Source Code Comprehension Strategies and Metrics to Predict Comprehension Effort in Software Maintenance and Evolution Tasks - An Empirical Study with Industry Practitioners

Kazuki NISHIZONO*, Shuji MORISAKI†, Rodrigo VIVANCO‡ and Kenichi MATSUMOTO*
* *Graduate School of Information Science, Nara Institute of Science and Technology*
8916-5 Takayama, Ikoma, NARA 630-0192 JAPAN
{*kazuki-n, matumoto*}*@is.naist.jp*
† *Faculty of Informatics, Shizuoka University*
3-5-1 Johoku, Naka, Hamamatsu, SHIZUOKA 432-8011 JAPAN
ismoris@ipc.shizuoka.ac.jp
‡ *Department of Computer Science, University of Manitoba, Winnipeg, Canada R3T 2N2*
rvivanco@cs.umanitoba.ca

***Abstract*—The goal of this research was to assess the consistency of source code comprehension strategies and comprehension effort estimation metrics, such as LOC, across different types of modification tasks in software maintenance and evolution. We conducted an empirical study with software development practitioners using source code from a small paint application written in Java, along with four semantics-preserving modification tasks (refactoring, defect correction) and four semantics-modifying modification tasks (enhancive and modification). Each task has a change specification and corresponding source code patch. The subjects were asked to comprehend the original source code and then judge whether each patch meets the corresponding change specification in the modification task. The subjects recorded the time to comprehend and described the comprehension strategies used and their reason for the patch judgments. The 24 subjects used similar comprehension strategies. The results show that the comprehension strategies and effort estimation metrics are not consistent across different types of modification tasks. The recorded descriptions indicate the subjects scanned through the original source code and the patches when trying to comprehend patches in the semantics-modifying tasks while the subjects only read the source code of the patches in semantics-preserving tasks. An important metric for estimating comprehension efforts of the semantics-modifying tasks is the Code Clone Subtracted from LOC(CCSLOC), while that of semantics-preserving tasks is the number of referred variables.**

***Keywords*-source code comprehension; comprehension effort estimation metrics; semantics-preserving; semantics-modifying; software maintenance and evolution**

I. Introduction

The cost of software evolution is 60% to 80% of the total software life cycle cost [1]. Identifying effort prediction metrics of software maintenance and evolution helps to prevent cost overruns and delivery slippage. Many investigations have been carried out to identify effort prediction metrics of software maintenance and evolution [2]–[8]. However, the modification tasks of software maintenance and evolution vary from corrective changes to enhancements for new functionality. As the percentage and the diversity of software maintenance and evolution grow, the importance for effort estimation of taking into consideration the types of modification tasks grows larger.

Benestad et al. investigated the relationship between effort and modification tasks in commercial software maintenance and evolution [4]. They constructed an effort estimation model with various parameters including the types of modification tasks to identify effort prediction metrics. The identified major metrics, based on data from version control system and interviews with the developers, were the diversity of the changed source code files and the volatility of the requirements for the modification tasks.

Basili et al. [9] and Nguyen et al. [10] investigated distributions of effort in change analysis, isolation, coding and unit testing across enhancive, reductive and corrective types. In this article [9], the distributions of effort are different among change types such as adaption, correction and enhancement from empirical data collected in commercial software maintenance and evolution. The article [10] reported on effort distribution from empirical data collected in a controlled experiment with students. The effort for isolation activity varied from 20% to 41% across reductive, enhancive and corrective types of modification tasks. This suggests that identifying effort estimation metrics for source code comprehension of an existing version could lead to better estimation.

As the software lifecycle and the number of project evolutions grow, it is more important to investigate the effort prediction metrics of software maintenance and evolution, especially isolation activity, e.g. comprehension of the existing source code. Previous researches imply that the comprehension strategies and effort prediction metrics are different between various types of modification tasks [4], [10].

In this paper, we conducted an empirical study with industry practitioners to answer the following research questions:

RQ1: Are comprehension strategies consistent across different maintenance tasks?

RQ2: Are metrics that are useful to predict source code comprehension effort the same across different maintenance tasks?

