An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks

Amy J. Ko, Brad A. Myers, *Senior Member*, *IEEE*, Michael J. Coblenz, and Htet Htet Aung

Abstract—Much of software developers' time is spent understanding unfamiliar code. To better understand how developers gain this understanding and how software development environments might be involved, a study was performed in which developers were given an unfamiliar program and asked to work on two debugging tasks and three enhancement tasks for 70 minutes. The study found that developers interleaved three activities. They began by searching for relevant code both manually and using search tools; however, they based their searches on limited and misrepresentative cues in the code, environment, and executing program, often leading to failed searches. When developers found relevant code, they followed its incoming and outgoing dependencies, often returning to it and navigating its other dependencies; while doing so, however, Eclipse's navigational tools caused significant overhead. Developers collected code and other information that they believed would be necessary to edit, duplicate, or otherwise refer to later by encoding it in the interactive state of Eclipse's package explorer, file tabs, and scroll bars. However, developers lost track of relevant code as these interfaces were used for other tasks, and developers were forced to find it again. These issues caused developers to spend, on average, 35 percent of their time performing the mechanics of navigation within and between source files. These observations suggest a new model of program understanding grounded in theories of information foraging and suggest ideas for tools that help developers seek, relate, and collect information in a more effective and explicit manner.

Index Terms—Program investigation, program understanding, program comprehension, empirical software engineering, information foraging, information scent.

1 Introduction

Most useful software undergoes a brief period of rapid development followed by a much longer period of maintenance and adaptation to new contexts of use [6], [31]. During this period, software developers spend much of their time exploring unfamiliar parts of a software system's source code in order to understand the parts of the system that are relevant to their current task [47]. Because these parts are typically distributed throughout a system's components and modules [18], [30], this exploration can be both time-consuming and difficult. Therefore, an important challenge in software engineering research is to invent tools that help software developers understand and reshape software more efficiently and effectively.

A central concept in the design of such tools, recently proposed by Murphy et al. [34] (and independently in our earlier work on this topic [26]), is that of a *task context*: the parts and relationships of artifacts relevant to a developer during work on a maintenance task. A number of important contributions have been built around this concept, including ways of representing task contexts [41], [43], tools that enable developers to manually build a task context by selecting program elements [42], and methods of automatically inferring the relevant task context based on a developer's investigations in a development environment [44], [46].

Although all of these ideas show promise in improving a developer's effectiveness on maintenance tasks, a more detailed understanding of how developers form their task contexts and how software development environments (SDEs) are related to this formation could lead to even greater gains, whether through improved tools, more rigorous processes, or other means. There are several unanswered questions in this regard:

- How do developers decide what is relevant?
- What types of relevant information do developers seek?
- How do developers keep track of relevant information?
- How do developers' task contexts differ on the same task?

To begin to answer these questions, we performed an exploratory study of 10 developers using the Eclipse 2.1.2 software development environment (www.eclipse.org) to work on five maintenance tasks on a small system with which they were not familiar. Our central goal was to investigate developers' strategies for understanding and utilizing relevant information and discover ways in which Eclipse and other environments might be related to these strategies. Because we wanted to understand the natural flow of information between developers and their workspace,

- *A.J. Ko and B.A. Myers are with the Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. E-mail: {ajko, bam}@cs.cmu.edu.*
- *M.J. Coblenz can be reached at mcoblenz@andrew.cmu.edu.*
- *H.H. Aung can be reached at hhaung@gmail.com.*

Manuscript received 4 Mar. 2006; revised 1 June 2006; accepted 13 Sept. 2006; published online 14 Nov. 2006.
Recommended for acceptance by W.G. Griswold and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0053-0306.

0098-5589/06/$20.00 © 2006 IEEE Published by the IEEE Computer Society

developers could complete their tasks in any order and had access to any tool or documentation they desired.

