How Effective Developers Investigate Source Code: An Exploratory Study

Martin P. Robillard, Wesley Coelho, and Gail C. Murphy, *Member*, *IEEE Computer Society*

Abstract—Prior to performing a software change task, developers must discover and understand the subset of the system relevant to the task. Since the behavior exhibited by individual developers when investigating a software system is influenced by intuition, experience, and skill, there is often significant variability in developer effectiveness. To understand the factors that contribute to effective program investigation behavior, we conducted a study of five developers performing a change task on a medium-size open source system. We isolated the factors related to effective program investigation behavior by performing a detailed qualitative analysis of the program investigation behavior of successful and unsuccessful developers. We report on these factors as a set of detailed observations, such as evidence of the phenomenon of inattention blindness by developers skimming source code. In general, our results support the intuitive notion that a methodical and structured approach to program investigation is the most effective.

Index Terms—Software evolution, empirical software engineering, program investigation, program understanding.

1 Introduction

Useful software systems change [14]. As observed by Boehm [2], modifying software generally involves three phases: understanding the existing software, modifying the existing software, and revalidating the modified software. Thus, before performing a modification to a software system, a developer must explore the system's source code to find and understand the subset relevant to the change task [27]. Often, the code relevant to a change task is scattered across many modules [10], increasing the difficulty of the task.

To help developers discover and understand the parts of a program[1] that may be related to a change task, software development environments (SDEs) provide features enabling users to perform lexical and structural searches on the source code of a program [4], [12], [15], [20], [30]. As an early example, the Interlisp-D environment provided cross-referencing support [25]. Cross-references allow developers to obtain information that cannot easily be gained through code inspection. For example, a classic cross-reference query produces all the callers of a function. Search features such as cross-references are now commonly found in current software development environments (e.g., the Eclipse platform [19]).

Unfortunately, the provision of search features is not sufficient to ensure the effective investigation of a program by a developer. For example, the keywords to provide as input to lexical searches, or the choice of cross-reference queries to perform, are determined by developers based on a mix of intuition, experience, and skill. The behavior of a developer investigating a program is thus a highly personal process which, combined with other factors, is responsible for the large variability in efficiency that is often observed between individual developers.

We were interested in isolating the factors influencing the success of a software modification task that are strictly associated with the behavior of a developer (rather than external factors such as the influence of the workplace [9], the programming environment used [8], [24], or the expressiveness of the notation used [29]). We believe understanding the nature of program investigation behavior that is associated with successful software modification tasks can help us improve the tool support and training programs offered to software developers.

To investigate the links between program investigation behavior and success at a software modification task, we conducted a study of five developers undertaking an identical software change task on a medium-sized system. The main goal of our experimental design was to achieve the highest level of realism for the task studied while still being capable of replicating the study. For this reason, our methodology focused on a context-rich, detailed qualitative analysis of a few replicated cases, rather than a statistical analysis of causality between dependent variables. By performing a detailed qualitative analysis contrasting the program investigation behavior of successful and unsuccessful developers, we were able to isolate the factors which we believe are associated with effectiveness during the program modification task.

Our results are presented as a theory of program investigation effectiveness that takes the form of a series of observations and associated hypotheses, each supported by detailed evidence taken from video recordings of the actions of each software developer. Overall, we found that

1. We use the terms *program* and *software system* interchangeably.

- *M.P. Robillard is with the School of Computer Science, McGill University, 3480 University Street, #318 Montréal QC Canada H3A 2A7. E-mail: martin@cs.mcgill.ca.*
- *W. Coelho and G.C. Murphy are with the Department of Computer Science, University of British Columbia, 201-2366 Main Mall, Vancouver BC Canada V6T 1Z4. E-mail: {Coelho, Murphy}@cs.ubc.ca.*

Manuscript received 7 June 2004; revised 11 Oct. 2004; accepted 18 Oct. 2004.
Recommended for acceptance by D. Rombach.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0109-0604.

successful developers exhibited a highly methodical approach to program investigation. Specifically, they investigated enough of the code to understand the high-level structures of the system, prepared a detailed plan of the change to be made, implemented the plan in a mostly linear fashion, and used structurally guided searches to investigate the system. In contrast, the behavior of unsuccessful developers was clearly ad hoc and opportunistic, relying heavily on code skimming and guessing. Surprisingly, the methodical approach did not lead to a longer time required to finish the task. On the contrary this approach allowed the successful developers to complete the task in close to half of the time allotted.

Our study has several novel features. First, it involved a more complex task and larger system than most previous multisubject studies of developers. Second, many of the previous studies (e.g., [5], [6], [7], [11], [18], [34]) have relied on analyses that were based on heavily abstracted characterizations of both developer behavior and success level. Although this strategy allowed the investigators to study a higher number of subjects, potentially increasing confidence in the results, it also limited the scope of the results. In contrast, our analysis involved a detailed study of the code examined by each developer and of the methods used by the developers to navigate between different locations in the code. Furthermore, our observations take into account a detailed analysis of the actual source code modified by each developer.

The contributions of this paper are twofold. First, we provide a set of detailed observations about the characteristics of effective program investigation behavior. These observations are accompanied by hypotheses that can be validated by additional research and practical experience. Second, we describe a methodology and analysis technique for studying the behavior of software developers that can be used by researchers to further test the hypotheses proposed in this paper and to perform other detailed investigations of programmer behavior.

The rest of the paper is organized as follows: In Section 2, we present the details of the modification task studied and our methodology. In Section 3, we describe how we performed an analysis of the data to determine the success level of each developer and to characterize their behavior. Section 4 is the presentation of our observations. In Section 5, we discuss the factors influencing the validity of our study. In Section 6, we present the related work, and conclude in Section 7.

2 Methodology

Our goal for this research was to be able to make hypotheses about the characteristics of the program investigation process that are associated with success at a software modification task. For this reason, we chose an exploratory study format. Our research relies on two main assumptions:

- There exists a relation between program investigation behavior and the success with which developers can perform a software modification task.
- Personal characteristics such as the skill, experience, and expertise of developers are reflected in their program investigation behavior.

We argue the validity of our assumptions by showing the absurdity of a situation in which they would not hold. Rejecting the first assumption would imply that a developer could investigate only irrelevant methods and still perform a correct change. Rejecting the second assumption would imply that the program investigation behavior of developers is either completely algorithmic or completely random.

These assumptions allow us to focus on the characteristics of program investigation behavior that influence developer effectiveness without having to consider the factors explaining the behavior. As a consequence, we believe our results are more useful and generalizable.

To fulfill our research goal, we undertook a study of the behavior of different developers as they performed an identical change task. To ensure we were studying the appropriate phenomenon, our experimental methodology needed to meet two specific requirements: realism and replication.

Realism. To explore the program investigation behavior of developers, we needed to study a situation representative of realistic program modification tasks. In practice, this meant a change task challenging enough to require developers to spend a significant amount of effort investigating the system, and a system large enough to preclude a "systematic" program understanding strategy [28], and developed by multiple people over multiple evolution cycles.[2]

Replication. To allow us to identify the characteristics of program investigation behavior associated with successful program modification tasks, we needed to be able to contrast the behavior of successful and unsuccessful developers. This requirement implies some form of replication. Additionally, because success at program change tasks can be heavily influenced by the nature of the task, programming language used, and other factors independent of developers, we needed to be able to control the task and the environment in which the work was performed.

These requirements are conflicting because, as we study more realistic tasks, we face increasing problems controlling for factors that might influence the results, and increasing difficulty analyzing enough of the data collected to account for the complexity of the phenomenon observed. To achieve a balance, we chose a three-hour software modification task on a 65kLOC[3] text editor system written in Java. We replicated the study with five different developers as this amount of replication offered a reasonable tradeoff between the cost of replication and detailed qualitative analysis and the generalizability of the results [22], [35]. Since our study was exploratory, we were interested in making observations based on a detailed, qualitative investigation of program investigation behavior rather that testing causality hypotheses using statistical inference.