To answer RQ1, we used free-form answers to the question "How do you comprehend the source code?" To answer RQ2, we used source code metrics and the amount of time needed to comprehend source code. The modification tasks were divided in terms of semantics of the software into two types: semantics-modifying (corrective and refactoring) or semantics-preserving (enhancive and modifying), which were proposed by Buckley et.al [11].

In Section II, we present the related work, while in Section III, we describe the design of our study. In Section IV, the results of the experiment are described. We discuss the results in Section V, and draw conclusions in Section VI.

II. Related works

Many investigations on software maintenance and evolution have been conducted [4], [6], [9]–[12]. The investigations are categorized into controlled experiments and field data analysis. In a controlled experiment, subjects are asked to carry out the same task under the same conditions. Observations including quantitative and qualitative data, videos, and questionnaires about the experiment are collected and analyzed. Curtis et al. investigated the relationship between three software complexity metrics and programmer performance on a source code recall task and a modification task [2]. The complexity metrics were Halstead's E, McCabe's cyclomatic complexity and the number of statements. The results showed that the three metrics were effort predictors especially for less experienced programmers. In article [3], five developers performed change tasks that added new functionalities to unfamiliar source code. The results showed that a systematic approach for source code comprehension was more effective than an opportunistic approach. Ngyuen et al. [10] conducted a controlled experiment with 24 students. The results of the experiment showed that the distributions of effort classified by development phases were different among modification types. However, these researches do not refer to consistency of comprehension strategies across modification types.

In a field data analysis, effort estimation models are proposed and effort prediction metrics are identified by using empirical data collected in commercial software development [4], [9], [13]. Basili et al. report the distributions of efforts classified by specification analysis, isolation, design, coding and testing phases in a commercial software development [9]. Basili et al. also proposed a two effort estimation model. In this model, the effort estimation model is determined by the percentage of corrective modification and enhancive modification. However, as described in article [9], models were built just for the estimation of the effort to release, not for comprehension of modification tasks. De Lucia et al. proposed effort estimation models for corrective maintenance [13]. The modification tasks were limited to corrective modification. Benestad et al. investigated 336 modification tasks in commercial software development and identified the effort prediction metrics as dispersion of changed code and volatility of the requirements for the modification task [4]. The results indicated that the effort estimation model should have modification type as a parameter.

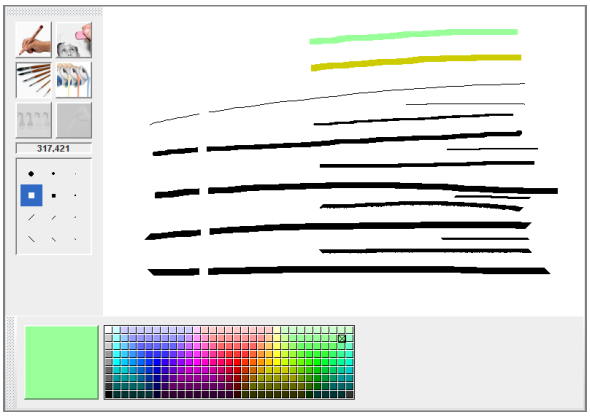


Figure 1. The paint application

III. Research Design

A. Overview

We conducted this research as part of various workshops on source code comprehension. We announced a call for participation in the workshop to IBM Rational mail magazine subscribers, most of whom are software development practitioners. For this study we prepared a paint tool application written in Java. Figure 1 shows a screen shot of the paint application. The application has basic painting functions including drawing a line with specified colors, deleting lines, and filling a specified area with a color. The application has 1,053 LOC and consists of 10 classes. The sample application had to be small enough to allow subjects to complete the tasks in the time allotted in the workshop for the study. The source code of the application was given to the subjects as version 1.0. They were then asked to comprehend the source code in preparation for future changes.

We prepared different types of modification tasks and corresponding source code patches. Each modification task was explained by a change specification that explained why the modification was required and what the modification was. Figure 2 shows an example of a modification task document, which consists of a natural language change specification and the corresponding diff-style patches.