This paper extends our work presented at the 2005 International Conference on Software Engineering [26] by taking a closer look at variations in developers' task structures, task contexts, and perceptions of relevance. We also present a new model of program understanding as a process of searching, relating, and collecting information of perceived relevance, in which the development environment plays a central role in influencing developers' perceptions. This model is consistent with existing models of program understanding, while explaining the structure and sequence of developers' actions in more detail. We also found several ways in which development environments help, hinder, and ignore developers' strategies for seeking and managing relevant information. For example, developers spent, on average, 35 percent of their time with the mechanics of redundant but necessary navigations between relevant code fragments. Furthermore, failed searches for task-relevant code began with developers observing misleading information cues in Eclipse. We use these findings to discuss the design of current software maintenance tools and also to motivate the design of new tools.

In the next section, we review related work in the area of program understanding and investigation. In Section 3, we describe the design and methodology of our study. In Section 4, we investigate Eclipse's relationship to developers' work both qualitatively and quantitatively. In Section 5, we discuss the limitations of our method and analyses, and in Section 6, we present our model of program understanding and its relationship to prior theoretical work. In Section 7, we conclude with implications of our results on the design of software development environments.

2 Related Work

There is a long history of studies of program understanding, investigating a variety of factors. Many studies have investigated the *strategies* that developers use to understand programs. Some have discovered that developers ask questions about the structure, intent, and behavior of software when asked to verbalize their thoughts during maintenance tasks [27], [32], [49]. There is a general consensus that these questions lead to informal hypotheses [8], [24], which are then tested using two kinds of strategies. Developers' "systematic" strategies involve forming a concrete plan and performing guided navigations of a program's dependencies. Developers' "as-needed" strategies tend to be more ad hoc, unplanned, and opportunistic [29]. It has been shown in a variety of contexts that the systematic strategies are more productive than the as-needed strategies [7], [32], [37], [45], but that developers generally use a combination of the two [4]. There is also evidence that successful developers (of varying definitions) write explicit implementation plans, including notes about relevant information as they find it during a task [30], [45].

Another line of research has considered the *knowledge* that developers form while understanding programs. For example, studies have shown that developers often determine the statements that could influence a variable (a backward slice) [50] and that automated tools for helping developers determine these statements improve developer success on debugging tasks [20]. Other studies suggest that developers form mental models of the structure, intent, and relationships in code, which guide their decision making [35], [51]. Some studies found that developers have a quickly degrading memory for such information [1], [17], explaining why developers rely so extensively on external memory sources, such as digital or paper notes and whiteboards [16], [39].

Many studies have considered the influence of the *representation* of various aspects of code on program understanding. For example, a number of factors can influence the speed and correctness of comprehension, including the typographic appearance of code [3], the indentation style used [33], the language in which a program is written [23], and naming schemes of identifiers [48]. Although these effects can be quite profound when comparing developers of different expertise, studies have shown that many of these effects disappear with increasing experience [11], [12].

A number of studies have explored the influence of *collaboration* on program understanding. This work has focused on workplace interruptions, finding that developers are interrupted, on average, every three minutes by themselves or others [22], [38] and that developers frequently interrupt others in order to obtain difficult to find or undocumented information about software [5], [30].

The study presented in this paper differs from the prior work in several ways. Our study is the first to explore the relationship between interactive aspects of modern SDEs and program understanding. Studies that have investigated similar issues have focused on other aspects of software development and have involved older tools that differ in their interactive nature from modern SDEs. Many of the studies cited above also placed numerous restrictions on the developers' work in order to isolate the measurement of a single variable. For example, some required developers to separate their work into reading phases and editing phases [45], despite evidence that developers frequently edit code for the sole purpose of understanding (for example, by inserting debug statements to understand a program's execution) [15]. Other studies disallowed the use of debuggers and other tools, again despite evidence of their common and essential use during maintenance tasks [30]. Our study relaxed these constraints, allowing us to study maintenance work involving multiple tasks, multiple tools, and developer-selected orderings.

