triggered by automated editing operations (such as the "Organize Imports" command), the Recorder will log the type of the command and the relevant text.

Information Extractor

Low-level editing information, such as the entry of a keystroke, requires the context of a program for proper interpretation. For example, the length and offset of copied text does not provide enough information to monitor future changes to that copied text. Providing semantic information, such as the enclosing scope of the text involved in an editing operation, is needed to ease the construction of applications that make use of editing process information.

To regenerate the programming context involved in document changes, we propose the Information Extractor. The Information Extractor will determine the method or static block in which the document change occurred. The enclosing type of the relevant method or static block will also be determined. Clients will then use the Information Extractor's API to access semantically meaningful information about the editing process. We intend to carefully design a straightforward API that encapsulates the voluminous low-level details of the editing process while remaining useful to a broad range of clients.

POSSIBLE SOFTWARE ENGINEERING TOOLS THAT EMPLOY EDITING PROCESS INFORMATION

Support for Programmer Observation

We envision the use of the Recorder and Information Extractor as a platform to conduct observational studies of programming practices. Observational studies are the most effective way to gauge how programmers interact with their tools and perform actual tasks. However, observational

studies are challenging to complete. The presence of an observer in the room is distracting and can substantially alter the work environment. Videotaping is an alternative that removes the physical observer but adds additional privacy concerns.

A replayer is an attractive, lightweight alternative. By adding a few controls such as play, stop, and jump to the Information Extractor, we can replay document changes previously performed by a programmer. With a replayer, researchers can efficiently "observe" programmers at their own pace offline.

Support for C&P Programming

The Recorder and the Information Extractor can be used to log copy and paste operations. As mentioned above, tools can then use this information to learn "structural templates" by observing repetitive duplication of code fragments followed by modifications (the "copy-then-remove" pattern.) When a programmer intends to duplicate a "structural template" via C&P, the tool can provide advanced statement (or block) completion or assist in removing the portion of pasted code that is not part of the structural template. Structural templates can also be used to customize the automatic restructuring feature currently offered by the Eclipse IDE. The tool can suggest possible candidates for refactoring, warn a programmer when he/she attempts to change a structural template, or propagate changes to other uses of a structural template automatically. A tool can also notify other programmers when a structural template changes. These features can prevent inconsistency within a code base.

Support for Program Understanding

Editing process information can be used to create new tools that support program understanding and design decision-making. These tools can supplement and improve programmers' existing use of editing operations to recall important design decisions. For example, a tool could infer that a dependency exists between two files if they are often edited together. A visualization of such empirically observed dependencies could assist programmers as they try to understand a new system or make changes to an existing code base. Existing dependency visualization systems that use invocation or containment could also be augmented with editing process relationships.

To capture the editing process, the infrastructure will retain knowledge about the order in which document changes are made. This feature enables the creation of tools that provide fine-grained information about the true age of code fragments. This fine-grained information could be used to

understand the stability of various fragments of code and also to infer which design decisions were made first. Knowledge about the churn rate of code or the age of existing design decisions is useful to programmers when they make new design decisions. Programmers could also review the evolution of code from a past state to its current state in detail.

Support for Programming by Demonstration

Programming by demonstration (PBD) tools construct a generalized program that accomplishes a user's task when the user demonstrates the result of a computation. PBD can be applied in the domain of program editing to identify repetitive patterns and assist programmers automatically by performing such patterns. PBD has been successfully applied in the domain of repetitive text editing [3]. We believe that the Information Extractor and the Recorder can ease the development of PBD tools that identify program editing patterns. Commonly used editing patterns can be replaced by macros. The discovery of frequent patterns could also be used to improve programming language designs.

CONCLUSION

From our case study of copy and paste programming practices, we concluded that editing process information can be used to design customized tools that support software evolution tasks. We propose an infrastructure that records editing history and abstracts logged editing information. Given this infrastructure, editing process information can be used by software engineering tools to identify common patterns, reduce inconsistent changes to software, and increase programmers' understanding of software. The proposed infrastructure will provide a platform for collecting large amounts of data, and study of that data by us and other researchers will uncover even more uses of editing process information.

ACKNOWLEDGMENTS

We thank Lawrence Berman and Tessa Lau for their advice and assistance with the ethnographic study.

REFERENCES

1. H. Beyer and K. Holtzblatt, Contextual Design. Morgan Kaufmann Publishers, 1998.
2. M. Kim, Ethnographic Study of Copy and Paste Programming Practices in OOPL. Qualification Exam Report, University of Washington, Oct. 2003.
3. T. Lau, Programming by Demonstration: a Machine Learning Approach. PhD thesis, University of Washington, 2001