Doctoral Dissertation

Measuring and Characterizing Eye Movements
for Performance Evaluation of Software Review

Hidetake Uwano

Department of Information Systems,
Graduate School of Information Science,
Nara Institute of Science and Technology

February, 2009

A Doctoral dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Hidetake Uwano

Thesis Committee:
       Professor Ken-ichi Matsumoto   (Supervisor)
       Professor Hirokazu Nishitani   (Co-supervisor)
       Associate Professor Akito Monden   (Co-supervisor)
       Associate Professor Masahide Nakamura   (Kobe University)

Measuring and Characterizing Eye Movements for Performance Evaluation of Software Review [1]

Hidetake Uwano

Abstract

Software review is a technique to improve the quality of software documents and to detect defects by reading the documents. To increase review performance (the number of defect detections and/or defect detection efficiency), many review techniques and support environments have been proposed. However, the difference between individuals is more dominant than the review techniques and the other factors. Hence, understanding the factor of individual differences between a high-performance reviewer and a low-performance reviewer is necessary to develop practical support and training methods. This thesis reveals the factors affecting review performance from an analysis of the reading procedure in software review.

To analyze the reviewers' reading procedure quantitatively, this thesis proposes to use the eye movements of the reviewer. Measuring eye movements on each line and in a document allow us a correlation analysis between reading procedure and review performance. In this thesis, eye movements are classified into two types: Eye movements between lines and eye movements between documents. Two experiments analyzed the relationship between the type of eye movements and review performance.

In the first experiment, eye movements between lines of source code were recorded. As a result, a particular pattern of eye movements, called a scan, in the subjects' eye movements was identified. Quantitative analysis showed that reviewers who did not spend enough time on the scan took more time on average to find defects. These results depict how the line-wise reading procedure affects review performance. The results suggest that a more concrete direction of reading improves review performance.

Then, in the second experiment, eye movements between multiple documents (software requirements specifications, detailed design document, source code, etc.) were recorded. Results of the experiments showed that reviewers who concentrated their eye movements on high-level documents (software requirements specifications and detailed design document) found more defects in the review target document efficiently. Especially, in code

---

ii

review, reviewers who balanced their reading time to software requirements specifications and detailed design document found more defects than reviewers who concentrate to detailed design document. These results are good evidences to encourage developers to read high-level documents in review.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Improvement of software quality has become extremely important today because of the increase in large-scale software systems. Software defects (i.e. bugs and failures) in large-scale systems such as banking systems cause serious economic and social damage to system users. The defects especially in power plants, airplanes, and train systems cause fatal accidents [Leveson 95]. The literature reported some serious crises caused by software defects such as the breakdown of the train station ticket examination system and engine trouble in running automobiles [Hirayama 07].

Elimination of defects in software before delivery is a necessary activity in software development organizations. In most software projects, defects in a system are detected by testing and are removed by debugging the system. However, due to growth in the size of recent software systems, defect detection requires massive costs. In addition, many defects usually remain in delivered software, and this increases the maintenance cost for fixing defects in the field. Since the size of today's software systems keeps growing, effective/efficient defect detection techniques in software development are indispensable.

1.2 Quality Improvement Activities in Software Development

Many of techniques to improve software quality have been proposed in the field of software engineering. A brief introduction of each method is described below.

Software testing is one of the most used techniques in software development organizations. Testing activities within the software development and maintenance process verify system functionality while identifying remaining defects. Hence, testing is most often used to reveal the presence (and not the absence) of defects[Laitenberger 98a]. There are many of studies about testing, such as proposals of new testing techniques and comparison of defect detection performance with other techniques [Cleve 05, Galli 04, Jones 02, Laitenberger 98a, Laitenberger 98b]. For example, Jones et al. proposed a visualization technique of test coverage to evaluate the completeness of implemented test units [Jones 02].

Software review is also applied in the many software organizations. This software review is a peer review of software system documents such as source code or requirements specifications. This review is intended to find and fix defects (i.e., bugs) overlooked in early development phases, thus, improving overall system quality [Boehm 81]. Hundreds of studies are performed to improve the software review performance [Laitenberger 00]. In particular, studies about software review techniques are most performed. In Chapter 2, a literature review of the software review is presented.

In several organizations, which develop highly reliable software such as artificial satellites, aviation systems, and military systems, Software Independent Verification and Validation (SW IV&V) employed. SW IV&V is a set of frameworks and techniques to improve software reliability [Arthur 98, Lewis 92]. In the IV&V framework, a third organization independent from development organizations and clients can confirm the correctness of software documents. Confirmation from the third organization provides an objective and wide-range evaluation. The National Aeronautics and Space

Administration (NASA) uses IV&V techniques for confirmation of software embedded in spacecrafts.

From these quality improvement techniques, software testing and software review are widely used. In particular, the review can apply in the early phase of the development; hence, the software review is a more effective technique than testing [Melo 01, Fagan 76, Laitenberger 98a, Wiegers 02]. 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 walkthrough and inspection can discover 50 to 70 percent of defects in software products [Wiegers 02].

1.3 Problem – Individual Differences

In much of the literature on software review *individual differences* of the developer are considered to heavily affect the effectiveness of software reliability [Boehm 75, Boehm 81, Bucher 75, Davis 95, Demarco 99, McBreen 01, Myers 78, Sackman 68]. As Bucher described in his paper: "the prime factor affecting the reliability of software is the selection, motivation and management of the personnel who design and maintain it" [Bucher 75]. Several books have described the human issue as a more important factor than technical issuew such as tools, development techniques, program languages, and processes [Boehm 81, Demarco 99]. Research that analyzed the project data of software development showed how the individual differences of the developer vary from 5 times to 28 times [Boehm 75, Sackman 68].

In the quality improvement activities in software development, the performance of developer is also affected by individual differences. Myers indicated that there is a tremendous amount of variability in individual performance at inspections (a sort of software review) and software testing [Myers 78]. For example, one subject found nine defects at the inspection; on the other hand, another subject who inspected the same document found only three defects. Laitenberger et al. also showed in their paper many differences in the performance of the individuals [Laitenberger 02a]. In their

experiment, the number of defects, which the subject found, varied from zero to eighteen.

The Software Engineering Institute (SEI) at Carnegie Mellon University (CMU) proposed a People Capability Maturity Model (People-CMM) to manage and develop the workforce in software organizations [P-CMM]. The People CMM helps organizations characterize the maturity of their workforce practices, establish a program of continuous workforce development, set priorities for improvement actions, integrate workforce development with process improvement, and establish a culture of excellence.

While much of the literatures mentions the importance of individual differences in software development, only a few studies examine a factor of the differences. Analyzing the factor of individual differences from an empirical evaluation is an important way to improve software development effectiveness. That is, analyzing experienced developers' activities leads us to establish a novel development technique that encourages other developers to perform similar activities for performance improvement. Also, analyzing inexperienced developers' activities advances the training method of a novice developer. Hence, analyzing the factor of the individual differences from developers' activities is a fruitful thesis topic.

1.4   Characterizing Individual Differences in Software Review

This thesis clarifies the factor of individual differences on one of the quality improvement activities, *software review*[1]. In the review, a reviewer reads the document, understands the structure and/or functions of the system, then detects and fixes defects if any. The software review is an *off-line task* conducted by human reviewers without executing the system. Basically, software review is a human-centered activity, hence impact of the individual differences on the review is quite dominant.

---

[1]Hereafter, the word "review" indicate software review, inspection, walkthrough and/or other reading techniques.

To characterize the developers' performance in the review, we propose using the *eye movements* of the reviewer. The way of reading software documents (i.e., *reading strategy*) should vary among different reviewers. The reading strategies are 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.

In this thesis, two experiments to measure reviewers' eye movements on software review are described. To record the reviewers' eye movements, gaze-based review evaluation system was created. Using the system, the way of reading documents was analyzed from two different viewpoints: eye movements between lines and eye movements between documents.

1.5   Thesis Stamement

The main claim of this thesis is that analysis of individual differences in software review is a fruitful research topic. As mentioned in Section 1.3, performance of individual reviewers is more dominant than the review techniques and other factors. However, although hundreds of studies about software review were performed, there is no research that evaluates individual differences in software review. This thesis is characterized by analyzing individual differences in software review from empirical experiments.

Another claim of this thesis is that measuring the eye movements of reviewer is a suitable way to analyze reviewers' reading procedure quantitatively. In order to measure human activity and psychological status, biological information such as eye movements, brain waves (EEG: electroencephalograms), and heart rate variability (HRV) were used [Murata 91, Hazlett 03, Nakayama 02]. Biological information can be measured without training the subjects, and can collect quantitative, objective information. Hence, biological informantion is widely used in the field of psychology and Human Computer Interaction.

As the related work, we proposed to use an EEG to evaluate the usability of the software. An EEG measurement allows us an objective and quanti-

tative analysis of software usability. The measurement of eye movements
in software review also allow us a quantitative/objective analysis without
subject training. In addition, eye movements in the software review directly
reflect the reviewers' reading procedure. Hence, measuring eye movement
in software review is a novel, suitable method for evaluation of individual
differences.

1.6   Thesis Organization

This thesis is organized into the following chapters:

- Chapter 1 discussed the problem and an approach is described.

- Chapter 2 describes a variety of software reviews and individual dif-
  ferences in the reviews.

- Chapter 3 describes a detailed approach with related work.

- Chapter 4 reports on an experiment, which evaluates reading proce-
  dures between lines.

- Chapter 5 reports on an experiment, which evaluates reading proce-
  dures between documents.

- Chapter 6 concludes this thesis with a discussion of its contributions
  and future directions.

Chapter 2

Software Review

Software review is a technique to improve the quality of software documents and detect defects (i.e. bugs or faults) by reading the documents [Boehm 81]. In software review, a developer reads software requirements specifications, designs documents, source code, and other documents to understand the systems' functions and structures, and then detects defects from the documents.

Defect detection by review can be performed in the early phases of software development without an implemented system; therefore, rework costs can be reduced [Laitenberger 02b]. Especially in large-scale projects, defect detection and defect correction consume huge resources, and defect detection by review is necessary. A study shows that review and its variants such as walkthrough and inspection can discover 50 to 70 percent of defects in software product [Wiegers 02].

This chapter starts with explanation of target documents, which are used mainly in the experiment on software review in Section 2.1. Section 2.2 gives a list of review techniques used to improve review performance. Section 2.3 describes the effects of the review techniques and individual differences. Section 2.4 summarizes this chapter.
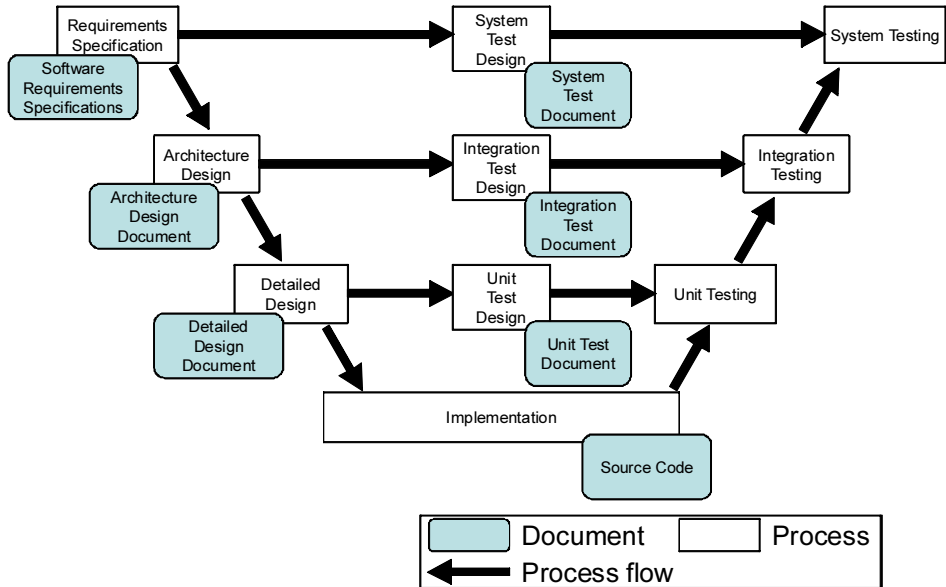
Figure 2.1: Example of the software documents created at each development process.

2.1   Target Document

In software development, several software documents are created in each development process. Figure 2.1 describes an example of the software documents created at each development process.

Basically, the developer can review every document that was created at each process. IEEE Standard lists 37 software documents as the review target [IEEE 1028-1997]. Here, we describe three documents used mainly in studies of software review.

- Software Requirements Specifications
  A software requirements specification (SRS) is a description of the purpose and environment for software under development. The SRS fully describes what the software will do and how it will be expected

to perform. Software review is the most common ways of validating an SRS [Porter 94]. The SRS is the highest level document in the software development. Defect detection in an early phase decreases the entire development costs. Hence, a software review of SRS is vital.

- Software Design Document
  A software design document (SDD) is a description of a software product that a software designer writes in order to give software development teams an overall guidance of the architecture of the software project. There are two kinds of design documents: HLDD – High Level Design Document and LLDD – Low Level Design Document. In the study of software review, SDD is written in natural language or as a Unified Modeling Language (UML).

- Source Code
  In the software development, source code is implemented after the requirement specification phase and design phase. Source code is frequently used as a review target in the literature about software review. Several studies showed that code review was significantly more effective than testing for defects detection [Melo 01, Ciolkowski 02, Laitenberger 98a].

2.2 Review Technique

Several methodologies that can be used for software review have been proposed so far. The idea behind these methods is to propose a certain criteria for reading the documents.

A review without any reading criteria is called *Ad-Hoc Reading* (AHR). *Checklist-Based Reading* (CBR) [Fagan 76] introduces a checklist with which the reviewers check typical mistakes in the document. A method, called *Perspective-Based Reading* (PBR), is used when the reviewers read the document from several different viewpoints, such as the designer's viewpoint, the programmers and the testers [Shull 00]. *Usage-Based Reading* (UBR) [Thelin 01] is reviews the document from the users' viewpoint. *Defect-Based Reading* (DBR) [Porter 95] focuses on detecting specific types of defects.

2.2.1   Checklist-Based Reading

Checklist-Based Reading (CBR) has been a commonly used technique in inspections since the 1970s. Checklists are based on a set of specific questions that are intended to guide the inspector during inspection[Sabaliauskaite 02]. Table 2.1 describes an example of a checklist for CBR[Thelin 03].

The checklist assists the reviewer in remembering which aspects are to be checked, but offers little guidance on what specifically to do. The reviewer has to map checklist questions to tasks and plan how to traverse the document. The reading process that builds up on a given checklist is often not repeatable and is prone to human variability and fallibility [Halling 01].

Checklists can cover a broad range of issues but may require the reviewer to read through the document several times sequentially, which restricts the applicability to documents of a limited suitable size, or makes the reviewer decide the AHR, which parts of the document to actually review, and which should actually be a part of the review planning.

Today, CBR is considered to be the standard reading technique in software organizations[Laitenberger 00]. Therefore, CBR is often used as a baseline method in empirical studies when investigating reading techniques.

2.2.2   Scenario-Based Reading

Scenario-Based Reading (SBR) uses procedures to detect specific classes of defects [Halling 01]. Based on the document and information that is supposed to be important to a stakeholder in development, a scenario consists of three parts: The introduction of the readers' role and interest, how to extract information, and questions, which can be answered with the extracted information. The scenarios guide readers through a document with a particular emphasis or viewpoint, and, thus, must be combined to provide complete coverage of the document.