3 Method

The developers in our study were asked to correctly complete as many of five maintenance tasks over a 70-minute period as possible, while responding to intermittent, automated interruptions. Three of the tasks were debugging tasks, requiring developers to test the program and diagnose a particular failure. The other two tasks were enhancement tasks, which required developers to understand some portion of the system and modify it in order to provide a new feature. We included interruptions because of recent evidence that interruptions are frequent in

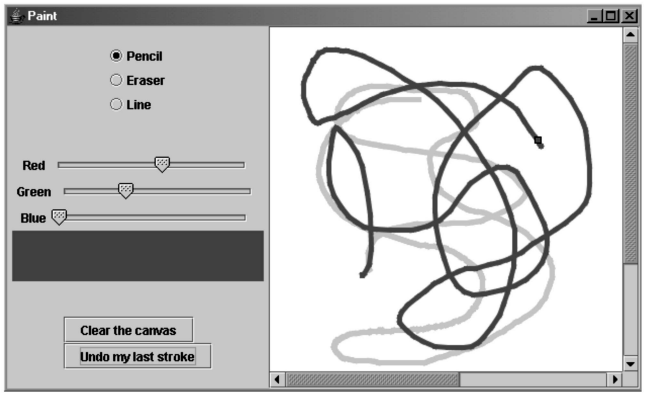


Fig. 1. The *Paint* application.

software engineering workplaces [22], [38] and we wanted to see in what ways development environments could help developers manage and recover from them. Our decision to study developers in the lab instead of in the context of developers' actual work was driven by our interest in comparing developers' strategies on identical tasks. Had we studied developers working on different code, as would have been the case in a more realistic context, we would not know whether differences in developers' work, if any, were due to variations in their strategies, in their code, or more likely, some combination of these and other factors.

3.1 Participants

We recruited 31 developers with experience with Java from the local community, including both undergraduate and graduate students and staff programmers. Analyses of various subsets of these developers' data have appeared in other publications [19], [25], [26]; our analyses in this paper focus on the 10 developers most experienced with Java, based on a pretest, self-report survey: seven described themselves as "Java experts" and the remaining three described themselves as having "above-average" Java expertise. All reported using either Eclipse or Visual Studio "regularly," and reported programming a median of 17.5 hours a week (the distribution was bimodal, with developers programming either less than 20 hours or more than 35). Although all claimed some degree of Java expertise, we did not expect nor want a strictly uniform sample because we were interested in seeing a variety of approaches to completing the tasks that we designed. We did, however, want to avoid analyzing novice Java programmers' work because of the high variability in their knowledge and decision making [14]. The 10 developers we studied were all male, had ages ranging from 19 to 28, and included six senior computer science students, two doctoral students in computer science, and two MS students in computer engineering and information systems.

3.2 The Paint Application

All of the tasks involved a program called *Paint* (shown executing in Fig. 1). This was a Java Swing application, implemented with nine Java classes across nine source files and 503 noncomment, nonwhitespace lines (available at http://www.cs.cmu.edu/~marmalade/studies.html). The application allowed users to draw, erase, clear and undo colored strokes on a white canvas. Its implementation was based on the `PaintObjectConstructor` class, which created a single `PaintObject` for each list of mouse locations accumulated between mouse down and up events. The canvas consisted of an ordered list of `PaintObject` instances, which was rendered from least to most recent. The application declared two subclasses of `PaintObject`: `PencilPaint` and `EraserPaint`. The `PencilPaint` class painted itself by iterating through the list of mouse coordinates and drawing beveled line segments between consecutive pairs. The `EraserPaint` class subclassed `PencilPaint`, overriding its `getColor()` method to return the color of the canvas, simulating the effect of an eraser. Developers were given no documentation about the implementation and the code was not commented.

Although the program was reasonably complex given its small size and the lack of documentation about its design, it was not as complex as programs that have been used in other recent studies [45], which were on the order of tens of thousands of lines long. Our primary reason for studying a smaller program was that it allowed us to investigate developers' work on several different tasks and detect variations in developers' strategies on these different tasks; most prior studies have focused on a single task on a larger system.

3.3 Tasks

The developers were given a sheet of paper with the text in the middle column of Table 1, which describes five invented user complaints and requests. The task names in Table 1 are used throughout this paper but were not given to developers. The descriptions explained the requirements for each task, so that each developer would have a similar understanding of the functional requirements. The last column of Table 1 describes a solution to each problem, including the minimum number of lines that had to be added, removed, or modified, and in how many files. These solutions were deemed by the author of *Paint* to be most consistent with the program's design. Because there were many valid solutions for each task, we accepted any solution that led to appropriate behavior (developers' actual code was verified later before our analyses). The errors for the debugging tasks were not artificial, but emerged during the creation of *Paint*.