Table I
MODIFICATION TASKS IN THE EXPERIMENT

ID	Title	Modification type	Detail
C1	defect correction in the mouse pointer's coordinates in the panel	Corrective	(before) The mouse pointer's coordinates are always displayed in the panel even if the mouse pointer goes out of the canvas area. (after) The panel displays the coordinates only when the mouse pointer is in the canvas area.
C2	defect correction in the line tool	Corrective	(before) Line with specific line size is not correctly drawn due to a wrong calculation caused by lack of significant digits. (after) Line is correctly drawn with correct calculation.
R1	class name replacement	Refactoring	Class name *PaintTool* is replaced with *PenTool*.
R2	changing variable type	Refactoring	An instance variable declared as float is replaced with an instance variable declared as integer.
E1	implementation of the air brush tool	Enhancive	Air brush-like drawing function is added.
E2	implementation of the canvas clear button	Enhancive	A function that paints the whole canvas area with background color is added.
M1	new feature to eraser tool	Modifying	(before) The eraser tool deletes brushstrokes by inverting them with background color. (after) The eraser tool deletes brushstrokes by inverting them with specified color.
M2	new feature to the color chooser	Modifying	(before) Color can be specified by choosing a color from the color palette that consists of 31 X 9 color table. (after) Color can also be specified by specifying RGB values with text box and by manipulating a GUI slider component.

Table II
SOURCE CODE METRICS OF THE PATCHES

Metrics	Explanation
LOC	lines of code of the patch
cyclomatic complexity	cyclomatic complexity of the patch
lack of cohesion of methods	the difference lack of cohesion of methods [14] patched source code and version 1.0
afferent couplings	the number of the references from methods and local variables outside of the patch to the modified methods and local variables [15]
efferent couplings	the number of the references from the patch to methods and local variables defined in outside of the patch [15]
number of referred variables	the number of the defined variables outside of the patch and referred in the patch
number of chunks of change	the number of places that the patch adds, replaces and deletes
CCSLOC(code clone subtracted LOC)	lines of code of the patch excluding Type-2 code clone [16], [17]

The subjects recorded the time needed to comprehend each patch and judge whether each maintenance patch was correct or not. The subjects also provided the reasons for their judgments. Furthermore, they were asked to explain their comprehension strategy for each patch, such as: "I started with the constructor of class A and methods invoked by the constructor. Then, I confirmed an I/O exception was raised and correctly caught."

The descriptions of comprehension strategies were analyzed to answer RQ1. The time and source code metrics of each patch were analyzed to answer RQ2.

B. Modification tasks

The modification tasks in the study were categorized into either a semantics-preserving type or a semantics-modifying type of task. The semantics-preserving tasks did not change the features of the software, while the semantics-modifying tasks change the features of the software [11]. We prepared 8 modification tasks categorized into 4 sub-types as follows:

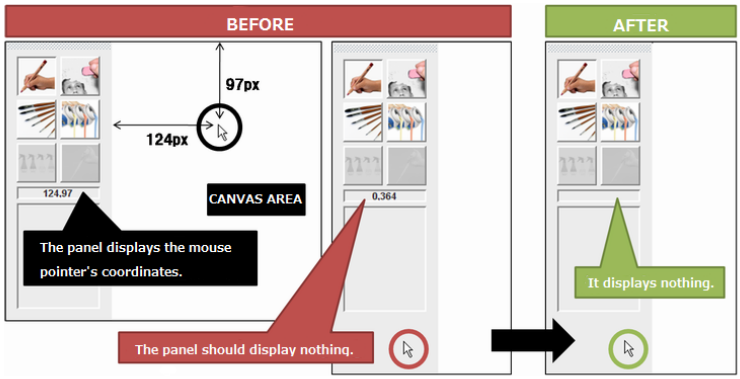
- semantics-preserving

Modification Task C1

Defect correction in the mouse pointer's coordinates in the panel

Change specification

Changed contents: The task changes the application so that the mouse pointer's coordinates display panel does not show the coordinates when the mouse cursor is out of the canvas area.
Current status: The last updated mouse pointer's coordinate remains displayed after the mouse cursor goes out of the canvas area.
Reason for change: To clearly specify that the mouse cursor is not in the canvas area.