Scenarios offer algorithmic guidance and encourage the reader to actively work with the document by taking notes, annotating the document, and con-

No.	Where to look ?	What to check ?	How to detect ?	Check when done			
1.	Modules	Consistency	Are the names of the modules consistent?				
2.		Correctness	Are the modules described correctly?				
3.		Complete-ness	Are all modules included in the design?				
4.	Signals & Parameters	Consistency	Are the names of the signals and parameters consistent?				
5.		Correctness	Is the description of the signals and parameters correct?				
6.		Correctness	Is the number of parameters correct for every signal?				
7.		Correctness	Are the signals specified related to the correct modules?				
8.		Correctness	Is the signal sequencing correct?				
9.		Complete-ness	Are all signals specified?				
10.	MSC:s	Consistency	Are the MSC:s and the signal specification consistent regarding signals and parameters?				
11.		Correctness	Are all signals included in the MSC:s specified and named correctly?				
12.		Correctness	Is the number of parameters correctly specified?				
13.		Correctness	Is the signal sequencing correct?				
14.		Complete-ness	Are all modules included?				
15.		Complete-ness	Are all signal routes specified?				
16.	Introduc-tory text	Consistency	Is the description in the SDL design consistent?				
17.		Correctness	Is the description in the SDL design correct?				
18.		Complete-ness	Is the description in the SDL design complete?				

Table 2.1: Example of checklist for CBR.

structing a mental model, which is supposed to lead to more coherence in the particular view of a reader. The scenario gives guidance on different levels of detail starting at major organizational entities and teaching inspectors how to recognize them, how to actively abstract relevant information and how to integrate this new information with the analysis so far. These steps are repeated on several levels of detail. SBR is typically distinguished by the following three types of applied scenarios.

Perspective-Based Reading

The main idea of the Perspective-Based Reading (PBR) technique is that a software product should be reviewed from the perspective of different stakeholders [Sabaliauskaite 02]. The perspectives depend on the roles people have within the software development and the maintenance process such as users, designers, implementers, and testers. To examine a document from a particular perspective, PBR technique provides guidance for the inspector in the form of a PBR scenario on how to read and examine the document.

The PBR scenario consists of three major sections: introduction (describing the quality requirements, which are most relevant to this perspective); instructions (describing what kind of documents to use, how to read them, and how to extract the necessary information) and questions (a set of questions which the inspector has to answer during the inspection). The main objective of the instructions for reading a document from different perspectives is to gain a better defect detection coverage of a software artifact.

An example of scenario (tester perspective) used in PBR is as follows [Ciolkowski 97]:

- Do you have all the information necessary to identify the item being tested and to identify your test criteria? Can you make up reasonable test cases for each item based upon the criteria?

- Is there another requirement or functional specification for which you would generate a similar test case, but would get a contradictory result?

- Can you be sure that the test you generated should yield the correct value in the correct units?

- Are there other interpretations of this requirement that the implementer might make based upon the way the requirement or functional specification is defined? Will this affect the tests you make up?

- Does the requirement or functional specification make sense from what you know about the application or from what is specified in the general description?

Usage-Based Reading

The principal idea behind Usage-Based Reading (UBR) is to focus the reading effort on detecting the most critical faults in the review target. Hence, faults are not assumed to be of equal importance, and the UBR method is aimed at finding the faults that have the most negative impact on the users' perception of system quality. The UBR method focuses the reading effort guided by a prioritized, requirements-level use case model during the individual preparation of a software review.

In the UBR, reviewers read the design document by following the procedure:

1. Select the use case with the highest priority.

2. Trace and manually executing the use case through the design document and use the requirements document as a reference.

3. Ensure that the document under review fulfills the goal of the use case, and that the needed functionality is provided, that the interfaces are correct, etc. Identify and report the issues found.

4. Repeat the review procedure using the next use case until all use cases are covered, or until a time limit is reached.

Data type inconsistencies
  1.a Are all data objects mentioned in the ...
  2.a Is the object's specification consistent ...
  ...

Incorrect functionality
  1.a Are all values written to each output ...
  1.b Identify at least one function that uses ...
  ...

Missing (or ambiguous) functionality
  1.a Identify the required precision, response ...
  2.a Is the specification of these events ...
  ...
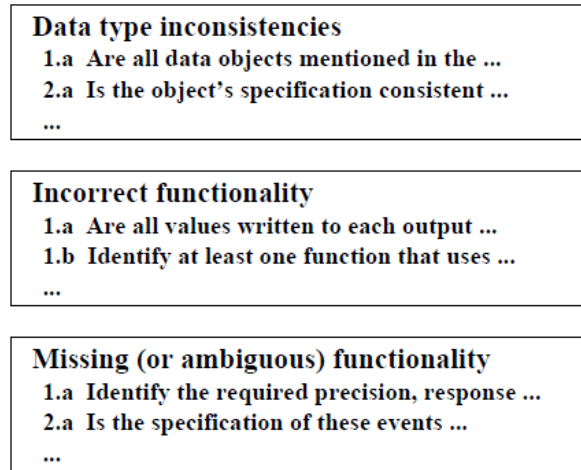
Figure 2.2: Example of questions used at DBR.

Defect-Based Reading

The main idea behind Defect-based Reading (DBR) is for different reviewers
to focus on different defect classes while scrutinizing the requirements docu-
ments [Laitenberger 00]. For each defect class, there is a scenario consisting
of a set of questions an inspector has to answer while reading. Answering
the questions primarily helps an inspector detect defects of the particular
class.

The defect-based reading technique has been validated in a controlled
experiment with students as subjects. The major finding was that inspectors
applying Defect-based Reading detect more defects than inspectors applying
either Ad-hoc or checklist-based reading. Figure 2.2 illustrates an example
of questions used at DBR [Porter 95].

2.2.3 Ad-Hoc Reading

Ad-hoc reading, by nature, offers very little reading support since a software product is simply given to inspectors without any direction or guidelines on how to proceed through it and what to look for [Laitenberger 00]. However, Ad-hoc does not mean that inspection participants do not scrutinize the inspected product systematically. The word 'Ad-hoc' only refers to the fact that no support is given to them. In this case, defect detection fully depends on the skill, the knowledge, and the experience of an inspector who may compensate for the lack of reading support.

2.3 Impact of Review Techniques and Individual Differences

To evaluate the performance of review techniques, hundreds of empirical studies have been conducted [Ciolkowski 02]. Some empirical reports have shown that CBR, which is the most used method in the software industries, is not more efficient than AHR[Porter 98, Porter 95].

UBR, PBR and DBR achieved a slightly better performance than CBR and AHR [Basili 96, Porter 98, Porter 95, Shull 98, Thelin 03]. Porter et al. showed in their study that in most cases DBR reviewers were more effective than CBR or AHR reviewers at finding the faults the scenario was designed to uncover [Porter 95]. At the same time, all reviewers, regardless of which detection method each used, were equally effective at finding those faults not targeted by any of the scenarios.

On the other hand, Halling et al. [Halling 01] report an opposite observation that CBR is better than PBR. In their work, reviewers who use a checklist are more effective regarding all defects in the document than reviewers who use a scenario. Several case studies have shown that these methods had no significant difference [Fusaro 97, Lanubile 00, Miller 98, Sabaliauskaite 02, Sandahl 98].

Sabaliauskaite et al. compared CBR and PBR when reviewing multiple

documents of UML[Sabaliauskaite 02]. In the experiment, four diagrams —
class, activity, sequence and component — were used in the review. Their
results revealed that PBR required less time than CBR to find defects on
average, but had no significant difference. For the period of the detection
rate, the difference between CBR and SBR was not observed.

The reason why the results vary among the empirical studies is that the
*performance of individual reviewers is more dominant than the review tech-
nique itself.* This is because the review task involves many *human factors.*

Thelin et al. [Thelin 03] compared the effectiveness between UBR and
CBR. Figure 2.3 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 com-
prises 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
CBR on average. However, as seen in the dotted lines in the figure, the in-
dividual performance in the same review technique varies much more than
the technique-wise difference.

Laitenberger et al. mentioned individual differences of review perfor-
mance in their study [Laitenberger 02a]. Figure 2.4 shows individual dif-
ferences in a review, which was performed in their experiment. The Figure
shows the inspector's (reviewer) performance across the inspections per-
formed. The number at the top indicates the number of inspections in
which an inspector participated. Moreover, the number of defects detected
are the ones individually detected normalized with the number of functional
requirements (size measure) to remove the size effect. As can be seen, there
is quite a difference in the performance of the individuals.

These results showed that the individual performance in the same re-
view technique varies much more than the method-wise difference. That is,
large differences of individuals in the same review technique come from more
detailed reviewers' activities, not from abstract reading guidelines. Foe ex-
ample, PBR instructs reviewers to use perspectives of stakeholders such as
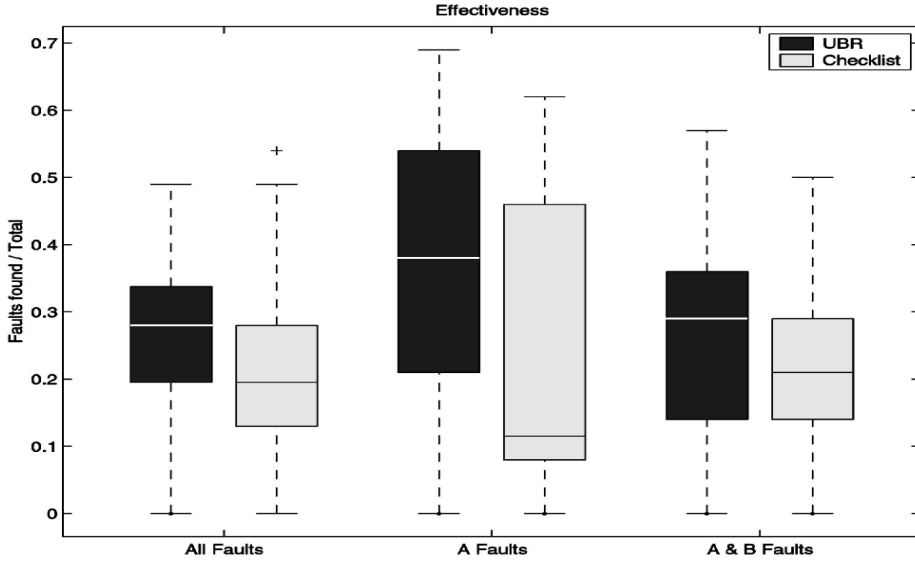
Figure 2.3: Effectiveness of Usage-Based Reading and Checklist-Based Reading.

user, developer, tester, and so on. Each perspective gives scenarios describing how the reviewer reads the software documents. However, this scenario shows only guidelines or criteria, and a more detailed/concrete reading procedure is not described. Hence, each reviewer reads the documents in their own way. The results of previous studies showed that the individual differences of detailed reading procedure are a major factor of review performance.

Analyzing the detailed reading procedure in the review from empirical evaluation is an important way to improve software development effectiveness. That is, analyzing experienced developers' activities leads us to establish a novel development technique that encourages other developers to perform similar activities for performance improvement. Also, analyzing inexperienced developers' activities advances a training method for the novice developer. Hence, analyzing the factor of individual differences from the
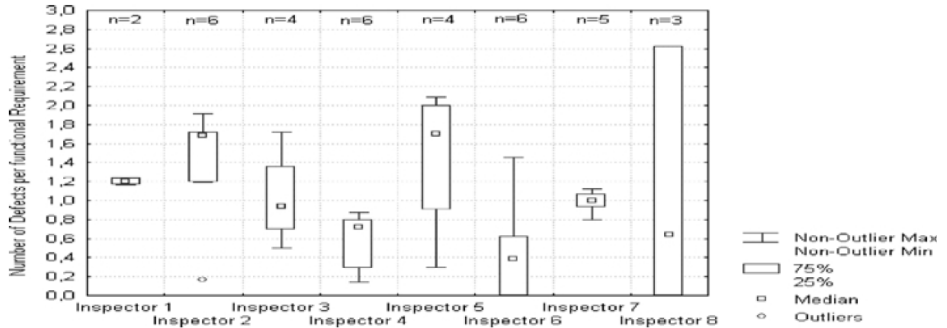
Figure 2.4: Individual differences in a review.

developers' activities is a fruitfl research activity.

2.4   Chapter Summary

This chapter discussed review target documents and review techniques. The software review is performed on a number of software documents, with different techniques, such as CBR and PBR. This chapter also described the impact of individual differences in the software review. From the viewpoint of review performance, individual differences are more dominant than technique-wise differences.

Chapter 3

Eye Movement in Software Review

To characterize the reviewers' performance in an objective way, we propose to use *eye movements* of the reviewer. The way of reading software documents (i.e., *reading strategy*) should vary among different reviewers. The reading strategy 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 performance in the software review.

Advantages of using eye movements to evaluate software review are as follows:

- **Measurement of detailed review activity**
  Generally, most knowledge of the reviewer is difficult to express in words; hence, a common analysis method such as an interview or a *think-aloud protocol* [Ericsson 84] could not extract the reviewers' characteristics. Detailed analysis of eye movements will capture the reviewers' knowledge from their reading procedure.

- **Easy to measure**
  The second advantage in adopting the eye movements is that the eye movements provide us a data without any training of the reviewers. A related work, a method called think-aloud protocol, tapes the au-

dio and video characteristics 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 reviewers.

- **Quantitative evaluation**
  Using the eye movements allows us to observe the reviewers' reading procedure with quantitative data. Quantitative analysis of eye movements and review performance data (i.e. the number of defect detection, detection time per review time) will reveal the correlation between reading procedure and review performance.

This chapter begins with an explanation of two measurement perspectives for eye movement evaluation in Section 3.1. Section 3.2 defines terminology used in this following thesis. Section 3.3 relates my thesis to previous studies of eye movements. Section 3.4 summarizes this chapter.

3.1   Measurement Perspectives

We chose two measurement perspectives for a structured analysis of eye movements: (1) Eye movement between lines, and (2) Eye movement between documents.

3.1.1   Eye Movement between Lines

One of the measurement perspectives is eye movements between lines in a document. 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 *software requirements specifications*.

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. On the other hand, the reviewer who reads the source code without the control flow (i.e. reads from the top of the file) requires more review time to understand the program. Figure 3.1 describes
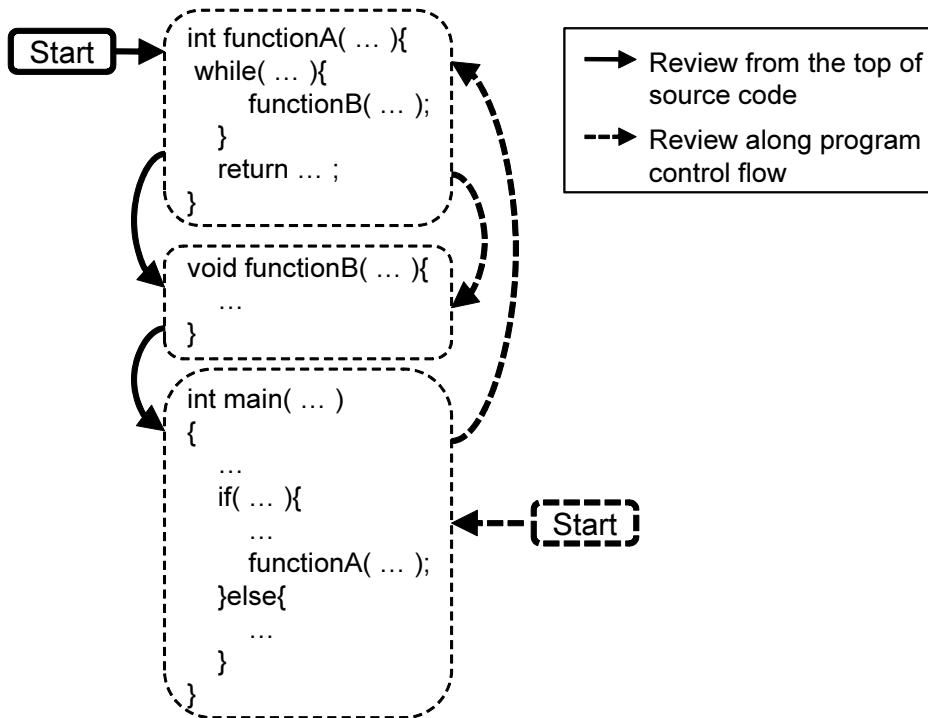
Figure 3.1: Reading procedure between lines.

an example of different reading procedures, review along program control flow, and review from the top of source code.

