A Survey on Methods of Recording Fine-grained Operations on Integrated Development Environments and their Applications*†

Takayuki Omori
Department of Computer Science,
Ritsumeikan University

Shinpei Hayashi
Department of Computer Science,
Tokyo Institute of Technology

Katsuhisa Maruyama
Department of Computer Science,
Ritsumeikan University

Abstract

This paper presents a survey on techniques to record and utilize developers' operations on integrated development environments (IDEs). Especially, we let techniques treating fine-grained code changes be targets of this survey for reference in software evolution research. We created a three-tiered model to represent the relationships among IDEs, recording techniques, and application techniques. This paper also presents common features of the techniques and their details.

1 Introduction

Software has to keep changing according to changes in user requirements and the external environment even after shipment [35, 48]. Software developers and researchers rarely have information about such *software evolution* [78]. Thus, not understanding how and why systems have been changed is a real problem [37]. Moreover, it is said that over 80% of the total cost of software arises after shipment. Therefore, elucidating how and why software evolution occurs is important also from the economic perspective.

Many studies have targeted time-series changes of source code from the perspective of program comprehension. Most leveraged information derived from version control systems (VCSs), such as CVS, Subversion, and Git. However, several researchers have pointed out their limitations [42, 53]. To understand software changes and their underlying intentions, not only understanding the details of how source code was changed but also understanding how and when the developer worked on the software project is helpful [76]. Robbes et al. took the fact that traditional software evolution studies have been based on VCS data as a problem and propounded change-based software evolution [53]. Change-based software evolution treats software evolution as history composed of developers' code changes recorded on an integrated development environment (IDE). Thus, change-based software evolution allows us to analyze finer-grained evolution that cannot be done by traditional approaches. Furthermore, the change history can be used for analyzing developers' usage of development support tools so that these tools can be improved based on the analysis.

This paper introduces methods for recording fine-grained operation history on IDEs and their applications. Since the paper would be mainly referred in software evolution studies, it is focused on methods for recording details of source code changes. We call a method of this survey's target an *operation-oriented* method hereafter[1]. Several recording methods have

*This paper is a translated version of a Japanese reviewed paper [86]. The electronic copy of the original version can be obtained from http://www.jstage.jst.go.jp/article/jssst/32/1/32_1_60/_pdf.

[1]Ebraert et al. used the term *change-oriented* [10]. Since this paper targets methods that can treat not only edit operations but also interactions (defined in Section 2), we use the term operation-oriented for this paper.

been proposed according to a specific IDE and programming language. Application methods have also been proposed based on these recording methods. However, to the best of our knowledge, there has not been a comprehensive survey of such methods at this point. Since the target of this paper spreads into various research fields, such as software maintenance and evolution, program comprehension, mining software repositories, automated software engineering, and human aspects, the authors believe a survey paper on them would be particularly valuable.

Operation-oriented methods discussed in this paper treat the operation history developers conducted on IDEs and recorded in chronological order. The history includes all changes performed on source code. There are several methods for recording developers' behaviors or code changes in narrower time ranges than VCSs that do not record details of edit operations on IDEs (e.g., [28, 29]). However, the authors considered these methods are not suitable for investigating software evolution since they have several problems regarding understanding code changes, as mentioned in Section 3. Therefore, this paper is not focused on those methods. However, the authors give a brief overview of several methods in Section 6 since the authors surveyed them for this study for narrowing down the survey targets. Survey papers regarding interaction history without edit operations include the book by Maalej et al. [36], which surveys methods for recording operation history for applying recommendation systems and the paper by Sarma [61], which targets supporting collaborative software development.

We classify operation-oriented methods into two types: recording and application. A recording method is used for obtaining and recording operation history derived from an IDE. An application method is used for supporting software development activities with data recorded by a recording method and/or for analyzing the recorded data. An IDE, recording method, and application method can be modeled as the three-tiered model shown in Fig. 1. An application method depends on the recording method, which generates history data. A recording method depends on an IDE, i.e., the base of recording. Hence, we have to determine the prerequisites, such as which IDE we should select, to use an operation-oriented method. Therefore, this paper shows tables that show each method and its dependent method in Sections 4 and 5.

This paper is structured as follows. Section 2 defines the terms used in this paper, and Section 3 presents the characteristics of operation-oriented methods, comparing traditional methods based on revision history. Section 4 introduces recording methods, and Section 5 introduces application methods. Section 6 introduces operation-recording methods outside the scope of this survey. Section 7 presents discussions about the format of operation history, leveraging it, and the public use. Section 8 concludes this paper.

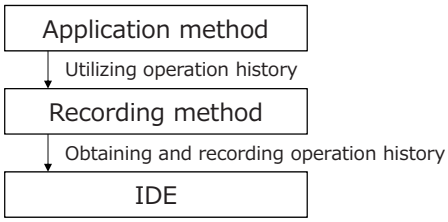

Figure 1: Three-tiered model of operation-oriented methods.

2 Terminology

This section explains terms that appear in this paper. The authors carefully defined each term not to contradict definitions in other studies, though the definition of each term depends on each study. The authors found that several studies have different definitions of a term. For example, [36] defines the term "interactions" as including "edits".

An *operation* is a developer's activity performed on an IDE. It can be classified as an edit or interaction.

An *edit* is an operation that changes source code content. It involves both a manual change and automatic one provided as an IDE function. This paper does not treat a change in file content except for source code.

An *interaction* is an operation that invokes a function provided by the IDE, such as automated refactoring, code completion, opening and closing a file, undo, redo, build, execution, and launching a debugger. An interaction itself never changes source code content. When invoking a function with code change (e.g., refactoring), we deem that an edit operation occurs accompanied by the interaction. We consider a method that does not record source code changes, such as Mylyn [29], as collecting interactions.

A *history* is time-series sequence(s) of operations. An edit history and interaction history respectively

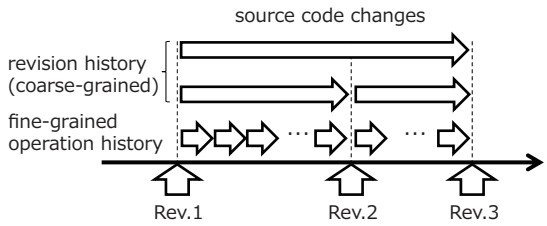

Figure 2: Revision history and fine-grained operation history.

mean a history of edits and interactions. They are differentiated from a revision history mentioned later. In case a history involves both edits and interactions and we do not need to differentiate the operation types, the history is called an operation history.

A *change* means a developer's activity or an execution of an IDE's function that causes changing source code content. It also involves a change in a syntax tree that accompanies the code change.

A *snapshot* means a state of source file content at a particular time.

A *revision history* represents how source files have been changed. In many studies, a revision history was obtained by calculating the differences between two snapshots. As Fig. 2 shows, differences between two consecutive snapshots correspond to code change resulting from multiple fine-grained operations (edits). Therefore, individual edits cannot be obtained using only a revision history.

In this paper, the word *fine-grained* means the unit of operation recording is finer than a revision history so that every source code change on an IDE can be traced using a fine-grained operation. However, actual units of recorded changes have little differences among recording methods. For example, Fluorite [74] mentioned in Section 4 records developers' keystrokes. In contrast, Syde [20] organizes changes in the unit of an abstract syntax tree (AST) node (syntax element).

3 Characteristics of operation-oriented methods

This section presents the characteristics of operation-oriented methods. Since all recording methods in this paper are aimed at solving problems in using revision history, most description is based on comparison with using revision history.

(1) Suitability for understanding code changes

Fluri et al. pointed out that traditional software evolution research depends on revision history preserved in CVS repositories and code changes are not associated with program elements [12]. In their paper, ChangeDistiller, a tool for restructuring a revision history as operations, such as insertion and deletion of AST nodes, was proposed. This is similar to operation-oriented methods in this paper. However, their method does not record operations actually performed on an IDE. Hence, it cannot precisely retrieve the order of performed operations.

