PAPER

Exploiting Eye Movements for Evaluating Reviewer's Performance in Software Review

Hidetake UWANO[†a)], ***Student Member***, **Masahide NAKAMURA**[†], **Akito MONDEN**[†], ***and*** **Ken-ichi MATSUMOTO**[†], ***Members***

SUMMARY This paper proposes to use eye movements to characterize the performance of individuals in reviewing software documents. We design and implement a system called DRESREM, which measures and records eye movements of document reviewers. Based on the eye movements captured by eye tracking device, the system computes the line number of the document that the reviewer is currently looking at. The system can also record and play back how the eyes moved during the review process. To evaluate the effectiveness of the system we conducted an experiment to analyze 30 processes of source code review (6 programs, 5 subjects) using the system. As a result, we have identified a particular pattern, called scan, in the subject's eye movements. Quantitative analysis showed that reviewers who did not spend enough time on the scan took more time to find defects on average.

key words: *software review, human factor, eye movement, experimental evaluation, system development*

1. Introduction

Software review is peer review of software system's document such as source code or requirements specifications. It is intended to find and fix defects (i.e., bugs) overlooked in early development phases, improving overall system quality [1]. Basically, the software review is an *off-line task* conducted by human reviewers without executing the system. In the software review, a reviewer reads the document, understands the structure and/or functions of the system, then detects and fixes defects if any. Especially in developing large-scale software applications, the software review is vital, since it is quite expensive to fix the defects in later integration and testing stages. A study shows that review and its variants such as walk-through and inspection can discover 50 to 70 percent of defects in software product [2]. Our long-term goal is to establish an efficient method that allows the reviewer to find as many defects as possible.

Several methodologies that can be used for the software review have been proposed so far. The idea behind these methods is to pose a certain criteria on reading the documents. Review without any reading criteria is called *Ad-Hoc Review* (AHR). A method where the reviewers read the document from several different viewpoints, such as designers, programmers and testers, is called *Perspective-Based Reading* (PBR) [3]. *Checklist-Based Reading* (CBR) [4] introduces a checklist with which the reviewers check typical mistakes in the document. *Usage-Based Reading* (UBR) [5] is to review the document from user's viewpoint. *Defect-Based Reading* (DBR) [6] focuses on detecting specific type of defects.

To evaluate the performance of these methods, hundreds of empirical studies have been conducted [7]. However, there has been no significant conclusion on which review method is the best. Some empirical reports have shown that CBR, which is the most used method in the software industries, is not more efficient than AHR. As for UBR, PBR and DBR, they achieved slightly better performance than CBR and AHR [6], [8]–[11]. On the other hand, Halling et al. [12] reports an opposite observation that CBR is better than PBR. Several case studies have shown that these methods had no significant difference [13]–[17].

The reason why the results vary among the empirical studies is that the *performance of individual reviewers is more dominant than the review method itself.* This is because the review is a task involving many *human factors*. Thelin et al. [11] compared the effectiveness between UBR and CBR. Figure 1 describes one of the results in their experiment. In this Figure, the horizontal axis shows the defect detection ratio (the number of defects found / the total number of defects in the documents) of each method, and the vertical axis represents a fault classification, which comprises class A (crucial), class B (important) and class C (not important.) The Figure shows that the effectiveness of UBR is 1.2–1.5 times better than the one of CBR on average. However, as seen in the dotted lines in the figure, the individual performance in the same review method varies much more than the method-wise difference. Unfortunately, the performance variance in individual reviewers has not been well studied. Thus, we consider it essential to investigate the performance of reviewers rather than to devise review method. Hence, the key is how to capture the difference among good and bad reviewers.

To characterize the reviewer's performance in an

Manuscript received January 4, 2007.
Manuscript revised May 2, 2007.
Final manuscript received June 29, 2007.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-0192 Japan.
a) E-mail: hideta-u@is.naist.jp

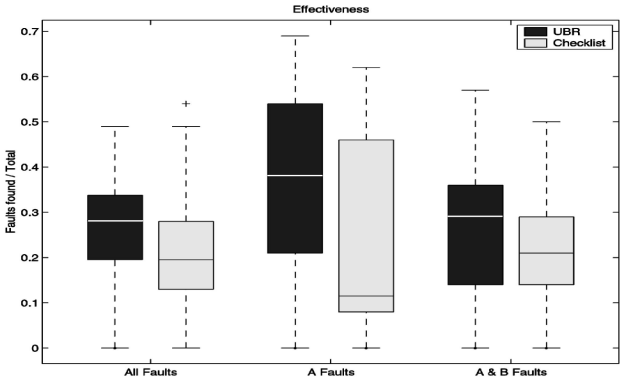


Fig. 1 Effectiveness of UBR and CBR [11].

objective way, this paper proposes to use *eye movements* of the reviewer. In this paper, we present a system called DRESREM to measure and record the eye movements during the software review. The system can calculate the line number of the document that the reviewer is currently looking at. This feature allows us to identify how the reviewer reads the document.

We first point out the following five requirements, which clarify the purpose of the system: (a) gaze point tracking on display, (b) line-wise classification of gaze point, (c) focus identification, (d) record of time-sequenced eye movements and (e) analysis supports. Based on these requirements, we then design and implement three sub-systems of DRESREM. Specifically, as for (a), we use an *eye gaze analyzer*. To achieve (c), we adapted a *fixation analyzer*. For (b), (d) and (e), we developed a *review platform* to calculate the line number that the reviewer is currently looking at.

As an effective application of DRESREM, we then conduct an experiment of source code review with 5 graduate students. Through the experiment, we have identified a particular pattern, called *scan*, in the subject's eye movements. The scan pattern characterizes an action that the reviewer reads the *entire* code before investigating the details of each line. Quantitative analysis showed that reviewers who did not spend enough time for the scan took more time for finding defects on average. Thus, it is expected that eye movement is promising to establish a more human-centered software review method reflecting human factors. Also, in the subsequent interviews, it was found that reviewers tend to comment more detailed and code-specific issues with the eye movements in the reviews. This fact indicates that the eye movements involve much information reflecting the reviewer's thought during the code review. Therefore, captured data of expert reviewers might be used for *educational/training* purposes.