The software requirements specification is typically structured, where a requirement contains several sub-requirements. Each requirement is written in a *labeled paragraph*, the set of lines. 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.

As seen in above examples, a primary construct of a software document

is a *statement*. Thus, it is reasonable to consider that the reviewer reads the document in units of *lines*. Therefore, reading procedure between lines in a document affects review performance. In the first experiment reported in Chapter 4, we evaluate the relationship between review performance and reading procedure between lines.

3.1.2   Eye Movement between Documents

Another measurement perspective is an eye movement between documents. According to Wiegers, software review in the industry uses not only the target document but also other relevant documents, such as a **High-level document** [Wiegers 02]. The reviewer reads the high-level document to confirm the target document correctly contains the system requirements.

For example, in source code review, reviewers read source code as well as the requirements specification and design document to understand system structures, functions, and data structures.

Figure 3.2 describes the relationship between source code and other documents at source code review. Source code has several blocks of functions, methods, classes, etc. The reviewer reads all the blocks to understand the program entirely (e.g. through *Function A* to *Function C*) and tries to find any defect during program understanding. In addition, the reviewer reads related blocks in the system requirements specification as well as in the detailed design document (e.g. *Requirement A* and *Design A*) to find any inconsistencies among different levels of documents. This activity is a "comparison" rather than an "understanding."

As in other relevant documents, the checklist in CBR and the scenario in SBR are also used in the review [Laitenberger 00, Sabaliauskaite 02]. In the review using these relevant documents, the reviewer reads the target document and relevant document alternately to follow these scenarios.

In such multi-document review, the reading procedure between documents affects the defect detection performance. In the second experiment
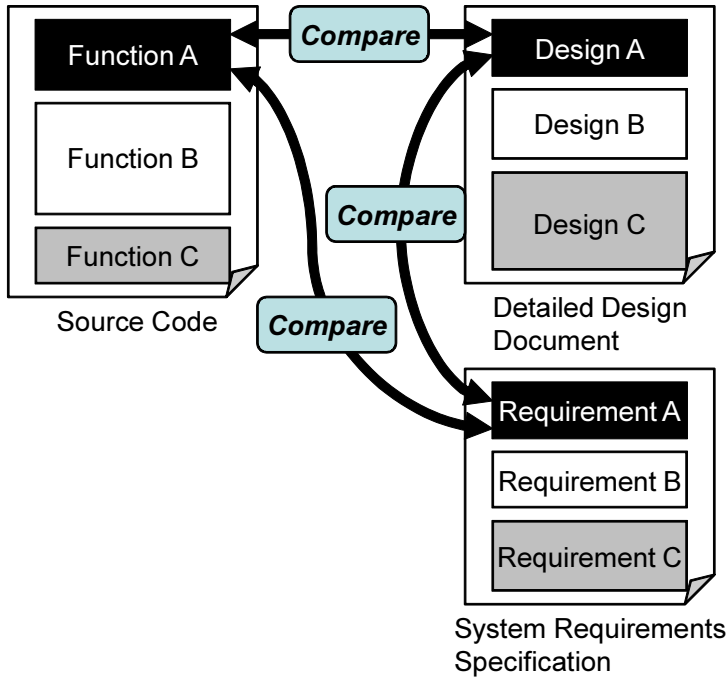
Figure 3.2: Source code review with multiple documents.

reported in Chapter 5, we evaluate the relationship between review performance and the reading procedure between documents.

3.2 Terminologies

Here, we define several technical words used throughout this thesis. A *gaze point* over an object is the point on the object where the user is currently looking. Strictly speaking, the gaze point refers to an intersection of the users' sight line and the object. A *fixation* is a condition where the 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 the interests of the user.

The pair (f_a, f_t) characterizing the fixation is called *fixation criteria*. In this thesis, we determined the fixation criteria as the area of 30 pixels in diameter where the eye mark stays more than 50ms. The *fixation point* is a gaze point where the fixation criteria hold. The *fixation time* describes how long the gaze points fixate to the criteria.

3.3   Related Work

Eye movements have been used often for the purpose of evaluating human performance, especially in cognitive science. Law et al. [Law 04] analyzed eye movements of experts and novices in a laparoscopic surgerical training environment. This study showed that experts tend to watch the affected parts more than the tool in their hands, compared with novices. Kasarskis et al. [Kasarskis 01] investigated eye movements of pilots in a landing task using a flight simulator. In this study, novices tended to concentrate more on watching the altimeter than the experts did, while the experts watched the airspeed.

In the field of software engineering, research exists regarding to the exploitation of eye movements, for the purpose of, for instance, monitoring an online debugging processes [Stein 04, Torii 99], usability evaluation [Bojko 05, Nakamichi 03], human interface [Robert 95, Zhai 99].

Stein and Brennan evaluated the usefulness of eye movement information for the support of the debugging process [Stein 04]. They captured the eye movements of experienced developers while the developers detected a defect from a source code. The authors then compared the defect detection time of the developers who read the source code with the visualized eye movements of an expert and a developer without visualized eye movements. The results showed that visualized eye movements of experts accelerate the defect detection of other developers.

As far as we know, no research has directly applied the eye movements to evaluate the performance of the software review.

Few studies have conducted an analysis of a software developers' eye movements. Crosby et al. [Crosby 90] and Bednarik et al. [Bednarik 05] analyzed developers' eye movements in program understanding. In these studies, the source code and visualized program behavior were displayed to the developer. The authors confirmed that eye movements are useful in revealing differences between the experts' and novices' program reading behavior. While these studies focused on effective program understanding where executable programs are available, we focus on document review where related upstream documents are available, but executable programs are not.

3.4   Chapter Summary

This chapter showed our proposed method for the evaluation of the reading procedure of the reviewers. Using the eye movements of the reviewer, the reading procedures are evaluated from two measurement perspectives. The following two chapters report experiments performed to evaluate eye movements between lines and eye movements between documents.

Chapter 4

Eye Movement between Lines

This chapter reports on an empirical experiment of code review for evaluating eye movements between lines. Using eye movements to measure an environment, which we made to evaluate reading procedure on software review, 60 review processes were analyzed. As a result, we have identified a particular pattern, called scan, in the subjects' eye movements. Quantitative analysis showed that reviewers who did not spend enough time for the scan tend to take more time for finding defects.

This chapter begins by clarifying system requirements to evaluate the reading procedure on software review as described in Section 4.1. Section 4.2 gives an explanation of the system implementation that we have developed based on the requirements. Section 4.3 explains the experimental settings and materials. Section 4.4 describes the results of the experiment. Section 4.5 summarizes this chapter.

4.1   System Requirements

To evaluate reviewers' eye movements, an integrated environment to measure and record the eye movements during the code review is necessary. In this Section, we present five requirements to be satisfied by the system.

Requirement R1: Sampling Gaze Points over a Computer Display
First, the system must be able to capture the reviewers' gaze points over the software documents. Usually, reviewed documents are either shown on the computer display, or provided as printed papers. Considering 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 on a one-statement-per-line basis. Thus, 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 appears at a certain line in the document, it does not necessarily mean that the reviewer is reading that line. That is, the system has to be able to distinguish a focus (i.e., interest) from reviewers' eye movements. It is reasonable that the fixation over a line reflects the fact that the reviewer is currently reading that 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 that line. The duration of the focus may indicate the strength of the reviewers' attention to the line. Therefore, the system must record the lines focused on as *time sequence* data.
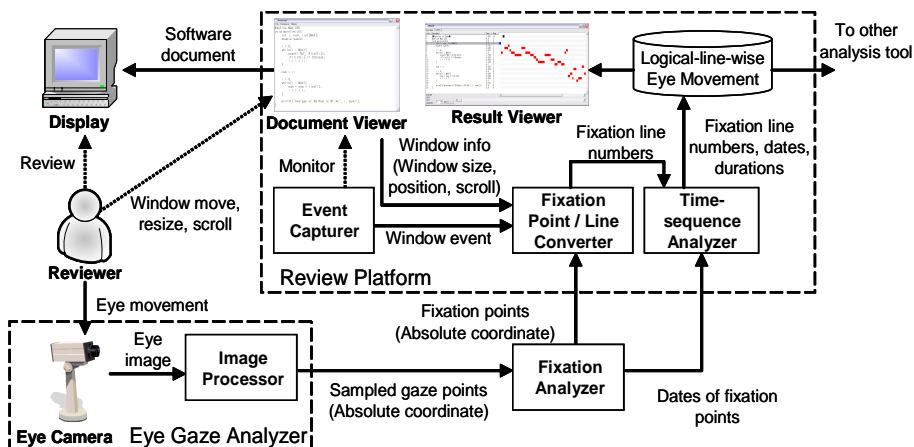
Figure 4.1: System architecture of DRESREM.

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 to novice reviewers for subsequent interviews or for educational purposes.

4.2   Implementation

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

4.2.1   System Architecture

As shown in Fig. 4.1, DRESREM is composed of three sub systems: (1) an *eye gaze analyzer*, (2) a *fixation analyzer* and (3) a *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 image processing, the gaze points are sampled as absolute coordinates. Next, the fixation analyzer converts the sampled gaze points into fixation points into 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 support for the analysts.

In the following subsections, a more detailed explanation for each of the sub systems is given.

4.2.2   Eye Gaze Analyzer

To achieve Requirement R1, the eye-gaze analyzer samples reviewers' 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/`). Figure 4.2 describes the eye-gaze analyzer used in the system. 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: Sampling Gaze Points over Computer Display.

EMR-NC consists of an eye camera and image processor. The system detects reviewers' eye images, 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. 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 resolutions with a dot pitch of 0.3893 millimeters. To minimize the noise data, we prepared a fixed and non-adjustable chair for the reviewers.

Figure 4.2: Eye gaze analyzer EMR-NC.

4.2.3 Fixation Analyzer

For a given fixation criteria (see Section 3.2) and the gaze points sampled
by the eye gaze analyzer, the fixation analyzer derives fixation points (as
absolute coordinates) and their observation date. Extracting the fixation
points from the gaze points is necessary to achieve Requirement R3: Identi-
fying Focuses. To implement the fixation analyzer, we have used the existing
analysis tool `EMR-ANY.exe`, which is a bundled application of EMR-NC.

4.2.4 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 a SWT (Standard Widget Tool), comprising about
4,000 lines of code.

What is most technically challenging is to satisfy Requirement R2: Ex-
tracting Logical Line Information from Gaze Points. 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 the *fixation line*. The goal is to capture the line numbers of the fixation lines.

Note that the line numbers must be captured as *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 to such correspondence as *point/line correspondence*.
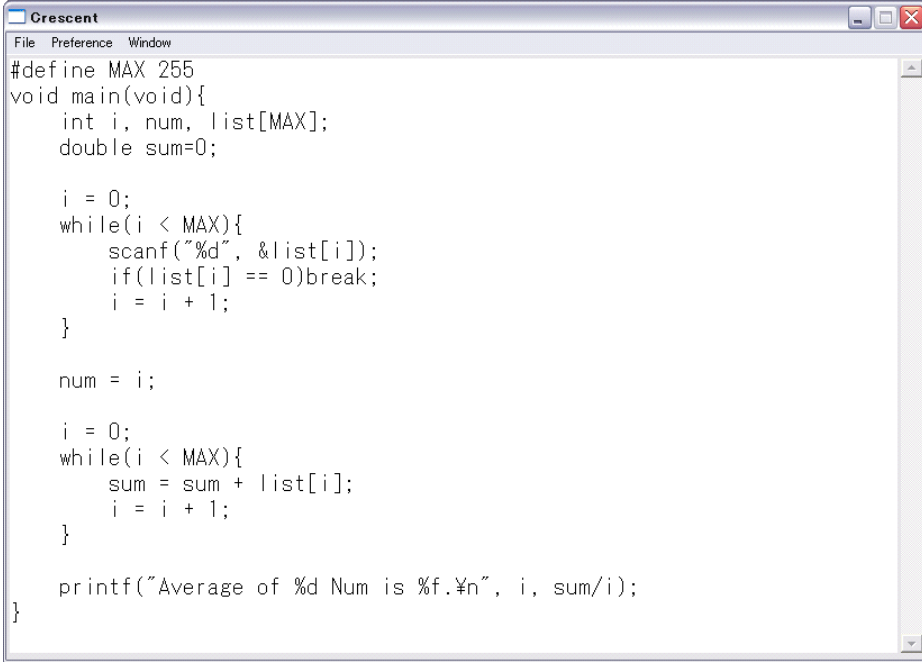
As seen in Fig. 4.1, the review platform consists of the following five components.

Document Viewer

The document viewer shows the software document to the PC display, with which the reviewer reads the document. As shown in Fig. 4.3, the viewer has a slider bar to scroll up and down 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 consistent point/line correspondence.

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 notes the event and forwards it to the fixation point/line converter.

```
Crescent                                                         _ □ ✕
File  Preference  Window
#define MAX 255
void main(void){
    int i, num, list[MAX];
    double sum=0;

    i = 0;
    while(i < MAX){
        scanf("%d", &list[i]);
        if(list[i] == 0)break;
        i = i + 1;
    }

    num = i;

    i = 0;
    while(i < MAX){
        sum = sum + list[i];
        i = i + 1;
    }

    printf("Average of %d Num is %f.¥n", i, sum/i);
}
```

Figure 4.3: Example of document viewer.

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:

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

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

Note that l_{p_r} is changed by the users' 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 $p_{w'}$. The converter is then 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: Extracting Logical Line Information from Gaze Points.

Time-Sequence Analyzer

The time-sequence analyzer summarizes the fixation line numbers as time-sequenced data to satisfy Requirement R4: Recording Time-Sequenced Transitions. Using the date information sampled by the fixation analyzer, the time-sequence analyzer sorts the fixation line numbers by date. This is done to represent the order of lines in which the reviewer reads the document. It also aggregates successive appearances of the same fixation line number into one with the *duration*. The duration for a fixation line then reflects the strength of the reviewers' interest in the line.

Result Viewer

The result viewer visualizes the line-wise eye movements using a horizontal bar chart, based on the time-sequenced fixation line numbers. Figure 4.4
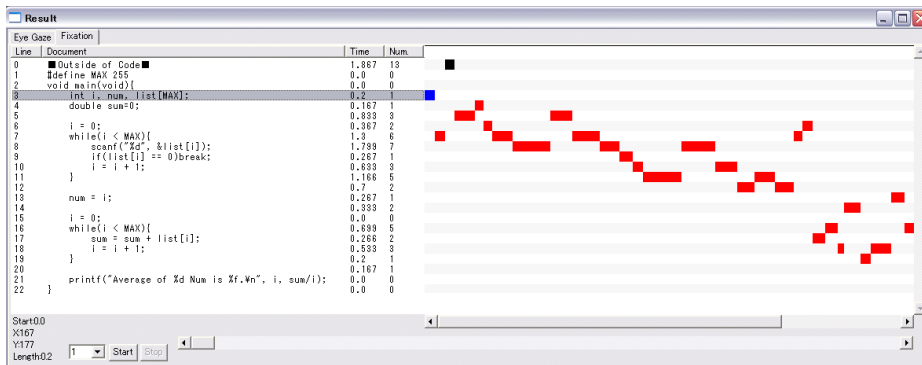
Figure 4.4: Result viewer.

shows a snapshot of the result viewer. In the figure, the left side of the window shows the document reviewed by the reviewer. On 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 a slider bar placed under the viewer, the analyst can control the replay position and the speed. On the result viewer, the time-sequenced transition of fixation lines is described by the highlighting of the line and the emphasis of bar. Moreover, the result viewer has a feature that can *superimpose* the recorded gaze points and fixation points onto the document viewer. This feature helps the analyst 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: Supporting Analysis.