Revision history stored in VCS repositories entangles multiple changes into a single one since changes can be obtained as the difference between revisions. Hence, we cannot determine the chronological order (and time) and intentions of individual changes. As a result, understanding code changes becomes difficult [32, 53]. Several researchers have pointed out the difficulty in tracing a program element between revisions, especially when rename and move refactorings were performed [53, 63, 79].

Negara et al. reported that 37% of code changes do not reach VCSs, that is, they are shadowed (overwritten); thus, a part of the history is lost [42]. Herzig et al. investigated five Java open source projects and found that at least 7–20% of bug fixes are tangled changes [26]. By using operation history, such information can be precisely obtained. Maruyama et al. proposed a method for extracting an operation history slice corresponding to changes in individual methods or classes, which are generally tangled with other changes [38].

(2) Ability of restoring past source code

We can easily proceed and rewind source code states by using operation history [53]. Text editors can generally restore past states of source code by its undo function. However, source code states prior to the last close of the editor cannot be restored because the undo history is discarded. Moreover, the undo function can rewind only the most recent code change at once [72]. On the other hand, a VCS repository keeps only source code committed by developers. Hence, we cannot restore other source code states [72], whereas we can restore any source code state with operation history.

(3) Availability for investigating tool usages

Several operation-oriented methods can record how developers used IDEs. Vakilian et al. pointed out that

existing information, such as revision history, is insufficient for investigating the use of automated refactoring functions provided by IDEs [71]. For example, by using only revision history, we cannot precisely know whether a refactoring was performed manually or automatically.

(4) Dependency on programming languages

In general, a VCS manages source code in a language-free format, such as files. Hence, they can ensure independency from a programming language [53]. In addition, they can also manage other types of resources. Most of the operation-oriented methods this paper introduces record code changes in a unit of a syntax element. In such methods, we can determine what element was changed without parsing. However, they impose additional work for supporting other languages.

(5) Dependency on IDE

A revision history can hold code changes regardless of whether they were performed on an IDE. All the operation-oriented methods in this paper record on their target IDEs and do not record changes performed outside of the IDEs. Moreover, they cannot be used in other IDEs as is.

(6) Necessity for protecting privacy

Since operation history often includes privacy-sensitive information, we need to be careful in disclosing history data. Such information includes, for instance, a conversion history in kanji conversion software and pasting unintended characters due to a mistake[2]. Hayashi et al. claimed the necessity of preprocessing prior to disclosing such data [83]. Vakilian et al. and Negara et al. mentioned the difficulty in assembling research participants due to privacy issues [39, 71].

4 Methods for recording operations

To tackle problems in revision history and precedent methods for recording operation history, recording methods have been proposed, as shown in Table 1. Each recording method collects different information based on its base IDE and main purpose of applications. This section presents the background of each recording method and what types of information it can record. Each subsection presents one recording method. Since EclipsEye and ChEOPSJ are respectively ported versions of SpyWare and ChEOPS to work on Eclipse, they are presented in the same subsections of the respective base tools.

[2]Based on the authors' experiences.

4.1 SpyWare

Robbes et al. developed SpyWare, a tool for recording developers' source code edits performed on the Squeak IDE [50, 51, 53].

Version control systems record only source code snapshots at developers' commits and do not hold changes performed between the commits. Thus, research on software evolution has been severely limited. Moreover, IDEs are not equipped with software evolution analysis tools. Hence, they are separated from actual programming activities. Robbes et al. claimed that collecting precise evolution data is necessary for effective software evolution research.

A main characteristic of change-based software evolution propounded by Robbes et al. is that a program change is treated as a first-class entity, that is, it can be (dynamically) generated in runtime and assigned to the value of an argument, return value, or variable.

Recording changes is performed by hooking the IDE's change notifications. A change is represented as an atomic change (finest operation) or more high-level operation (e.g., refactoring, development session). Each atomic change is recorded as an addition, deletion, or move of a node or subtree of the AST.

Sharon developed an operation history recording tool called EclipsEye by applying the method of SpyWare into Java program development on Eclipse [63]. EclipsEye records automated refactorings (renaming and moving) and additions, deletions and modifications of syntax elements, such as packages, import statements, classes, methods, and fields. Operation history data are output by Java object serialization.

4.2 ChEOPS

Ebraert et al. developed ChEOPS (Change- and Evolution-Oriented Programming Support), a tool enabling to record operation history on VisualWorks Smalltalk IDE [7, 10].

Most Smalltalk IDEs involve a mechanism for keeping snapshots in change files after every source code change. VisualWorks contains a tool called Change List, which is a more advanced mechanism treating fine-grained changes. After each change is performed, the tool generates a Smalltalk object from the change and records it. The change is recorded in a change

Table 1: Methods for recording operations.

	Unit of change	Base IDE	Target language
SpyWare [51, 53]	syntax element	Squeak	Smalltalk
EclipsEye [63]	syntax element	Eclipse	Java
ChEOPS [7, 10]	syntax element	VisualWorks	Smalltalk
ChEOPSJ [66]*[1]	syntax element	Eclipse	Java
OperationRecorder [44, 76]*[2]	text	Eclipse	Java
Syde [20]*[3]	syntax element	Eclipse	Java
Fluorite [74]*[4]	text	Eclipse	Java
CodingTracker [39, 42]*[5]	syntax element	Eclipse	Java

1 Available at `http://win.ua.ac.be/~qsoeten/other/cheopsj/`.
2 Available at `http://www.fse.cs.ritsumei.ac.jp/~takayuki/operec.html`.
The authors checked that the tool actually works.
3 Available at `http://www.inf.usi.ch/phd/hattori/syde/`.
Source code is included in the available jar file.
4 Available at `http://www.cs.cmu.edu/~fluorite/`. The authors checked that the tool actually works.
Source code is also available at GitHub.
5 Available at `http://codingtracker.web.engr.illinois.edu/`. The authors checked that the tool actually works.
Source code is also available at GitHub.

file, which is mainly used for restoration after an IDE crash. By using Change List, we can view and reorder a change list and delete and reapply its changes. Ebraert et al. pointed out the following four defects regarding the tool.

1. The granularity level of changes is restricted. For example, a change for an attribute is lost, whereas a change for a class or a method is recorded.
2. Definitions of change objects are inconsistent.
3. It cannot record high-level changes.
4. Investigating changes is difficult due to the above three factors.

They implemented ChEOPS to compensate for the above defects. ChEOPS treats a change as a first-class entity. It models changes in accordance with a meta-model of object-oriented design, which is based on FAMIX [69] extended for Smalltalk. It can record any addition, deletion, and modification of a package, class, method, attribute, or variable as an atomic change, i.e., the finest unit of a change. Atomic changes can be grouped as a more abstract unit called a composite change. A developer can freely define such an abstract change. Moreover, a pre-condition and post-condition can be specified to it so that a relationship between changes can be represented. For example, we can declare that addition of a method can be performed to only an existing class. This can prevent inconsistency; adding a method to a class before the class is created. Furthermore, declaring an intensional change is also supported. Generally, an operation that affects multiple parts of source code, such as renaming, should be represented as multiple sub-changes. Such a sub-change is called an extensional change. An intensional change can comprehensively represent the change based on its characteristics (e.g., "changing every part referring the variable").

Soetens et al. developed ChEOPSJ by applying the approach of ChEOPS to Java [66]. It enables the change-based programming of ChEOPS in the context of Java programs. Similar to ChEOPS, it generates a FAMIX-based change model of a Java program when editing it. ChEOPSJ automatically records every addition, deletion, and modification of a package, class, field, and method. However, statement-level changes, such as adding a method call and local variable, are not automatically recorded. We have to manually generate the differences to investigate such changes.