2. In this paper, we consistently use the term "systematic" in the sense of Soloway et al., to mean a line-by-line read of the entire source code of a program.

3. Thousand lines of source code, excluding comments and blank lines.

In the rest of this section, we describe the details of our experimental setting, our data collection methods, and the coding and analysis methodology we used to process the raw data in preparation for the detailed analysis.

2.1 The Study

Our study setup was to ask five developers to individually perform an identical software modification task.

2.1.1 The Task

The target system for our study was the jEdit text editor (version 4.6-pre6).[4] jEdit is written in Java and consists of 64,994 noncomment, nonblank lines of source code, distributed over 301 classes in 20 packages. Among other features, jEdit saves open file buffers automatically. Our study focuses on this autosave feature.

In version 4.6-pre6, any changed and unsaved (or dirty) file buffer is saved in a special backup file at regular intervals (e.g., every 30 seconds). This frequency can be set by the user through an Options page brought up with a menu command in the application's menu bar. If jEdit crashes with unsaved buffers, the next time it is executed, it will attempt to recover the unsaved files from the automatically saved backups. In the initial version of jEdit used for our study, a user can disable the autosave feature by specifying the autosave frequency as zero. However, this option is undocumented, and can only be discovered by inspecting the source code.

The task we requested of subjects consisted of the following modification request.

Modify the application so that the users can explicitly disable the autosave feature. The modified version should meet the following requirements.

1. *jEdit shall have a check box labeled "Enable Autosave" above the autosave frequency field in the Loading and Saving pane of the global options. This check box shall control whether the autosave feature is enabled or not.*
2. *The state of the autosave feature should persist between different executions of the tool.*
3. *When the autosave feature is disabled, all autosave backup files for existing buffers shall be immediately deleted from disk.*
4. *When the autosave feature is enabled, all dirty buffers should be saved within the specified autosave frequency.*
5. *When the autosave feature is disabled, the tool should never attempt to recover from an autosave backup, if for some reason an autosave backup is present. In this case the autosave backup should be left as is.*

The change corresponding to this request for modification, as initially performed by an author of this paper in preparation for the study, amounted to about 65 lines of modified, deleted, or added code, scattered in six files.

2.1.2 Subject Selection

Subjects for this study were recruited through a mailing list in the Department of Computer Science at the University of British Columbia, and through personal contacts. Subjects were required to have a minimum level of Java programming experience and experience with the maintenance of software systems of at least medium size. Student applicants with programming experience gained through cooperative work terms and graduate research projects were accepted for the study, since this level of experience corresponds to that of entry-level professional developers. No current member of our research group was accepted for this study. Subjects were paid for their time.

2.1.3 Study Setting

The study was divided into three phases. To minimize potential investigator bias, each phase was described entirely through written instructions. In any phase, the subjects could ask questions, but we established guidelines restricting answers from the investigator to clarification of the written material.

Eclipse Training Phase. To investigate the code and to perform the change, subjects used the Eclipse integrated programming environment configured for Java [19]. To ensure that all subjects had sufficient familiarity with the development environment, we first had the subjects complete a tutorial on how to use the principal features of Eclipse required for the study: code browsing and editing, and performing searches and cross-references. This phase was limited to 30 minutes. Subjects already familiar with Eclipse were asked to read through the tutorial, but could end the training period at their discretion. Before continuing on to the next phase, the subjects had to pass a simple proficiency test, in which the investigator asked them to perform various tasks covered in the tutorial. All subjects passed the Eclipse training test.

Program Investigation Phase. After the training phase, the subjects were asked to read some preparatory material about the change to perform. This material included excerpts from the jEdit user manual describing file buffers and the autosave feature, instructions on how to launch jEdit and test the autosave feature, the change requirements listed in Section 2.1.1, and a set of eight test cases covering the basic requirements. The test cases were sequences of operations on jEdit, described in English, that could be performed using jEdit's graphical user interface. The subjects were allowed to perform these tests at any point during the study. The written material for that phase also included two pointers to the code, intended to simulate expert knowledge available about the change task. These pointers consisted of the classes `Autosave` and `LoadSaveOptionPane`, the classes dealing with the autosave timer and the option pane where the autosave save frequency was set, respectively. We felt such pointers were a reasonable starting point since this type of information is often found in requests for modifications and bug descriptions managed by systems such as Bugzilla.[5] In our case, using such starting points enabled us to reduce the variability of the behavior analyzed, with the benefit of increasing the value of any contrast observed between subjects.

The subjects were then given one hour to investigate the code pertaining to the change in preparation to the actual task. The subjects were to investigate the code using the

4. http://www.jedit.org.

5. http://www.bugzilla.org.

features of Eclipse covered in the training phase, but were not instructed which features to use or how and when to use them. The subjects were informed about the possibility of taking notes in a text file, but were not explicitly instructed to do so. The subjects were also allowed to execute the jEdit program, but not to change any code, even temporarily, nor to use the debugger. We set these restrictions to reduce the influence of debugging skills in Eclipse on the results. The intent of the initial investigation phase was to introduce some control over the study, and to prevent the case where a developer would try to perform the change with almost no prior investigation. Although this is possible in the case of trivial tasks, we knew from designing the study that the possibility of succeeding at the task without investigating the program was extremely unlikely. We thus wanted to prevent developers from coming to this realization halfway through the study. Section 5.3 discusses the potential impact of this choice on the generalizability of our results.

Program Change Phase. In this phase, subjects were simply instructed to implement the requirements, and that this implementation needed to pass the tests. The subjects were given two hours to complete this phase of the study. The instructions for this phase were purposefully left as general as possible so as not to be leading. However, it elicited questions to determine when the change was "completed." These questions were answered with the clarification to implement the requirements as well and as efficiently as possible and to end the study when they felt they had done so. The subjects who completed the study before the prescribed time independently identified the completion point for the study. In this phase, use of the debugger was again disallowed. At the end of the phase (either triggered by the subject or after two hours), an investigator ran through the test cases that had been provided to the subjects.

2.2 Data Collection

We collected two types of data as part of the study: artifacts produced or modified by the subjects and a record of the actions of the subjects during their task. The artifacts collected include any source code changed during the task, and any additional documents produced, such as a free-form text file with personal notes. To record the actions of a developer in the investigation and modification phases, we captured a video of the screen using the Camtasia screen recording program[6] operating at 5 frames/second and a resolution of $1,280 \times 1,024$ pixels.

3 Data Analysis

Our basic strategy for data analysis is inspired by the method described by Seaman [26]. First, we looked for patterns, contrasts, and commonalities in the behavior of successful versus unsuccessful developers. We then derived observations based on a detailed study of the behavior of developers surrounding these patterns, contrasts, and commonalities.

6. http://www.techsmith.com.

TABLE 1
Time Taken to Complete the Change Phase of the Study

Subject	1	2	3	4	5
Time (minutes)	125	62	72	114	120

This methodology required two types of data analysis:

- an evaluation of the success level of each developer, and
- an abstraction of the behavior of each developer to guide the detailed analysis process.

3.1 Analysis of Developer Success

We evaluated the success level of each developer by analyzing two types of data: the time taken to complete the change and the quality of the coded solution.

3.1.1 Time

Table 1 presents the time taken by each subject to complete the change phase of the study (this time was capped at 125 minutes: two hours plus a five-minute grace period).

As this table shows, an important difference exists between subjects 2 and 3, who completed the change in just above half of the total allocated time, and subjects 1, 4, and 5, who used practically the entire time allocated to the change phase of the study.

