A Change-Aware Development Environment by Recording Editing Operations of Source Code

Takayuki Omori
Graduate School of Science and Engineering
Ritsumeikan University, Kusatsu, Japan
takayuki@fse.cs.ritsumei.ac.jp

Katsuhisa Maruyama
Department of Computer Science
Ritsumeikan University, Kusatsu, Japan
maru@cs.ritsumei.ac.jp

ABSTRACT

Understanding a program and its evolution is not satisfied only by looking at a current snapshot of its source code. Thus, a developer often examines a sequence of its snapshots stored in repositories of versioning systems, and identifies differences between two successive snapshots. Unfortunately, such differences do not represent individual changes of the source code. This paper proposes a mechanism for recording all editing operations a developer has applied to source code on an integrated development environment. The paper also shows a running implementation of the mechanism built as an Eclipse plug-in, which is called OperationRecorder. The experimental results with a small-scale program substantiate that it has a practical use from the viewpoint of its performance.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Integrated environments; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Version control; D.2.3 [**Coding Tools and Techniques**]: Program editors

General Terms

Management

Keywords

Integrated development environments, Versioning systems, Software evolution, Source code modification

1. INTRODUCTION

In software maintenance, it is important for developers to understand a program that they will extend or modify. Existing file-based or snapshot-based versioning systems, such as Concurrent Versions System (CVS) or Subversion, help this maintenance activity since they keep track of changes performed on source code of the program [3]. The developers have a chance to obtain knowledge of how the program has been ever evolved (e.g., extended, modified, or improved) by analyzing its changes.

However, this analysis task is in general troublesome [12]. This is because changes between two snapshots might be tangled each other. Unfortunately, the conventional versioning systems neither individually store changes nor untangle entwined ones. Therefore, they still have a crucial limitation that loses useful information on past changes [9]. To overcome this limitation, a number of mechanisms for identifying differences between two programs have been proposed [6]. Although some of them achieve high accuracy differencing, their effects cannot be guaranteed for every source code.

We believe that separate changes are useful for developers to understand a program which the changes have been applied to. From this viewpoint, the problem is that the conventional versioning systems unwisely dismiss editing operations between two snapshots of the program and casually fold every change into a single transaction. Moreover, almost all program analysis techniques can be effectively utilized only when snapshots before and after the transaction are both compilable. These problematic factors make it harder to capture the minimal set of individual changes. To sweep them, a development environment recording a complete history of logical editing operations is essential [6, 9]. The history-based differences facilitate developers capturing changes which are more understandable than tangled ones.

This paper proposes a mechanism for providing history-based changes by recording all editing operations a developer has applied to source code on an integrated development environment (IDE). We have developed a running implementation of the proposed mechanism built as an Eclipse plug-in, which is called **OperationRecorder**. All the recorded operations are stored into a repository (database) so that they can be easily retrieved. In the paper hereafter, a change indicates either an editing operation or a composite of editing operations. Introducing OperationRecorder in development, developers can easily obtain all changes of source code without loss of its modification information.

Here, we should mention that the basic concept of change-aware IDE recording editing operations is not novel. OperationRecorder is inspired by SpyWare [9] which is an IDE plug-in for a Squeak environment. Although both approaches are almost same with respect to change-aware plug-ins of IDEs, respective mechanisms are different in two features. One is that OperationRecorder extracts editing operations from an undo history managed by an Eclipse editor. The other is that OperationRecorder can handle not only compilable source code but un-compilable one, whereas SpyWare directly handles operations for an abstract syntax tree (AST) derived from compilable source code. In other words, OperationRecorder does not assume the existence of an AST.

The remainder of the paper is organized as follows: Section 2 describes conventional techniques for identifying differences between two programs. Section 3 describes an over-view of OperationRecorder and explains the details of its implementation. Section 4 shows several results on experimental development with Op-

MSR'08, May 10-11, 2008, Leipzig, Germany.

erationRecorder. Finally, Section 5 concludes with a brief summary and future work.