The digest version of this paper was published as a conference paper in ETRA'06 [18]. Changes were made to this version, most significantly the refinement of the system requirements and architectures, definitions of variables in an analysis, and an addition of a statistical analysis. These include explanations of how the proposed system calculates the line number of a document from reviewer's eye movements on a PC display, the details of an analysis method of the eye movements in the experiment, and the statistical significance of the result of the analysis. We believe that these refinements clarify the applicability and limitations of the proposed system against practical review process.

2. Exploiting Eye Movements for Software Review Evaluation

2.1 Software Review – Reading Software Documents

We propose to use *eye movements* of the reviewer for evaluation of software review processes. The primary reason why we exploit the eye movements is that the software documents are not read as ordinary documents such as newspapers and stories.

For instance, let us consider two kinds of software documents: *source code* and *requirements specification*. The source code has a *control flow* (branches, loops, function calls, etc.), which defines the execution order among program statements. The reviewer often reads the code according to the control flow, in order to simulate exactly how the program works. The requirements specification is typically structured, where a requirement contains several sub-requirements. Each requirement is written in *labeled paragraph*. If a requirement R depends on other requirements R_1 and R_2, R refers R_1 and R_2 by their labels. Hence, when the reviewer reads the document, he/she frequently jumps from one requirement to another by traversing the labels.

The way of reading software documents (i.e., *reading strategy*) should vary among different reviewers. The reading strategies is indicated by the eye movements of the reviewers. Thus, we consider that the eye movements can be used as a powerful metric to characterize the performance in the software review.

To support the eye-gaze-based evaluation efficiently, we develop an integrated system environment for capturing and analyzing eye movements during the software review.

2.2 Terminologies

We define several technical words used throughout this paper. The *gaze point* over an object is the point on the object where the user is currently looking. Strictly speaking, it refers to an intersection of the user's sight line and the object. The *fixation* is a condition where gaze points of a user remain within a small area f_a on a object during a given period of time f_t. The fixation is often used to characterize interests of the user. The pair (f_a, f_t) characterizing the fixation is called *fixation criteria*. The *fixation point* is a gaze point where the

fixation criteria holds.

2.3 System Requirements

To make clear the purpose of the system, we present five requirements to be satisfied by the system.

Requirement R1: Sampling Gaze Points over Computer Display
First of all, the system must be able to capture the reviewer's gaze points over the software documents. Usually, reviewed documents are either shown on the computer display, or provided as printed papers. Considering the feasibility, we try to capture gaze points over a computer display. To precisely locate the gaze points over the documents, the system should sample the coordinates with sufficiently fine resolutions, distinguishing normal-size fonts around 10–20 points.

Requirement R2: Extracting Logical Line Information from Gaze Points
As seen in source code, a primary construct of a software document is a *statement*. Software documents are structured, and often written in one-statement-per-line basis. Thus, it is reasonable to consider that the reviewer reads the document in units of *lines*. The system has to be capable of identifying which *line* of the document the reviewer is currently looking at. Note that the information must be stored as *logical* line numbers of a document, which is independent of the font size or the absolute coordinates where the lines are currently displayed.

Requirement R3: Identifying Focuses
Even if a gaze point comes at a certain line in the document, it does not necessarily mean that the reviewer is reading the line. That is, the system has to be able to distinguish a focus (i.e., interest) from their eye movements. It is reasonable to consider that the fixation over a line reflects a fact that the reviewer is currently reading the line.

Requirement R4: Recording Time-Sequenced Transitions
The order in which the reviewer reads lines is important information that reflects individual characteristics of software review. Also, each time the reviewer gazes at a line, it is essential to measure *how long* the reviewer focuses on the line. The duration of the focus may indicate the strength of reviewer's attention to the line. Therefore, the system must record the lines focused on as *time sequence* data.

Requirement R5: Supporting Analysis
Preferably, the system should provide tool supports to facilitate analysis of the recorded data. Especially, features to play back and visualize the data significantly contribute to efficient analysis. The tools may be useful for subsequent interviews or for educational purposes to novice reviewers.

3. DRESREM – The Proposed System

Based on the requirements, we have developed a gaze-based review evaluation system called *DRESREM* (Document Review Evaluation System by Recording Eye Movements).

3.1 System Architecture

As shown in Fig. 2, DRESREM is composed of three sub systems: (1) *eye gaze analyzer*, (2) *fixation analyzer* and (3) *review platform*. As a reviewer interacts with these three sub systems, DRESREM captures the line-wise eye movements of the reviewers. While a reviewer is reviewing a software document, the eye gaze analyzer captures his/her gaze points over the display. Through an image processing, the gaze points are sampled as absolute coordinates. Then, the fixation analyzer converts the sampled gaze points into fixation points, to filter gaze points irrelevant for the review analysis. Finally, the review platform derives the logical line numbers from the fixation points and corresponding date information, and stores the line numbers as time-sequenced data. The review platform also provides interfaces for the reviewers, and analysis supports for the analysts.

In the following subsections, we will give a more detailed explanation for each of the sub systems.

3.2 Eye Gaze Analyzer

To achieve Requirement R1, the eye gaze analyzer samples reviewer's eye movements on a computer display. To implement the analyzer, we have selected a non-contact eye gaze tracker EMR-NC, manufactured by nac Image Technology Inc (`http://www.nacinc.jp/`). EMR-NC can sample eye movements within 30Hz. The finest resolution of the tracker is 5.4 pixels on the screen, which is equivalent to 0.25 lines of 20 point letters. The resolution is fine enough to satisfy Requirement R1.

EMR-NC consists of an eye camera and image processor. The system detects reviewer's eye image, and calculates the position, direction, and angle of an eye. Then the system calculates the position of a display where the reviewer is currently looking at. Each sample of the data consists of an *absolute* coordinate of the gaze point on the screen and sampled date.

To display the document, we used a 21-inch liquid crystal display (EIZO FlexScanL771) set at 1024x768 resolution with a dot pitch of 0.3893 millimeter. To minimize the noise data, we prepared a fixed and non-adjustable chair for the reviewers.