3.1.2 Quality of Change

We assessed the quality of the change carried out by each developer by inspecting the details of the source code changes performed. This analysis was carried out by the first author of this paper, who designed the experiment and had a detailed understanding of the code of jEdit pertaining to the autosave feature. Our analysis involved three steps:

1. We analyzed the jEdit source code in detail to determine the characteristics of an ideal solution. We considered a solution to be ideal if it correctly implements the requirements *and* respects the design of jEdit.
2. Based on our analysis of the implementation of jEdit and the requirements of the task, we divided the task into five subtasks. A subtask was defined as the implementation of a relatively independent subset of the requirements (with the exception that it could depend on other subtasks). Although this separation into subtasks is subjective, we found that it naturally aligned with the requirements.
3. We examined how each subject had implemented each subtask, and characterized its quality.

Task breakdown. Fig. 1 is a diagram representing each subtask and the dependencies between them. The top-level task is the complete modification task. To realize this task, developers had to correctly implement subtasks 1 to 5, where subtasks 2 to 5 in turn depend on the correct implementation of subtask 1. To provide context for the

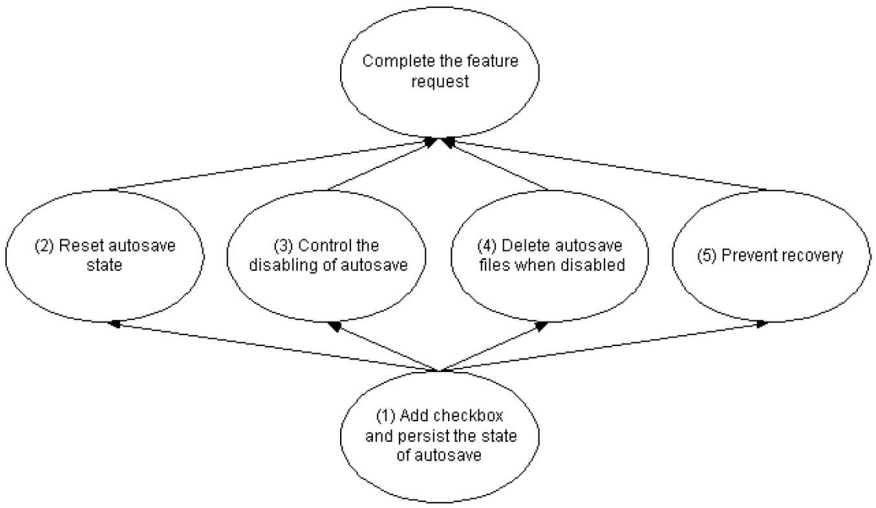

Fig. 1. Decomposition of the requirements of the task.

analysis described in the rest of the paper, we describe the challenges associated with each subtask.

1. *Add check box and persist the state of autosave*. This subtask involves adding a check box widget in the option pane for loading and saving files. Performing this subtask requires adding code to methods `_init` and `_save` of class `LoadSaveOptionPane` to respectively show a check box to disable autosave and to save its state when the option pane is closed. This subtask can be considered relatively easy since the `LoadSaveOptionPane` class was given as an initial hint to the subjects, and because the code to add has a high similarity to code that already exists in the file to display other check boxes.
2. *Reset autosave state*. In the jEdit application, files are represented in memory as instances of a `Buffer` class. Buffers can be in different states, such as "dirty" (the file contains changes not saved explicitly) and "autosave-dirty" (the file contains changes saved neither explicitly nor automatically). The implementation of this subtask is trivial, involving one line of code, but requires a detailed knowledge of how the state of file buffers is managed. This knowledge involves elements such as methods `setFlag` and `getFlag` of class `Buffer`, information about accesses to a `flag` field in class `Buffer`, and the various `flag` constants (e.g., `AUTOSAVE_DIRTY`). This subtask is particularly difficult because it is not explicitly mentioned in the requirements for the change: subjects must discover that this subtask is necessary in order to ensure a correct implementation of the requirements. However, the test suite given to the subjects exercises this functionality.
3. *Control the disabling of autosave*. Once a check box is installed in the option pane, its selection must cause the autosave feature to be suspended. This requires discovering 1) how to suspend the autosave feature and 2) where to trigger the suspension. In jEdit, the timing of the autosave feature is performed by a class named `Autosave`. This class is provided as an initial hint and is short (78 LOC). It is thus possible to completely understand how to deactivate the autosave feature through a systematic (i.e., line-by-line) study of class `Autosave`. The problem of identifying where to deactivate (and reactivate) the autosave feature is much more complex. To respect the design of jEdit, this action should be performed in a method of the `jEdit` class called `propertiesChanged`, which is used to respond to various changes in the properties of the application. For example, in the version of the code provided to the subjects, `propertiesChanged` contains the code to modify the frequency of the autosave events.
4. *Delete autosave files when disabled*. One of the requirements states that whenever the autosave feature is disabled, all automatically saved backups corresponding to files currently open should be deleted. Performing this subtask requires discovering how to obtain a list of all of the open buffers and how to delete the autosave file associated with the buffers. In jEdit, the simplest way to do this is to call the static method `getBuffers` in class `jEdit`, to use the method `getAutosaveFile` for each buffer, and then to call `delete` on the returned `File` object. Logically, this implementation should take place at the same location as the one used to disable/enable the autosave feature (subtask 3).
5. *Prevent recovery*. This subtask represents requirement 5 of our study. To implement this subtask, the subject must identify where recovery from automatically saved backups is normally triggered, and bypass the recovery stage if the autosave feature is disabled. This requires identifying a call between methods `load` and `recoverAutosave` of class `Buffer`, and to insert a conditional statement in either method to bypass the recovery if the autosave feature is disabled.

Evaluation. Given the task breakdown described above, we inspected the solution provided by each subject and

TABLE 2
Solution Quality for Each Subject

Sub-task/Subject	1	2	3	4	5
1–Check box	**Success**	**Success**	**Success**	Inelegant	**Success**
2–State reset	Not attempted	Buggy	**Success**	Buggy	Not attempted
3–Disabling	Unworkable	**Success**	**Success**	**Success**	Unworkable
4–Deletion	Unworkable	**Success**	**Success**	Buggy	Unworkable
5–Recovery	Unworkable	**Success**	**Success**	**Success**	Not attempted

characterized the success level for each subtask using the following classifications:

- **Success**. The subject provided a correct solution that respects the original design of jEdit.
- **Inelegant**. The subject provided a correct solution that did not respect the original design of jEdit.
- **Buggy**. The subject provided a generally workable solution that contains one or more bugs.
- **Unworkable**. The subject provided a solution that does not work in most cases.
- **Not attempted**. The subject did not provide a solution.

In the evaluation, correctness can be assessed objectively by determining whether the solution meets the requirements and does not contain faults. However, the assessment of whether a solution respects the existing design is subjective. For this reason, we have been conservative in our analysis, and judged that a solution did not respect the existing design only if it clearly broke an existing structure (e.g., by hard-coding a value that should have been obtained from a property object).

Table 2 presents the characterization of the success level for each subtask and each subject. As this table shows, all subjects were able to complete subtask 1. The success level for subtask 1 is not surprising given the hints provided to the subjects, and given the natural inclination of developers to code by example [13], [23]. The other subtasks discriminate the subjects. Subjects 1 and 5 can be considered to have generally failed at the task. Subjects 2 and 3 were highly successful. Specifically, subject 3's solution is ideal, and subject 2's solution contains only a very small logic error, where a flag is reset to `false` instead of the value of another flag. Finally, subject 4's solution can be considered "average," as it is a workable but generally low-quality implementation.