3.4 Tools and Instrumentation

The developers were given the Eclipse 2.1.2 IDE (released in March of 2004) and a project with the nine source files. They were allowed to use debuggers, text editors, paper for notes, and the Internet. The only resource they were not allowed to use was the experimenter, who was only permitted to answer clarifying questions about the functional requirements described in the task descriptions. The browser's default page was the Java 1.4 API documentation. Developers used a PC with Windows XP, a keyboard, a mouse with a scroll wheel, and a 17″ LCD. Because our analyses would involve a careful inspection of the developers' actions, even at the level of mouse cursor movements, we recorded every detail of developers' work with full screen-captured videos at 12 frames per second in 24-bit color, as well as audio. The display was limited to a resolution of $1{,}024 \times 768$ to prevent any impact of the recording on the PC's performance.

TABLE 1
The Five Maintenance Tasks

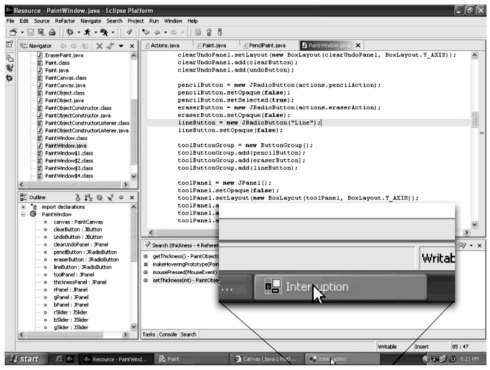
Task Name	Complaint or Request and Task	Ideal Solution
SCROLL	Users complained that scroll bars don't always appear after painting outside the canvas, but when they do appear, the canvas doesn't look right. Fix Paint so that (1) the scroll bars appear immediately when painting outside the visible canvas and (2) the canvas is correctly rendered when using the scroll bars to navigate the canvas.	The "preferred size" of the canvas in the scroll pane was not updated as strokes were created, preventing the scroll bars from appearing correctly and causing the pane to only repaint a fixed region. Developers needed to understand the behavior of the `JScrollPane` component of the Swing API. The best correction involved adding code to call `setPreferredSize()` on the canvas to update its size whenever a mouse release event occurred outside the canvas's current boundaries. **Minimal change:** add 5 lines in PaintCanvas.java.
YELLOW	Users complained that they can't select yellow. Fix Paint so that users can paint with the color yellow.	The green slider's value was referenced twice in the `colorChangeListener`, which responded to slider events. As a result, the blue slider's value was ignored. The best correction involved changing the reference to `gSlider` to `bSlider`. **Minimal change:** modify 1 line in PaintWindow.java.
UNDO	Users complained that the "Undo my last stroke" button doesn't always work. Fix Paint so that the Undo my last stroke button undoes the last stroke or clear of the canvas.	There was no repaint call after the undo operation, causing the window to repaint only after some other operation caused the window to repaint. Other nearby and related methods did call repaint after their operation. The best correction involved adding a call to `repaint()` after the operation. **Minimal change:** add 1 line in PaintWindow.java.
LINE	Users requested a line tool. There's a radio button for it, but it doesn't work yet. Create a line tool that allows users to draw a line between two points. Users should be able to see the line while dragging.	The most straightforward solution involved subclassing the `PencilPaint` class and revising its painting algorithm to draw a single line segment between the first and most recent points in the list of mouse coordinates. **Minimal Change**: create a new class file, override 1 method with a new painting algorithm of at least 10 lines, and add 5 lines in two other files to attach to the class to radio button in the user interface.
THICKNESS	Users requested control over the stroke thickness of the pencil, eraser, and line tools. Create a thickness slider that controls the stroke thickness for all tools, with a pixel range of 1 to 50.	This task required the instantiating, initializing and adding a new slider to the user interface, and implementing an slider event listener to call `setThickness()` on the `PaintObjectConstructor` using the slider's current value. **Minimal change**: Add 12 lines in PaintWindow.java and 1 in EraserPaint.java.