Patch

Unified diffs

```
--- C:/Documents and Settings/yuichiro-y/workspace/reviewpaint/trunk/ReviewPaintMain.java    Mon Jun 01 05:12:12 2009
+++ C:/Documents and Settings/yuichiro-y/workspace/reviewpaint/branches/modify_cursor_position/ReviewPaintMain.java Fri Jun 12 01:09:36 2009
@@ -158,6 +158,12 @@

             jPanelCanvasPaper = new CanvasPanel(pt_pencil); // make a canvas and designate a default paint tool

+            // an event listener called when the mouse cursor goes out of the canvas area
+            jPanelCanvasPaper.addMouseListener(new java.awt.event.MouseAdapter() {
+                public void mouseExited(java.awt.event.MouseEvent e) {
+                    jLabelCursorPosition.setText(""); // clear the displayed mouse cursor's cordinate
+                }
+            });
             // an event listener called when the mouse cursor is moved or dragged
             jPanelCanvasPaper.addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {
                 public void mouseMoved(java.awt.event.MouseEvent e) {
```

Modified files

Change type	Files after modification	Diffs
Error correction	/branches/modify_cursor_position/ReviewPaintMain.java	Side-by-side diffs

Figure 2. An example of a modification task document

  - **C** corrective: bug fix
  - **R** refactoring: maintainability improvement
- semantics-modifying
  - **E** enhancive: almost all source code is newly added to implement a feature
  - **M** modifying: existing source code is modified to implement a feature

Table I summarizes all modification tasks in the study.

C. Experimental procedure

The study consisted of two steps. In the first step, the subjects were asked to comprehend version 1.0 of the application by reading the explanation and source code, and by executing the application. The modification task documents were also provided to the subject. However, in the first step, the subjects were asked not to read the modification task documents. The criterion of comprehension was to understand version 1.0 in preparation for future changes.

In the second step, the subjects were asked to understand the change specifications, comprehend the corresponding patches and decide whether the patches are correct or not. The subjects were also asked to note the time needed to understand each modification task. The subjects recorded the following information in a sheet for each modification task;

- time to understand each modification task
- the patch is correct(meets the change specification) or not
- the reason why they consider the patch correct or not
- comprehension strategy of the patch

The subjects were not allowed to execute the updated application by applying the patches. Within the workshop,

Table III
COMPREHENSION STRATEGIES

comprehension strategy	Semantics-Preserving			Semantics-Modifying		
	All	Corrective	Refactoring	All	Enhancive	Modifying
spotlight	42.3%	50.0%	36.1%	41.1%	44.9%	35.3%
skim	2.3%	1.3%	3.1%	8.5%	6.4%	11.8%
dependencies	16.6%	15.4%	17.5%	19.4%	21.8%	15.7%
patch files	2.9%	2.6%	3.1%	1.6%	1.3%	2.0%

the time allotted for the study was three hours. However, the subjects were not required to completely finish both steps within this time. We didn't expect that all the subjects would finish all the modification tasks in the second step or even the first step. The subjects were able to refer to technical books, specifications of the Java API, and information on the Internet.

D. Analysis Procedure

To answer RQ1, the comprehension strategies used in each modification task was analyzed. After comprehension strategies were identified from the subject's answers, the strategies were grouped into several categories of comprehension strategies. The strategies were enumerated for each task. A subject can use multiple comprehension strategies for a patch.

To answer RQ2, the time to finish the modification tasks and the source code metrics of the corresponding patches were compared to identify whether the metrics which can predict the comprehension time are the same across different maintenance tasks. The source code metrics of the patches are shown in Table II.

IV. RESULT

There were 24 subjects who finished the eight modification tasks within three hours. In the rest of this paper, we investigate these 24 subjects. All the subjects have more than five years experience in commercial software development.

A. Comprehension strategies

The following four comprehension strategies were identified;

- Spotlight strategy
  Descriptions that indicate localizing a specific part of the patch or version 1.0 source code are categorized into this strategy. Also, descriptions expressing that the subject had a certain point of view, hypothesis or assumption about a specific part are categorized into this strategy. Words such as "check," "make sure," "compare," "point of view" and "determine" are found in the descriptions in this category.