3.1.3 Overall Analysis

Given that subjects 2 and 3 completed the task in close to half the time allotted and provided a high-quality solution, we can confidently characterize these subjects as "successful." Because subjects 1 and 5 could not provide a working solution in the time alloted, we consider these subjects "unsuccessful." With a workable yet low-quality solution, subject 4 cannot be associated with either of the two extremes, and will be referred to as "average." Table 3 provides information on the background of the subjects, showing the approximate and self-reported number of years of programming experience of each subject. Not surprisingly, the unsuccessful subjects are also the less experienced. However, the two successful subjects are not the two most experienced subjects. In any case, the focus of our study was to determine why certain developers are efficient, independently of their background. Our intent is in part to help make novice developers more quickly reach the efficiency level of more experienced developers.

3.2 Abstraction of Developer Behavior

To perform a high-level analysis of the behavior of each developer in our study, we needed an abstraction of their actions. For this reason, we transcribed the videos produced during the investigation and change phases for each subject. Transcribing information-rich media like video inevitably involves deciding which information to include and which to leave out. Because we were mostly interested in studying how each developer navigated through the program code, we produced transcripts listing each method a developer examined during the study. Specifically, for each phase of the study and each subject, we transcribed the corresponding video into a list of events defined as evidence that a developer may be examining the implementation of a method in a code editor. A new event is created every time a new method is displayed in the code editor. For each event we recorded the following information:

- **Time**. The time at which a program investigation event is observed.
- **Method**. The method examined by the developer at the time of the event.
- **Navigation**. The way the method was accessed.
- **Modification**. Whether the method examined was also modified (for the change phase only).

TABLE 3
Programming Experience of Subjects

Subject	1	2	3	4	5
Experience (years)	1	3	5	5	1

The navigation techniques we considered were the following:

- **Scrolling**. The subject reveals a method by scrolling up or down a source file.
- **Browsing**. The subject reveals a method by selecting it in a code browser.[7]
- **Cross-reference**. The subject reveals a method by following a cross-reference in the code.
- **Recall**. The subject reveals a method by returning to an already open editor window.
- **Keyword search**. The subject reveals a method by performing a keyword search.

As an example, Table 4 shows a partial transcript from subject 1 in the study. As this transcript shows, the behavior observed involves the subject revealing method `B89`[8] through a keyword search, then switching to previously examined method `A4`, then recalling method `L3`, and finally scrolling within the file to reveal method `L2`.

The main challenge with coding video data into a transcript that abstracts program navigation as a sequence of method examination events is to determine which method a developer is examining at any point in time. In many cases, it is possible to unambiguously determine this information. For example, when there is only one method displayed in the editor, or when a method is revealed explicitly through an action such as selecting the method from a list. However, in many cases, it was necessary to determine which method was being examined based on more subtle clues such as a mouse pointer hovering over a certain area of the code. This problem made it necessary for an investigator to manually code the entire transcript, introducing a certain amount of subjectivity into the coding.

To address the subjectivity of the coding, two investigators independently coded each video. Table 5 shows the correlations between the transcripts for each subject in terms of common events versus the total number of events (averaged over the two transcripts). The length of the transcripts varied between 35 and 112 events for the investigation phase and between 30 and 164 events for the change phase.

These results show an inter-coder accuracy varying between 73 percent and 83 percent. Most of the discrepancies between coders fall into two categories: not including an event because a method was examined too fast by a subject, or not including methods because there is not enough evidence that the method on the screen is actually the one investigated. We feel this level of accuracy is sufficient for our purposes since the observations we make based on the transcripts are not affected by the differences in the coding of transcripts and because most of the observations relied on both the transcripts and a detailed reinvestigation of the videos.

TABLE 4
Transcript Excerpt

0:26:51	B89	Keyword
0:27:55	A4	Recall
0:28:05	L3	Recall
0:28:09	L2	Scrolling

4 Results

Based on a detailed analysis of the data collected during the study, we made several observations about the program investigation behavior of both the successful and unsuccessful subjects. From these observations, we derived hypotheses about the intrinsic factors influencing the success of developers. We express our findings as a theory of effective program investigation. This theory is formulated as a high-level statement that summarizes our observations and hypotheses. Our general theory of program investigation effectiveness can be stated as follows:

Theory. *During a program investigation task, a methodical investigation of the code of a system is more effective than an opportunistic approach.*[9]

What this theory synthesizes is that during a program investigation session, the successful subjects seemed to be trying to answer specific questions using focused searches. They also recorded their findings in detail, prepared a modification plan, and mostly kept to this plan while implementing the change. In contrast, the unsuccessful subjects exhibited a more opportunistic approach that can be summarized as involving more guessing. Perhaps surprisingly, the effort spent by subjects planning their change did not translate into a longer time required to perform the task. In fact, subjects who were methodical performed the task in half of the time. This being said, the theory does not imply that a purely systematic approach to program investigation is the most effective. Successful subjects also exhibited some opportunistic behavior. However, they exhibited opportunistic behavior much less often than unsuccessful subjects.

7. In our study, the code browser corresponded to the Eclipse Package Explorer View.

8. Indices are used instead of method names for simplicity.

9. We use the term *methodical* in its general sense, to indicate a behavior characterized by method and order. This term can be contrasted with *systematic*, which we use to refer to a line-by-line investigation of the entire source code. We use the term *opportunistic* as an antonym of *methodical*.

TABLE 5
Transcript Correlation

Subject	1	2	3	4	5
Correlation	73%	83%	79%	81%	73%

In the rest of this section, we present specific observations we have made about the program investigation behavior of developers during our study. For each observation, we summarize the supporting evidence, propose a hypothesis to explain the observation, detail any additional evidence that specifically supports the hypothesis, and briefly discuss the implications of the observation for the field of software engineering.

Observation 1 (Locality of Changes). *Unsuccessful subjects made all of their code modifications in one place even if they should have been scattered to better align with the existing design.*

The inspection of the solution for each subject showed that the subjects who provided an unacceptable or average solution (i.e., subjects 1, 4, and 5) had an implementation where all of the solution implementation was located predominantly in one method, which did not respect the existing design. Specifically, subjects 1 and 5 tried to implement the entire functionality in methods `_save` and `_init` of class `LoadSaveOptionPane`, respectively, and subject 4 implemented all the functionality in method `autosave` of class `Buffer`. This observation of the code is corroborated by the following characterization of the behavior of the subjects during the program change phase. Fig. 2 shows, for each subject, the three methods that were modified the most often, with the bars representing the total number of modifications. A method is modified if, during the event associated with it, the subject modified code in the method (see Section 3.2). The name of the method modified most often is also indicated on the figure, and the error bars represent the difference between the two independent codings of the transcripts. As Fig. 2 shows, all subjects except for subject 3 (ideal solution) present a high number of modification events for one method corresponding to the method where most of the change was implemented, and little activity for other methods. In contrast, subject 3 shows a much more even pattern of code modification. This points to the possibility that developers (in our case, even some successful developers) have the tendency to focus all of their changes in one location.

Based on this evidence we propose the following hypothesis.

Hypothesis. *Inadequate investigation prior to performing a change task limits the understanding of the existing design of a system, leading to a change performed in a single location in the code that is better understood by the developer. In other words, inadequate investigation leads to "ignorant surgery" [21].*

Additional support for our hypothesis that colocated modifications are the result of inadequate planning comes from observing that the investigation of the unsuccessful subjects during the investigation phase was desultory and limited. This observation is made by characterizing the investigation performed by each subject during the investigation phase.