3.5 Interruptions

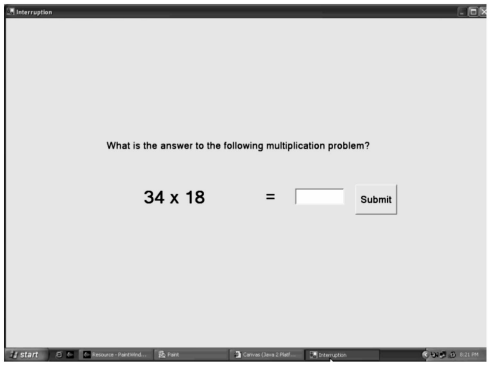
Interruptions came from a server on the experimenter's machine and appeared on the developer's machine as a flashing taskbar item with an audible alert, as shown in Fig. 2a. The interruptions were designed to require the developers' full attention, mimicking real interruptions such as requests from coworkers for help on unrelated projects [38]. Thus, when clicked, a full-screen dialog appeared with a 2-digit multiplication problem and a text box for the answer, as shown in Fig. 2b. Although the developers were told that they were not allowed to use any external resources to solve these problems, most found them so difficult that they used the text field to store intermediate results while the experimenter was not looking. The server sent interruptions every two and a half to three and a half minutes. The order of the multiplication questions was fixed and identical for all developers. Each question was unique and did not contain 0 digits.

3.6 Procedure

The developers worked alone in a lab and began by completing a survey on their programming expertise. They were then told that they would be given five user complaints and requests for an application and would have 70 minutes to complete as many as possible (the 70 minute limit was made to keep the full session under two hours). Developers were told they would be paid $10 for each request correctly completed. They were then told that a flashing taskbar item would occasionally interrupt them and that they should click it "when they were ready" and answer the arithmetic problem presented. The experimenter then explained that they would lose $2 for each interruption



(a)



(b)

Fig. 2. (a) The flashing taskbar notification and (b) one of the arithmetic interruption tasks.

TABLE 2
Developer Actions Transcribed from the Screen-Captured Videos and Examples of Events in the Videos that Indicated the Action had Occurred

Developer Action
Reading code, identified by text caret movement, mouse cursor hovering, text selection, and hovering of scroll bars in a fixed region.
Editing code, by typing, copy and pasting, refactoring, or quickfixes.
Navigating a static dependency, including any navigation from or to method declarations, method calls, class declarations, class references, and declarations, assignments, and uses of variables.
Navigating an indirect dependency, between code fragments that were indirectly related by two or more static dependencies.
Searching for text strings, within a file or the whole project.
Testing *Paint* by executing it from Eclipse.
Switching to documentation, either in the browser or inside of Eclipse.
Switching to Eclipse from *Paint*, web browser, interruption, etc.
Switching to a source file, and the Eclipse user interface used to do so.
Reading the task description, indicated by experimenter notes.
Starting a new task, identified by the task structure inferred.
Acknowledging an interruption by clicking on the taskbar item.
Introducing an error that later caused *Paint* to behave inappropriately.

ignored or answered incorrectly (this was used to give the interruptions some cost during the study, but was not actually enforced when developers were paid). Developers were then told that their work would be recorded with screen capturing software and were then given the user complaints and requests and asked to begin. Afterward, the experimenter tested the developers' solutions for correct behavior on the five tasks, paid the developers accordingly, and then answered any questions about the study.

4 Results

In this section, we provide both qualitative and quantitative evidence for a number of patterns, based on about 12 hours of screen-captured video across 10 developers' work. Our method for analyzing the videos involved two phases. In the first phase, we looked ahead in each developer's video to find what code they inspected and modified and what behaviors they tested. Because there were only five tasks, this was enough information to determine the task they were working on. Once we determined the task, we scanned backward in the video to find the moment when the developer began the task. This was obvious from pauses in activity after the developer tested the behavior they were modifying or implementing in their previous task. Once we had the sequence of tasks that a developer worked on, we then observed each task in detail, studying developers' individual actions, navigations, and choices, attempting to infer their high-level goals, and noting any interesting patterns regarding information seeking and management. While we did this, we also generated a list of the developer actions that we felt were important to understanding their behavior. These are listed in Table 2. Two of the authors performed all of these observations together over about 40 hours.