2. DIFFERENCING MECHANISMS

Existing versioning systems use a differencing tool for identifying changes between files. A famous differencing tool is a UNIX *diff* utility or its variants. Unfortunately, it naively compares respective text files of two programs without respect to their syntactical structure and then presents only added and/or deleted lines between the programs.

Several differencing techniques have been proposed to remove the limitations of a simple line-based textual differencing [6]. For example, techniques [4, 7, 10] structurally compare ASTs of two versions of source code. Semantic-based differencing techniques use control flow graphs, program dependence graphs, or abstract semantic graphs of source code [2, 5, 8]. A tree-based or graph-based technique can identify more exact differences between two programs than a text-based one by exploiting structural equality (or similarity) of their trees or graphs. Moreover, a technique [3] compares the sets of lines added and deleted in a *diff* change set, combining the uses of Vector Space Models and the Levenshtein distance.

Although these techniques succeed at identifying differences between two programs, resulting differences are insufficient for appropriately capturing consecutive changes done in the past. This is because, in development using versioning systems, a single transaction between two commitments contains frequently multiple modifications. That is, a delta between two snapshots stored in CVS or Subversion contains multiple changes. The differencing mechanisms are weak in such delta and tend to fail in its identification. A technique [11] tries to avoid this problem by automatically committing source code to a secondary CVS repository at short intervals. However, this idea does not guarantee that only one change is stored in the specified interval.

In summary, the larger the delta is, the harder it is to distinguish individual changes by using a differencing mechanism [9]. If a delta contains an interference of multiple changes (which were performed at the same location), the conventional differencing techniques are almost impossible to identify individual changes from the delta. In this case, they might require expensive matching algorithms to extract individual changes.

3. CHANGE-AWARE IDE

OperationRecorder is a plug-in of Eclipse, which provides the functionality of recording editing operations performed on an editor and distilling changes from the recorded operations. In this section, we first overview its architecture and behavior, and then explain the details of its implementation.

3.1 Overview of OperationRecorder

An overall architecture of OperationRecorder is shown in Figure 1. Its implementation contains about 2,000 non-comment, non-blank lines of code written in Java. An extended Java editor is derived from a regular Java editor of Eclipse. OperationRecorder collects in the backend all editing operations a developer has applied to every source file. A history of the collected operations is stored in a MySQL database. Moreover, OperationRecorder automatically outputs a backup copy of each source file so as to make the operation history robust. The editing operations stored in the database and the backup files are utilized by any tool that a tool vendor provides or a developer uses.

The recording process requires no special task of developers during their editing, which is analogous to usual development. When a source file is saved, OperationRecorder extracts editing operations from the undo history of Eclipse for the file. When the file is closed, OperationRecorder stores its operation history into the MySQL database.

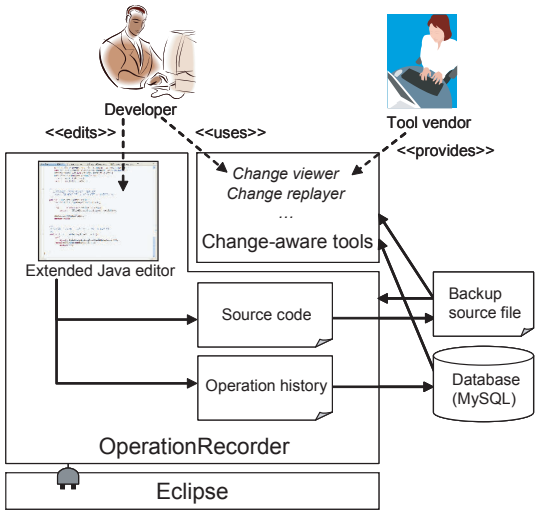

Figure 1: Architecture of OperationRecorder.