Table 6 gives a high-level characterization of the behavior of each subject during the program investigation phase. The second column lists the number of distinct methods examined by each subject (presented as an average over the two transcripts, with the boundaries). The third column shows the proportion of intent-driven discovery events (cross-references and keyword searches) versus the total number of events (including scrolling, browsing, and recalling). This table again presents a sharp contrast between unsuccessful subjects (1, 5) and successful subjects (2, 3). The successful subjects are seen to have investigated more methods during the investigation phase and to have relied more on cross-reference and keyword searches. In contrast, unsuccessful subjects examined very few methods, and relied more on unstructured techniques such as

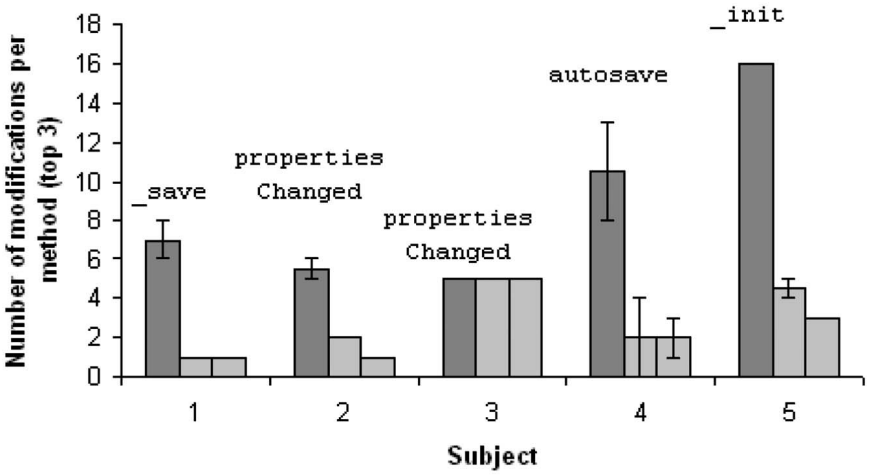

Fig. 2. Methods modified often during the change phase.

TABLE 6
Characterization of the Investigation Phase

Subject	Number of methods	Ratio of cross-reference and keyword events
1	8.5 ± 1.5	$2.0\% \pm 0.7$
2	27.5 ± 0.5	$23.3\% \pm 2.0$
3	34 ± 2	$31.7\% \pm 1.4$
4	38 ± 9	$30.8\% \pm 6.2$
5	17.5 ± 4.5	$10.7\% \pm 3.6$

scrolling and browsing. However, this high-level characterization does not help to explain the behavior of the average subject (subject 4). Although this subject's solution exhibited the problem of being colocated (although to a lesser degree), the abstract behavior of the subject is associated with the one of the successful subjects. This suggests that factors that cannot be characterized by a coarse abstraction of the programmer behavior also come into play, such as how well the information investigated is retained. Observation 3 (below) potentially explains the case of subject 4.

Implications. Assuming the design of a system is not known, a broad cross-section of the system potentially impacted by the change needs to be investigated prior to making a change in order to allow reasoning about design. In other words developers should not rely on studying specific locations in detail with the hope of performing a surgical change as this may lead to code decay.

Observation 2 (Inattention Blindness During Program Investigation). *Program segments that were clearly relevant to the change task were not acknowledged when displayed accidentally.*

This observation was made when we tried to understand why subjects 1 and 5 did not realize that method `propertiesChanged` in class `jEdit` was the key to the success of the task, and as a result could not produce a workable solution (see subtask 3, Section 3.1.2). Subject 5 never investigated the method, which explains the problem. However, the case of subject 1 is more interesting. This subject displayed method `propertiesChanged` while scrolling the code of class `jEdit`. At one point, the subject pauses and the code of the method is shown on the screen, with keywords visible that are strongly associated with the task (e.g., "Autosave"). However, at this point the subject takes no action that indicates that the information was acknowledged. Based on these observations we propose the following hypothesis.

Hypothesis. *Program information relevant to a change is discovered only if it is searched for explicitly. In other words, the hypothesis of "no conscious perception without attention" elaborated in the field of cognitive psychology [17] may apply to program investigation activities.*

The hypothesis of inattention blindness is not refuted by the successful and average subjects since all three found the method through a cross-reference (i.e., explicit) search, and the two successful subjects incorporated the method in the design of their solution.

Implications. The impact of this hypothesis for program investigation tasks is that broad searches without a specific discovery purpose (i.e., skimming the source code) are of little value and should be limited. If our hypothesis holds, unfocused code investigation behavior is open to the problem of inattention blindness.

Observation 3 (Planning). *The successful subjects created a detailed and complete plan prior to the change whereas the unsuccessful and average subjects did not.*

The written instructions provided to the subjects as part of the study stated that any notes they wished to make should be saved in a text file. During the investigation phase, every subject chose to record some information in a text file. Some subjects wrote vague comments on what they intended to do and what they had discovered about how the system worked. Other subjects created a detailed implementation plan that included the names of methods to be modified. We found that subjects who produced the best solutions created and followed a detailed change plan.

Specifically, we considered the content of each subject's notes file as they began the execution phase of the change task. For each subject, we compared the methods intended for modification according to the notes file with the methods actually modified. We also considered the nature and level of detail of the recorded notes.

Subject 3 implemented an ideal solution to the requested task and subject 2's solution was very close to ideal. Both subjects 2 and 3 chose to produce a detailed plan for making the change that included the names of methods that were to be modified. Subject 2 did not modify any methods that were not previously documented as part of the change task. Subject 3 modified only a single method that was not part of his plan and later returned this method to its original state.

TABLE 7
Average Reinvestigations per Window

Subject	10 Method Window	20 Method Window
1	4.6	11.8
2	3.0	8.4
3	3.5	9.3
4	3.5	9.6
5	4.7	12.0

TABLE 8
Average Method Investigation Cycle Length

Subject	Average Cycle Length
1	6.9
2	11.7
3	14.0
4	13.8
5	9.6

This shows that subjects 2 and 3 created complete plans and followed them to produce quality solutions.

The notes for subject 4, whose solution was of average quality, identified three methods to be modified, but seven methods were actually modified during the modification phase. This suggests that subject 4 did not create a complete plan before proceeding with the implementation. Although subject 4 was able to implement the requested change, the solution was of average quality.

Subject 5's notes included a 10-step plan for implementing the required changes. However, the notes were of a high-level nature that described what was to be discovered but did not include actual changes to be made. Although related classes were identified, the plan did not include the name of any methods. Subject 5 modified four methods but was unable to implement the requested change.

Subject 1 was also unable to implement the required changes. Subject 1's notes identified three methods to be modified. All three of these methods plus an additional method that was not in the plan were modified during the execution phase. This shows that Subject 1 created a plan and followed it to some extent. However, the plan was incomplete, as was the implemented change.

Based on this evidence we propose the following hypothesis.

Hypothesis. *Making a detailed change plan allows developers to 1) reason about the extent of the code investigated and assess whether it is sufficient, 2) perform focused program searches in the context of the elaboration of a solution, and 3) eases the cognitive load on developers.*

Implications. Developers should investigate the code of a system related to a change in the context of a change plan explicitly supported by an external medium (e.g., a text file or a dedicated software engineering tool).

Observation 4 (Reinvestigation Frequency). *Successful subjects did not reinvestigate methods as frequently as unsuccessful subjects.*

One way to characterize the behavior of subjects is to look at how often they return to previously investigated methods. Intuitively, a methodical change should involve a mostly linear traversal of the code, with few cycles, whereas we expect an opportunistic approach to involve more iterations. Indeed, in our study we found that the successful subjects navigated the code with longer method investigation cycles.