In the second phase, we used the actions in Table 2 to create a log of each developer's work to facilitate our analyses. The same two authors, on separate computers, stepped through the video, cooperatively creating a single log of each action, its start and stop time, and, if relevant, a description of the code that was operated on and the user interface that was used to perform the action (for example, static dependencies could be followed using the Eclipse *Open Declaration* command, using one of the commands in the *Java Search* dialog, or manually). In addition to the actions in Table 2, we also recorded our inferences about developers' questions and hypotheses, based on the information they investigated. To help us detect navigations of dependencies in the program, we enumerated the *Paint* application's static dependencies prior to transcription. Each 70 minute video took about 3 to 4 hours to transcribe, resulting in 2,870 actions (shown by task and developer in Table 3). During this process, there were never disagreements about whether a developer action had actually occurred, but there were many cases where one author missed an action that the other found. This synchronized logging allowed us to catch many actions that would have otherwise been omitted.

Once we had created transcripts for each developer, we set out to analyze the patterns that we had observed in the first phase of our analyses. As we discuss our results in this section, we will report per-developer averages for reasonably normal distributions, as *average* ($\pm$ *standard deviation*) and medians for other distributions. All time proportions that we report *exclude* any time spent on handling the interruptions, which accounted for an average of 22 percent ($\pm$ 6) of the developers' time.

TABLE 3
Task Completion Statistics for the 10 Developers, Including the Average Time Spent on Each Task and the Number of Actions per Task per Developer

Task	Time on Task	Actions per Task for each Developer and Success on each Task (✓ = success × = failure ■ = not attempted)										
		A	*B*	*C*	*D*	*E*	*F*	*G*	*H*	*I*	*J*	*Average*
SCROLL	17 (±13) minutes	91 ×	27 ×	50 ×	56 ×	181 ×	131 ×	6 ×	27 ×	13 ×	63 ✓	64.5 (±55)
YELLOW	10 (±8) minutes	12 ✓	94 ✓	30 ✓	124 ✓	25 ✓	42 ✓	25 ✓	49 ✓	36 ✓	57 ✓	49 (±35)
UNDO	6 (±5) minutes	13 ✓	5 ×	18 ✓	17 ✓	17 ✓	15 ✓	63 ✓	66 ✓	44 ✓	38 ✓	30 (±22)
LINE	22 (±12) minutes	84 ×	0 ■	90 ✓	54 ✓	63 ✓	0 ■	208 ×	50 ✓	72 ✓	49 ✓	67 (±58)
THICKNESS	17 (±8) minutes	71 ✓	150 ✓	64 ✓	70 ✓	52 ✓	101 ✓	66 ✓	103 ✓	38 ✓	50 ✓	77 (±33)

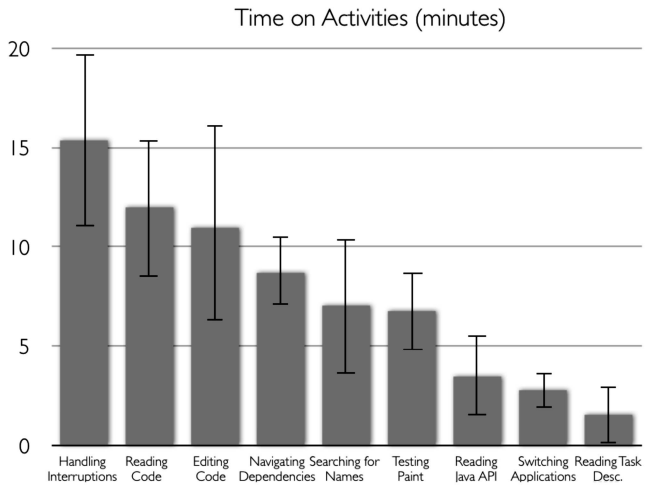


Fig. 3. Developers' division of labor in terms of time on activities. The vertical bars represent one standard deviation above and below the mean.