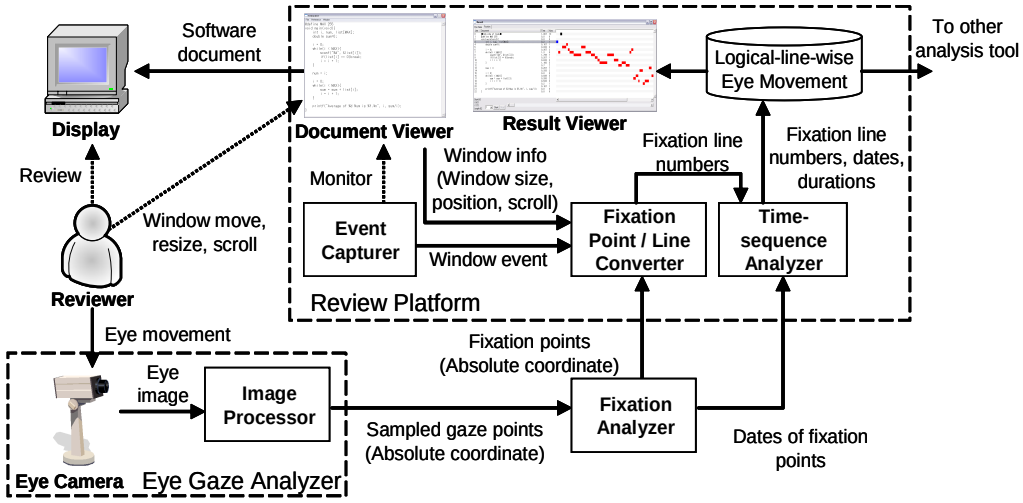


Fig. 2 System architecture of DRESREM.

3.3 Fixation Analyzer

For a given fixation criteria (see Sect. 2.2) and the gaze points sampled by the eye gaze analyzer, the fixation analyzer derives fixation points (as absolute coordinates) and their observed date. Extracting the fixation points from the gaze points is necessary to achieve Requirement R3. To implement the fixation analyzer, we have used the existing analysis tool `EMR-ANY.exe`, which is a bundled application of EMR-NC.

3.4 Review Platform

The review platform is the *core* of DRESREM, which handles various tasks specific to the software review activities. We have implemented the platform in the Java language with SWT (Standard Widget Tool), comprising about 4,000 lines of code.

What most technically challenging is to satisfy Requirement R2. In order to judge if the reviewer is looking at a line of the document, we use fixation points derived by the fixation analyzer. Here we define a line on which a fixation point overlaps as *fixation line*. The goal is to capture the line numbers of the fixation lines.

Note that the line numbers must be captured as the *logical line numbers*. The logical line number is a sequence number attached to every line within the document. The line number is basically independent of the font size or the absolute position of the line currently being displayed. Hence, we need a sophisticated mechanism to derive the logical line numbers from fixation points captured as absolute coordinates. For this, we carefully consider the correspondence between absolute coordinates of points on the PC display and the lines of the documents displayed over those coordinates. We refer such correspondence as *point/line correspondence*.

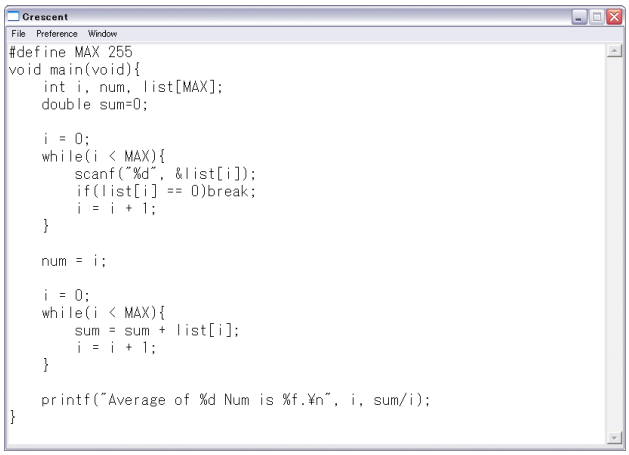


Fig. 3 Example of document viewer.

As seen in Fig. 2, the review platform consists of the following five components.

3.4.1 Document Viewer

The document viewer shows the software document to the PC display, with which the reviewer reads the document. As shown in Fig. 3, the viewer has a slider bar to scroll the document. By default, the viewer displays 25 lines of the document in a 20-point font, simultaneously. The viewer polls window information (such as window size, font size, position, scroll pitch) to the fixation point/line converter. This information is necessary to manage the consistent point/line correspondence.

3.4.2 Event Capturer

As a reviewer interacts with the document viewer, the reviewer may scroll, move, or resize the window of the

document viewer. These window events change the absolute position of the document within the PC display, thus, modifying the point/line correspondence. To keep track of the consistent correspondence, the event capturer monitors all events issued in the document viewer. When an event occurs, the event capturer grabs the event and forwards it to the fixation point/line converter.

3.4.3 Fixation Point/Line Converter

The fixation point/line converter derives the logical line numbers of fixation lines (referred as *fixation line numbers*) from the given fixation points. Let $p_a = (x_a, y_a)$ be an absolute coordinate of a fixation point on the PC display. First, the converter converts p_a into a *relative* coordinate p_r within the document viewer, based on the current window position $p_w = (x_w, y_w)$ of the viewer, i.e., $p_r = (x_r, y_r) = p_a - p_w = (x_a - x_w, y_a - y_w)$. Then, taking p_r, the window height H, the window width W, the font size F and the line pitch L into account, the converter computes a fixation line number l_{p_r}. Specifically, l_{p_r} is derived by the following computation:

$$l_{p_r} = \begin{cases} \lfloor y_r/(F+L) \rfloor + 1, \\ \quad \cdots \text{ if } ((0 \le x_r \le W) \text{ and } (0 \le y_r \le H)) \\ 0 \quad \texttt{(OUT\_OF\_DOCUMENT)}, \\ \quad \cdots \text{ otherwise} \end{cases}$$

Thus, the point/line correspondence is constructed as a pair (p_a, l_{p_r}).

Note that l_{p_r} is changed by the user's event (e.g., window move or scroll up/down). Therefore, the converter updates l_{p_r} upon receiving every event polled from the event capturer. For instance, suppose that the reviewer moves the document viewer to a new position $p_{w'}$. Then, the converter notified of a window move event. Upon receiving the event, the converter re-calculates p_r as $p_a - p_{w'}$, and updates l_{p_r}.

Thus, for every fixation point, the fixation point/line converter derives the corresponding fixation line number, which achieves Requirement R2.

3.4.4 Time-Sequence Analyzer

The time-sequence analyzer summarizes the fixation line numbers as time-sequenced data to satisfy Requirement R4. Using the date information sampled by the fixation analyzer, the time-sequence analyzer sorts the fixation line numbers by date. This is to represent the order of lines in which the reviewer read the document. It also aggregates successive appearances of the same fixation line number into one with the *duration*. The duration for a fixation line would reflect the strength of reviewer's interest in the line.