Specifically, we analyzed the transcripts of each subject's program investigation behavior to determine how frequently methods were reinvestigated. To characterize the iterative nature of the program investigation behavior, we proceeded as follows. Given a sequence of investigated methods from each transcript, we defined a window of n methods. The window was then advanced through the entire transcript one event at a time to consider all consecutive sets of n events. For each window, the number of times a method was reinvestigated was computed. To summarize our analysis, we computed the average number of reinvestigated methods per window. This number provides a measure of the degree to which method navigation events were focused on a small number of methods at a time. Table 7 shows this high-level characterization of the behavior for each subject for window sizes of 10 and 20 methods, respectively.

This table shows a contrast between subjects 2, 3, and 4 (the successful and average subjects, with a value below 3.5), and subjects 1 and 5 (the unsuccessful subjects, with a value above 4.6). This contrast is robust in terms of window size since we observe it for window sizes of both 10 and 20 events (with different values).

To increase our confidence in our high-level assessment, we also characterized the iterative behavior of subjects in terms of the average length of method investigation cycles. We defined an investigation cycle as the investigation of a method followed by the investigation of one or more other methods followed by reinvestigation of the original method. The length of a cycle is simply the number of methods in the cycle. The analysis of subject behavior using this characterization is consistent with our previous one. Specifically, subjects 2, 3, and 4 navigated with longer cycles than subjects 1 and 5. Table 8 lists the average cycle length for each subject.

There are several possible hypotheses that may explain this relationship between reinvestigation frequency and change task implementation success.

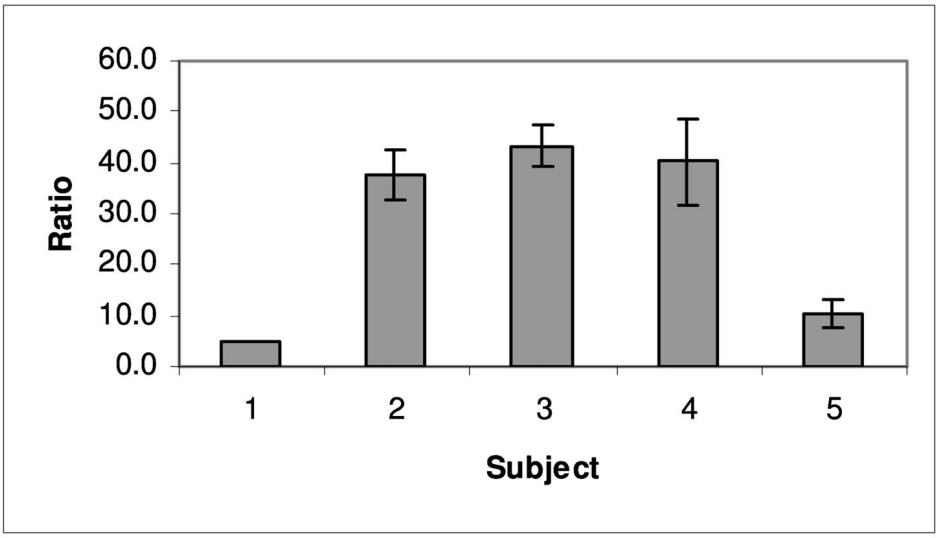

Fig. 3. Ratios of structurally guided searches.

Hypotheses.

- *Developers who are better able to assess the importance of methods they are investigating are more effective. They can identify key methods and spend more time understanding them early in the investigation rather than returning to them only later when their relevance is more evident.*
- *Developers who are better able to understand and remember the methods they have investigated are more effective; they do not need to reinvestigate those methods as frequently.*
- *Developers who have difficulty discovering new relevant methods are less effective; they spend more time reinvestigating methods that have been previously identified as important to the change task.*

Implications. The first hypothesis suggests that developers should carefully assess the potential relevance of each method investigated. If it is possible that the method is relevant to the change task, it may be beneficial to understand its function thoroughly before proceeding, as this strategy may avoid repeated future reinvestigations.

The second and third hypotheses suggest that tools should be used to assist programmers with discovering, understanding and remembering relevant methods. For example, when a developer identifies a relevant method a tool could be used to identify related methods that are also potentially relevant. Tools could also be used to organize and annotate methods that have been discovered and understood.

Observation 5 (Structurally Guided Searches). *The successful subjects performed mostly structurally guided searches (e.g., keyword and cross-reference searches), rather than searches based on intuition (browsing) or aligned with the file decomposition of the system (scrolling).*

We consider keyword searches to be structurally guided because, as we observed, the input to keyword searches and their output is very often the name of program structures (e,g, fields and methods), rather than purely lexical information such as comments.

Evidence of the effectiveness of structurally guided searches comes from two sources involving both a high-level analysis of the transcripts and a detailed analysis of the videos.

A first source of evidence is a high-level analysis of the navigation methods used by each subject. To characterize the general style of navigation method used by a subject we computed the ratio of events in both phases where the navigation method was "keyword" or "cross-reference," and divided this number by the total number of nonrecall events. This characterization differs from the one presented in Table 6 because it does not include recall events in the denominator of the ratio. This has an important consequence for the interpretation of the results: including recall events provides a characterization of what subjects *do*, while excluding recall events provides a characterization of how subjects *search*. Fig. 3 illustrates this last characterization of subject behavior. Each bar represents the ratio for a subject (in percentage points), and the error bars represent the difference between the two independent codings of the transcripts. Again, this figure shows a contrast between the unsuccessful subjects (1 and 5) and the successful and average subjects (2, 3, 4).

A second source of evidence for this observation is simply a corroboration with the evidence provided for Observation 2 stating that the successful and average subjects had all identified the most critical method for the change through a cross-reference query.

Based on this evidence, we propose the following hypothesis.

Hypothesis. *Without a detailed knowledge of the implementation of a system, guessing which method to look at based on its name (browsing) or looking at methods simply because they are in the same file as another relevant method (scrolling) is not as effective as performing structurally guided searches.*

Implications. Developers should resist the temptation to try to guess the code relevant to a change based on nonstructural clues. In other words, developers should minimize the use of scrolling. Even though this strategy may seem more effective that methodically investigating the structural links in a system, our study provides evidence that it is not.

5 Experimental Critique

Several factors potentially affect the validity of our study.

5.1 Construct Validity

The test of *construct validity* questions whether the operational measures used correctly reflect the concept studied [35]. Ensuring construct validity is generally a difficult problem in empirical studies of programmers involving multiple subjects. Abstracting the behavior of developers or the result of their task inevitably introduces the possibility for the variables studied to not truly represent the phenomenon observed. Realizing this challenge, one of the main goals of our study was to analyze the behavior of developers in detail and thus, implicitly, ensure a high level of construct validity. Specifically, our study relies on two main measures: the evaluation of each developer's solution and an analysis of the behavior of the developers. To evaluate the solution produced by the subjects, we performed a detailed inspection of the source code for the solution of each developer. Although our final evaluation relies on a subjective assessment of the quality of the solution, this assessment is directly based on the unabstracted raw data collected, and as such we have a high confidence in the validity of the evaluation. To analyze the behavior of the developers, we partly relied on transcripts which are an approximation of the data recorded. To compensate for the potential errors of interpretation due to the use of transcripts, we corroborated our observations with detailed analysis of the videos whenever possible and, in some cases (e.g., observation 1), we corroborated our findings with data from the code modifications performed by the subjects. The use of multiple sources of evidence is generally considered to increase the validity of qualitative assessments [3].

5.2 Internal Validity

The test of *internal validity* for a study questions whether the results truly represent "a causal relationship, whereby certain conditions are shown to lead to other conditions, as distinguished from spurious relationships." [35, p. 33]. Since our study was exploratory, internal validity in our case relates to the soundness of the evidence used to make hypotheses. We thus discuss the factors potentially affecting our observations, and our attempts to limit them.