4.3 OperationRecorder

OperationRecorder [44, 76] is an Eclipse plug-in that records history of every edit performed on Java source code, launching a function via a menu item, and so on.

In software maintenance, it is important to understand not only the latest source code snapshot but also past changes. Most traditional research used revision

history for understanding changes. However, changes under multiple intentions are mixed. Therefore, individual changes cannot be obtained. To solve this problem, OperationRecorder, a tool for recording every change performed between revisions, was developed.

OperationRecorder obtains code changes from Eclipse's undo history. Hence, it records just textual changes, having no concern with whether the code is compilable. Therefore, we can restore a non-compilable source code snapshot with operation history of OperationRecorder. Moreover, comments and white spaces that are not included in general ASTs can be also restored. On the other hand, we cannot determine what syntax element the recorded operation is performed for by only seeing recorded data. For this, a method for mapping an edit and an AST syntax element with the offset value of the edit was also proposed [44].

Though earlier versions [44] of OperationRecorder stored operation history into a MySQL database, recent versions output data into XML files [45]. It can record not only edits but also file operations (open, save, close, activation), cut, copy, paste, and invoking menu functions provided by Eclipse. In addition, a source code snapshot is recorded when a file operation was performed if needed.

4.4 Syde

Hattori et al. extended change-based software evolution propounded by Robbes et al. and proposed recording and using changes in collaborative development by multiple developers. Moreover, they implemented Syde, which records operation history in Java development on Eclipse [20].

In traditional approaches of collaborative development, multiple developers concurrently change source code that was checked-out from a VCS. In such a situation, if multiple developers change the same part or multiple parts that have a relationship with each other (e.g., caller and callee of a method, parent class and its child), the program threatens to behave in an unintended manner. Therefore, changes have to be properly merged when they are committed in development with a VCS. However, developers actually tend to hesitate to edit and commit to avoid merging [19, 22]. Since existing tools supporting developers' collaboration (e.g., Jazz[3], CollabVS[4], TUKAN [62]) treat changes on a file basis, information derived from the program's model is lost. There is a tool that transfers change data in real time so that it can alleviate the merge problem [60]. However, it has several problems, such as that the unit of conflict detection is coarse.

Syde can immediately transfer changes on the IDE to other developers. Thus, it can improve developers' awareness [5] regarding changes in collaborative development. Syde is a client-server application. The client is implemented as an Eclipse plug-in. The client sends change history of when the project was built to the server. The change history includes the addition, deletion, and move of a syntax element or AST subtree. The server saves the data sent from clients and broadcasts the data to active clients.

4.5 Fluorite

Fluorite [74] is a plug-in that can record various events (operations' occurrences) on Eclipse. Its characteristics include focusing on how developers are using the IDE.

It is important to know how features of programming languages and tools on IDEs are used to evaluate and improve them. Main data sources available for it are categorized as follows. The first is a direct interview with a developer. However, the answer sometimes lacks reliability. The second is video recording of the development. However, its analysis is time-consuming and error-prone. The third is mining from VCS repositories. However, we cannot determine what operations were performed between revisions from repository data. Moreover, interaction data recorded using several interaction recording tools (e.g., Mylyn [29]) lack detailed information.

Fluorite was extended Practically Macro[5] in terms of recording file open and stability. Fluorite can record an edit (insertion, deletion, and replacement), cursor move, text selection, search, execution (debug and run), file operation (open and activation), and content assist (code completion and quick fix). Developers can leave annotations within the operation history. The tool can record not only what kind of operation occurred but also its details, such as the target words of text search and replacement.

[3] http://jazz.net/

[4] http://research.microsoft.com/en-us/projects/collabvs/

[5] http://sourceforge.net/projects/practicalmacro/

4.6 CodingTracker

CodingTracker can record various operations performed on Eclipse by replacing plug-ins included in standard distributions of Eclipse [39, 42].

Version control systems are used by a large number of developers. A large quantity of development history of open source software development is available to the public. Thus, researchers have been using source code snapshots stored in VCSs as the main data of source code evolution. Negara et al. claimed that precedent studies based on VCS data are dangerous in terms of the following three viewpoints.

The first is derived from the fact that history data are incomplete. The same code may be changed multiple times in a single transaction (changes between commits). Hence, a part of history is not committed and does not remain in the history data. This is represented as a shadowing, or a change is shadowed.

The second is derived from the fact that history data are imprecise. Multiple changes whose intentions are different to the same element may occur in a single transaction. This is called an overlap. For example, when a refactored program element is fixed due to another reason, intentions of the change become obscure. Thus, they are difficult to estimate from revision history.

The third is derived from the fact that the relationships between code changes and developers' behaviors are unclear. In traditional VCSs, the places of changed code can be traced. However, they do not record developers' behaviors before and after the changes. Therefore, it is impossible to answer questions, such as whether a test or refactoring was performed.

To promote research with precise understanding of source code evolution, history data without the above three problems are required. Negara et al. developed CodingTracker to collect such data. CodingTracker can record (1) an edit operation and its undo and redo, (2) editing, creating, updating, saving, and closing a file, (3) opening, saving, and closing a compare editor, (4) execution, undo, redo, and completion of an automated refactoring, (5) creating, copying, moving, deleting, and externally modifying a resource, (6) check-in and check-out on CVS/Subversion, (7) starting and ending of JUnit testing, and (8) changes in IDE options, and so on. Moreover, the tool identifies the target syntax element of the change by using the change's offset value indicating where the change was performed. In particular, it calculates an offset value indicating the changed place based on performed insertions and deletions and glues them together. The changed region is presented by the offset values of the glued text edits and its length. By comparing the calculated region and region of each element on the AST that Eclipse JDT creates, the corresponding element can be identified.

To be exact, refactoring executions are recorded using a tool called CodingSpectator. However, this paper lumps CodingTracker and CodingSpectator together as CodingTracker.

4.7 Other recording methods

There have been several studies with implementation of operation recording tools for investigating operation history and improving IDE tools, though their main focus is not proposing recording tools.

For instance, Kim et al. created a tool that records operations performed on the Eclipse Java editor [30, 31]. Moreover, they implemented a replayer that enables us to manually replay recorded operations. Using the replayer, they investigated how copy and paste are performed in software development and what intentions underlie them. Their tool records fine-grained operation history including code edits in contrast to the tools mentioned in Section 6.

5 Application methods of operation history

This section presents application methods of operation history. Table 2 lists these methods. The names of the recording tools in the table indicate that each application depends on the corresponding recording tool. The authors classified the application methods in terms of instantaneousness of using operation history, supporting method, and supporting object, explained as follows.

Instantaneousness

This indicates whether operation history is used instantly or after they are accumulated.

[I] Instant
A form of development support that instantly uses developers' operation data in the development session once the operations have been detected.

[A] Accumulated
A form of development support by accu-

mulating past operation data and processing them collectively with mining techniques etc.

Supporting method

This indicates how operation history is applied to development support.

- [P] Operation processing
  A method for altering the operations' form or generate a more abstract presentation of the operations. Filtering and grouping operations are included in this category.
- [R] Operation replaying
  A method for restoring past source code by replaying operations.
- [A] Operation analysis
  A method for extracting operation history's characteristics by analyzing them.
- [V] Visualization
  A method for visualizing operation history.
- [C] Change conflict detection
  A method for detecting change conflicts among developers with operations recorded in a multiple-developer project.

Supporting object

This indicates what task the method supports in software development.

- [Co] Supporting comprehension
  To support comprehension regarding source code and/or code changes.
- [Ch] Supporting change
  To support code change directly (e.g., applying past changes to other development contexts).
- [T] Tool improvement
  To improve tools on an IDE to support editing, testing, and/or other development activities with operation history. Analyzing operations for the purpose is also included.
- [P] Process improvement
  To clarify and alleviate the problem of the development process with past operations. A method focused on development stagnation is included.
- [CD] Collaborative development
  To support collaboration in a multiple-developer project. This category includes detecting change conflicts in an early stage of development to reduce the burden of merging.
- [E] Investigation of software evolution [78]
  To clarify how software evolution occurs. A method in this category is not focused on directly supporting development but elucidating the evolution phenomenon.