3.4.5 Result Viewer

The result viewer visualizes the line-wise eye movements using horizontal bar chart, based on the time-sequenced fixation line numbers. Figure 4 shows a snapshot of the result viewer. In the figure, the left side of the window shows document which was reviewed by the reviewer. In the right side of the window, the sequential eye movements of the reviewer are described as a bar chart. In this chart, the length of each bar represents the duration for the fixation line.

The result viewer can play back the eye movements. Using the start/stop buttons and slider bar placed under the viewer, the analyst can control replay position and speed. On the result viewer, time-sequenced transition of fixation lines is described by highlighting of line and emphasizing of bar. Moreover, the result viewer has a feature which can *superimpose* the recorded gaze points and fixation points onto the document viewer. This feature helps the analyst to watch more detailed eye movements over the document. Thus, the result viewer can be extensively used for the subsequent analysis of the recorded data, which fulfills Requirement R5.

4. Evaluating Performance of Source Code Review using DRESREM

To demonstrate the effectiveness of DRESREM, we have conducted an experiment of source code review.

4.1 Experiment Overview

The *source code review* is a popular software review activity, where each reviewer reads the source code of the system, and finds bugs without executing the code.

The purpose of this experiment is to watch how the eye movements characterize reviewer's performance in the source code review. In the experiment, we have instructed individual subjects to review *source code* of small-scale programs, each of which contains a *single defect*. Based on a given specification of the program, each subject tried to find the defect as fast as possible. The performance of each reviewer was measured by the time taken until the injected defect was successfully detected (we call the time *defect detection time*).

During the experiment, the eye movements of the individual subjects were recorded by DRESREM. Using the recorded data, we investigate the correlation between the review performance and eye movements.

4.2 Experiment Design and Procedure

Five graduate students participated in the experiment as the reviewers. The subjects have 3 or 4 years of programming experience, and have experienced the source

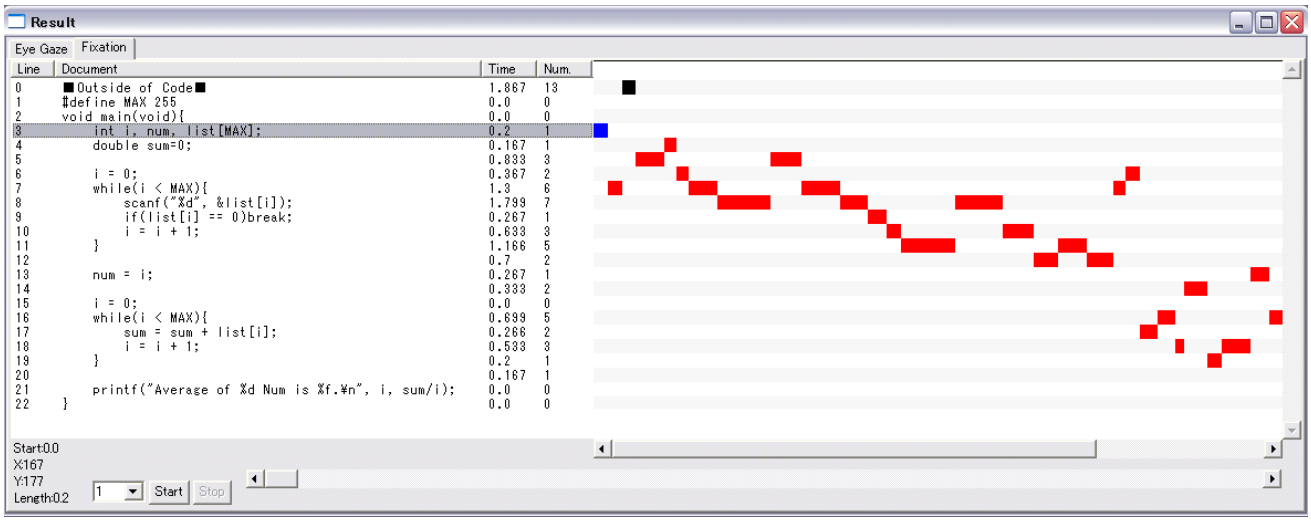


Fig. 4 Result viewer.

code review before the experiment at least once.

We have prepared six small-scale programs written in the C language (12 to 23 lines of source code). To measure the performance purely with the eye movements, each program has no comment line. For each program, we prepared a specification, which is compact and easy enough for the reviewers to understand and memorize. Then, in each program we intentionally injected a single *logical* defect, which is an error of program logic, but not of program syntax. Table 1 summarizes the programs prepared for the experiment.

We then instructed individual subjects to review the six programs with DRESREM. The review method was the ad-hoc review (AHR, see Sect. 1). The task for each subject to review each program consists of the following five steps.

1. Calibrate DRESREM so that the eye movements are captured correctly.
2. Explain the specification of the program to the subject verbally. Explain the fact that there exists a single defect somewhere in the program.
3. Synchronizing the subject to start the code review to find defect, start the capture of eye movements and code scrolling.
4. Suspend the review task when the subjects says he/she found the defect. Then, ask the subject to explain the defect verbally.
5. Finish the code review task if the detected defect is correct. Otherwise, resume the task by going back to the step 3. The review task is continued until the subject successfully finds the defect, or the total time for the review exceeds 5 minutes.

The above task is repeated for each of the six programs. Thus, total 30 review tasks (= 6 programs × 5 subjects) have been conducted. The order of assigning the six programs may yield *learning/fatigue effects* to the reviewer. To minimize the effects, we have used the *Latin square* to shuffle and balance the order.

4.3 New Finding — Scan Pattern

After the experiment, we have investigated the recorded data. Using the result viewer extensively, we played back the eye movements of the individual reviewers, and examined statistics. As a result, we have identified a particular pattern of the eye movements.

It was observed that the subjects were likely to first read the whole lines of the code from the top to the bottom briefly, and then to concentrate on some particular portions. The statistics show that 72.8 percent of the code lines were gazed in the first 30 percent of the review time. We call this preliminary reading of the entire code, the *scan pattern*.