One possible source of interference for this study is the possibility that the success level for a subject was determined by prior knowledge, proficiency with the development environment, and investigator bias during the study. To reduce this possibility, we took steps to ensure that no subject had prior knowledge of jEdit, we asked subjects not to communicate the details of the study to others, we provided basic training with Eclipse to each subject, we precluded the use of powerful features of Eclipse, such as the debugger, and we scripted the entire study, limiting the role of the investigator to answering questions. There is always the possibility of investigator bias in the answers to the subject's questions. To limit this effect, we established guidelines at the start of the study for the investigator to use in answering questions: The investigator was to only answer questions about the features of the tools covered in the tutorial, and provide no comment about the task.

Another possibility is that the experimental setup may have affected the behavior of the subjects. In particular, the explicit separation of the task into an investigation and change phase, the use of test cases, and the constraint preventing subjects from modifying the code in the investigation phase, may have caused adjustments to the subjects' behavior. However, we feel that the impact of these necessary constraints is lessened by the fact that the primary object of our study is not the behavior of developers per se, but rather the contrast between the behavior of different developers.

Finally, there exists the possibility that the evaluation of the solution presented in Section 3.1.2 may influence the results. However, this analysis was carefully performed by an investigator who had detailed knowledge of the code of jEdit pertaining to the study. Subjectivity in assessing design was addressed through the use of a conservative criterion. Also, the differences observed between subjects were so large that the evaluation is likely to be robust in the face of a different categorization. In any case, our analysis is transparent: Independent researchers can access the code of jEdit and evaluate the solution. Our general belief is that the evaluation of the solution is stable and poses limited threats to the validity of the study.

5.3 External Validity

The applicability of our findings must be carefully established. All of the subjects were either students or recent graduates from a computer science department. Different results might be obtained from different populations such as, for example, a population of senior developers not formally trained in object-oriented programming. Another threat to the generality of our study is our use of a single task. Although our study involved a nontrivial task requiring developers to reason about different aspects of the system (e.g., control-flow, state transitions), there exists many different types of software modification problems. We do not expect that identical results will be obtained for all problems. In particular, modification tasks involving surgical changes to a single location may yield different results. Nevertheless, we believe that our use of a real task in a program large enough that it cannot be completely understood in a short amount of time contributes to achieving an acceptable level of external validity.

The software development environment and programming language used by the subjects were also fixed. This additional factor limits the generalizability of the study to similar conditions. However, in the case of the development environment, by limiting the use of advanced functionalities, we ensured that the results were not dependent on

platform-specific features. Many integrated software development environments offer features such as the ones used in this study, and we can expect that they are used in a similar fashion.

Finally, an important limit to the generalizability of our findings comes from the fact that we have based our observations on the analysis of the behavior of only five subjects. We could gain additional confidence in our hypotheses by studying more subjects from a larger cross-section of backgrounds. However, by focusing our study on the link between the program investigation behavior and the effectiveness of developers, we limit the risk of our hypotheses not being robust in the face of the personal characteristics of developers that are reflected in the program investigation behavior.

5.4 Reliability

The methodology of this study, including the data collection procedures, has been documented in this paper. The task was defined in detail and an open-source code base was used as a target system. The complete experimental material can be obtained from http://www.cs.mcgill.ca/~martin/tse1. As a result, it should be possible to replicate the study.

6 Related Work

Many empirical studies of programmers have been reported in the literature (see Basili et al. [1] for an overview of the foundational work on this topic). In this section, we provide a brief survey of the studies that focused on program understanding for the purpose of software change tasks, and discuss the novelty of our research in this context.

Letovsky and Soloway performed seminal work on the topic of program understanding. An early study of six professional programmers modifying a 300LOC Fortran 77 program focused on eliciting the causes of comprehension failures, and determined that such failures were often caused by the presence of "delocalized plans" [16]. In another study of 20 programmers modifying a 250LOC Fortran program, Soloway et al. focused on understanding the strategies developers used during a program understanding task [28], and concluded that programmers whose strategy was to systematically study a program line-by-line made fewer errors than the ones using an opportunistic (or "as needed") approach [28]. Our study shares some commonalities with these early studies of Fortran programmers. Notably, both our approaches were exploratory and focused on understanding the details of the programmer behavior in terms of source code modified. However, our experimental methods differ. Soloway et al. used protocol analysis (where subjects are asked to "think aloud"), whereas we judged this technique too disruptive and focused only on the analysis of screen recordings. More important, the target programs used by Letovsky and Soloway were small enough to be analyzed systematically. Judging this an unrealistic scenario in the general case, we used a target program too large to be fully understood by the subjects.

Corritore and Wiedenbeck [5], [6], [7], Engebretson and Wiedenbeck [11], Mosemann and Wiedenbeck [18], and Wiedenbeck et al. [34] carried out many studies to investigate various characteristics of developers involved in program understanding tasks. All these studies share a common methodology. First, a large set of subjects (30-100) studies (and in some cases, modifies) one or more small programs (less than 800LOC). Then, the subjects are asked questions intended to test different aspects of their understanding of the target program. Finally, statistical analysis is performed to test hypotheses concerning their understanding. The main difference between the studies described above and our work lies in the choice of methodology. In contrast to a highly controlled experiment, we used a more realistic setting and performed a more detailed analysis, at the cost of not being able to statistically infer causality. For this reason, the nature of the results are also different: Rather than statistical evidence supporting a small set of hypotheses, we contribute detailed and context-rich data potentially explaining the behavior of developers.

Using the technique of protocol analysis, Vans et al. [31], von Mayrhauser and Vans [32], and von Mayrhauser et al. [33] studied the comprehension processes of programmers during large-scale corrective and perfective maintenance. The approach of von Mayrhauser, et al. also involved the detailed analysis of few subjects (one, two, and four, respectively), and as such can be considered closer to our approach than the studies of Wiedenbeck et al. However, the focus of von Mayrhauser' work was to investigate the cognitive processes of programmers. As a result, the principal source of data is the utterances of the subjects. This is in contrast to our approach, where we have spent most of our effort analyzing the navigation behavior of the subjects.

To summarize, in contrast to previous work, the novelty of our study is that it not only looked at what developers do in general, but focused on determining what *successful* developers do in contrast to unsuccessful ones. This comparative analysis allowed us to make detailed hypotheses about the style of program investigation behavior that contributes to effectiveness at program modification tasks.

7 Conclusions

Prior to performing a software modification task, developers must inevitably investigate the code of the target system in order to find and understand the code related to the change. If we assume that the way a developer investigates a program influences the success of the modification task, then ensuring that developers in charge of modifying software systems investigate the code of the system effectively can yield important benefits such as decreasing the cost of performing software changes and increasing the quality of the change.

The empirical study described in this paper investigated the links between program investigation behavior and success at a software modification task. Although many empirical studies of programmers have analyzed what developers do during program modification tasks, our study is novel in that it considered specifically what *effective*

developers do in contrast to ineffective ones. This analysis was possible because we replicated the study of a developer performing a highly realistic task.

Based on our analysis of the data collected during the study, we observed many sharp contrasts between the behavior of successful and unsuccessful developers and came to the conclusion that *in the context of a program investigation task, a methodical investigation of the code of a system is more effective than an opportunistic approach*. This theory does not imply that a complete line-by-line investigation of a program is the most effective approach, but rather that developers should follow a general plan when investigating a program, should perform focused searches in the context of this plan, and should keep some form of record of their findings. Although this theory will sound intuitive to many, some of our detailed observations include more surprising results, such as the ineffectiveness of the code reading strategy, and the fact that a methodical approach to program investigation does not require more time than an opportunistic approach.