Note that the classification in Table 2 is not for encompassing the supporting methods and objects. Moreover, several items in the supporting method and supporting object partially overlap with each other. For example, regarding supporting method, a method of "visualization" performs "operation processing" to extract visualization targets. Methods specialized in comprehension and changing are respectively classified into "supporting comprehension" and "supporting change", even if they are based on "tool improvement". This classification helps to grasp the tendency of the proposed application methods. For example, most application methods are aimed at supporting comprehension, whereas their supporting methods are varied, such as operation processing, replaying, and analysis. In addition, methods aimed at tool improvement are classified into: (1) improving existing approaches by analyzing operation history and existing problems of tool use and (2) appending a novel tool, such as a code restore tool, to an IDE.

5.1 Understanding source code and code changes

Robbes et al. provided various tools that use operation history recorded using SpyWare [56]. The tools include a *metric graph* that presents transitions of metric values with operation history and a *change matrix* that comprehensibly presents when and where changes were performed.

Robbes et al. proposed the concept of *development session* [54]. A development session is a period of time which is obtained by dividing a single transaction in a VCS. Each development session is classified into several types, such as decoration (the finest operations, including modifying a method body), painting (adding a method), and restoration (refactoring). The classification is based on the session's corresponding task estimated from the involved operations. They also proposed a tool for visualizing development sessions, which helps in understanding developers' tasks.

Table 2: Application methods of operation history.

Summary	Instantaneous-ness	Supporting method	Supporting object
SpyWare			
Various tools based on SpyWare [56]	[A]	[V], etc.	[Co], etc.
Identifying and classifying development sessions [54]	[A]	[P]	[Co]
Example-based program transformation [55]	[A]	[P], [R]	[Ch]
Evaluating and improving code completion [57]	[I]	[A]	[T]
Evaluating and improving change prediction [59]	[A]	[A]	[T]
Detecting logical couplings [58]	[A]	[A]	[Co]
EclipsEye			
Detecting development stagnation and support [1]	[I]	[A]	[P]
ChEOPS			
Change-based FOP [6, 11]	[A]	[P], [V]	[Co], [Ch]
Change-based FOP (intensional change) [8]	[A]	[P]	[Ch]
Change-based FOP (FODA diagram) [9]	[A]	[P]	[Co]
ChEOPSJ			
Test case selection [66, 67]	[A]	[A]	[T]
Reconstruction of floss-refactorings [68]	[A]	[P]	[Co]
OperationRecorder			
Supporting divided commit by grouping operations [84]	[A]	[P]	[Co]
Supporting understanding history by grouping operations [32, 79]	[A]	[P]	[Co]
Filtering and grouping operations [80]	[A]	[P]	[Co]
Analyzing repetitive code completion [47]	[A]	[A]	[T]
Associating development intentions, history refactoring [23, 24]	[I]	[P]	[Co]
Operation replayer, highlight plug-in [46, 77]	[A]	[R], [V]	[Co]
Operation history slicing [38]	[A]	[R], [P]	[Co]
Operation history replayer, stagnation detection [45]	[A]	[R], [V]	[P]
Syde			
Change notification, visualization [20, 34]	[I]	[C], [V]	[CD], [Co]
Supporting collaborative development with change notifications [15, 16]	[I]	[C]	[CD]
Refining code ownership [19, 21]	[A]	[A]	[CD], [Co]
Understanding evolution by operation history replay [17, 18, 22]	[A]	[R]	[CD], [Co]
Supporting team awareness by visualization [33]	[I]	[C], [V]	[CD], [Co]
Fluorite			
Analyzing backtracking [72]	[A]	[A]	[T]
Azurite (time-line view, editor extension) [73]	[I]	[V], [R]	[T], [Co]
CodingTracker			
Problems of using revision history [42]	[A]	[A]	[E]
Investigation of automated refactorings [70]	[A]	[A]	[T]
Comparative experiment on manual and automated refactorings [40]	[A]	[A]	[T]
CodeSkimmer (time line view, replaying operation) [65]	[I]	[V], [R]	[Co]
Change pattern detection [41]	[A]	[A]	[Co]

Legend

Instantaneousness: [I]: Instant [A]: Accumulated

Supporting method: [P]: Operation processing [R]: Operation replaying [A]: Operation analysis [V]: Visualization
[C]: Change conflict detection

Supporting object: [Co]: Supporting comprehension [Ch]: Supporting change [T]: Tool improvement [P]: Process improvement
[CD]: Collaborative development [E]: Investigation of software evolution

Robbes et al. [58] also proposed logical coupling measures based on operation history recorded using SpyWare. A logical coupling means a combination of parts of code that are often edited in the same transaction. Thus, it indicates implicit relationships among program elements. They concluded that using operation history of SpyWare can improve measuring couplings with less data than those of traditional approaches based on software configuration management (SCM).

Ebraert et al. proposed change-based feature oriented programming (CFOP), so they applied operation history to FOP [6, 11]. The characteristics of CFOP include treating a feature as a set of first-class changes. In an evaluation experiment, they showed that an extension of an existing graph generation framework enables the visual check of the validity of feature compositions and improvement in features' reusability.

Ebraert et al. proposed feature oriented design analysis (FODA) diagram [9], which presents the design of a feature-oriented program. In a FODA diagram, individual changes are grouped in a feature that can be incrementally abstracted. The dependency among features

is presented as a link. By using this diagram, inconsistency between implementation (individual changes) and its design can be detected.

Soetens et al. proposed a method for reconstructing floss-refactorings using operation history recorded by ChEOPSJ [68]. A floss-refactoring is performed in the midst of other changes, such as adding a function. Thus, it is entangled with other kinds of edits. Reconstructing floss-refactorings from the differences in snapshots stored in VCSs is difficult. The method presents dependencies among code edits (insertions and deletions) and types of program entities (class and method) in a graph of a change pattern. Then, to reconstruct refactorings, it evaluates the correspondence between the graph and pre-defined graphs of change patterns corresponding to supported refactorings. Their paper showed that reconstructing refactorings is possible by applying their method to move-method and rename-method refactorings. It also showed the superiority of using fine-grained history data by comparing the method and RefFinder [49], which reconstructs refactorings with differences between snapshots.

Kitsu et al. proposed a method for associating operations recorded using OperationRecorder to program changes (e.g., adding, removing, and moving a field or method, renaming a method, and changing a method body) [79]. They also proposed a method for aggregating individual changes based on temporal distance and spatial distance between program changes [32]. The case studies showed that using the method contributed to understanding changes.

Kuwabara et al. proposed a method for making operation history recorded using OperationRecorder coarser grained by filtering, merging, and grouping operations to improve efficiency of replaying operations [80]. Their preliminary experiment showed how the number of replay units changed with their method.

Hayashi et al. proposed a method for supporting developers to commit their changes into revision history in a proper unit [24]. Committing changes to VCSs in a proper granularity is recommended in software configuration management. However, there are many cases of violating the policy in actual development [26]. Their method records operations with OperationRecorder and allows developers to annotate the operations. Then, the operation history is divided into sub-histories. Each of them corresponds to each intention. Thus, the method allows us to commit changes that are separated by intentions. Moreover, they summed up such modifications to edit history as edit history refactorings [23]. Annotating edits has to be performed manually. However, annotating based on several features, such as the program entity and time of the edit, is automated [84].

Omori et al. proposed a tool called OperationReplayer, which replays edits recorded using OperationRecorder [46, 77]. OperationReplayer provides a time-line bar for grasping operations. By implementing a plug-in, users can flexibly customize the visualization of the time-line bar and analyze operation history. They showed three cases; emphasizing operations performed in the specific method, emphasizing comment-out operations, and showing transitions of source code length [77].