- Skim strategy
  Descriptions indicating that the subject scanned through the patch or version 1.0 source code are categorized into this strategy. Also, descriptions expressing that the subject didn't read line by line are categorized into this strategy. Words such as "skip," "scan," "skim," and "glance over" are found in the descriptions in this category.

- Dependencies strategy
  Descriptions indicating that the subject followed control and data flow dependencies are categorized into this strategy. Also, descriptions expressing that the subject followed dependencies of variables, methods, or classes changed by the patch are categorized into this strategy. Words such as "invoked," "referenced," "relevant," "affect," and "access" are found in the descriptions in this category.

- Patch files strategy
  Descriptions indicating that the subject carefully read a patch and didn't read other parts of the source code. Expressions such as "I read only the patch" are found in the descriptions in this category.

Table III shows the distributions of the comprehension strategies of the modification tasks. A comprehension strategy description can be categorized into multiple comprehension strategies. The numbers represent percentages of the comprehension strategies relative to the entire number of comprehension strategies. Forty percent of the descriptions in both semantics-preserving and semantics-modifying tasks are categorized into the *spotlight* strategy. The description indicates that the subjects read a part of source code by focusing on a specific interest or by having an assumption and validating the assumption. The percentage of tasks categorized as semantics-modifying where the subject used a *skim* strategy is larger than that of semantics-preserving tasks. The percentage of tasks categorized as semantics-preserving where the subject used a *patch files* strategy is slightly larger than that of semantics-modifying.

B. Comprehension time and metrics

Figure 3 shows the distributions of the subjects' comprehension times of all the modification tasks in the experiment. The horizontal axis represents the modification tasks. The vertical axis represents comprehension time in minutes. The bottom and top of each box are the lower quartile and

Table IV
SOURCE CODE METRICS OF PATCH FILES OF MODIFICATION TASKS

Metrics	Semantics-Preserving				Semantics-Modifying			
	C1	C2	R1	R2	E1	E2	M1	M2
LOC	5	4	60	50	140	23	182	240
cyclomatic complexity	0	0	1	0	13	3	10	26
lack of cohesion of methods	0	0	-1.67	0	0.93	0.03	0	0.2
afferent couplings	1	1	15	9	10	2	29	20
efferent couplings	1	9	7	1	9	0	24	10
number of referred variables	2	5	4	20	22	4	36	30
number of chunks of change	1	1	11	10	2	4	18	8
CCSLOC	5	2	60	25	100	23	144	80

Table V
PEARSON'S CORRELATION COEFFICIENTS WITH MEDIAN COMPREHENSION TIME

	Semantics-preserving	Semantics-modifying
LOC	0.459	0.836
cyclomatic complexity	-0.333	0.510
lack of cohesion of methods	0.333	0.184
afferent couplings	0.245	0.890
efferent couplings	-0.566	0.904
number of referred variables	0.989	0.971
number of chunks of change	0.515	0.627
CCSLOC	0.050	0.963

upper quartile of comprehension time for each modification task. The black bar in each box represents the median of the comprehension time. The lower and upper ends of the whiskers represent the minimum and maximum of the comprehension time.

Table V shows the Pearson's correlation coefficients between the medians of the comprehension times and the source code metrics of the patches. As shown in Table V, the correlations are different between semantics-preserving tasks and semantics-modifying tasks. The medians of the times for comprehension of the modification due to semantics-preserving tasks have lower correlations than that of semantics-modifying tasks, except for the number of referred variables. The median times for comprehension of modification tasks of semantics-modifying have high correlation with the metrics, especially the number of referred variables, CCSLOC, efferent couplings, afferent couplings and LOC.

Figures 4 and 5 show the distributions of times ordered by the number of referred variables, highest correlation with comprehension effort. Figure 6 shows the distributions of times for semantics-modifying tasks ordered by CCSLOC. Figure 5 shows the distributions of comprehension times for semantics-modifying tasks ordered by efferent and afferent couplings. The order is the same by efferent and afferent couplings, number of referred variables. Figure 7 shows the distributions of comprehension times for semantics-modifying tasks ordered by LOC. In the semantics-modifying modification tasks, CCSLOC is a major indicator of the effort of source comprehension.