3.2 Recording Mechanism

OperationRecorder records editing operations that directly affect source code on the editor. The operations include manual typing (insertion, deletion, and modification of a character) and manual editing (cut and paste of a text). They also include automatic completions or refactorings. OperationRecorder excludes operations that do not change the contents of source code, such as the movement of a cursor, selection of a text, file opening and closing, switching a file currently edited, file renaming, and undoing/redoing. Every editing operation is obtained from information which Eclipse stores in its undo facility. In general, the low-level modifications such as developer's keystrokes are too primitive to understand. The employed undo facility aggregates successive keystrokes at neighboring locations on source code. As a result, a recoded operation has the potential to represent a coherent change.

OperationRecorder utilizes the following three classes packaged in `org.eclipse.core.commands.operations`, which implement the interface `IUndoableOperation`. The instances of these classes are obtained from an undo history by using the Java reflection API.

(1) `UndoableTextChange`

This class represents an elemental undoable operation editing a single character or continuously multiple ones. It contains information on the location (the offset value indicating) where a change is applied and a sequence of inserted or deleted characters (inserted or deleted text). When a developer replaces a part of the original text with a new one, the old and new texts are preserved as a deleted text and an inserted one, respectively.

(2) `UndoableCompoundTextChange`

This class represents a composite of undoable operations performed by automatic completions such as code assist, Java content assist, quick fix, or formatting provided by Eclipse. For example, when the Java content assist complete developer's input, the editor stores an elemental operation which inserts the input text and successively stores an elemental operation which replaces the in-

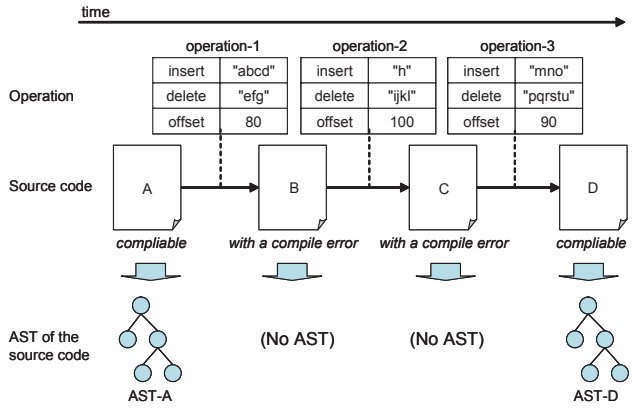

Figure 2: Offsets and their adjustment.

put text with the completion text. These two operations are compounded in one composite operation, each of which is stored in an instance of `UndoableTextChange`. An instance of `UndoableCompoundTextChange` nests instances of the the classes that implement `IUndoableOperation`.

(3) `TriggeredOperations`

This class represents an undoable operation triggered by a special action such as selection of a menu item. Refactoring is an explanatory example of this operation. Each of the changes is stored in an instance of a class that implements `IUndoableOperation`. An instance of `TriggeredOperations` nests all the stored instances and preserves the name of the applied refactoring.

3.3 Distillation Mechanism

OperationRecorder tracks the offsets of changed parts of source code and tries to associate all editing operations with respective nodes of its AST. An offset indicates the location where the operation was performed, and is recorded together with the operation. An AST contains syntactic information on source code and is useful for obtaining a variety of semantic information. Moreover, OperationRecorder adjusts recorded offsets according as the length of an inserted or deleted text. Thus, it bridges a gap between ASTs that respectively derived from two compilable contents of source code even if there are several incomplete contents between the two compilable contents.

In OperationRecorder, an inserted or deleted text stored in an elemental operation is separately associated with an AST node. For the inserted text of an operation, OperationRecorder uses the AST of compilable source code immediately after the operation, which is called a *succedent AST*. Conversely, for the deleted text, it uses the AST of compilable source code immediately before the operation, which is called a *precedent AST*. Each operation has always a precedent AST and a succedent one by preparing an empty AST derived from source code with no content. The reason why OperationRecorder uses a succedent AST for an operation in order to adjust the offset value of its inserted text is that the succedent AST is more likely to have a node corresponding to the inserted text than its precedent AST. OperationRecorder uses a precedent AST for an operation to adjust the offset value of its deleted text since its succedent AST has never a node corresponding to the deleted text.