4.3   Experiment

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

4.3.1   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 the reviewers' 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 quickly 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.

4.3.2   Experiment Settings

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

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. Next, in each program a single *logical* defect was intentionally injected, which is an error of program logic, but not

of program syntax. Table 4.1 summarizes the programs prepared for the experiment.

We then instructed individual subjects to review the six programs with DRESREM. The review technique was the ad-hoc review (AHR, see Sect. 2.2.3). The task for each subject to review each program consisted 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. Synchronize the subject to start the code review to find the defect; start the capture of eye movements and code scrolling.

4. Suspend the review task when the subject 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 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, a total of 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 these effects, we have used the *Latin square* to shuffle and balance the order.

4.4 Results

4.4.1 Qualitative Analysis

After the experiment, we investigated the recorded data. Using the result viewer extensively, we played back the eye movements of the individual re-

Table 4.1: 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.

viewers, 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 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 at in the first 30 percent of the review time. We call this preliminary reading of the entire code, the *scan pattern*.

Figures 4.5 and 4.6 describe the eye movements of two subjects C and E while 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. 4.5, this subject scans the code twice, then concentrates on the while loop block located in the middle of the code. In Fig. 4.6 this subject first locates the headers of two function declarations in lines 1 and 13. Next, the subject scans 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 identifies suspected portions where the defect is likely to exist. Therefore, scan quality significantly influences the efficiency of the defect detection in the review.

4.4.2   Quantitative Analysis of Scan Pattern

To verify the hypothesis, we conduct a quantitative analysis using the recorded data as follow. For each review in the experiment, we have measured a *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
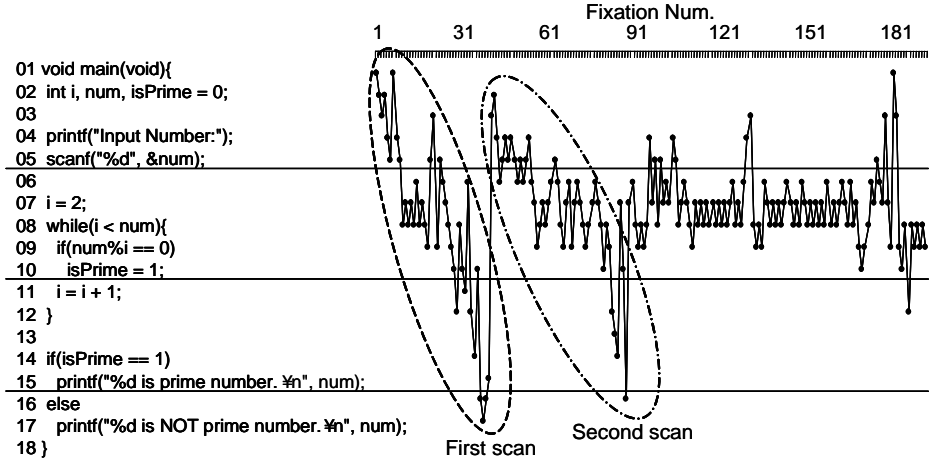
Figure 4.5: Eye movements of subject E reviewing program `IsPrime`.

as a metric characterizing the quality of the scan.

Note that both DDT and FST depend deeply on the *reading speed* of the reviewer. That is, a slow reader tends to spend more time to scan and for defect detection than a fast reader. The reading speed varies from subject to 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 scanning (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. Thus, we define *normalized defect detection time (nDDT)* and *normalized first scan time (nFST)* as follows.

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

$$nFST(r,p) = \frac{FST(r,p)}{\sum_{p' \in Prog} FST(r,p') \ / \ |Prog|}$$
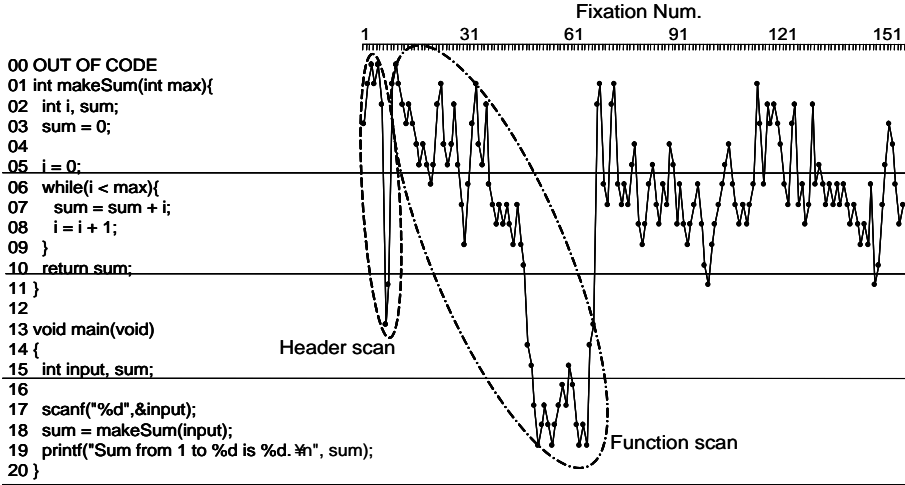
Figure 4.6: Eye movements of subject C reviewing program `Accumulate`.

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

Figure 4.7 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 $nDDT$. Pearsons' product moment showed a significantly negative correlation between $nFST$ and $nDDT$ ($r = -0.568, p = 0.002$). That is, the first scan time is less than the average, and yields the longer defect detection time. More specifically, the defect detection time increased to 2.5 times of the
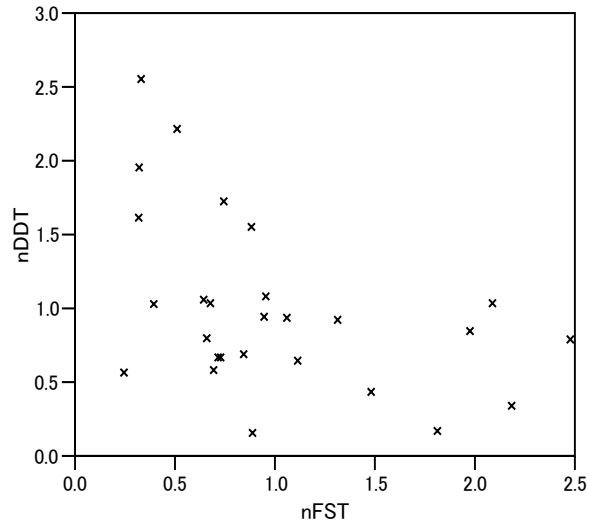
Figure 4.7: Normalized first scan time and defect detection time.

average detection time when the first scan time is less than 1.0. On the other hand, in the case where 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. 4.5 and 4.6, the reviewers focus their eye movements on a particular block or a function after the scanning of the code.

On the other hand, a reviewer with insufficient scanning often misses some critical code lines, and they stick to irrelevant lines involving no defects. Figure 4.8 depicts typical eye movements that could not address suspicious lines through the scanning of a program `IsPrime`, which is the same source
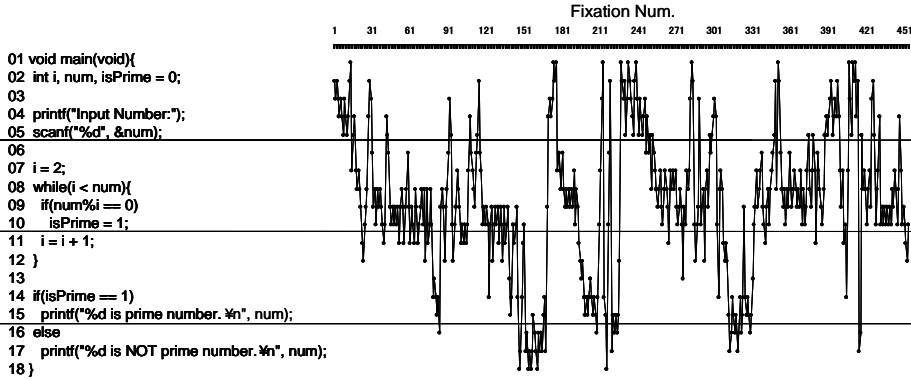
Figure 4.8: Eye movements of subject B reviewing program `IsPrime`.

code as in Fig. 4.5. This reviewer spent insufficient scanning time compared with his/her average scanning time ($nFST = 0.51$), and the reviewer could not detect the defect. Of course, our hypothesis has been proven only with this experiment. For more generality, we plan to continue more experiments in our future research.

4.4.3   Using Recorded Data for Review Training

After the experiment, we conducted two kinds of interviews to investigate what the eye movements actually reflect. In the first interview, each subject was shown the source code and asked what the subject had been thinking in the code review. Most subjects commented about 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 4.2.

In the second interview, the recorded eye movements were shown using the result viewer together with the source code, and the same questions were asked. As a result, more detailed and code-specific comments were gathered. As shown in the second column of Table 4.2, each subject explained reasons why he checked some particular lines carefully and why not for other lines. It seems that the record of the eye movements well reminded the subjects

Table 4.2: Comments gathered in interviews.

First interview (with source code only)	Second interview (with source code and eye movements)
· I thought something was wrong in the second while loop.	· I did not care about the conditional expression of the loop.
· First, I reviewed main function, and then read another one.	· I watched this variable declaration to see the initial value of the variable.
· I simulated the program execution in my 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 several times.	· I could not understand why this variable was initialized here.

of their thought.

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

4.5   Chapter Summary

In this chapter, 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 reviewers' eye movements, and derives the sequence of logical line numbers of the document in which the reviewer has focused. Thus, the system allows quantitative evaluation of how the reviewer reads the document.

An experimental evaluation of the source code review using DRESREM has therefore been conducted. As a result, a particular reading pattern, called a scan, has been found. Through the statistic analysis, it was shown that the reviewers who took sufficient time to scan the code tended 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 information reflecting the reviewers' thoughts during the code review.

Chapter 5

Eye Movement between Documents

This chapter reports on an experiment of review for evaluating eye movements between documents. For the measurement of eye movements between documents, the measurement environment was improved.

Using the extended system, 24 review processes were recorded. In the experiment, two types of review, design review, and code review were employed. The reasons why these reviews were employed are as follow: (1) multiple documents were used in these reviews, (2) both reviews require elaborate/detailed reading because the documents have more detailed information than other documents, such as SRS.

This chapter begins by redefining the requirements of the evaluation system for analyzing eye movements between documents as mentioned Section 5.1. Section 5.2 describes metrics for evaluation of eye movements in the experiment. Section 5.3 describes the hypotheses verified in the experiment. Section 5.4 explains the experiment settings and materials. Section 5.5 discusses the results of the experiment. Section 5.6 summarizes this chapter.

5.1   System Improvement

To observe multi-document review activities, the measurement environment must identify which document the reviewer reads. Usually, multiple documents were displayed in multiple windows or in a window that has a tab to switch documents displayed in the window; hence, documents can be overlapped with other documents during review tasks. This means that the current focus of the reviewer cannot be identified from the coordination of eye movements alone. Therefore, the system should have a functionality to identify which document the reviewer is currently focused on by recording tab-switching activities and window-focusing activities.

To achieve the modified requirement, DRESREM was improved. Figure 5.1 shows the architecture of improved system, DRESREM 2.

The Review Platform is improved to show multiple documents for the reviewers and to record their operations. The procedure of recording the reviewers' eye movements and operations is as follows. (Figure 5.1). Documents used in the review are displayed in the Document Viewer. The eye gaze analyzer outputs the reviewers' gaze points, represented as coordinates (x, y) on a display. These sampled gaze points are converted to fixation points by the Fixation Analyzer. The **Window Event Capturer** observes user operations on a Document Viewer and records Window information, i.e., window position and window size, and current scroll position (line number) of the document currently focused on.

Reviewer operations such as defect description recording, keyword searching and note taking are recorded by **Operation Recorder**, and then the **Review Information Integrator** combines the operations and eye movements to create the time series data of the review history.

Recorded eye movements and operations are visualized in the **Result Viewer**. Figure 5.2 shows an example of visualized eye movements and operations in a source code review. In this figure, the left side of the window shows a source code that is read in the review, and the right side of the window describes eye movement fixations and operations as a bar chart.
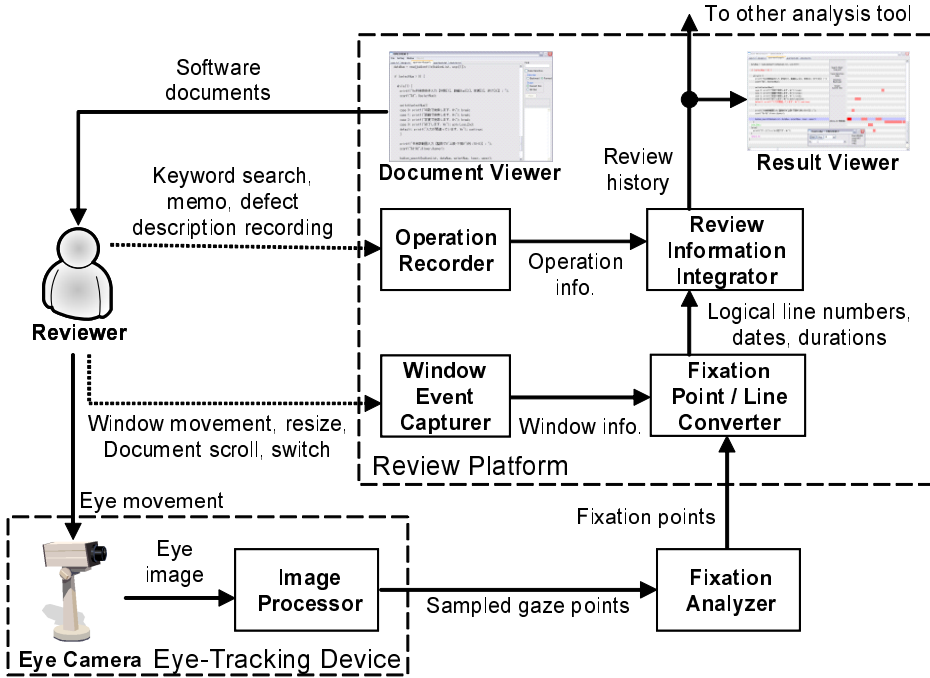
Figure 5.1: Improved Architecture of DRESREM 2.

Also, the sequence of the eye movements can be played back in this window.

DRESREM 2 outputs review history (i.e. time series of eye movements and operations) as three type formats: document-wise, block-wise, and line-wise. In each format, eye movements were recorded as a series of fixations on documents, blocks, or lines. These formats allow users to easily analyze the review history from different granularities.