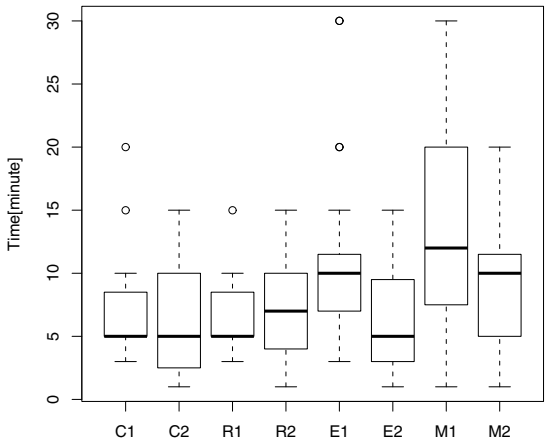


Figure 3. Comprehension times of 8 tasks

V. DISCUSSION

A. RQ1: Consistency of comprehension strategies

The results show that the subjects' comprehension strategies are similar among tasks of semantics-preserving and semantics-modifying tasks. The answer to research question 1 is that the comprehension strategies are relatively consistent based on analysis of the descriptions by the subjects. In both types of maintenance tasks, the subjects' descriptions indicated that 40% of the subjects had assumptions and hypotheses about the source code before they read the patches.

In the semantics-preserving tasks the number of descriptions indicating that the subjects read only the patch is

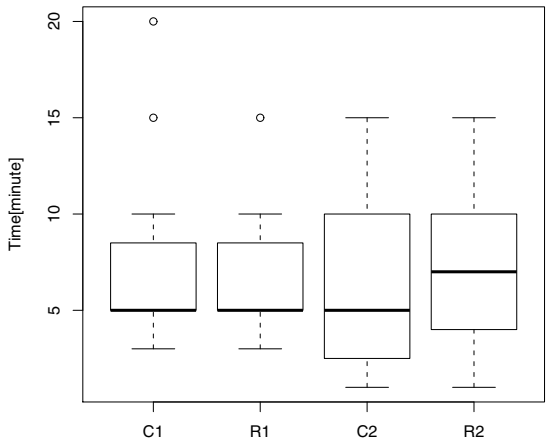


Figure 4. Comprehension times of semantics-preserving tasks ordered by the number of referred variables

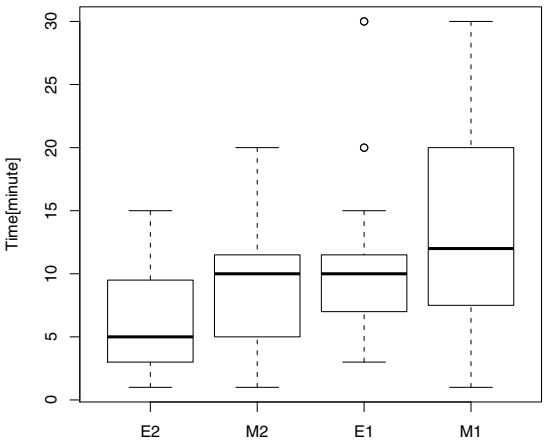


Figure 6. Comprehension times of semantics-modifying tasks ordered by CCSLOC

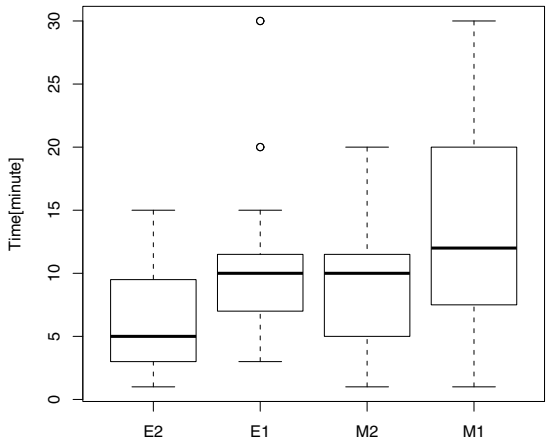


Figure 5. Comprehension times of semantics-modifying tasks ordered by the number of referred variables, efferent and afferent couplings