See Figure 2 to understand the offset adjustment mentioned above. There exist four contents of source code, two of which (A and D) are compilable but the remaining two (B and C) have any compile error. Moreover, two ASTs (AST-A and AST-D) are derived from A and D, respectively. Three editing operations were recorded

Table 1: Database schema of the operation history.

(a) Elemental table

Column Name	Description
time	When the operation was performed
xth	Suffix of identifier (ID)
developer	Who performed the operation
file	Changed file in the Eclipse workspace
sort	"inserted", "deleted", or "modified"
start	Offset for the starting point of the text
end	Offset for the ending point of the text
inserted	Inserted text
deleted	Deleted text
parent	Reference to a composite record
classNameForward	Class name for the inserted text
methodNameForward	Method name for the inserted text
nodePathForward	AST node for the insertion
classNameBackward	Class name for the deleted text
methodNameBackward	Method name for the deleted text
nodePathBackward	AST node for the deletion

(b) Composite table

Column name	Description
compositeID	Identifier (ID)
time	When the operation was performed
refactoringName	Name of the applied refactoring, etc.

between two successive contents. Each operation has its inserted text, deleted text, and offset value. For example, the inserted and deleted texts for operation-2 are "`h`" and "`ijkl`", respectively. Moreover, its offset value is equal to 100. The precedent or succedent AST for all the operations is AST-A or AST-D since these two ASTs are compilable and the operations are enclosed between them innermost. The offset value of the inserted text of operation-2 is affected by operation-3 since its value is more than that of operation-3 ($100 > 90$). The adjusted offset value is calculated as follows:

$$100 + 3 \text{ (for "mno")} - 6 \text{ (for "pqrstu")} = 97$$

The inserted text of operation-2 is associated with an AST node whose textual range (with its starting and ending points) encloses the offset value 97 in AST-D. Similarly, the offset value of the deleted text of operation-2 is affected by operation-1 since its value is more than that of operation-1 ($100 > 80$). The adjusted offset value is calculated as follows:

$$100 + 3 \text{ (for "efg")} - 4 \text{ (for "abcd")} = 99$$

The deleted text of operation-2 is associated with an AST node whose textual range encloses the offset value 99 in AST-A.

Note that not all operations require this adjustment. For example, the offset value of the inserted text of operation-1 is affected by neither operation-2 nor operation-3 since its value is less than those of operation-2 and operation-3 ($80 < 100$ and $80 < 90$).

3.4 Database Schema

Table 1 shows a schema of information which OperationRecorder stores in the MySQL database, consisting of an elemental table and a composite one. The elemental table preserves information on all elemental operations, which is obtained from an instance of the class `UndoableTextChange` mentioned in Section 3.2. The composite table preserves information on all composite operations that dangle several elemental operations the information of which is stored in the elemental table. This information is extracted from an instance of the class `UndoableCompoundTextChange` or `TriggeredOperations` mentioned in Section 3.2.

The combination of the time and xth columns is the primary key of the elemental table (xth is a suffix distinguishing multiple operations performed in the same time). The compositeID column is the primary key of the composition table. The last six columns in Table 1(a) denote syntactic information of each editing operation. An AST node of the columns nodePathForward and nodePathBackward is represented by a sequence of AST nodes concatenated with the slash symbol ("/"), which are on a path of the AST from its root to the specified AST node.

4. EXPERIMENTAL RESULTS

To assess the performance of OperationRecorder, we made an experiment with a small-scale program and collected actual editing operations during its development (coding). This program implements a reversi game as a Java applet, consisting of four source files. The total size of the files is equal to 618 LOC (13,239 bytes), containing comment and blank lines. As a result, a total number of 2,023 records in the elemental table and 278 records in the composite table have been collected in 3 hours and 5 minutes.