Figure 5.3 and Figure 5.4 show examples of the reviewers' eye movements recorded by the system. Using the quantitative data of the eye movements, the system allows a statistical analysis of the reviewers' activity.
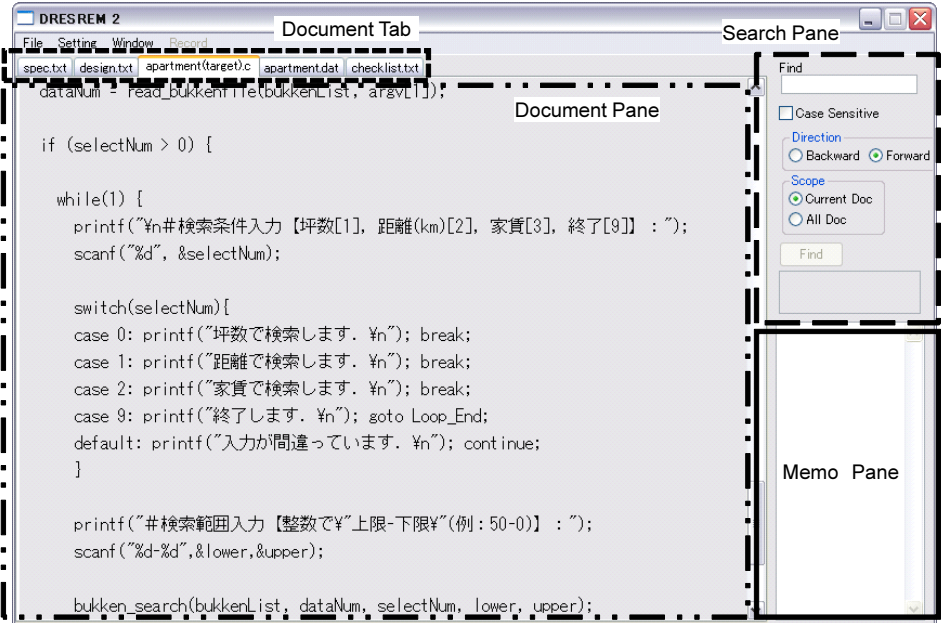
Figure 5.2: Screenshot of Review Platform.

5.2   Metrics

In this Section, two metrics of review performance are defined. *Review Quality* (RQ) is defined as the ratio of defect detection in a review. RQ characterizes how completely defects are detected from the document. *Review Efficiency* (RE) is defined as the average detection time needed to detect a defect from the document. RE characterizes how quickly defects are detected in a review.

Let us assume that there is a review target document *doc* and a set of high-level documents $HD = \{hd_1, hd_2, \ldots, hd_l\}$. Here, we suppose *doc* has n defects $F_x = \{f_1, f_2, \ldots, f_n\}$ which is categorized as defect type x. Now, a reviewer spent time t to review *doc* and detect defects $F_{x,found} = \{f_{i1}, f_{i2}, \ldots, f_{ik}\} \subseteq F_x$ from F_x. Here, review quality (RQ) and review

efficiency (RE) of defect type x in the *doc* is described as follows:

$$RQ(doc, x) = |F_{x,found}|/|F_x| = k/n$$

$$RE(doc, x) = t/|F_{x,found}| = t/k$$

The larger value of RQ shows that the review is performed more extensively, and the smaller value of RE shows that the review is performed efficiently.

We also introduce the metric *Gazing Time Ratio* (GTR) to evaluate how long it takes reviewers to read each document in the review. This metric is described as the ratio of fixation time to each document, and it characterizes the concentration to review time to the document.

Here, $gtime(d)$ describes the fixation time to a document $d(\in doc \cup HD)$. The gazing time ratio to d is defined as the following formula.

$$GTR(d) = \frac{gtime(d)}{\displaystyle\sum_{d' \in doc \cup HD} gtime(d')}$$

These three metrics are collected at each review. Hence, metrics recorded at each review instance r are described as $r.RQ(\ldots)$ $r.RE(\ldots)$ $r.GTR(\ldots)$.

5.3   Hypotheses

The following six hypotheses are about detailed design review and code review. These hypotheses describes how the review performance increases while a high-level document is read in both reviews.

In the following formula, SRS, detailed design document, and Source code are described as [Req], [Design], and [Code], for short. Here, r r' shows a review instance, and x describes defect type.

Hypothesis DQ: Review Quality in design review
In design review, reviewers who spend more time to read the SRS find more defects on average.

$$r.GTR([Req]) > r'.GTR([Req])$$
$$\Rightarrow r.RQ([Design], x) > r'.RQ([Design], x)$$

Hypothesis DE: Review Efficiency in design review
In design review, reviewers who spend more time to read the SRS find defects efficiently (with less time) on average.

$$r.GTR([Req]) > r'.GTR([Req])$$
$$\Rightarrow r.RE([Design], x) < r'.RE([Design], x)$$

Hypothesis SQ: Review Quality in code review
In source code review, reviewers who spend more time on average to read the SRS and the detailed design document find more defects. We verify this hypothesis for each high-level document.

Hypothesis SQ_{Req}
$$r.GTR([Req]) > r'.GTR([Req])$$
$$\Rightarrow r.RQ([Code], x) > r'.RQ([Code], x)$$

Hypothesis SQ_{Design}
$$r.GTR([Design]) > r'.GTR([Design])$$
$$\Rightarrow r.RQ([Code], x) > r'.RQ([Code], x)$$

Hypothesis SE: Review Efficiency in code review
In source code review, reviewers who spend more time on average to read

the SRS and the detailed design document find defects more efficiently (with less time). We verify this hypothesis for each high-level document.

Hypothesis SE_{Req}
$r.GTR([Req]) > r'.GTR([Req])$
$\qquad \Rightarrow r.RE([Code], x) < r'.RE([Code], x)$

Hypothesis SE_{Design}
$r.GTR([Design]) > r'.GTR([Design])$
$\qquad \Rightarrow r.RE([Code], x) < r'.RE([Code], x)$

5.4 Experiment

5.4.1 Overview

In the experiment, subjects were asked to find defects from a target document in design review and code review. Documents used in the experiment were about a rental house search system actually used in an industrial training workshop.

First, subjects performed the design review with four documents (the SRS, the detailed design document, the data file, and a checklist for design review) to detect defects injected into the detailed design document beforehand (See Section 5.4.2 for more details). Next, the subjects performed code review with five documents (the SRS, the detailed design document, C source code, the data file, and a checklist for code review) to find injected defects in the source code (See Section 5.4.2 for more details).

Reviews were finished when a subject (reviewer) concluded the target document had no more defects. In both reviews, time spent for a review and the number of detected defects were collected for the analysis.

Subjects were 12 graduate students and faculty members of the Nara Institute of Science and Technology. The average of their programming experience was 7.6 years, and 2.4 years for programming with C language.

Two of them had experience with software development in industries.

5.4.2   Review Type

Design Review

Design review in this thesis is defined as follows:

Target document: Detailed design document

Document used with target: SRS, checklist for design review, data file

Purpose of the review: Detect inconsistency with SRS.

Defect types: The following three types are adopted.

> **Non-conforming Design (ND):** This defect type means that the detailed design document contains a function described in the SRS, but this function does not fulfill the requirements.

> **Missing Design (MD):** MD means that the detailed design document has no description about a function described in the SRS.

> **Excessive Design (ED):** This defect type indicates the existence of excess descriptions in the detailed design document, which have not been described in the SRS. This defect can be also considered as an insufficient description of the SRS.

Code Review

Code review in this thesis is defined as follows:

Target document: Source code

Document used with target: SRS, detailed design document, checklist for code review, data file

Purpose of the review: Detect defects from source code without executing the program.

Defect types: The following four types are adopted refer to defect classification [Beizer 90, Chillarege 92]

> **Fault in Data (Data):** This defect type includes an incorrect definition and usage of variables.

> **Fault in Process (Process):** This defect type represents incorrect functional logic (such as incorrect conditional statements).

> **Fault in Screen (Screen):** This is a defect of display output, which causes user confusion and/or mistakes.

> **Excessive Implementation (Excess):** This type of defect indicates excess implementation in the source code, which has not described in the SRS and the detailed design document. Therefore, this defect can be also considered an "insufficient description" of the SRS and the detailed design document.

5.4.3 Materials

Documents used in the experiment were about a rental house search system actually used in an industrial training workshop. The documents consist of the requirements specification, the design document, and the data file.

This system reads a data file in which a set of rental houses is listed. A system user inputs a condition about the rental houses (e.g. distance from the nearest train station, floor space and rent) that he/she wants to look at. According to the user input, the system outputs a list of rental houses that match his or her conditions.

SRS: This document consists of 40 lines of Japanese text, describing system functions and requirements.

Detailed design document: This document describes details of each function interface, data, and processes. It consists of 30 lines of Japanese text.

Source code: This is a 5 function, 120 step C language program. All comments were removed before the experiment.

Data file: This file is read by the system when the system starts. The file consists of a list of rental houses.

Checklist for design review: This is a generic checklist for a design document review, written based on existing literature [Porter 95, Thelin 03].

Checklist for code review: This is a generic checklist for a C source code review built based on existing literature [Porter 95, Thelin 03]. Table 5.1 shows the checklist.

5.5   Results

5.5.1   Collected Data

The average review time was 60.4 minutes (25.3 minutes for design review, and 35.0 minutes for code review.) Twenty-four (12 subjects × 2 type review) eye movement data were collected in the experiment. Two of the subjects were removed from the analysis because of the insufficient data accuracy of their eye movements. As a result, 22 (11 subjects' data for each review type) data were used for analysis.

From the interview of reviewers conducted after the experiment, we confirmed that the motivation of subjects to find defects was kept high during the experiment. All subjects found at least three bugs (the average was 5.45).

Using the reviewers' eye movements allowed us to improve the accuracy of analysis, because eye fixations to the outside of documents (Search pane, out of Document Viewer, etc.) were removed. The number of detected defects of each subject was collected from operation logs.

Table 5.1: Checklist for source code review.

Class	Item
Completeness	Every requirement is implemented correctly.
Initialization	In the following places, variables and parameters are properly initialized. - Beginning of the program. - Beginning of loop blocks. - Before function call.
Function call	Types and order of the parameters are correct.
	Usage of pointer parameters is correct.
	The function returns a correct value.
Operation	Operations such as =, ==, && are used correctly.
	Each sign of inequality has a correct direction.
	Every operator is used with correct priority.
Data	All data in the system have correct types and values.
	Access to data is correct.
	Every input to the system fulfills the following conditions. - All input data are correctly stored. - Input files are opened before access. - Input files are closed after access.
Conditional statement	Statements start/end correctly.
	Statements have correct conditions.

Table 5.2 shows the fixation time for each document and the number of defect detections on the code review, and Table 5.3 shows the fixation time for each document and the number of defect detections on design review. The Table described, for example, how reviewer C gazed at documents for 692 seconds in total, and detected eight defects on the code review. On the other hand, although reviewer F took twice as long to gaze at the documents, he detected only six defects.

Table 5.4 describes metrics on the code review, and Table 5.5 describes metrics on the design review. Metrics GTR and RQ are values normalized by the total time of fixation and the number of detected defects (See Section 5.2). The average of GTR in Table 5.4 shows the source code (the review target document) was almost read entirely, but the data file and checklist were not read at all.

These metrics show, for example, on code review, that reviewer C spent 65.3% of his fixation time on the source code ($GTR(Code)$), spent 11.1% on the SRS, and 21.5% on the detailed design document. The same reviewer also detected all defects except the *Excess* in his review and he detected each defect at 86.5 seconds ($RE(Code, All)$). On the other hand, the metrics of reviewer F depicts his review performance is lower than reviewer C.

Figure 5.3 describes the eye movements of a high-performance reviewer on the code review (Reviewer C), and Figure 5.4 describes the eye movements of a low-performance reviewer on the code review (Reviewer F). This graph describes a time series of eye movements on each block of documents (function or paragraph); the horizontal axis shows fixation ID (transitions of fixation among lines), and the vertical axis shows the line number of documents.

The Figure shows that the high-performance reviewer read high-level documents frequently. Conversely, the low-performance reviewer scarcely read the high-level documents, and concentrated on the target document (source code). These results indicate that our hypothesis, "the review performance increases while high-level documents are read in the review" is correct.

Table 5.2: Fixation time and the number of defect detections on the code review.

Subject	Review time (Sec.)	Fixation time (Sec.)							# defect detection				
		Require-ments	Design	Code	Data	Checklist	Other	Total	Data	Process	Screen	Excess	All
A	716	7.5	55.9	638.3	0.0	11.7	3.0	716	2	2	0	1	5
B	1164	56.9	269.3	826.9	0.0	10.7	0.4	1164	1	3	2	2	8
C	692	76.7	149.2	451.9	0.0	14.4	0.0	692	3	3	2	0	8
D	2257	59.8	248.4	1872.4	6.9	53.2	16.0	2257	1	3	1	1	6
E	658	30.8	179.7	446.3	1.4	0.0	0.0	658	2	3	2	0	7
F	1618	5.3	10.9	1411.9	2.6	22.3	164.7	1618	3	2	1	0	6
G	2248	101.3	253.8	1783.4	1.2	20.5	87.6	2248	2	3	2	1	8
I	1741	129.5	189.6	1346.9	9.0	54.3	11.5	1741	2	3	0	4	9
J	1252	51.6	145.4	1035.1	0.0	0.0	19.5	1252	1	3	2	0	6
K	2042	110.8	136.4	1548.0	3.2	40.3	203.2	2042	2	2	1	3	8
L	1378	31.8	134.0	1209.0	0.3	0.0	2.5	1378	2	2	2	1	7
Average	1433.2	60.2	161.1	1142.7	2.2	20.7	46.2	1433.2	1.9	2.6	1.4	1.2	7.1

Table 5.3: Fixation time and the number of defect detections on the design review.

Subject	Review time (Sec.)	Fixation time (Sec.)						# defect detection			
		Require-ments	Design	Data	Checklist	Other	Total	ND	MD	ED	All
A	574	129.8	420.4	0.3	20.6	2.6	574	2	1	2	5
C	543	165.7	366.3	3.2	7.7	0.0	543	0	2	2	4
D	996	335.0	623.6	11.3	23.9	2.6	996	1	3	2	6
E	503	122.1	368.6	2.0	10.0	0.0	503	1	2	0	3
F	1349	266.4	982.7	16.4	31.5	51.9	1349	2	2	2	6
G	1778	491.8	1200.3	7.1	54.9	24.3	1778	1	2	2	5
H	1051	335.8	689.5	1.4	22.9	1.1	1051	0	3	1	4
I	1151	471.3	631.3	3.6	41.2	3.6	1151	1	3	3	7
J	968	284.7	644.5	11.9	19.9	7.5	968	2	3	3	8
K	1471	404.6	1063.3	0.6	0.0	2.3	1471	2	3	3	8
L	1126	287.4	808.8	13.2	13.1	3.8	1126	1	3	0	4
Average	1046.4	299.5	709.0	6.4	22.3	9.1	1046.4	1.2	2.5	1.8	5.5

Table 5.4: GTR for each document and review performance on code review.

Subject	Gaze time ratio						Review quality					RE (Code, All)
	GTR (Req)	GTR (Design)	GTR (Code)	GTR (Data)	GTR (Checklist)	GTR (Other)	RQ (Code, Data)	RQ (Code, Process)	RQ (Code, Screen)	RQ (Code, Excess)	RQ (Code, All)	
A	0.010	0.078	0.891	0.000	0.016	0.004	0.667	0.667	0.000	0.250	0.417	143.3
B	0.049	0.231	0.710	0.000	0.009	0.000	0.333	1.000	1.000	0.500	0.667	145.5
C	0.111	0.215	0.653	0.000	0.021	0.000	1.000	1.000	1.000	0.000	0.667	86.5
D	0.027	0.110	0.830	0.003	0.024	0.007	0.333	1.000	0.500	0.250	0.500	376.1
E	0.047	0.273	0.678	0.002	0.000	0.000	0.667	1.000	1.000	0.000	0.583	94.0
F	0.003	0.007	0.873	0.002	0.014	0.102	1.000	0.667	0.500	0.000	0.500	269.6
G	0.045	0.113	0.793	0.001	0.009	0.039	0.667	1.000	1.000	0.250	0.667	281.0
I	0.074	0.109	0.774	0.005	0.031	0.007	0.667	1.000	0.000	1.000	0.750	193.5
J	0.041	0.116	0.827	0.000	0.000	0.016	0.333	1.000	1.000	0.000	0.500	208.6
K	0.054	0.067	0.758	0.002	0.020	0.100	0.667	0.667	0.500	0.750	0.667	255.2
L	0.023	0.097	0.878	0.000	0.000	0.002	0.667	0.667	1.000	0.250	0.583	196.8
Average	0.044	0.129	0.788	0.001	0.013	0.025	0.636	0.879	0.682	0.295	0.591	204.6

Table 5.5: GTR for each document and review performance on design review.