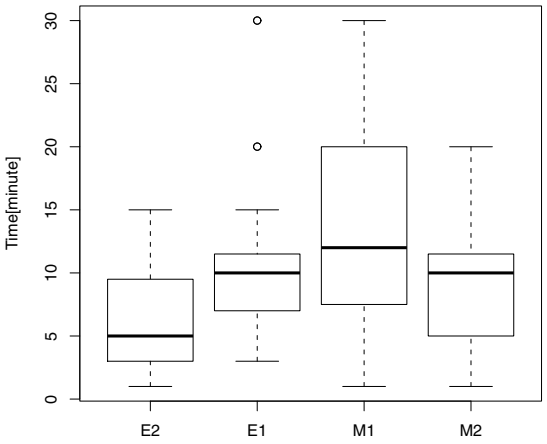


Figure 7. Comprehension times of semantics-modifying tasks ordered by LOC

slightly larger than those in semantics-modifying tasks. Also, some descriptions indicated that subjects read the patches in the semantics-modifying tasks line by line while they did not do so in semantics-preserving tasks. Corrective and refactoring tasks require careful comprehension so that modification will not cause inconsistency between existing code and added, replaced and deleted source code.

In the semantics-modifying tasks, the percentage of the descriptions indicating that the subjects skimmed the source code and the patch is slightly larger than those in semantics-preserving tasks. In the semantics-preserving tasks, the percentage of the descriptions indicating that the subjects read only the patch is slightly larger than those in semantics-modifying tasks. This may be caused by the tendency of the subjects' scanning through during enhancive and modification tasks while the subjects read the source code and the patch line by line with careful attention.

B. RQ2: Metrics to predict comprehension effort

From the point of view of comprehension effort estimation by using source code metrics of the patches, the metrics to predict comprehension effort in this experiment are quite different between the modifications categorized as semantics-preserving and semantics-modifying. The answer to research question 2 is that the metrics that predict comprehension effort differ across different types of the modification tasks.

While LOC, CCSLOC, efferent couplings and afferent couplings are metrics to predict the comprehension efforts of semantics-modifying tasks, these metrics are not those used for semantics-preserving tasks. The comprehension times of modifying tasks of semantics-preserving have lower correlations with most source code metrics.

CCSLOC was a major predictor for comprehension effort among the modification tasks in this experiment. The measured code clones were Type-2, where the names of

variables and methods were not important. In the reading of the patches for semantics-modifying tasks, the subjects tend to skim similar source code that they have already read. The descriptions of comprehension strategies for semantics-modifying tasks also indicate that the subjects scanned through the source code and the patches.

Article [4] reports that the dispersions of the patch is one of the major predictors for the evolution effort. The results of our experiment also indicate that the number of changed chunks affect comprehension times for both semantics-preserving and semantics-modifying tasks.

In article [9], two different effort estimation models using LOC were proposed. One is for enhancement release and the other is for corrective release. In our study, similar results were obtained.

C. Threats to validity

The number of source code lines in the application is smaller than that of a general application due to the fact that a limited amount of time within the workshop was allocated to the empirical study. The essential conclusions obtained from the experiment have limitations in effort estimation for larger sized software. However, the application includes essential elements such as design patterns, geometric algorithms, event driven architecture and basic object oriented programming concepts. The advantage of using a smaller size is that it allows many practitioners to participate in the study. We believe that trends and tendencies observed from many industry practitioners enable generalization of lessons learned.

The comprehension times self-recorded by the participants are potentially not as accurate as could be obtained. However, in this study, we are looking for trends and tendencies, not absolute times. Using many practitioners in the study allows us to track the general trends of which maintenance tasks take longer to comprehend for the average industry practitioner.

The source code metrics of the patches in semantics-preserving and semantics-modifying tasks are not symmetrical. This may affect the results of our investigations: time to comprehend maintenance task source code and the metrics that correlate to the comprehension effort. However, we believe that all the modification tasks are realistic ones and that in real situations similar maintenance tasks will result in patches with similar characteristics.