The contributions of our research to the software engineering community are twofold. First, we provide a set of detailed observations about the characteristics of effective program investigation behavior. Although our observations are descriptive, they are accompanied by hypotheses that can be validated by additional research and practical experience. Second, we provide a detailed methodology for performing empirical studies of programmers where it is important that programmer behavior be studied in detail. Researchers can reuse our study to help validate our hypotheses, or to study other aspects of programmer behavior.

ACKNOWLEDGMENTS

The authors are grateful to Davor Čubranić and to the anonymous reviewers for their valuable comments. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by IBM.

REFERENCES

[1] V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. Software Eng.,* vol. 12, no. 7, pp. 733-743, July 1986.

[2] B.W. Boehm, "Software Engineering," *IEEE Trans. Computers,* vol. 12, no. 25, pp. 1226-1242, Dec. 1976.

[3] L. Bratthall and M. Jørgensen, "Can You Trust a Single Data Source Exploratory Software Engineering Case Study?" *Empirical Software Eng.,* vol. 7, no. 1, pp. 9-26, Mar. 2002.

[4] Y.-F. Chen, M.Y. Nishimoto, and C.V. Ramamoorthy, "The C Information Abstraction System," *IEEE Trans. Software Eng.,* vol. 16, no. 3, pp. 325-334, Mar. 1990.

[5] C.L. Corritore and S. Wiedenbeck, "Mental Representation of Expert Procedural and Object-Oriented Programmers in a Software Maintenance Task," *Int'l J. Human-Computer Studies,* vol. 50, no. 1, pp. 61-83, Jan. 1999.

[6] C.L. Corritore and S. Wiedenbeck, "Direction and Scope of Comprehension-Related Activities by Procedural and Object-Oriented Programmers: An Empirical Study," *Proc. Eighth Int'l Workshop Program Comprehension,* pp. 139-148, June 2000.

[7] C.L. Corritore and S. Wiedenbeck, "An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers," *Int'l J. Human-Computer Studies,* vol. 54, no. 1, pp. 1-23, Jan. 2001.

[8] B. Curtis, "Substantiating Programmer Variability," *Proc. IEEE,* vol. 69, no. 7, pp. 846, July 1981.

[9] T. DeMarco and T. Lister, "Programmer Performance and the Effects of the Workplace," *Proc. Eighth Int'l Conf. Software Eng.,* pp. 268-272, 1985.

[10] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," *IEEE Trans. Software Eng.,* vol. 27, no. 1, pp. 1-12, 2001.

[11] A. Engebretson and S. Wiedenbeck, "Novice Comprehension of Program Using Task-Specific and Nontask-Specific Constructs," *Proc. IEEE 2002 Symp. Human Centric Computing Languages and Environments,* pp. 11-18, Sept. 2002.

[12] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[13] B.M. Lange and T.G. Moher, "Some Strategies of Reuse in an Object-Oriented Programming Environment," *Proc. SIGCHI Conf. Human Factors in Computing Systems,* pp. 69-73, 1989.

[14] M.M. Lehman and L.A. Belady, "Program Evolution: Processes of Software Change," *APIC Studies in Data Processing,* vol. 27, 1985.

[15] M. Lejter, S. Meyers, and S.P. Reiss, "Support for Maintaining Object-Oriented Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 12, pp. 1045-1052, Dec. 1992.

[16] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software,* vol. 3, no. 3, pp. 41-49, May 1986.

[17] A. Mack and I. Rock, *Inattentional Blindness.* MIT Press, 1998.

[18] R. Mosemann and S. Wiedenbeck, "Navigation and Comprehension of Programs by Novice Programmers," *Proc. Ninth Int'l Workshop Program Comprehension,* pp. 79-88, May 2001.

[19] Object Technology International, Inc., "Eclipse Platform Technical Overview," white paper, July 2001.

[20] P.D. O'Brien, D.C. Halbert, and M.F. Kilian, "The Trellis Programming Environment," *Proc. Conf. Object-Oriented Programming, Systems, and Applications,* pp. 91-102, Oct. 1987.

[21] D.L. Parnas, "Sofware Aging," *Proc. 16th Int'l Conf. Software Eng.,* pp. 279-287, May 1994.

[22] S.L. Pfleeger, "Experimental Design and Analysis in Software Engineering—Part 3: Types of Experimental Design," *Software Eng. Notes,* vol. 20, no. 2, pp. 14-16, Apr. 1995.

[23] D.F. Redmiles, "Reducing the Variability of Programmers' Performance Through Explained Examples," *Proc. Conf. Human Factors in Computing Systems,* pp. 67-73, 1993.

[24] H. Sackman, W.J. Erikson, and E.E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Comm. ACM,* vol. 11, no. 1, pp. 3-11, 1968.

[25] M. Sanella, *The Interlisp-D Reference Manual.* Xerox Corporation, Palo Alto, Calif., 1983.

[26] C.B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Trans. Software Eng.,* vol. 25, no. 4, pp. 557-572, July/Aug. 1999.

[27] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," *Proc. 1997 Conf. Centre for Advanced Studies on Collaborative Research,* pp. 209-223, 1997.

[28] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert, "Designing Documentation to Compensate for Delocalized Plans," *Comm. ACM,* vol. 31, no. 11, pp. 1259-1267, Nov. 1988.

[29] B. Teasley, L.M. Leventhal, K. Instone, and D.S. Rohlman, "Longitudinal Studies of the Relation of Programmer Expertise and Role-Expressiveness to Program Comprehension," *Proc. NATO Advanced Research Workshop User-Centred Requirements for Software Eng. Environments, NATO Advanced Science Institutes Series-Computer and Systems Science,* vol. 123, pp. 143-163, Sept. 1991.

[30] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer,* vol. 14, no. 4, pp. 25-33, Apr. 1981.

[31] A.M. Vans, A. von Mayrhauser, and G. Somlo, "Program Understanding Behavior during Corrective Maintenance of Large-Scale Software," *Int'l J. Human-Computer Studies,* vol. 51, no. 1, pp. 31-70, July 1999.

[32] A. von Mayrhauser and A.M. Vans, "Identification of Dynamic Comprehension Processes during Large Scale Maintenance," *IEEE Trans. Software Eng.,* vol. 22, no. 6, pp. 424-437, 1996.

[33] A. von Mayrhauser, A.M. Vans, and A.E. Howe, "Program Understanding Behaviour during Enhancement of Large-Scale Software," *J. Software Maintenance: Research and Practice,* vol. 9, no. 5, pp. 299-327, Sept./Oct. 1997.

[34] S. Wiedenbeck, V. Fix, and J. Scholtz, "Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study," *Int'l J. Man-Machine Studies,* vol. 39, pp. 793-812, 1993.
[35] R.K. Yin, "Case Study Research: Design and Methods," *Applied Social Research Methods Series,* vol. 5, second ed. 1989.

Martin P. Robillard received the BEng degree in computer engineering from École Polytechnique de Montréal in 1997 and the MSc and PhD degrees in computer science from the University of British Columbia in 1999 and 2004, respectively. He is currently an assistant professor in the School of Computer Science at McGill University. His research interests are in software evolution, aspect-oriented software development, and empirical software engineering.

Wesley Coelho received the Bachelor of Commerce degree from the University of Victoria in 2000. He is currently a master's student in the Department of Computer Science at the University of British Columbia. His research interests are in aspect-oriented programming and model-based development tools.

Gail C. Murphy received the BSc degree in computing science from the University of Alberta in 1987 and the MS and PhD degrees in computer science and engineering from the University of Washington in 1994 and 1996, respectively. From 1987 to 1992, she worked as a software designer in industry. She is currently an associate professor in the Department of Computer Science at the University of British Columbia. Her research interests are in software evolution, software design, and source code analysis. She is a member of the IEEE Computer Society.