Subject	Gaze time ratio					Review quality				RE(Design, All)
	GTR (Req)	GTR (Design)	GTR (Data)	GTR (Checklist)	GTR (Other)	RQ (Design, ND)	RQ (Design, MD)	RQ (Design, ED)	RQ (Design, All)	
A	0.226	0.733	0.001	0.036	0.005	0.667	0.333	0.667	0.556	114.7
C	0.305	0.675	0.006	0.014	0.000	0.000	0.667	0.667	0.444	135.7
D	0.336	0.626	0.011	0.024	0.003	0.333	1.000	0.667	0.667	166.1
E	0.243	0.733	0.004	0.020	0.000	0.333	0.667	0.000	0.333	167.6
F	0.198	0.729	0.012	0.023	0.038	0.667	0.667	0.667	0.667	224.8
G	0.277	0.675	0.004	0.031	0.014	0.333	0.667	0.667	0.556	355.7
H	0.320	0.656	0.001	0.022	0.001	0.000	1.000	0.333	0.444	262.7
I	0.409	0.548	0.003	0.036	0.003	0.333	1.000	1.000	0.778	164.4
J	0.294	0.665	0.012	0.021	0.008	0.667	1.000	1.000	0.889	121.1
K	0.275	0.723	0.000	0.000	0.002	0.667	1.000	1.000	0.889	183.9
L	0.255	0.718	0.012	0.012	0.003	0.333	1.000	0.000	0.444	281.6
Average	0.285	0.680	0.006	0.022	0.007	0.394	0.818	0.606	0.606	198.0
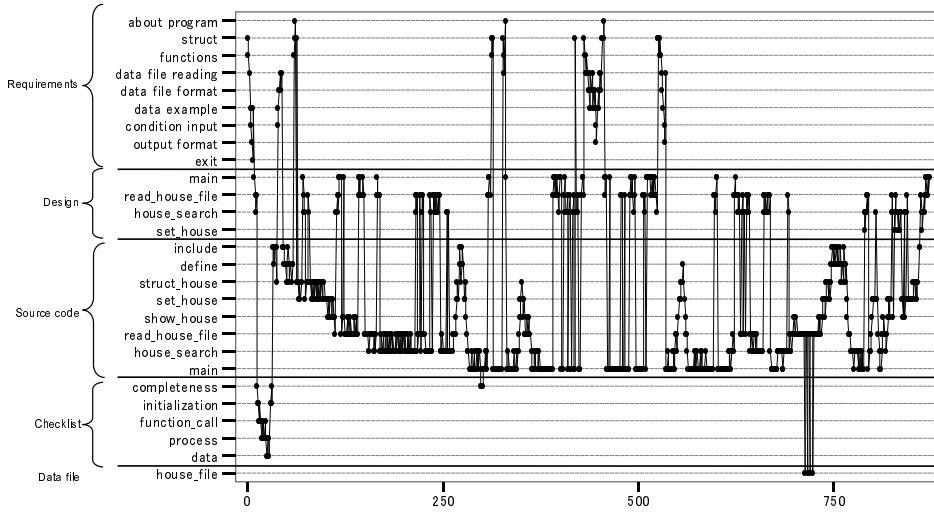
Figure 5.3: Eye movements of a high-performance reviewer on code review (Reviewer C).
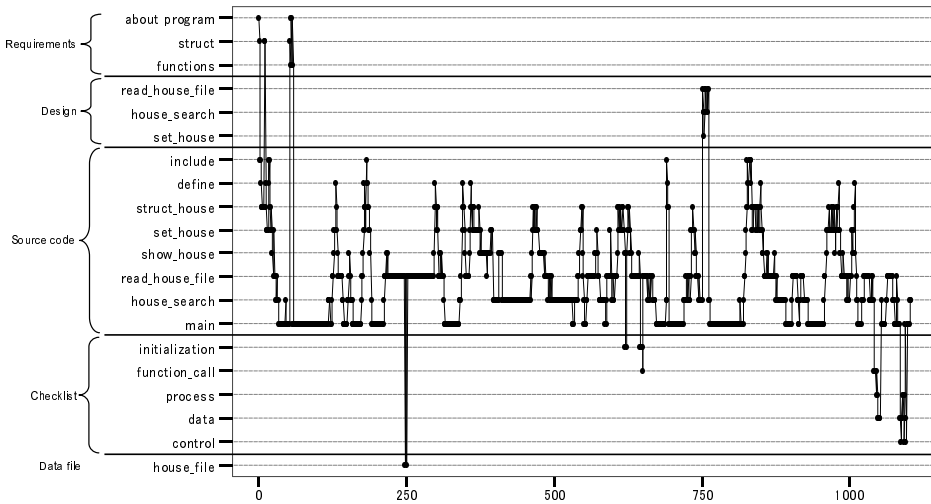
Figure 5.4: Eye movements of a low-performance reviewer on code review (Reviewer F).

Table 5.6: Correlation between fixation ratio and review performance on design review.

		Quality				Effectiveness
		RQ(Design, All)	RQ(Design, ND)	RQ(Design, ED)	RQ(Design, MD)	RE(Design, All)
GTR(Req)	Pearsons' r	0.272	−0.465	0.372	**0.593**	−0.123
	p−value	0.419	0.150	0.260	**0.054**	0.719

5.5.2 Design Review Performance

Table 5.6 describes the correlation between GTR and review performance on the design review. There is no correlation between $GTR(Req)$ and $RQ(Design, All)$ ($r = 0.272$, $p = 0.419$). The Table shows a positive correlation, however, between $GTR(Req)$ and $RQ(Design, MD)$ ($r = 0.593$, $p = 0.054$).

The result describes a reviewer, who, spending a longer time read the SRS, detected more omissions of the requirements. Hence, the hypothesis DQ is supported. Thus, reading the SRS yields more understanding of system requirements than reading only a detailed design document.

There is no correlation between $GTR(Req)$ and $RE(Design, All)$ ($r = -0.123$, $p = 0.719$). Hence, the hypothesis DE is not supported. The result might be interpreted such that the target software is simple; therefore, reviewers easily understood system requirements and structures. That is, differences of fixation time to each document did not make a difference in understanding efficiency.

5.5.3 Code Review Performance

Table 5.7 describes the correlation (Pearsons' r and its p-value) between GTR and review performance on code review. The Table shows a posi-

Table 5.7: Correlation between fixation ratio and review performance on source code review.

		Quality					Effectiveness
		RQ(Code, All)	RQ(Code, Data)	RQ(Code, Process)	RQ(Code, Screen)	RQ(Code, Excess)	RE(Code, All)
GTR(Req)	Pearsons' r	**0.739**	0.197	**0.561**	0.225	0.225	−0.445
	p−value	**0.009**	0.561	**0.072**	0.506	0.505	0.170
GTR(design)	Pearsons' r	0.339	−0.193	**0.669**	**0.539**	−0.169	**−0.644**
	p−value	0.308	0.570	**0.024**	**0.087**	0.619	**0.033**

tive correlation between $GTR(Req)$ and $RQ(Code, All)$ ($r = 0.739$, $p = 0.009$). Also, the Table shows a positive correlation between $GTR(Req)$ and $RQ(Code, Process)$ ($r = 0.561$, $p = 0.072$), and a positive correlation between $GTR(Design)$ and $RQ(Code, Process)$ ($r = 0.669$, $p = 0.024$). There is a positive correlation between $GTR(Design)$ and $RQ(Code, Screen)$ ($r = 0.539$, $p = 0.087$).

From the above results, the two hypotheses SQ_{Req} and SQ_{Design} were supported. The result suggests that spending more time on the high-level document promotes more understanding of the software requirements and improves defect detection performance in the source code review. For "Process" defects especially, reviewers found the defects effectively by reading the detailed design document, which describes the details of the program's functions.

Table 5.7 also shows a negative correlation between $GTR(Design)$ and $RE(Code, All)$ ($r = -0.644$, $p = 0.033$). The result suggests that a reviewer who spent a longer time reading the detailed design document, detected defects within a shorter time period.

From the above result, hypothesis SE_{Design} is supported. Obviously the design document had meaningful information about system structures and functions that helped the reviewer to understand the source code. Moreover, compared to the requirement specifications, the design specification was

more affinitive to implementation, hence, reading the design specification allows more effective defect detection in the source code. Hypothesis SE_{Req} is not supported from the result.

5.5.4   Detailed Analyses

This Section shows more detailed analyses of eye movements in the review. The analyses surpass the verification of the hypotheses described in Section 5.3. However, these analyses will indicate a fruitful way of understanding review activity.

Fixation to Review Target Document

We analyzed the correlation between review performance and fixation time for the review target document.

Table 5.8 shows the correlation between $GTR(Code)$ and review performance on the design review. Table shows a negative correlation between $GTR(Code)$ and $RQ(Code, All)$ ($r = -0.639$, $p = 0.034$) and a negative correlation between $GTR(Code)$ and $RQ(Code, Process)$ ($r = -0.604$, $p = 0.049$). In addition, there is a positive correlation between $GTR(Code)$ and $RE(Code, All)$ ($r = 0.529$, $p = 0.094$).

This result suggests that a concentration of review time to source code yields less understanding of the system requirements and design. The result strengthens the correctness of the hypotheses on code review.

Table 5.9 shows a correlation between $GTR(Design)$ and review performance on design review. The table shows there is no correlation on design review.

Fixation to Different High-level Documents

In the code review of this experiment, two high-level documents (SRS, and a detailed design document) were used. We suppose concentration of reading time for different high-level document yields different review performances. For quantification of which documents were read intensively, the following

Table 5.8: Correlation between fixation ratio to target document and review performance on code review.

		Quality					Effectiveness
		RQ(Code, All)	RQ(Code, Data)	RQ(Code, Process)	RQ(Code, Screen)	RQ(Code, Excess)	RE(Code, All)
GTR(Code)	Pearsons' r	**−0.639**	−0.094	**−0.604**	−0.418	−0.054	0.529
	p−value	**0.034**	0.784	**0.049**	0.201	0.874	0.094

Table 5.9: Correlation between fixation ratio to target document and review performance on design review.

		Quality				Effectiveness
		RQ (Design, All)	RQ (Design, ND)	RQ (Design, ED)	RQ (Design, MD)	RE (Design, All)
GTR(Design)	Pearsons' r	−0.311	0.371	−0.427	−0.502	0.073
	p−value	0.352	0.261	0.190	0.115	0.832

metrics are defined.

$$Weight_{Req} = \frac{gtime(Req)}{gtime(Req) + gtime(Design)}$$

$Weight_{Req}$ describes the ratio of fixation time on SRS to fixation time on high-level documents.

Figure 5.5 shows the relationship between $Weight_{Req}$ and $RQ(Code, All)$, Figure 5.6 shows the relationship between $Weight_{Req}$ and $RE(Code, All)$. The maximum value of $Weight_{Req}$ in the experiment is 0.448; hence, the reviewer plotted the right side of the figure to read two high-level documents equally.

Statistical analysis shows $Weight_{Req}$ and $RQ(Code, All)$ had a significant positive correlation ($r = 0.587$, $p = 0.057$). That is, spending a longer
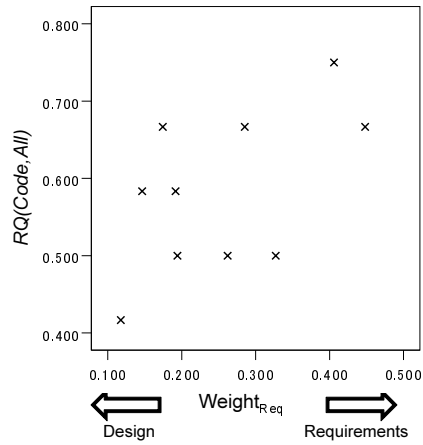
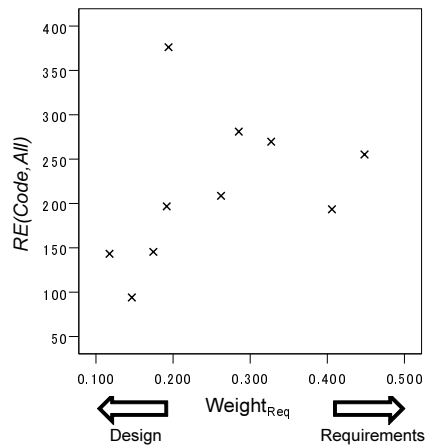Figure 5.5: Relationship between weight to reading requirement specification and review quality.

Figure 5.6: Relationship between weight to reading requirement specification and review effectiveness.

time to read the SRS than the design document detects more defects from the source code.

Also, Figure 5.6 suggests a positive correlation between $Weight_{Req}$ and $RE(Code, All)$. The figure indicates that spending a longer time on the design document than the SRS increases the defect detection efficiency. However, there is no significant correlation between them.
This result indicates that there was a different effect on review performance between spending a longer time to read the SRS and the detailed design document. Since the defect detection ratio and the detection time per defect are both important, developers need to read both the requirements specification and the design document in the source code review.

5.6 Chapter Summary

This chapter reported a second experiment performed to evaluate the correlation between eye movements between documents and the review performance. Using the improved evaluation environment, design review and code review with multiple documents were measured, and six hypotheses were verified.

Table 5.10 summarizes the results of verification. The results of the experiment showed that spending a longer time to read high-level documents increases review performance and spending a longer time to read the target document decreases the performance. Also, the result suggests that concentration to the SRS increases the defect detection ratio, and concentration to the design document increases the defect detection efficiency. This is good evidence to encourage developers to read high-level documents when reviewing low-level documents.

Table 5.10: Results of testing hypotheses.

Hypothesis	Result
DQ	Supported (MD)
DE	Not supporter
SQ_{Req}	Supported (All, Process)
SQ_{Design}	Supported (Process, Screen)
SE_{Req}	Not supported
SE_{Design}	Supported

Chapter 6

Conclusion

This chapter presents the final conclusion of this thesis research. It begins with a summary of the research in Section 6.1, and then lists the main contributions in Section 6.2. Finally, it discusses future directions are discussed in Section 6.3.

6.1 Research Summary

This thesis research improves software developers' performance from the viewpoint of defect detection. Many of the development techniques to improve software quality, such as software testing and review exist. However, in software development, the impact of individual differences is a more dominant factor than the techniques themselves. Analyzing the factor of individual differences from an empirical evaluation is important to improving software development effectiveness.

This thesis clarifies the factor of individual differences on software review. Software review is one of the most adopted quality improvement techniques in the organizations. Several studies showed the review is a more efficient/effective technique than software testing. The software review is a human-centered activity; hence, the impact of individual differences on the review is quite dominant.

To characterize the developers' performance in the review quantitatively, we used the eye movements of the reviewer. The reading strategy is indicated by the eye movements of the reviewers. Thus, eye movements can be used as a powerful metric to characterize performance in the software review.

In this thesis, a gaze-based review evaluation has been designed and implemented in an environment called DRESREM, for the eye-gaze-based evaluation of software review. The system allowed us to evaluate how the reviewers read the software document quantitatively. The system also provided features to play back the eye movements and the operations for a qualitative analysis of review activities.

Using this environment, two experiments were used to analyze the reviewers' eye movements from different perspectives: eye movements between lines, and eye movements between documents.

In the first experiment, we analyzed eye movements between lines in a source code. The result shows that reviewers who took sufficient time for a preliminary reading of the entire code tended to detect defects efficiently.

In the second experiment, eye movements between documents were used in the design and code review. The results of the experiments showed that spending a longer time to read high-level documents increases review performance, and spending a longer time to review a target document decreases performance. Also, the results suggest that concentration to the SRS increases the defect detection ratio, and concentration to the design document increases defect detection efficiency. These results are good evidence to encourage developers to read high-level documents when reviewing low-level documents.