Figures 5 and 6 describe the eye movements of two subjects C and E reviewing programs `IsPrime` and `Accumulate`, respectively. The graphs depict the time sequence of fixation lines. In the figures, the scan patterns are well observed. As seen in Fig. 5, this subject scans the code twice, then concentrates the while loop block located middle of code. In Fig. 6 is seen that this subject firstly locates the headers of two function declarations in lines 1 and 13. Then, the subject scan the two functions `makeSum()` and `main()` in this order. After the scan, he concentrates on the review of `makeSum()`.

We hypothesize that the scan pattern reflects the following review strategy in source code review: A reviewer first tries to understand the program structure by scanning the whole code. During the scan, the reviewer should identify suspected portions where the defect is likely to exist. Therefore, we consider that the scan quality would significantly influence the efficiency of the defect detection in the review.

Table 1 Programs reviewed in the experiment.

Program	LOC	Specification	Injected Defect
`IsPrime`	18	The user inputs an integer n. The program returns a verdict whether n is a prime number or not.	Logic in a conditional expression is wrongly reversed, yielding an opposite verdict.
`Accumulate`	20	The user inputs a non-negative integer n. The program returns the sum of all integers from 1 to n.	A loop condition is mistaken. The condition must be $(i <= n)$ but is actually $(i < n)$.
`Sum-5`	12	The user inputs five integers. The program outputs the sum of these integers.	A variable for accumulating the sum is not initialized.
`Average-5`	16	The user inputs five integers. The program outputs the average of these.	An explicit type conversion from integer to double is forgotten, yielding a round margin in the average.
`Average-any`	22	The user inputs an arbitrary number of integers (up to 255) until zero is given. The program outputs the average of the given numbers.	The number of loops is wrong. The program always calculates the average of 255 numbers regardless of the number of integers actually entered.
`Swap`	23	The user inputs two integers n_1, n_2. The program swaps values of n_1 and n_2 using function `swap()`, and outputs them.	Pointers are misused. As a result, the two numbers are not swapped.

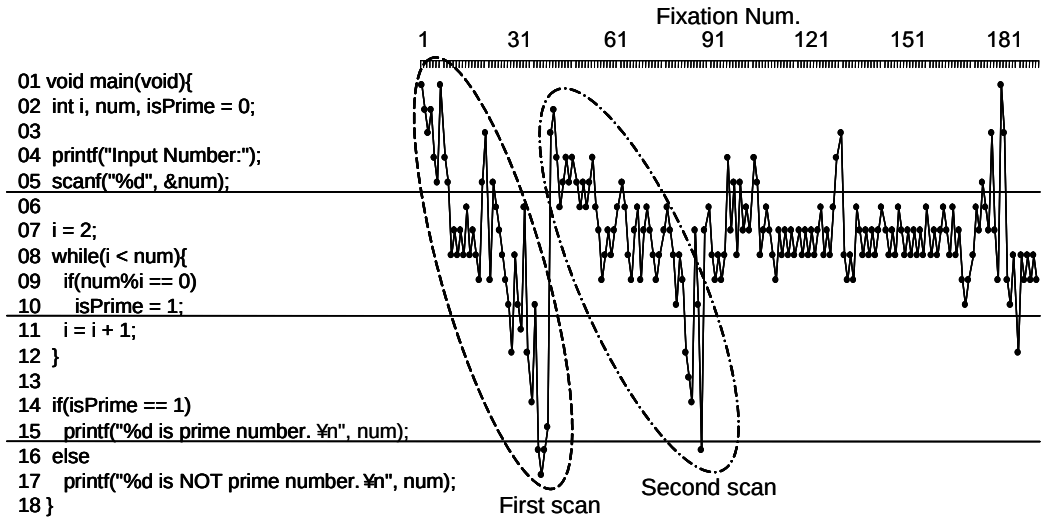


Fig. 5 Eye movements of subject E reviewing program `IsPrime`.

4.4 Quantitative Analysis of Scan Pattern and Review Performance

To verify the hypothesis, we here conduct a quantitative analysis using the recorded data. For each review in the experiment, we have measured *defect detection time (DDT)* and *first scan time (FST)*. For a reviewer r and a program p, $DDT(r,p)$ is defined as the time taken for r to detect the injected defect within p. DDT is supposed to be a metric reflecting the performance (efficiency) of the review task. On the other hand, $FST(r,p)$ is defined as the time spent from the beginning of the review until r reads 80 percent of the total lines (except blank lines) of p. FST might be used as a metric characterizing the quality of the scan.

Note that both DDT and FST deeply depends on the *reading speed* of the reviewer. That is, a slow reader tends to spend more time for scan and defect detection than a fast reader. The reading speed varies from a subject to a subject according to individual experience. Hence, for each reviewer r, the absolute value of $FST(r,p)$ (or $DDT(r,p)$) does not necessarily reflect his/her quality of scan (or performance, respectively). To minimize the effect of the reading speed, we *normalize* $DDT(r,p)$ and $FST(r,p)$ by the total average. Let r be a reviewer, p be a given program, and $Prog$ be a set of all programs reviewed. Then, we define *normalized defect detection time (nDDT)* and *normalized first scan time (nFST)* as follows.

$$nDDT(r,p) = \frac{DDT(r,p)}{\sum_{p' \in Prog} DDT(r,p') \ / \ |Prog|}$$

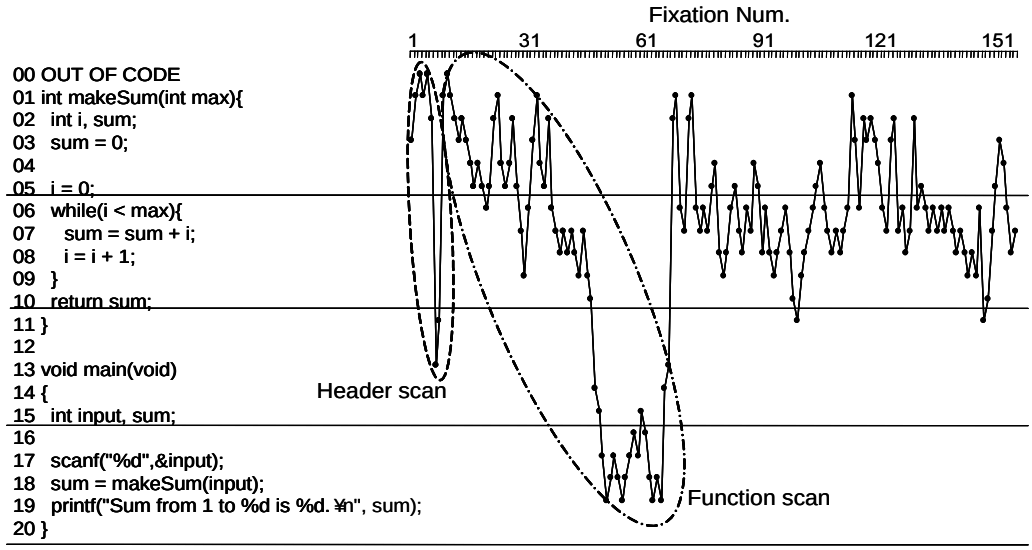


Fig. 6 Eye movements of subject C reviewing program `Accumulate`.