Maruyama et al. proposed a method and a tool for slicing history data to improve the efficiency of replaying operation history recorded using OperationRecorder [38]. In understanding a specific program element (e.g., method or field), replaying edits performed out of the element is unnecessary. Their method constructs an edit operation graph, which presents the correspondences between edit operations and program elements based on offset values in the code where the operation was performed. By identifying a set of reachable vertices in the graph, we can replay only edit operations within the specific program element.

Simmons proposed an replayer for operation history recorded using CodingTracker [65]. The characteristics of this tool include time-line visualization and emphasizing operations related to selected code.

Negara et al. tried to detect code change patterns with history data collected using CodingTracker [41]. Several precedent studies identified refactoring-specific code change patterns using change history data directly collected from IDEs [13, 14]. However, such methods only identify patterns that conform to pre-defined ones. Whereas, by applying data mining techniques, the method of Negara et al. can detect unknown change patterns without pre-defined templates. In the method, a code change pattern is captured by pairs of operation kinds (e.g., add or change) and AST node types. They are the target items of mining. Then, by presenting an item by a bag (not set), the method allows transactions overlap. Finally, based on the mining from history data of CodingTracker, they succeeded in detecting 10 change patterns. The usability of several change patterns were shown by an experiment with developers.

5.2 Supporting code changes

Robbes et al. proposed a method for supporting source code transformation with operation history recorded using SpyWare. Small-scale changes can be done by edits or refactorings. Large-scale changes require dedicated program transformation languages. This method is focused on medium-scale transformations that are not well supported with traditional methods. In the method, developers perform a change as an example. A change that can be applied in other contexts is generated by using the example. The applicability of the method was shown by several examples of transformations.

Ebraert et al. proposed a formal language to present intensional changes in CFOP [8]. With this language, changes across multiple modules can be grouped in a change set.

5.3 Analyzing and improving use of IDE Tools

Robbes claimed operation history is effective for evaluating the performance of recommendation systems (e.g., change prediction tool and code completion tool) on IDEs [52]. Robbes et al. proposed a benchmark for evaluating change prediction methods and evaluated the performance of existing change prediction methods with the benchmark [59]. The results show that using recent changes is the best approach in predicting classes and methods to be changed.

Robbes et al. proposed a method for improving code completion with operation history [57]. They also proposed a method for evaluating performance of code completion based on change data and a user interface of code completion. Furthermore, they proposed a benchmark-based evaluation of code completion performance. Based on their evaluation, they showed that their proposed code completion method drastically improved existing code completion methods.

To improve code completion tools, Omori et al. conducted an experiment with operation history and reported that code completion operations inserting the same text tend to be repeated in a short time [47]. They analyzed repetitive code completion with OperationReplayer and gave five examples of repetitive code completion.

Soetens et al. proposed a method for selecting test cases with operation history collected using ChEOPSJ [66, 67]. When software becomes large, the size of the test suite for automated unit testing also becomes large. Thus, executing all tests after every fine-grained change is not realistic. When a developer edits source code, changed program elements are extracted with ChEOPSJ. Then, tests regarding the elements and other elements depending on them are executed. Therefore, the execution time of automated testing can be reduced.

Yoon et al. investigated developers' backtracking on source code, especially when and how backtracking occurs, with output data of Fluorite [72]. They showed problems regarding backtracking, such as remaining debug code that should be deleted and time-consuming restoration of source code deleted by a mistake.

Yoon et al. proposed Azurite[6], which uses output data of Fluorite [73]. This tool not only visualizes change history on time-line bars but also provides various functions, such as selective undo on the Eclipse source code editor, filtering and searching on the history, and showing differences between past and current snapshots. They showed that the tool helps developers answer questions regarding code change history (e.g., how and when the code was changed and what changes were performed recently).

Negara et al. conducted a comparative experiment of manual and automated refactorings with operation history of CodingTracker [40]. They presented a method for inferencing automated refactorings with operation history. The experimental results on the ratio of performed manual refactorings to automated ones, their size, and required time were also presented.

Vakilian et al. investigated how developers are using automated refactoring tools on Eclipse [70]. They categorized developers' usage into use, disuse, and misuse and discussed each one. For example, need, awareness, and naming were presented as factors of disuse.

5.4 Supporting distributed and collaborative development

Syde [20] provides a tool for recording operations and its applications, which help in collaborative software development with developers distributed geographically. One of the purposes with Syde is improving team awareness by notifying changes to the developers as the change happens. Syde includes the following plug-ins[7].

[6] http://www.cs.cmu.edu/~azurite/

[7] Inspector plug-in is excluded since it is a recording tool.

1. Scamp: It extends Eclipse's Package Explorer view to help the checking of changed files. It also shows a graph representing recent changes in each class.

2. Conflict Plug-in: It notifies change conflicts to developers. It can also semi-automatically resolve conflicts.

Lanza et al. applied Syde to actual collaborative development [34]. As a result, duplicated work among developers can be avoided and the number of conflicts on merging can be reduced.

Based on operation history of Syde, Hattori et al. [19] presented an experiment regarding code ownership. In their study, the owner of a source file was deemed as a developer who performed the greater number of changes on it. In contrast, in the traditional CVS, the owner of a file is a person who changed the greater number of lines of code in it. After an evaluation, they concluded that the ownership classification with their proposed method is more accurate than the traditional one. In a large development project, identifying code ownership is particularly important since not all developers are familiar with every code. Hattori et al. [21] extended the above study, given the forgetting curve. As a novel finding, no optimal value for a memory strength parameter exists.

Hattori introduced a mechanism for detecting code change conflicts in real time and alerting based on the check-in status of the code [16]. Hattori et al. [15] analyzed the effect of preemptive detection of code change conflicts. The results include: (1) developers can communicate earlier to decide what operations should be checked-in first, and (2) developers can break their commits into small ones to reduce the complexity of merging.

Moreover, Hattori et al. implemented a tool called Replay, which can replay recorded operations and showed that replaying is useful for understanding software evolution [17, 18, 22]. Since the number of operations within the history data is generally massive, the tool provides functions of grouping and filtering operations based on change time, developer, and artifact (package, class, method). They showed that the tool is helpful for reducing the time for answering software evolution questions and the precision in the answers can also be improved, compared to understanding evolution with Subversion [17]. The later study [18] extended the experiment and included research participants' comments on tool improvement and details of the experiment.

Lanza et al. proposed a tool called Manhattan for supporting a developer notify other developers' work with visualization based on a city metaphor [33]. Collecting code changes and detecting conflicts are conducted with Syde. In a preliminary evaluation in which participants used the tool, several positive comments were obtained (e.g., the tool is intuitive in understanding code, change conflicts were informed before they become too painful to fix).

5.5 Analyzing development stagnation periods

Omori et al. proposed a method for identifying development terms with a comparatively large number of deleted characters so that stagnation periods are shown on time line bars of OperationReplayer [45]. Examples of stagnation periods actually detected include trial and errors in debugging, extracting a class, and tackling an error. By precisely analyzing such stagnation periods, recurrences of similar stagnations can be prevented.

Carter et al. created a tool to identify stagnations (having difficulty) in development [1]. Their work adopted the extended EclipsEye and their recording tool for Visual Studio. They also proposed a method of identifying stagnation periods with operation kinds and showed that the resultant accuracy was 100%. As a result, it has become possible for developers to help each other sooner and prevent deterioration in development efficiency.

5.6 Investigation of software evolution

Negara et al. investigated five research questions in terms of the reliability of revision history data [42]. The research questions and experimental results are as follows:

Q1 How much code evolution data is not stored in VCS?: 37% of code changes are shadowed and are not stored in VCSs.