VI. Conclusion

To assess the consistency of effort prediction metrics and comprehension strategies in software maintenance and evolution, an experiment was carried out as part of workshops on source code inspection. The subjects, industry practitioners, were asked to comprehend the Java source code of a paint application (version 1.0) and then eight modification tasks. The tasks were categorized into 4 semantics-preserving (refactoring and defect correction) tasks and 4 semantics-modifying (enhancement and modification) tasks. The tasks were described with change specifications in natural language and the corresponding source code patches. The subjects were asked to understand the change specifications and comprehend the patches, to describe their own patch comprehension strategies and to record the time used in comprehension. The subjects judged whether each patch met the change specification or not and described the reasons for their judgment.

The results of the 24 industry practitioners who finished all eight tasks indicate that the comprehension strategies for modification tasks are similar. However, the results indicate that in the comprehension of the modification tasks categorized as semantics-modifying tasks, some of the subjects scanned through the source code in patches and version 1.0 source code potentially affected by modification task, while in the modification tasks categorized as semantics-preserving tasks, some of the subjects read only the source code within the patches. The relationships between the source code metrics of the patches and the subjects' time to comprehend show that the effort prediction metrics are different between semantics-preserving and semantics-modifying tasks. In the semantics-preserving tasks, the metric that most highly correlates to the comprehension effort is the number of referred variables. In the semantics-modifying tasks, the metric that most highly correlates to the comprehension effort is CCSLOC.

Acknowledgment

The authors would like to thank the participants of the workshops. The authors also appreciate Kyoko Hattori (IBM Japan) and Hidehito Sunohara (IBM Japan) for helping conducting the workshops. This work was conducted as part of StagE Project (the Development of Next Generation IT Infrastructure), Grant-in-Aid for Scientific Research (B), 23300009, 2011, and Grant-in-Aid for Young Scientists (B), 21700033, 2011 by the Ministry of Education, Culture, Sports, Science and Technology, Japan. The measurement of source code metrics and the events for collecting empirical data in this paper were partially supported by the IBM Academic Initiative Program (http://www.ibm.com/developerworks/university/academicinitiative/).

References

[1] G. Canfora and A. Cimitile, *Handbook of Software Engineering and Knowledge Engineering*. River Edge NJ: World Scientific, 2001, ch. Software maintenance, pp. 91–120.

[2] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics," *IEEE Transactions on software engineering*, vol. 5, no. 2, pp. 96–104, September 1979.

[3] M. P. Robillard, W. Coelho, and G. C. Murphy, "How effective developers investigate source code: an exploratory study," *IEEE Transactions on software engineering*, vol. 30, pp. 889–903, December 2004.

[4] H. C. Benestad, A. Bente, and A. Erik, "Understanding cost drivers of software evolution: a quantitative and qualitative investigation of change effort in two evolving software systems," *EMPIRICAL SOFTWARE ENGINEERING*, vol. 15, pp. 166–203, April 2010.

[5] C. L. Corritore, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *International Journal of Human-Computer Studies*, vol. 54, pp. 1–23, 2001.

[6] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, December 2006.

[7] F. Fioravanti and P. Nesi, "Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems," *IEEE Transactions on software engineering*, vol. 27, no. 12, pp. 1062–1084, December 2001.

[8] M. Jorgensen, "Experience with the accuracy of software maintenance task effort prediction models," *IEEE Transactions on software engineering*, vol. 21, no. 8, pp. 674–681, August 1995.

[9] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance releases," in *Proceedings of the 18th international conference on Software engineering*, 1996, pp. 464–474.

[10] V. Nguyen, B. Boehm, and P. Danphitsanuphan, "Assessing and estimating corrective, enhancive, and reductive maintenance tasks: A controlled experiment," in *Proceedings of the 16th Asia-Pacific Software Engineering*, 2009, pp. 381–388.

[11] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, pp. 309–332, 2005.

[12] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 492–497.

[13] A. De Lucia, E. Pompella, and S. Stefanucci, "Assessing effort estimation models for corrective maintenance through empirical studies," *Information and Software Technology*, vol. 47, pp. 3–15, January 2005.

[14] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, December 1995.

[15] R. Martin, "OO design quality metrics - an analysis of dependencies," in *Proceedings of the Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, October 1994.

[16] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 804–818, September 2007.

[17] S. Bellon, "Detection of software clones," Institute for Software Technology, University of Stuttgart, http://www.bauhaus-stuttgart.de/clones/, Tech. Rep., 2003.