6.2 Contributions

There are three main contributions from this research:

- **The Idea of Eye Movement Measuring in Software Review**
  To analyze the reviewers' activity, their eye movements for software documents were recorded. Eye movements in the software review directly reflect the reviewers' reading procedure. Hence, reviewers' intentions on the review were easily understandable from the data. This is a great advantage of eye movement compared to interview or think-aloud protocols. Employment of eye movements also enabled an objective, quantitative analysis of review procedure.

- **The System Implementation**
  A gaze-based review evaluation environment is the result of this thesis research. The environment automatically captures reviewers' eye movements, and derives the sequence of logical line numbers of the document on which the reviewer has focused. This enables a user in the evaluation environment to conduct a line-wise analysis of reading procedures. Analysis of the line-wise eye movement data is much easier than coordinate-wise eye movement data. The system also supported an analysis to visualize the eye movements. Visualized eye movements and the operation of the reviewer helps the user to understand the reviewers' activity.

- **Insight from the Empirical Studies**
  Two empirical experiments revealed the correlation between reading procedure and the review performance of the review. The reviewers spent a longer time "scanning" the source code, and spent a longer time reading the SRS and/or design document. This result is suggested from studies of program comprehension. The experiments show empirical evidence for the researchers, and this is one of the achievements of this thesis.

  Also, the result suggested that spending a longer time reading the SRS increases the defect detection ratio, and conversely, spending a longer time reading the design document increases the defect detection efficiency. Since the defect detection ratio and the detection time

per defect are both important, developers need to read both the requirements' specification and the design document in a source code review.

These results show a more concrete reading technique that can be used to increase review performance as follows:

1. Scan the all elements in a high-level document such as an SRS to comprehend roughly the structures of the system.

2. Select one of the elements in the high-level document and read intensively for a detailed understanding.

3. Scan a low-level document such as a design document that describes the elements selected above.

4. Read the low-level document intensively for a detailed understanding.

This reading technique defines detailed action in the review that was not mentioned in the existing techniques. The techniques may improve the effectiveness of a low-performance reviewer. Moreover, the reading technique is applicable with existing review techniques such as CBR and PBR. Hence, developers can implement the technique without enormous process modification.

The results of the two interviews indicated that subjects perceived their unconscious activities by watching their own eye movements. Subjects perceived their 'scan' pattern from eye movements recorded from the first experiment. The results suggest that measuring the reviewers' eye movements enables experimenter to externalize implicit knowledge, i.e., the knowledge of how to go about doing something.

6.3   Future Directions

In this thesis, we have conducted experiments of the source code design review. However, we believe that DRESREM (and DRESREM 2) is applicable to other kinds of practical software documents as well.

As seen in the Section 4.2.4, the primary feature of the environment is the line-wise gaze tracking, which is especially suitable for (a) structured documents, (b) documents formed by multiple statements, (c) documents written in a 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. In object-oriented programming, diagram-centered documents such as class diagrams, sequence diagrams, and state chart diagrams are created. Line-wise analysis of eye movements is inadequate for such documents. However, for these documents, the eye movements should be tracked and evaluated on 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 Section 4.2.4). We can cope with this by customizing DRESREM 2, or by using other existing systems. Thus, we believe that this is not a critical problem.

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

This thesis focused on the individual differences in software review. Likewise, an evaluation of individual differences in debugging, testing, and implementation is also an interesting research question. Tracking eye gaze over the IDE may also be a helpful mechanism to measure the developers' behaviors regarding debugging, testing, and implementation.

In this thesis, we extract eye movement patterns manually. Recent study about eye movement measurement proposed a support system named eye-Patterns to extract the patterns from subjects' eye movements [West 06].

Such a support system is suitable for extracting patterns from reviewers' eye movements efficiently. Also, some modeling methods such as Hidden Markov Model (HMM) and Dynamic Time Warping (DTW) are useful for pattern matching.

These are all very interesting research themes, but we are unable to perform them at this stage of the research. However, we believe empirical studies of individual differences in software development will produce extremely useful insights for excel training methods and support environments.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my supervisor, Professor Ken-ichi Matsumoto, for his continuous support and encouragement during this work.

I am also very grateful to Professor Hirokazu Nishitani, for his valuable comments and helpful criticism of this work as a member of my thesis review.

I am also deeply grateful to Associate Professor Akito Monden. He gave me the opportunity to study in the field of human factors in software development. For five years, I have received invaluable assistance from him. I will always remember his encouragement and enthusiasm.

I also want to thank Associate Professor Masahide Nakamura, for his patient advice and guidance. I have learned a lot from his knowledge and positive attitude toward research. I could not have finished this dissertation without his encouragement.

I also wish to thank Professor Hajimu Iida. I have received helpful advice and warm support from him for five years.

I also wish to thank Assistant Professor Masao Ohira. His support and advice were very helpful in the completion of this dissertation.

I also wish to thank Assistant Professor Shuji Morisaki. His comments and advice encouraged me through this research.

I also want to thank Assistant Professor Noboru Nakamichi. He provided me with knowledge about how to evaluate eye movement using an eye-tracking device.

I have been fortunate to have received assistance from many colleagues. I wish to thank all the members of the Software Engineering Lab., Graduate School of Information Science, Nara Institute of Science and Technology. I can only mention a few of my helpful colleagues here because the list is long. I wish to extend thanks to Masateru Tshunoda, Tomoko Matsumura, Takeshi Kakimoto, Hiroki Yamauchi, Koji Toda, Yasutaka Kamei, Shinsuke Matsumoto, and Junko Inui.

Finally, I would like to express my warmest gratitude to my parents, my grandparents, and my friends for their constant encouragement and generous help.

References

[Arthur 98]   Arthur, J. D., Groener, M. K., Hayhurst, K. J., and Holloway, C. M.: Adding Value to the Software Development Process: A Study in Independent Verification and Validation, Technical report, Technical Report 98-15, Virginia Tech (1998).

[Basili 96]   Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S., and Zelkowitz, M. V.: The Empirical Investigation of Perspective-Based Reading, *An International Journal of Empirical Software Engineering*, Vol. 1, No. 2, pp. 133–163 (1996).

[Bednarik 05]   Bednarik, R., Myller, N., Sutinen, E., and Tukiainen, M.: Applying Eye-Movement Tracking to Program Visualization, in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 302–304, Washington, DC, USA (2005), IEEE Computer Society.

[Beizer 90]   Beizer, B.: *Software testing techniques (2nd ed.)*, Van Nostrand Reinhold Co., New York, NY, USA (1990).

[Boehm 75]   Boehm, B. W.: The High Cost of Software, Practical Strategies for Developing Large Software Systems (1975).

[Boehm 81]   Boehm, B. W.: *Software Engineering Economics*, Prentice Hall (1981).

[Bojko 05]   Bojko, A. and Stephenson, A.: Supplementing Conventional Usability Measures with Eye Movement Data in Evaluating Visual Search Performance, in *Proceedings of the 11th International Conference on Human-Computer Interaction* (2005).

[Bucher 75]   Bucher, D. W.: Maintenance of the computer sciences teleprocessing system, *SIGPLAN Not.*, Vol. 10, No. 6, pp. 260–266 (1975).

[Chillarege 92] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., and Wong, M.: Orthogonal Defect Classification-A Concept for In-Process Measurements, *IEEE Trans. Softw. Eng.*, Vol. 18, No. 11, pp. 943–956 (1992).

[Ciolkowski 97] Ciolkowski, M., Differding, C., Laitenberger, O., and Münch, J.: Empirical Investigation of Perspective-based Reading:A Replicated Experiment, Technical Report 97-13, ISERN Technical Report (1997).

[Ciolkowski 02] Ciolkowski, M., Laitenberger, O., Rombach, D., Shull, F., and Perry, D.: Software Inspection, Reviews and Walkthroughs, in *Proceedings of the 24th International Conference on Software Engineering*, pp. 641–642 (2002).

[Cleve 05]   Cleve, H. and Zeller, A.: Locating causes of program failures, in *Proceedings of the 27th international conference on Software engineering*, pp. 342–351 (2005).

[Crosby 90]   Crosby, M. E. and Stelobsky, J.: How Do We Read Algorithms? A Case Study, *IEEE Computer*, Vol. 23, No. 1, pp. 24–35 (1990).

[Davis 95]   Davis, A. M.: *201 Principles of Software Development*, McGraw-Hill (1995).

[Demarco 99]   Demarco, T. and Lister, T.: *Peopleware: Productive Projects and Teams, 2nd Edition*, Dorset House (1999).

[Ericsson 84]   Ericsson, K. A. and Simon, H. A.: *Protocol analysis: Verbal reports as data*, MIT Press, Cambridge, MA, USA (1984).

[Fagan 76]   Fagan, M. E.: Design and Code Inspection to Reduce Errors in Program Development, *IBM Systems Journal*, Vol. 15, No. 3, pp. 182–211 (1976).

[Fusaro 97]   Fusaro, P., Lanubile, F., and Visaggio, G.: A Replicated Experiment to Assess Requirements Inspection Techniques, *An International Journal of Empirical Software Engineering*, Vol. 2, No. 1, pp. 39–57 (1997).

[Galli 04]   Galli, M., Lanza, M., Nierstrasz, O., and Wuyts, R.: Ordering Broken Unit Tests for Focused Debugging, in *Proceedings of the 20th International Conference on Software Maintenance*, pp. 114–123, Washington, DC, USA (2004), IEEE Computer Society.

[Halling 01]   Halling, M., Biffl, S., Grechenig, T., and Köhle, M.: Using Reading Techniques to Focus Inspection Performance, in *Proceedings of the 27th Euromicro Workshop Software Process and Product Improvement*, pp. 248–257 (2001).

[Hazlett 03]   Hazlett, R.: Measurement of user frustration: a biologic approach, in *CHI '03 extended abstracts on Human factors in computing systems*, pp. 734–735, New York, NY, USA (2003), ACM Press.

[Hirayama 07]   Hirayama, M.: http://www.ipa.go.jp/ (2007).

[IEEE 1028-1997]   IEEE/ANSI: *1028-1997, Standard for Software Reviews* (1997).

[Jones 02]   Jones, J. A., Harrold, M. J., and Stasko, J.: Visualization of test information to assist fault localization, in *Proceedings of the 24th International Conference on Software Engineering*, pp. 467–477, New York, NY, USA (2002), ACM Press.

[Kasarskis 01]   Kasarskis, P., Stehwien, J., Hichox, J., Aretz, A., and Wickens, C.: Comparison of Expert and Novice Scan Behaviors during VFR Flight, in *Proceedings of the 11th International Symposium on Aviation Psychology* (2001).

[Laitenberger 98a] Laitenberger, O.: Studying the Effects of Code Inspection and Structural Testing on Software Quality, in *Proceedings of the The Ninth International Symposium on Software Reliability Engineering*, p. 237, Washington, DC, USA (1998), IEEE Computer Society.

[Laitenberger 98b] Laitenberger, O.: Studying the Effects of Code Inspection and Structural Testing on Software Quality, in *Proceedings of the The 9th International Symposium on Software Reliability Engineering*, p. 237, Washington, DC, USA (1998), IEEE Computer Society.

[Laitenberger 00] Laitenberger, O. and DeBaud, J.: An encompassing life cycle centric survey of software inspection, *Journal of Systems and Software*, Vol. 50, No. 1, pp. 5–31 (2000).

[Laitenberger 02a] Laitenberger, O., Beil, T., and Schwinn, T.: An Industrial Case Study to Examine a Non-Traditional Inspection Implementation for Requirements Specifications, *Empirical Softw. Engg.*, Vol. 7, No. 4, pp. 345–374 (2002).

[Laitenberger 02b] Laitenberger, O., Beil, T., and Schwinn, T.: An Industrial Case Study to examine a non-traditional Inspection Implementation for Requirements Specifications, in *Proceedings of the The 8th IEEE International Symposium on Software Metrics*, pp. 97–106 (2002).

[Lanubile 00]   Lanubile, F. and Visaggio, G.: Evaluating Defect Detection Techniques for Software Requirements Inspections, Technical Report 08, ISERN Technical Report (2000).

[Law 04]   Law, B., Atkins, M. S., Kirkpatrick, A. E., Lomax, A. J., and Mackenzie, C. L.: Eye Gaze Patterns Differentiate Novice and Expert in a Virtual Laparoscopic Surgery Training Environment, in *Proceedings of the 2004 Symposium on Eye Tracking Research and Applications*, pp. 41–48 (2004).

[Leveson 95]   Leveson, N.: *Safeware : System Safety and Computers*, Addison-Wesley Professional (1995).

[Lewis 92]   Lewis, R. O.: *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*, Wiley (1992).

[McBreen 01]   McBreen, P.: *Software Craftsmanship: The New Imperative*, Addison-Wesley (2001).

[Melo 01]   Melo, W., Shull, F., and Travassos, G. H.: Software Review Guideline, Technical report, COPPE/UFRJ Systems Engineering and Computer Science Program Technical Report ES556 /01 (2001).

[Miller 98]   Miller, J., Wood, M., Roper, M., and Brooks, A.: Further Experiences with Scenarios and Checklists, *An International Journal of Empirical Software Engineering*, Vol. 3, No. 3, pp. 37–64 (1998).

[Murata 91]   Murata, A.: Measurement of Mental Workload during Location Task, *IEICE Transactions on Fundamentals*, Vol. 74, No. 4, pp. 706–714 (1991).

[Myers 78]   Myers, G. J.: A controlled experiment in program testing and code walkthroughs/inspections, *Commun. ACM*, Vol. 21, No. 9, pp. 760–768 (1978).

[Nakamichi 03]   Nakamichi, N., Sakai, M., Hu, J., Shima, K., Naka-
                      mura, M., and Matsumoto, K.: WebTracer: Evaluating
                      Web usability with browsing history and eye movement,
                      in *Proceedings of the 10th International Conference on
                      Human-Computer Interaction*, pp. 813–817 (2003).

[Nakayama 02]   Nakayama, M., Takahashi, K., and Shimizu, Y.: The
                      act of task difficulty and eye-movement frequency for the
                      'Oculo-motor indices', in *Proceedings of the 2002 sympo-
                      sium on Eye tracking research & applications*, pp. 37–42,
                      New York, NY, USA (2002), ACM Press.

[P-CMM]   People   Capability   Maturity   Model,
                      http://www.sei.cmu.edu/.

[Porter 94]   Porter, A. A. and Votta, L. G.: An Experiment to As-
                      sess Different Defect Detection Methods for Software Re-
                      quirements Inspections, in *Proceedings of the 16th inter-
                      national conference on Software engineering*, pp. 103–
                      112 (1994).

[Porter 95]   Porter, A. A., Votta, L. G., and Basili, V. R.: Compar-
                      ing Detection Methods for Software Requirements In-
                      spection - A Replicated Experiment, *IEEE Transaction
                      on Software Engineering*, Vol. 21, No. 6, pp. 563–575
                      (1995).

[Porter 98]   Porter, A. and Votta, L.: Comparing Detection Methods
                      for Software Requirements Inspection: A Replication
                      Using Professional Subjects, *An International Journal
                      of Empirical Software Engineering*, Vol. 3, No. 4, pp.
                      355–380 (1998).

[Robert 95]   Robert, J. K. J.: *Eye tracking in advanced interface de-
                      sign*, pp. 258–288, Oxford University Press (1995).