Q2 How much do developers intersperse refactorings and edits in the same commit?: 46% of refactored program entities are also edited in the same commit.

Q3 How frequently do developers fix failing tests by changing the test itself?: 40% of test fixes involve changes to the tests.

Q4 How many changes are committed to VCS without being tested?: 24% of changes committed to VCSs are untested.
Q5 What is the temporal and spacial locality of changes?: To answer this question, they employed three time windows spanning 15, 30, and 60 minutes. That is, for a base change operation in a method, they counted how many changes were performed in the method within 7.5, 15, and 30 minutes before and after the base one, respectively. By summing the results for all code changes of each method, the total number of code changes in each method was obtained. Moreover, by adding up the sums, the number of all code changes was obtained. Based on these calculations, they calculated the frequency of each method. As a result, 85% of changes to a method during an hour interval are clustered within 15 minutes on average.

6 Other methods for recording interaction history

This section presents history recording tools that are not this paper's main survey target.

The following tools can store copies of source code as a history: Moraine [85], AJC Active Backup[8], File Hamster[9], Save Dirty Editor Eclipse Plugin[10]. Tani et al. proposed an operation-history-recording tool called plog [81, 82]. This tool records transitions of source code snapshots. It also records the execution environment and execution results when the program is executed. By using these tools, we can trace source code changes in shorter intervals than using snapshots stored in VCSs. However, the operation-oriented methods this paper introduced can treat much finer-grained operations.

The following methods treat only interaction history not detailed source code changes.

Hackystat [28] monitors developers' activities on software development tools by embedding a sensor in them. It provides sensors to Eclipse, Emacs, Ant, JUnit, CVS, Jira, and so on. It gathers various information including the time when a file was accessed and when a build or commit was performed. Moreover, it provides transitions of several metric values, such as file size. With these types of information, it supports decision making, particularly in software project management.

Mylyn (previously called Mylar)[11] [29] is a task and application lifecycle management tool that is installed on Eclipse by default. Mylyn records selected and/or edited files and used functions on Eclipse. With such history, it calculates a degree-of-interest (DOI) value, which indicates how much a task and resource (e.g., source file) are related. Based on this value, it can filter, except for items related to the current task on the Package Explorer view.

Eclipse Usage Data Collector (UDC)[12] is a tool for collecting interaction history on Eclipse. Eclipse UDC collects information on a performed function, its time, and basic user properties (e.g., running operating system). Users can send the recorded data to the Eclipse UDC server. Some of the collected data are publicly available.

Empirical Project Monitor (EPM) [43, 75] is a system for improving the software process by collecting software development data. It consists of four functions: data collection, format translation, data store, and data analysis/visualization. Thus, it can collect time-series data, such as the number of lines of code, that of posted mails, and that of issues, in development support systems (e.g., configuration management system and mailing list).

PROM [64] is a tool to automatically collect metric data on software development. It is supposed to be mainly used for personal software process (PSP) [27]. PROM uses a plug-in architecture so that each plug-in installed by a developer (client) actually collects data. The collected data are sent to the PROM server via the plug-ins server of each client. With this tool, we can avoid error-prone and time-consuming manual data collection. In addition, developers' tasks are not prohibited since the data collection is fully automated.

Recording tools dedicated to refactoring recording include CatchUp! [25] and MolhadoRef [2, 3, 4]. Moreover, recent implementation of Eclipse provides Refactoring History, which preserves the history of executed automated refactorings.

[8] http://www.ajcsoft.com/active-backup.htm
[9] http://www.filehamster.com/
[10] http://sourceforge.net/projects/savedirtyeditor/
[11] http://wiki.eclipse.org/Mylyn
[12] http://wiki.eclipse.org/UDC
The authors checked that Eclipse UDC was not available and the webpage was deprecated in March 2014.

7 Discussions on operation history

This section describes discussions on the format, leveraging, and public use of operation history to the best of our knowledge.

7.1 Format of operation history

Each recording method this paper introduced adopted a different policy of the output format. For example, OperationRecorder emphasizes the readability of XML files. Hence, we can understand the order of code changes in a source file by reading the output file from top to bottom. Meanwhile, Fluorite records a code change and input command (what key was pressed) in different elements since it emphasizes the keeping of an operation's trigger that caused the code change. Therefore, understanding the output file is comparatively difficult for humans.

The amount of recorded data depends on the recording method and development situation. Usually, the size of history data is larger than that of revision history. For reference, in a sample case with OperationRecorder, the last snapshot consisted of 7,569 lines (approximately 205 kilobytes) of code in total. The size of the operation history recorded during its implementation was approximately 24.8 gigabytes (121-fold).

Some studies tried to reduce the size of output files. For example, the data format of CodingTracker is considered more lightweight than XML-based ones, though it is not appropriate for human reading. Another example is Fluorite. It can abbreviate repeated elements with `repeat`-attribute. EclipsEye can automatically compress the output files into a zip file [63].

7.2 Leveraging operation history

When we consider an application of operation history, what kinds of operations can be recorded by the base recording method is an important factor. The recorded operations may be incomplete if the application was not considered in implementation of the recording method. For example, in analyzing testing, the target or results of the testing may not be recorded within the history, even if a running event itself was recorded. In addition, the ease of using operation history is also an important factor in implementation of application methods. Since SpyWare treats a program change as a first-class entity, it can be easily accessed from other applications running on Squeak. Since EclipsEye outputs operation history with serialization, they can be read with standard APIs of Java. OperationRecorder provides APIs to easily read and write operation history. Its APIs also include those for transformation of operation history, such as filtering based on operation kinds.

7.3 Public use of operation history

By allowing operation history recorded in an experiment be for public use, problems with software development can be shared. The data can be used also as a benchmark. Robbes et al. proposed using operation history recorded by SpyWare as a benchmark [57]. However, to the best of our knowledge, there is no case in which operation history is actually opened to the public. The authors consider the reason as a privacy and low-demand issues, as mentioned in Section 3.

8 Conclusion

This paper presented a survey on methods for recording operation history on IDEs, including edit history, and their application methods, supposed to be referred by future software evolution studies. With the presented methods, most problems of traditional approaches with revision history can be solved.

Revision history inevitably arises when source code is stored (committed) to a repository. Hence, it is appropriate or necessary for the history to be shared among development team members. Therefore, collecting such revision history data from an actual software development project is comparatively easy. This is a major factor that revision history has been used in many studies. However, as mentioned in Section 3, understanding changes based on revision history is limited in its accuracy.

Meanwhile, operation-oriented methods enable more accurate analysis by allowing the granularity of code changes to be finer. However, in some cases, finer history data should not be shared or are not desirable for sharing due to privacy issues. Whether the data should be shared among team members in actual development is controversial. Thus, developers' cooperation and receptivity are required at this point.

Since recording methods have become available for actual use only recently, there are still a few cases of their applications to real software projects. Hence, the number of implemented operation-oriented applications is still limited. Moreover, when developers use

operation-based support tools, they are forced to change their traditional development style. As more developers recognize the effectiveness of operation-oriented methods in future, recording methods and their applications will become more enriched. Furthermore, software development techniques based on them will be gradually established.

If an operation-oriented tool is introduced in actual software development or data collection in a study, ease of installation and absence of bugs are important. However, no tools are fully satisfactory at this point. For example, though CodingTracker can record various kinds of operations by replacing standard plug-ins included in Eclipse, it works on limited versions of Eclipse due to dependency restrictions among plug-ins. Moreover, when the authors tried several recording tools, problems in recording accuracy were observed (i.e., some operations were missing). To implement operation recording tools, deep knowledge of the base IDE's implementation is required. Moreover, the updating along with upgrades of the IDE is also required. Thus, it is very difficult to accurately record operation history. It is required to establish a system to obtain accurate history with easy installation and access to operations.