$$nFST(r,p) = \frac{FST(r,p)}{\sum_{p' \in Prog} FST(r,p') \ / \ |Prog|}$$

$nDDT$ and $nFST$ are *relative* metrics for individual reviewer. When $nDDT(r,p)$ is greater than 1.0, r spent more time than usual to detect the defect in p, which means the lower performance. When $nFST(r,p)$ is greater than 1.0, r spent more time than usual to scan the code, which means the higher quality of scan.

Figure 7 depicts a scattered plot, representing the pairs of $(nFST(r,p), nDDT(r,p))$, for every reviewer r and every program p. In the figure, the horizontal axis represents $nFST$, whereas the vertical axis plots $nDDT$. The figure clearly shows a negative correlation between $nFST$ and $nDDT$. Pearson's product moment showed significantly negative correlation between $nFST$ and $nDDT$ ($r = -0.568, p = 0.002$). That is, the first scan time less than the average yields the longer defect detection time. More specifically, the defect detection time increased up to 2.5 times of average detection time when the first scan time is less than 1.0. On the other hand, in the case that the scanning time is more than 1.0, the defect detection time is less than the average.

Thus, the experiment showed that the longer a reviewer scanned the code, the more efficiently the reviewer could find the defect in the code review. This observation can be interpreted as follows. A reviewer, who carefully scans the entire structure of the code, is able to identify many candidates of code lines containing defects during the scan. In Figs. 5 and 6, the reviewers focus their eye movements to a particular block or a function after the scanning of the code. On the other hand, a reviewer with insufficient scan often misses some critical code lines, and they stick to irrelevant lines involving no defect. Figure 8 depicts typical eye movements that could not address suspicious lines through the scanning of a program `IsPrime`, which is the same source code as Fig. 5. This reviewer spent insufficient scan time compared with his/her average scan time ($nFST = 0.51$), and the reviewer could not detect a defect at last. Of course, our hypothesis has been proven within this experiment only. For more generality, we plan to continue more experiments in our future work.

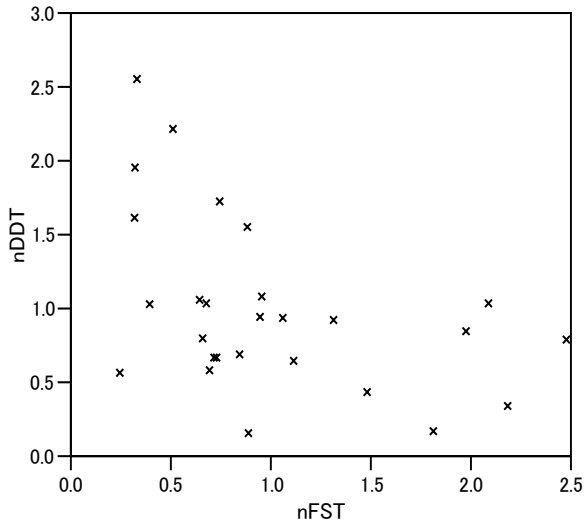


Fig. 7 Normalized first scan time and defect detection time.

4.5 Using Recorded Data for Review Training

After the experiment, we conducted two kinds of in-

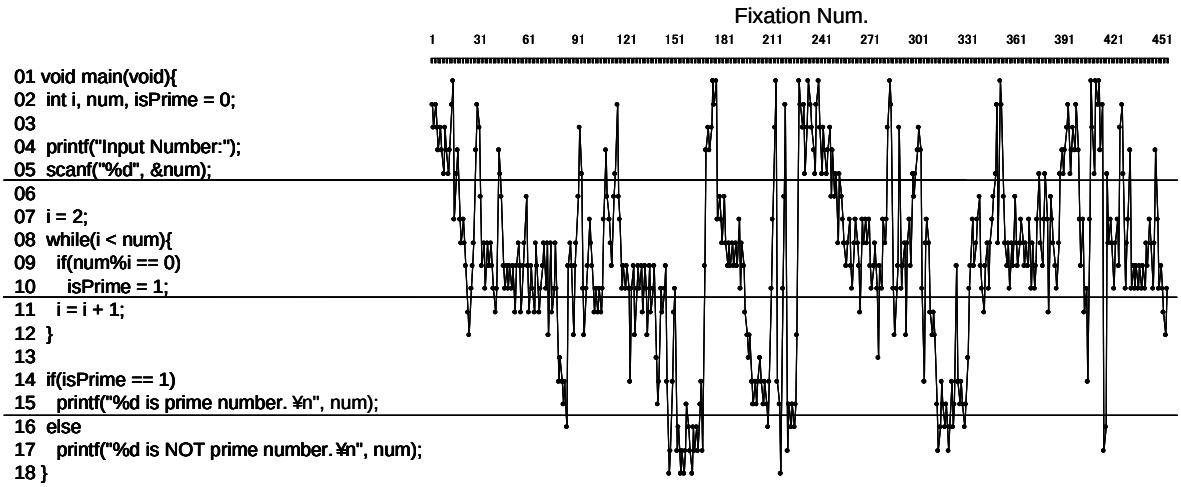


Fig. 8 Eye movements of subject B reviewing program `IsPrime`.

terviews to investigate what the eye movements actually reflect. In the first interview, for each subject we showed the source code and asked what the subject had been thinking in the code review. Most subjects commented abstract review policies, including the strategy of understanding the code and the flow of the review. Typical comments are summarized in the first column of Table 2.

In the second interview, we showed the recorded eye movements using the result viewer together with the source code, and asked the same questions. As a result, we were able to gather more detailed and code-specific comments. As shown in the second column of Table 2, each subject told reasons why he checked some particular lines carefully and why not for other lines. It seems that the record of the eye movements reminded the subjects of their thought well.