Through this experiment, we obtained the following results and insights.

(1) Storage Size

The total size of the operation history stored is 478,236 (47,1842 for an elemental table and 6,394 for a composite table) bytes, which were calculated based on the sizes of respective data size [1]. Moreover, OperationRecorder always generates a backup file whose size is equal to that of the original source file. Consequently, the total size of data stored in the experiment is 504,714 (13,239 + 13,239 + 478,236) bytes. It is approximately 38 (504,714/13,239) times larger than that of the original source files.

Further storage capacity will be consumed in long-term development. Moreover, the execution time of database search is a debatable subject since this performance much affects the feasibility of a tool using OperationRecorder.

(2) Accuracy

We have developed a change replayer using OperationRecorder, which can detect an error resulting from inconsistent recovery of a snapshot. Using this tool, all the source files were restored without any recovery error. Moreover, the contents of each of the restored source files are the same as those of its corresponding source file which was being edited at that time. This fact shows that a complete sequence of editing operations was stored in the database without loss of any operation and all the stored operations were consistent with each other in this experiment.

A close look at the recorded operations reveals a problematic modification between neighboring two versions of source code, which inserts a new text and then deletes a part of the inserted text. In this case, OperationRecorder surely records these two editing operations. However, it associates the deleting operation with an inadequate AST node. This might confuse a developer in understanding the actual change.

(3) Effects on Coding

The recording process obviously increases response time in the interaction with an editor; nevertheless a developer (one of the authors) did not get frustrated in this experiment on a computer with a Pentium4 3.2GHz CPU and a 2GB of RAM, running Windows XP and Eclipse 3.2. The MySQL server was running on the same computer as OperationRecorder.

This shows that OperationRecorder has a practical use on development where each source file is closed in a short period of time. However, long-term editing without closing files might annoy a developer and adversely affect its usability.

5. CONCLUSIONS

Individual changes of a program, consisting of fine-grained editing operations, facilitate a developer understanding the program and its evolution. This paper proposes a mechanism for recording all editing operations which have been applied to source code on an IDE. We have developed OperationRecorder which is a running implementation of the proposed mechanism built as an Eclipse plug-in. To demonstrate ease of development of change-aware tools, we have been implementing them using OperationRecorder.

In the current version of OperationRecorder, all editing operations are simply extracted from the undo history of Eclipse; therefore, they are sometimes considered too crude for program understanding. We are addressing this problem by grouping some of the recorded operations based on their spatial distance (the distance of positions where they were performed on source code) and/or temporal distances (the interval of times when they were performed). Information on the dependencies between program elements (or AST nodes) involved by the same edit operation or different ones is also useful for such grouping. From the viewpoint of accuracy and usefulness of OperationRecorder, we must make a large number of experiments with various sizes of programs and check its applicability. The support for development where multiple developers collaboratively write a program by using CVS or Subversion is future work.

6. REFERENCES

[1] MySQL 5.0 Reference Manual, 9.5 Data Type Storage Requirements. http://www.mysql.com/.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proc. ASE'04*, pages 2–13, 2004.

[3] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proc. MSR'07*, page 14, 2007.

[4] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, 2007.

[5] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proc. PLDI'90*, pages 234–245, 1990.

[6] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *Proc. MSR'06*, pages 58–64, 2006.

[7] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proc. MSR'05*, pages 1–5, 2005.

[8] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *Proc. ICSM'04*, pages 188–197, 2004.

[9] R. Robbes. Mining a change-based software repository. In *Proc. MSR'07*, page 15, 2007.

[10] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar java classes using tree algorithms. In *Proc. MSR'06*, pages 65–71, 2006.

[11] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proc. MSR'04*, 2004.

[12] W. Yang. Identifying syntactic differences between two programs. *Software–Practice and Experience*, 21(7):739–755, 1991.