As mentioned in this paper, the target programming languages of the current recording methods are only Java and Smalltalk. The number of target IDEs is also limited. To introduce operation-oriented methods to many software development projects, supporting more languages and IDEs is necessary. In addition, by devising a common representation for operation history, dependency between recording and application methods may be cut off. Such basic improvements are required for the future.

Acknowledgments

This work was partially supported by MEXT/JSPS KAKENHI Grant Numbers 24500050, 26730042, and 23700030.

References

[1] Carter, J. and Dewan, P.: Design, Implementation, and Evaluation of an Approach for Determining when Programmers Are Having Difficulty, *Proceedings of the 16th ACM International Conference on Supporting Group Work*, 2010, pp. 215–224.

[2] Dig, D.: *Automated Upgrading of Component-based Applications*, PhD Thesis, University of Illinois, 2007.

[3] Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. N.: Refactoring-Aware Configuration Management for Object-Oriented Programs, *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 427–436.

[4] Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. N.: Effective Software Merging in the Presence of Object-Oriented Refactorings, *IEEE Transactions on Software Engineering*, Vol. 34, No. 3(2008), pp. 321–335.

[5] Dourish, P. and Bellotti, V.: Awareness and Coordination in Shared Workspaces, *Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work*, 1992, pp. 107–114.

[6] Ebraert, P.: First-class Change Objects for Feature-oriented Programming, *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 319–322.

[7] Ebraert, P.: *A Bottom-up Approach to Program Variation*, PhD Thesis, Vrije Universiteit Brussel, 2009.

[8] Ebraert, P., D'Hondt, T., Molderez, T., and Janssens, D.: Intensional Changes: Modularizing Crosscutting Features, *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 2176–2182.

[9] Ebraert, P., Soetens, Q. D., and Janssens, D.: Change-based FODA Diagrams: Bridging the Gap Between Feature-oriented Design and Implementation, *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011, pp. 1345–1352.

[10] Ebraert, P., Vallejos, J., Costanza, P., Van Paesschen, E., and D'Hondt, T.: Change-oriented Software Engineering, *Proceedings of the 2007 International Conference on Dynamic Languages*, 2007, pp. 3–24.

[11] Ebraert, P., Vallejos, J., and Vandewoude, Y.: Flexible Features: Making Feature Modules More Reusable, *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 1963–1970.

[12] Fluri, B., Würsch, M., Pinzger, M., and Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, *IEEE Transactions on Software Engineering*, Vol. 33, No. 11(2007), pp. 725–743.

[13] Foster, S. R., Griswold, W. G., and Lerner, S.: WitchDoctor: IDE Support for Real-time Autocompletion of Refactorings, *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 222–232.

[14] Ge, X., DuBose, Q. L., and Murphy-Hill, E.: Reconciling Manual and Automatic Refactoring, *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 211–221.

[15] Hattori, L., Lanza, M., and D'Ambros, M.: A Qualitative User Study on Preemptive Conflict Detection, *Proceedings of the 7th International Conference on Global Software Engineering*, 2012, pp. 159–163.

[16] Hattori, L.: Enhancing Collaboration of Multi-developer Projects with Synchronous Changes, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2*, 2010, pp. 377–380.

[17] Hattori, L., D'Ambros, M., Lanza, M., and Lungu, M.: Software Evolution Comprehension: Replay to the Rescue, *Proceedings of the 19th International Conference on Program Comprehension*, 2011, pp. 161–170.

[18] Hattori, L., D'Ambros, M., Lanza, M., and Lungu, M.: Answering Software Evolution Questions: An Empirical Evaluation, *Information and Software Technology*, Vol. 55, No. 4(2013), pp. 755–775.

[19] Hattori, L. and Lanza, M.: Mining the History of Synchronous Changes to Refine Code Ownership, *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, 2009, pp. 141–150.

[20] Hattori, L. and Lanza, M.: Syde: a Tool for Collaborative Software Development, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2*, 2010, pp. 235–238.

[21] Hattori, L., Lanza, M., and Robbes, R.: Refining Code Ownership with Synchronous Changes, *Empirical Software Engineering*, Vol. 17, No. 4-5(2012), pp. 467–499.

[22] Hattori, L., Lungu, M., and Lanza, M.: Replaying Past Changes in Multi-developer Projects, *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010, pp. 13–22.

[23] Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K., and Saeki, M.: Refactoring Edit History of Source Code, *Proceedings of the 28th International IEEE Conference on Software Maintenance*, 2012, pp. 617–620.

[24] Hayashi, S. and Saeki, M.: Recording Finer-grained Software Evolution with IDE: an Annotation-based Approach, *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010, pp. 8–12.

[25] Henkel, J. and Diwan, A.: CatchUp!: Capturing and Replaying Refactorings to Support API Evolution, *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 274–283.

[26] Herzig, K. and Zeller, A.: The Impact of Tangled Code Changes, *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 121–130.

[27] Humphrey, W. S.: *A Discipline for Software Engineering*, Addison-Wesley Professional, 1995.

[28] Johnson, P. M., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., and Yamashita, T.: Improving Software Development Management Through Software Project Telemetry, *IEEE Software*, Vol. 22, No. 4(2005), pp. 76–85.

[29] Kersten, M. and Murphy, G. C.: Mylar: A Degree-of-interest Model for IDEs, *Proceedings of the 4th*

International Conference on Aspect-oriented Software Development, 2005, pp. 159–168.

[30] Kim, M.: *Analyzing and Inferring the Structure of Code Changes*, PhD Thesis, University of Washington, 2008.

[31] Kim, M., Bergman, L., Lau, T., and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, 2004, pp. 83–92.

[32] Kitsu, E., Omori, T., and Maruyama, K.: Detecting Program Changes from Edit History of Source Code, *Proceedings of the 20th Asia-Pacific Software Engineering Conference*, 2013, pp. 299–306.

[33] Lanza, M., D'Ambros, M., Bacchelli, A., Hattori, L., and Rigotti, F.: Manhattan: Supporting Real-time Visual Team Activity Awareness, *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 207–210.

[34] Lanza, M., Hattori, L., and Guzzi, A.: Supporting Collaboration Awareness with Real-time Visualization of Development Activity, *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, 2010, pp. 207–216.

[35] Lehman, M. M.: Programs, Life Cycles, and Laws of Software Evolution, *Proceedings of IEEE*, Vol. 68, 1980, pp. 1060–1076.

[36] Maalej, W., Fritz, T., and Robbes, R.: *Collecting and Processing Interaction Data for Recommendation Systems*, Springer, 2014.

[37] Madhavji, N. H., Fernandez-Ramil, J., and Perry, D.(eds.): *Software Evolution and Feedback - Theory and Practice*, Wiley, 2006.

[38] Maruyama, K., Kitsu, E., Omori, T., and Hayashi, S.: Slicing and Replaying Code Change History, *Proceedings of the 27th IEEE/ACM International Conference on Automated Software*, 2012, pp. 246–249.

[39] Negara, S.: *Towards a Change-oriented Programming Environment*, PhD Thesis, University of Illinois, 2013.

[40] Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D.: A Comparative Study of Manual and Automated Refactorings, *Proceedings of the 27th European Conference on Object-Oriented Programming*, 2013, pp. 552–576.

[41] Negara, S., Codoban, M., Dig, D., and Johnson, R. E.: Mining Fine-Grained Code Changes to Detect Unknown Change Patterns, Technical report, University of Illinois, 2013.

[42] Negara, S., Vakilian, M., Chen, N., Johnson, R. E., and Dig, D.: Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012, pp. 79–103.

[43] Ohira, M., Yokomori, R., Sakai, M., Matsumoto, K., Inoue, K., and Torii, K.: Empirical Project Monitor: A Tool for Mining Multiple Project Data, *Proceedings of the 1st International Workshop on Mining Software Repositories*, 2004, pp. 42–46.

[44] Omori, T. and Maruyama, K.: A Change-aware Development Environment by Recording Editing Operations of Source Code, *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, 2008, pp. 31–34.