This fact indicates that the eye movements involve much information reflecting the reviewer's thought during the code review. Therefore, we consider that data captured by DRESREM can be used for training and educational purposes. Especially, the eye movements of expert reviewers would be helpful for novice reviewers.

5. Discussion

5.1 Advantage and Limitations

The major advantage in adopting the eye movements is that the eye movements provide us with quantitative and objective analysis on how the reviewer reads the software document. As a related work, there exists a method called *think-aloud protocol* [19], which tapes audio and video of subjects to record their intellectual activities. However, compared to the think-aloud protocol, the eye movements do not impose training or expensive preparation upon the subjects.

The limitation is that capturing eye movements requires an eye camera with high resolution and extreme precision. Such eye cameras and surrounding devices are still uncommon and expensive. However, we believe that the limitation will be alleviated with the future technologies and devices. A report [20] says that until 2010 the price of eye cameras will be reduced to 1/100 and that the precision will be improved 10 times.

5.2 Applicability to Practical Software Review

In this paper, we have conducted an experiment of the source code review only. However, we consider that DRESREM is applicable to other kinds of practical software documents as well.

As seen in the Sect. 3.4, primary feature of DRESREM is the line-wise gaze tracking. Therefore, it is especially suitable for (a) structured documents, (b) documents formed by multiple statements, (c) documents written in one-statement-per-line basis, or (d) documents that have special flows (e.g., control flow, labeled references, etc.). Such software documents include requirement specifications, use case descriptions, program code (source, assembly) and test cases.

On the other hand, the documents mainly constructed from figures, diagrams and charts are beyond the scope of DRESREM. Such documents include sequence diagrams and state chart diagrams. However, as for these documents, the eye movements should be tracked and evaluated in a point-wise basis, rather than the line-wise basis. Therefore, we can use the sampled fixation points directly, without performing the point/line conversions (see Sect. 3.4.3). We can cope with this by down-grading DRESREM, or by using other existing systems. Thus, we believe that this is not a critical problem.

Table 2 Comments gathered in interviews.

First interview (with source code only)	Second interview (with source code and eye movements)
· I thought something is wrong in the second while loop.	· I did not care the conditional expression of the loop.
· Firstly, I reviewed main function, and then, read the another one.	· I watched this variable declaration to see the initial value of the variable.
· I simulated the program execution in mind assuming an input value.	· I thought this input process was correct because it is written in a typical way.
· I checked the while loop for times.	· I could not understand why this variable is initialized here.

5.3 Related Work

Eye movements have been often used for the purpose of evaluating human performance, especially in cognitive science. Law et al. [21] analyzed eye movements of experts and novices in laparoscopic surgery training environment. This study showed that experts tend to watch affected parts more than the tool in hand, compared with novices. Kasarskis et al. [22] investigated eye movements of pilots in a landing task at flight simulator. In this study, novices tend to concentrate watching the altimeter than experts, while the experts watch the airspeed.

In the field of software engineering, there are several research works exploiting the eye movements, for the purpose of, for instance, monitoring online debugging processes [23], [24], usability evaluation [25], [26], human interface [27], [28]. As far as we know, there has been no research that directly applies the eye movements for evaluating performance of the software review.

Crosby et al. analyzed eye movements of university students to characterize their performance of *program comprehension* [29]. In their experiment, the students are first instructed to watch movies showing correct execution of Pascal programs. Then, the students read the source code. However, their task is to understand the code, but not to find defects. Thus, the context is quite different from that of source code review, where the reviewers have to find as many bugs as possible without executing the code.

6. Conclusion

In this paper, we have designed and implemented a system, called DRESREM, for the eye-gaze-based evaluation of software review. Integrating three sub-systems (the eye gaze analyzer, the fixation analyzer and the review platform), DRESREM automatically captures reviewer's eye movements, and derives the sequence of logical line numbers of the document that the reviewer has focused. Thus, the system allows us to evaluate quantitatively how the reviewer reads the document.

We have also conducted an experimental evaluation of the source code review using DRESREM. As a result, we have found a particular reading pattern, called scan. Through the statistic analysis, it was shown that the reviewers taking sufficient time for scanning the code tend to detect defects efficiently. In the subsequent interviews, reviewers made more detailed and code-specific comments when the recorded eye movements were shown. This fact indicates that the eye movements involve much information reflecting the reviewer's thought during the code review.

As future work, we are planning to conduct experiments with more large-scale source code and/or different document such as requirement specifications. In review of large-scale documents, reviewers would tend to aggregate multiple lines as a meaningful unit, called *chunk* [30]. Therefore, we plan to conduct *chunk-wise analysis*, which is coarser-grained analysis than what was presented in this paper.

Evaluation of tool-assisted review is also an interesting topic to establish an efficient review method. The recent sophisticated IDEs (Integrated Development Environments) provide many convenient features for writing/reading software documents, including word search and the call hierarchy viewer. Tracking eye gaze over the IDE may derive an efficient way of the tool assistance for software review.

Acknowledgments

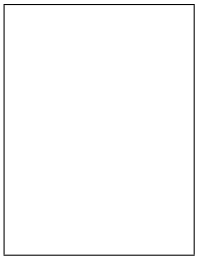
This work was supported by Grants-in-Aid for Scientific Research (B) No. 17300007 and by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

References

[1] B.W. Boehm, Software Engineering Economics, Prentice Hall, 1981.
[2] K. Wiegers, Peer Reviews in Software - A Practical Guide, Addison-Wesley, 2002.
[3] F. Shull, I. Rus, and V. Basili, "How perspective-based reading can improve requirements inspections," Computer, vol.33, no.7, pp.73–79, 2000.
[4] M.E. Fagan, "Design and code inspection to reduce errors in program development," IBM Syst.J., vol.15, no.3, pp.182–211, 1976.
[5] T. Thelin, P. Runeson, and B. Regnell, "Usage-based reading :An experiment to guide reviewers with use cases," Information and Software Technology, vol.43, no.15, pp.925–938, 2001.
[6] A.A. Porter, L.G. Votta, and V.R. Basili, "Comparing detection methods for software requirements inspection - A