[Sabaliauskaite 02]   Sabaliauskaite, G., Matsukawa, F., Kusumoto, S., and
                      Inoue, K.: An Experimental Comparison of Checklist-
                      Based Reading and Perspective-Based Reading for UML

| | Design Document Inspection, in *Proceedings of the 2002 International Symposium on Empirical Software Engineering*, p. 148, Washington, DC, USA (2002), IEEE Computer Society. |

[Sackman 68] Sackman, H., Erikson, W. J., and Grant, E. E.: Exploratory experimental studies comparing online and offline programming performance, *Commun. ACM*, Vol. 11, No. 1, pp. 3–11 (1968).

[Sandahl 98] Sandahl, K., Blomkvist, O., Karlsonn, J., Krysander, C., Lindvall, M., and Ohlsson, N.: An Extended Replication of an Experiment for Assessing Methods for Software Requirements Inspections, *An International Journal of Empirical Software Engineering*, Vol. 3, No. 4, pp. 327–354 (1998).

[Shull 98] Shull, F. J.: *Developing Techniques for Using Software Documents: A Series of Empirical Studies*, PhD thesis, Univ. of Maryland (1998).

[Shull 00] Shull, F., Rus, I., and Basili, V.: How Perspective-Based Reading Can Improve Requirements Inspections, *IEEE Computer*, Vol. 33, No. 7, pp. 73–79 (2000).

[Stein 04] Stein, R. and Brennan, S. E.: Another Person's Eye Gaze as a Cue in Solving Programming Problems, in *Proceedings of the 6th International Conference on Multimodal Interface*, pp. 9–15, ACM Press (2004).

[Thelin 01] Thelin, T., Runeson, P., and Regnell, B.: Usage-Based reading :An Experiment to Guide Reviewers with Use Cases, *Information and Software Technology*, Vol. 43, No. 15, pp. 925–938 (2001).

[Thelin 03] Thelin, T., Runeson, P., and Wohlin, C.: An Experimental Comparison of Usage-Based and Checklist-Based Reading, *IEEE Transaction on Software Engineering*, Vol. 29, No. 8, pp. 687–704 (2003).

[Torii 99]   Torii, K., Matsumoto, K., Nakakoji, K., Takada, Y.,
                  Takada, S., and Shima, K.: Ginger2: An Environment
                  for Computer-Aided Empirical Software Engineering,
                  IEEE Transactions on Software Engineering, Vol. 25,
                  No. 4, pp. 474–492 (1999).

[West 06]   West, J. M., Haake, A. R., Rozanski, E. P., and
                  Karn, K. S.: eyePatterns: software for identifying pat-
                  terns and similarities across fixation sequences, in *Pro-
                  ceedings of the 2006 symposium on Eye tracking re-
                  search & applications*, pp. 149–154, New York, NY, USA
                  (2006), ACM Press.

[Wiegers 02]   Wiegers, K.: *Peer Reviews in Software - A Practical
                  Guide*, Addison-Wesley (2002).

[Zhai 99]   Zhai, S., Morimoto, C., and Ihde, S.: Manual and Gaze
                  Input Cascaded (MAGIC) Pointing, in *Proceedings of
                  the SIGCHI conference on Human factors in computing
                  systems*, pp. 246–253 (1999).

List of Major Publications

Journal Papers

1. Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, "Exploiting Eye Movements for Evaluating Reviewers' Performance in Software Review," IEICE Transactions on Fundamentals, Vol.E90-A, No.10, pp.317-328, October 2007. (Relate with Chapter 2, 3, and 4)

2. Hidetake Uwano, Kyoko Ishida, Yuko Matsuda, Shota Fukushima, Noboru Nakamichi, Masao Ohira, Ken-ichi Matsumoto, and Yasunori Okada, "Evaluation of Software Usability Using Electroencephalogram – Comparison of Frequency Component between Different Software Versions," Human Interface Society, Vol.10, No.2, pp.233-242, May 2008. (Relate with Chapter 1) (In Japanese)

International Conference Papers

1. Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, "Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement," In the Eye Tracking Research & Applications Symposium (ETRA 2006), pp.133-140, March 2006. (Relate with Chapter 2, 3, and 4)

2. Hidetake Uwano, Akito Monden, and Ken-ichi Matsumoto, "Are Good Code Reviewers Also Good at Design Review?," In the 2nd International Symposium on Empirical Software Engineering and Measure-

ment (ESEM 2008), pp.351-353, October 2008. (Relate with Chapter 2, 3, and 5)

3. Hidetake Uwano, Akito Monden, and Ken-ichi Matsumoto, "Dresrem 2: An Analysis System for Multi-Document Software Review Using Reviewers' Eye Movements," In The Third International Conference on Software Engineering Advances (ICSEA 2008), pp. 177-183, October 2008. (Relate with Chapter 2, 3, and 5)

Domestic Conference Papers with Review

1. Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, "Analyzing Performance of Source Code Review Using Reviewers' Eye Movement", Foundation of Software Engineering Workshop (FOSE 2006), Japan Society for Software Science and Technology, pp.103-112, November 2006. (in Japanese)

2. Hidetake Uwano, Noboru Nakamichi, Hiroshi Igaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, "Analysis of Review Process using Programmers' Eye Movement", Technical Report of IEICE, Vol.105, No.128, pp.21-26, June 2005. (in Japanese)

3. Hidetake Uwano, Hiroshi Igaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, "Analysis of Defect detection Process using Programmers' Eye Movement", In the Second Forum on Reliable Computer Softwar (FORCE 2005), June 2005. (in Japanese)

Appendix

A   Source Code Used on First Experiment

A.1   IsPrime

```c
void main(void){
    int i, num, isPrime = 0;

    printf("Input Number:");
    scanf("%d", &num);

    i = 2;
    while(i < num){
        if(num%i == 0)
            isPrime = 1;
        i = i + 1;
    }

    if(isPrime == 1)
        printf("%d is prime number.\n", num);
    else
        printf("%d is NOT prime number.\n", num);
}
```

A.2   Accumulate

```
int makeSum(int max){
    int i, sum;
    sum = 0;

    i = 0;
    while(i < max){
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}


void main(void)
{
    int input, sum;

    scanf("%d",&input);
    sum = makeSum(input);
    printf("Sum from 1 to %d is %d.\n", input, sum);
}
```

A.3   Sum-5

```
void main(void){
    int i, input, sum;

    i = 0;
    while(i < 5){
        scanf("%d", &input);
        sum = sum + input;
        i = i + 1;
    }

    printf("Sum:%d\n", sum);
}
```

A.4   Average-5

```
void main(void){
    int i, input, sum;
    double ave;

    sum = 0;

    i = 0;
    while(i < 5){
        scanf("%d", &input);
        sum = sum + input;
        i = i + 1;
    }

    ave = sum / i;
    printf("Average:%f\n", ave);
}
```

A.5   Average-any

```
#define MAX 255
void main(void){
    int i, num, list[MAX];
    double sum=0;

    i = 0;
    while(i < MAX){
        scanf("%d", &list[i]);
        if(list[i] == 0)break;
        i = i + 1;
    }

    num = i;

    i = 0;
    while(i < MAX){
        sum = sum + list[i];
        i = i + 1;
    }

    printf("Average of %d Num is %f.\n", i, sum/i);
}
```

A.6   Swap

```
void swap(int a, int b){
    int tmp;
    tmp = a; a = b; b = tmp;
}


void main(void){
    int list[2], i;

    i = 0;
    while(i < 2){
        printf("Data #%d:", i+1);
        scanf("%d", &list[i]);
        i = i + 1;
    }

    swap(list[0], list[1]);

    i = 0;
    while(i < 2){
        printf("Data #%d:%d\n", i+1, list[i]);
        i = i + 1;
    }
}
```

B   Documents Used on Second Experiment

B.1   Software Requirements Specifications

User is requesting a rental house search system using the
following structure:

```
struct house {
  char   name[20];  /* Name of house */
  int    area;      /* Floor space */
  double distance;  /* Distance from nearest station (km) */
  int    rent;      /* Rent per month (JPY)*/
};
```

Requirement specification of the system is as follow:

- Read data file to get a list of houses.
  File name is given as command-line argument:
  e.g. $./a.exe house.dat

- Each line of data file consist from name, floor space,
  distance from nearest station, and rent per month
  as tab separated values.
  The maximum number of the data is 100.
  More than the maximum number of data is ignored.
  The system is exit when the data file is empty.

  Example of data file:
    LionHouse   40 5.7 150000
    PensionTamada 10 2.5 70000

- At first, user selects a search condition from area,
  distance, and rent.
  Then input a range of the search. Only a integer is accepted.

- The system output a list of rental house which satisfy the
  condition. Output consists from name, area, distance, and
  price.

- The system exit when the user selects exit from the condition.

- The system does not consider file format error and file read
  error.

B.2   Detailed Design Document used on Design Review

```
int main(int argc, char *argv[]);
```
At first, this function gets a name of data file from command-
line argument.
Then read the data file using read_housefile function, and
store the list of house to filename which used by every other
functions. When the file contains no data, output the message,
then exit. At next, give the user input (search condition and
range) to house_search.
To add the new house data (newhlist), function write_housefile
is used.
The system exit when the user select exit.

```
int read_housefile(struct house hlist[], char filename[]);
```
This function reads the data file (filename) and stores a list
of house to house list (hlist).
Each house data is stored to the structure using set_house
function. Return of the function is the number of house read
from the file.

```
int write_housefile(struct house hlist[], char filename[]);
```
This function adds new house data (newhlist) to the end of the
data file.
Return is the number of the house added successfully.

```
void house_search(struct house hlist[], int dataNum,
                  int selectNum, double lower, double upper);
```
This function searches hlist using search condition (selectNum)
and search range (lower, upper).
Then output the list of the house at ascending order.
To internal process, the number of the house contained in hlist
(dataNum) is given.
Output of each house data is performed by show_house function.

```
selectNum: area[0], distance(m)[1], exit[9]
```

```
struct house set_house(char name[], int area, double distance);
This function set a data of house to house structure.
Return is a structure containes the house data.

void show_house(struct house *hs);
Output the house data (name, area, distance, rent) from hs.
```

B.3 Detailed Design Document used on Code Review

```
int main(int argc, char *argv[]);
```
At first, this function gets a name of data file from command-
line argument.
Then read the data file using read_housefile function, and
store the list of house to filename which used by every other
functions. When the file contains no data, output the message,
then exit. At next, give the user input (search condition and
range) to house_search function.
The system exit when the user select exit.

```
int read_housefile(struct house hlist[], char filename[]);
```
This function reads the data file (filename) and stores a list
of house to house list (hlist). The maximum number of the data
is 100. More than the maximum number of data is ignored.
Each house data is stored to the structure using set_house
function. Return of the function is the number of house read
from the file.

```
void house_search(struct house hlist[], int dataNum,
                              int selectNum, int lower, int upper);
```
This function searches hlist using search condition (selectNum)
and search range (lower, upper). Then output the list of the
house. To internal process, the number of the house contained
in hlist (dataNum) is given.
Output of each house data is performed by show_house function.

selectNum: area[0], distance(km)[1], rent(JPY)[2], exit[9]

```
struct house set_house(char name[], int area, double distance,
                                                   int rent);
```
This function set a data of house to house structure.
Return is a structure containes the house data.

```
void show_house(struct house *hs);
Output the house data (name, area, distance, rent) from hs.
```

B.4   Source Code

```
#include<stdio.h>
#include<string.h>

#define MAXHOUSE 255

struct house {
  char   name[20];
  int    area;
  double distance;
  int    rent;
};

struct house set_house(char name[], int area,
                                 double distance, int rent){
  struct house temp;

  strcpy(temp.name, name);
  temp.area = area;
  temp.distance = distance;
  temp.rent = rent;

  return temp;
}

void show_house(struct house *hs){
  printf("%s\t", hs->name);
  printf("%d\t", hs->area);
  printf("%lf\t",hs->distance);
}

int read_housefile(struct house hlist[], char filename[]){
  int    i;
  FILE   *fp;
  char   name[20];
```

```
  int    area;
  double distance;
  int    rent;

  fp = fopen(filename,"r");
  if(fp == NULL){
    printf("Cannot open file %s. \n",filename);
    fclose(fp);
    return 0;
  }else{
    i = 0;
    while(fscanf(fp,"%s\t%d\t%lf\t%d", name, &area, &distance,
                                                &rent) != EOF){
      hlist[i] = set_house(name, area, distance, rent);
      i++;
    }
    fclose(fp);
    return i;
  }
}

void house_search(struct house hlist[], int dataNum,
                               int selectNum, int lower, int upper){
  struct house *ph;

  printf("Name          area    distance   price\n");

  for(ph= hlist; ph < hlist+dataNum;ph++){
    switch(selectNum){
    case 0:
      if(ph->area >= lower && ph->area <= upper)
show_house(ph);
      break;
    case 1:
      if(ph->distance >= (double)lower ||
                         ph->distance <= (double)upper)
```

```
show_house(ph);
      break;
    case 2:
       if(ph->rent >= lower && ph->rent <= upper)
show_house(ph);
      break;
    }
  }
}

int main(int argc, char *argv[]){
  struct house houseList[MAXHOUSE];
  int i;
  int dataNum;
  int selectNum;
  int lower,upper;

  if(argc != 2){
    printf("Usage: %s datafile\n", argv[0]);
    return 1;
  }

  dataNum = read_housefile(houseList, argv[1]);

  if (selectNum > 0) {

    while(1) {
      printf("Area[1], distance(km)[2], price[3], exit[9]: ");
      scanf("%d", &selectNum);

      switch(selectNum){
      case 0: printf("Search by Area.\n"); break;
      case 1: printf("Search by distance.\n"); break;
      case 2: printf("Search by price.\n"); break;
      case 9: printf("Exit.\n"); goto Loop_End;
      default: printf("Invalid input.\n"); continue;
```

```
      }

      printf("#Input range by integer \"Upper-Lower\":");
      scanf("%d-%d",&lower,&upper);

      house_search(houseList, dataNum, selectNum, lower, upper);
    }
  Loop_End:;
  }else{
    printf("Empty data file.\n");
  }
  return 0;
}
```

B.5   Data file

```
LionHouse 40 5.7 150000
PensionTamada 10 2.5 70000
DormitoryNaka 25 3.1 80000
PalaceIgaki 50 1.1 350000
RabbitHome 7 1.7 40000
YamadaSo 12 3.2 60000
OtogiriSo 45 10.0 120000
PrismTanaka 31 15.1 115000
FrontIshii 25 2.2 100000
TowerSuit 33 0.7 250000
HillSasaki 18 7.0 80000
MountTakata 15 3.2 55000
HeavenCry 17 5.5 100000
TearHamada 62 4.8 330000
SkyCourtArima 35 3.2 150000
VogueYoshino 6 2.7 52000
GrandYukari 8 4.1 35000
BeautyUmezawa 21 7.2 72000
SkynetTunoda 9 2.8 68000
RandomMurakami 11 0.9 170000
```

B.6   Checklist for Detailed Design Review

```
Completeness
  Required functions are entirely comprised by design specifi-
  cation.
  Every required output is created correctly.
  Every input to the system is provided.
  External data such as file is provided.

Data structure
  Data structure of the system is defined.
  Every roles and functions of data is clarified.

Function
  Necessary functions are listed correctly.
  Every roles of each function is clarified.
  Every interface of each functions is defined.
```

B.7 Checklist for Code Review

```
Completeness
  Every requirement is implemented correctly.


Initialization
  At following steps, valiables and parameters are initialized.
    Beginning of the program.
    Beginning of loop blocks.
    Before function call.


Function call
  Types and order of parameter are correct.
  Usage of pointers is correct.
  The function returns correct value.


Operation
  Operations such as =, ==, && are used correctly.
  Each sign of inequality has correct direction.
  Every operator is used with correct priority.


Data file
  Every data has correct types and values.
  Access to data is correct.
  Every file fulfills following conditions.
  Define correctly before access.
  It is opened before access.
  It is closed after access.


Conditional statement
  Statements start/end correctly.
  Statements have correct condition.
```

Index