[45] Omori, T. and Maruyama, K.: Identifying Stagnation Periods in Software Evolution by Replaying Editing Operations, *Proceedings of the 16th Asia-Pacific Software Engineering Conference*, 2009, pp. 389–396.

[46] Omori, T. and Maruyama, K.: An Editing-operation Replayer with Highlights Supporting Investigation of Program Modifications, *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 2011, pp. 101–105.

[47] Omori, T. and Maruyama, K.: A Study on Repetitiveness of Code Completion Operations, *Proceedings of the 28th International Conference on Software Maintenance*, 2012, pp. 584–587.

[48] Pfleeger, S. L.: *Software Engineering (2nd Edition)*, Pearson Education, 2001.

[49] Rachatasumrit, N. and Kim, M.: An Empirical Investigation into the Impact of Refactoring on Regression Testing, *Proceedings of the 28th International Conference on Software Maintenance*, 2012, pp. 357–366.

[50] Robbes, R.: Mining a Change-based Software Repository, *Proceedings of the 4th International Workshop on Mining Software Repositories*, 2007, pp. 15.

[51] Robbes, R.: *Of Change and Software*, PhD Thesis, University of Lugano, 2008.

[52] Robbes, R.: On the Evaluation of Recommender Systems with Recorded Interactions, *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, 2009, pp. 45–48.

[53] Robbes, R. and Lanza, M.: A Change-based Approach to Software Evolution, *Electronic Notes in Theoretical Computer Science*, Vol. 166(2007), pp. 93–109.

[54] Robbes, R. and Lanza, M.: Characterizing and Understanding Development Sessions, *Proceedings of the 15th IEEE International Conference on Program Comprehension*, 2007, pp. 155–166.

[55] Robbes, R. and Lanza, M.: Example-Based Program Transformation, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, 2008, pp. 174–188.

[56] Robbes, R. and Lanza, M.: SpyWare: A Change-aware Development Toolset, *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 847–850.

[57] Robbes, R. and Lanza, M.: Improving Code Completion with Program History, *Automated Software Engineering*, Vol. 17(2010), pp. 181–212.

[58] Robbes, R., Pollet, D., and Lanza, M.: Logical Coupling Based on Fine-grained Change Information, *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008, pp. 42–46.

[59] Robbes, R., Pollet, D., and Lanza, M.: Replaying IDE Interactions to Evaluate and Improve Change Prediction Approaches, *Proceedings of the 7th IEEE International Working Conference on Mining Software Repositories*, 2010, pp. 161–170.

[60] Sarma, A., Redmiles, D., and van der Hoek, A.: Palantír: Early Detection of Development Conflicts Arising from Parallel Code Changes, *IEEE Transactions on Software Engineering*, Vol. 38, No. 4(2012), pp. 889–908.

[61] Sarma, A.: A Survey of Collaborative Tools in Software Development, Technical report, Institute for Software Research, University of California, Irvine, 2005.

[62] Schümmer, T. and Schümmer, J.: *Support for Distributed Teams in Extreme Programming*, Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 355–377.

[63] Sharon, Y.: EclipsEye Spying on Eclipse, Undergraduate thesis, University of Lugano, 2007.

[64] Sillitti, A., Janes, A., Succi, G., and Vernazza, T.: Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data, *Proceedings of the 29th Euromicro Conference*, 2003, pp. 336–342.

[65] Simmons, C.: CodeSkimmer: A Novel Visualization Tool for Capturing, Replaying, and Understanding Fine-grained Change in Software, Master's thesis, University of Illinois, 2013.

[66] Soetens, Q. and Demeyer, S.: ChEOPSJ: Change-Based Test Optimization, *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 535–538.

[67] Soetens, Q., Demeyer, S., and Zaidman, A.: Change-Based Test Selection in the Presence of Developer Tests, *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 101–110.

[68] Soetens, Q., Perez, J., and Demeyer, S.: An Initial Investigation into Change-Based Reconstruction of Floss-Refactorings, *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013, pp. 384–387.

[69] Tichelaar, S., Ducasse, S., and Demeyer, S.: FAMIX: Exchange Experiences with CDIF and XMI, *Proceedings of the Workshop on Standard Exchange Format 2000*, 2000.

[70] Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P., and Johnson, R. E.: Use, Disuse, and Misuse of Automated Refactorings, *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 233–243.

[71] Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Moghaddam, R. Z., and Johnson, R. E.: The Need for Richer Refactoring Usage Data, *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, 2011, pp. 31–38.

[72] Yoon, Y. and Myers, B.: An Exploratory Study of Backtracking Strategies Used by Developers, *Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2012, pp. 138–144.

[73] Yoon, Y., Myers, B., and Koo, S.: Visualization of Fine-Grained Code Change History, *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2013, pp. 119–126.

[74] Yoon, Y. and Myers, B. A.: Capturing and Analyzing Low-level Events from the Code Editor, *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, 2011, pp. 25–30.

[75] Ohira, M., Yokomori, R., Sakai, M., Iwamura, S., Ono, E., Shinkai, T., and Yokogawa, T.: Designing a System for Real-Time Software Development Management, *IEICE Transactions on Information and Syatems*, Vol. 88, No. 2(2005), pp. 228–239. (In Japanese)

[76] Omori, T. and Maruyama, K.: A Method for Extracting Source Code Modifications from Recorded Editing Operations, *Journal of Information Processing*, Vol. 49, No. 7(2008), pp. 2349–2359. (In Japanese)

[77] Omori, T., Kuwabara, H., and Maruyama, K.: An Editing-operation Replayer to Ease Investigation of Program Modifications, *Computer Software*, Vol. 28, No. 4(2011), pp. 371–376. (In Japanese)

[78] Omori, T., Maruyama, K., Hayashi, S., and Sawada, A.: A Literature Review on Software Evolution Research, *Computer Software*, Vol. 29, No. 3(2012), pp. 3–28. (In Japanese)

[79] Kitsu, E., Omori, T., and Maruyama, K.: Detecting Program Changes based on the Edit History of Source Code, *Computer Software*, Vol. 29, No. 2(2012), pp. 168–173. (In Japanese)

[80] Kuwabara, H. and Omori, T.: Coarse-Grained Frame for Replaying Editing Operation History, *Computer Software*, Vol. 30, No. 4(2013), pp. 61–66. (In Japanese)

[81] Tani, T., Kaneko, N., Yamamoto, S., and Agusa, K.: plog: Programming Activity Recording System Aiming at Extraction of Programming Experiences, *JSSST Symposium on Foundations of Software Engineering (FOSE2007)*, 2007, pp. 161–166. (In Japanese)

[82] Tani, T., Kobayashi, T., Yamamoto, S., and Agusa, K.: Retrieving Experiences of Solving Problems during Programming with Stack Trace Information, *JSSST Symposium on Foundations of Software Engineering (FOSE2008)*, 2008, pp. 99–104. (In Japanese)

[83] Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K., and Saeki, M.: A Technique for Refactoring Editing Histories of Source Code, *JSSST Symposium on Foundations of Software Engineering (FOSE2011)*, 2011, pp. 61–70. (In Japanese)

[84] Hoshino, D., Hayashi, S., and Saeki, M.: Automated Grouping of Editing Operations of Source Code, *Computer Software*, Vol. 31, No. 3(2014), pp. 277–283. (In Japanese)

[85] Yamamoto, T., Matsushita, M., and Inoue, K.: Accumulative File System Moraine and A Metrics Environment MAME., *Computer Software*, Vol. 18, No. 3(2001), pp. 250–260. (In Japanese)

[86] Omori, T., Hayashi, S., and Maruyama, K.: A Survey on Methods of Recording Fine-grained Operations on Integrated Development Environments and their Applications, *Computer Software*, Vol. 32, No. 1(2015), pp. 60–80. (In Japanese)