replicated experiment," IEEE Trans. Softw. Eng., vol.21, no.6, pp.563–575, 1995.
[7] M. Ciolkowski, O. Laitenberger, D. Rombach, F. Shull, and D. Perry, "Software inspection, reviews and walkthroughs," International Conference on Software Engineering (ICSE), pp.641–642, 2002.
[8] V.R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, and M.V. Zelkowitz, "The empirical investigation of perspective-based reading," Empirical Software Engineering: An International Journal, vol.1, no.2, pp.133–163, 1996.
[9] A. Porter and L. Votta, "Comparing detection methods for software requirements inspection: A replication using professional subjects," Empirical Software Engineering: An International Journal, vol.3, no.4, pp.355–380, 1998.
[10] F.J. Shull, Developing Techniques for Using Software Documents: A Series of Empirical Studies, Ph.D. Thesis, Univ. of Maryland, 1998.
[11] T. Thelin, P. Runeson, and C. Wohlin, "An experimental comparison of usage-based and checklist-based reading," IEEE Trans. Softw. Eng., vol.29, no.8, pp.687–704, 2003.
[12] M. Halling, S. Biffl, T. Grechenig, and M. Köhle, "Using reading techniques to focus inspection performance," 27th Euromicro Workshop Software Process and Product Improvement, pp.248–257, 2001.
[13] P. Fusaro, F. Lanubile, and G. Visaggio, "A replicated experiment to assess requirements inspection techniques," Empirical Software Engineering: An International Journal, vol.2, no.1, pp.39–57, 1997.
[14] F. Lanubile and G. Visaggio, "Evaluating defect detection techniques for software requirements inspections," Tech. Rep. 08, ISERN Technical Report, 2000.
[15] J. Miller, M. Wood, M. Roper, and A. Brooks, "Further experiences with scenarios and checklists," Empirical Software Engineering: An International Journal, vol.3, no.3, pp.37–64, 1998.
[16] K. Sandahl, O. Blomkvist, J. Karlsonn, C. Krysander, M. Lindvall, and N. Ohlsson, "An extended replication of an experiment for assessing methods for software requirements inspections," Empirical Software Engineering: An International Journal, vol.3, no.4, pp.327–354, 1998.
[17] G. Sabaliauskaite, F. Matsukawa, S. Kusumoto, and K. Inoue, "An experimental comparison of checklist-based reading and perspective-based reading for UML design document inspection," 2002 International Symposium on Empirical Software Engineering (ISESE '02), pp.148–157, IEEE Computer Society, 2002.
[18] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," 2006 symposium on Eye tracking research & applications (ETRA '06), pp.133–140, ACM Press, 2006.
[19] K.A. Ericsson and H.A. Simon, Protocol analysis: Verbal reports as data, MIT Press, Cambridge, MA, USA, 1984.
[20] "The eye prize." http://hcvl.hci.iastate.edu/IPRIZE/
[21] B. Law, M.S. Atkins, A.E. Kirkpatrick, A.J. Lomax, and C.L. Mackenzie, "Eye gaze patterns differentiate novice and expert in a virtual laparoscopic surgery training environment," ACM Symposium of Eye Tracking Research and Applications (ETRA '04), pp.41–48, 2004.
[22] P. Kasarskis, J. Stehwien, J. Hichox, A. Aretz, and C. Wickens, "Comparison of expert and novice scan behaviors during VFR flight," 11th International Symposium on Aviation Psychology, http://www.aviation.uiuc.edu/UnitsHFD/ conference/proced01.pdf, 2001.
[23] R. Stein and S.E. Brennan, "Another person's eye gaze as a cue in solving programming problems," 6th International Conference on Multimodal Interface, pp.9–15, ACM Press, 2004.
[24] K. Torii, K. Matsumoto, K. Nakakoji, Y. Takada, S. Takada, and K. Shima, "Ginger2: An environment for computer-aided empirical software engineering," IEEE Trans. Softw. Eng., vol.25, no.4, pp.474–492, 1999.
[25] A. Bojko and A. Stephenson, "Supplementing conventional usability measures with eye movement data in evaluating visual search performance," 11th International Conference on Human-Computer Interaction (HCI International 2005), 2005.
[26] N. Nakamichi, M. Sakai, J. Hu, K. Shima, M. Nakamura, and K. Matsumoto, "Webtracer: Evaluating Web usability with browsing history and eye movement," 10th International Conference on Human-Computer Interaction (HCI International 2003), pp.813–817, 2003.
[27] J.K.J. Robert, Eye tracking in advanced interface design, pp.258–288, Oxford University Press, 1995.
[28] S. Zhai, C. Morimoto, and S. Ihde, "Manual and gaze input cascaded (MAGIC) pointing," SIGCHI conference on Human factors in computing systems, pp.246–253, 1999.
[29] M.E. Crosby and J. Stelobsky, "How do we read algorithms? a case study," Computer, vol.23, no.1, pp.24–35, 1990.
[30] I. Burnstein, K. Roberson, F. Saner, A. Mirza, and A. Tubaishat, "A role for chunking and fuzzy reasoning in a program comprehension and debugging tool," 9th International Conference on Tools with Artifical Intelligence, pp.102–109, 1997.

Hidetake Uwano received the BE degree in Software and Information Sciences from Iwate Prefectural University, Japan in 2004, and the ME degree in information science from Nara Institute of Science and Technology, Japan in 2006. He is currently a PhD candidate in Graduate School of Information Science, Nara Institute of Science and Technology, Japan. His research interests include human-computer interaction, human factor, and software measurement. He is a student member of IEEE and IPSJ.



Masahide Nakamura received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at University of Ottawa, Canada. He joined Cybermedia Center at Osaka University from 2000 to 2002. From 2002 to 2007, he worked for the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. He is currently an associate professor in the Graduate School of Engineering at Kobe University. His research interests include the service-oriented architecture, Web services, the feature interaction problem, V&V techniques and software security. He is a member of IEEE.

Akito Monden received the BE degree (1994) in electrical engineering from Nagoya University, Japan, and the ME degree (1996) and DE degree (1998) in information science from Nara Institute of Science and Technology, Japan. He was honorary research fellow at the University of Auckland, New Zealand, from June 2003 to March 2004. He is currently Associate Professor at Nara Institute of Science and Technology. His research interests include software security, software measurement, and human-computer interaction. He is a member of the IEEE, ACM, IPSJ, JSSST, and JSiSE.

Ken-ichi Matsumoto received the BE, ME, and PhD degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include software metrics and measurement framework. He is a senior member of the IEEE, and a member of the ACM, IPSJ and JSSST.