4.1 Division of Labor

Table 3 lists the number of developers attempting and completing each task and the average time spent on each. Developers finished an average of 3.4 ($\pm$ 0.8) tasks in 70 minutes. Almost everyone finished the **YELLOW**, **UNDO**, and **THICKNESS** tasks but spent most their time on the more difficult tasks, **SCROLL** and **LINE**. One developer completed all five correctly.

The bar chart shown in Fig. 3 portrays developers' average division of labor in terms of the actions in Table 2. Developers spent about a fifth of their non-interrupted time reading code, a fifth of their time editing code, a quarter of their time performing textual searches and navigating dependencies, and a tenth of their time testing the *Paint* application. An average of 5 percent ($\pm$ 2) of each developer's time was spent switching and reorienting between Eclipse, the Web browser, interruptions, and Paint. Of the 6 percent ($\pm$ 4) of time that was spent reading the Java APIs, nearly all of it was read in the context of the Javadoc documentation within the Web browser, despite evidence that each developer knew that documentation was accessible within Eclipse. In a few cases, developers used Google to search documentation and examples. Of course, each developer had a unique distribution of labor, as noted by the error bars. For example, some developers spent more time editing than others and correspondingly less time on other activities.

4.2 Task Structure

The activities in Fig. 3 were not independent: Before editing code, the developers had to determine what code to edit, and before determining this, they had to find it. Although all of these low-level actions were interleaved to some degree, our observations of developers' work indicated a higher-level sequence of choosing a task to work on, searching for task-relevant information, understanding the relationships between information, and editing, duplicating, and otherwise referencing the necessary code. Because developers' searches often failed and developers often inserted errors that had to be fixed, portions of this sequence were interleaved.

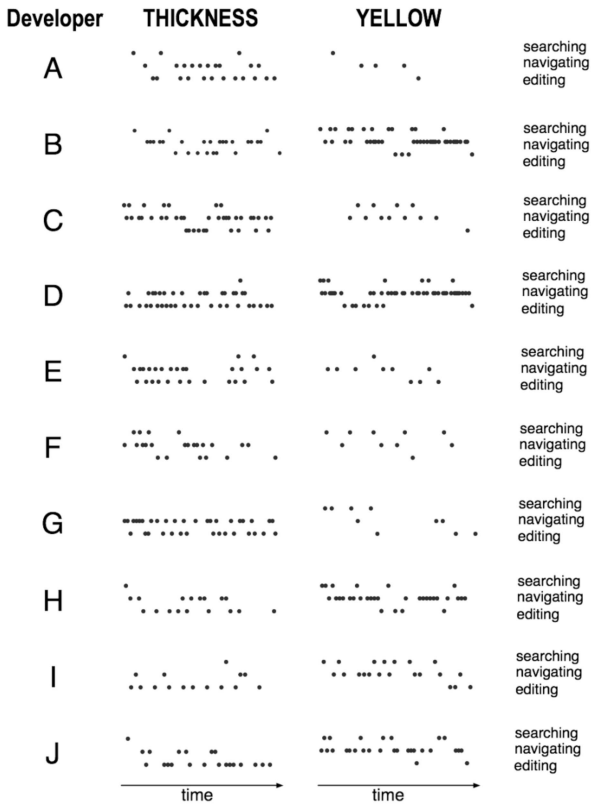


Fig. 4. The developers' actions for the **THICKNESS** and **YELLOW** tasks.

To attempt to illustrate this sequence with data, we grouped the actions listed in Table 2 into four categories: *searching*, *navigating*, *editing*, and *other*. In the first category were textual searches and reading task descriptions; in the second were static dependency navigations, switching between files, and reading API documentation to understand API dependencies; in the third were copying and editing code and indirect dependency navigations (as defined in Table 2), which occurred later in the task, once the developer had comprehended the necessary code. The remaining actions, such as testing and switching files, were categorized as *other*, since they were activities that seemed to occur throughout the developers' work. We then categorized each action from the **THICKNESS** and **YELLOW** tasks (allowing us to compare an enhancement task and debugging task that all developers completed).

Using these categorizations, we plotted each developer's action sequence, resulting in Fig. 4. The vertical axis is the category of action (we exclude the *other* category for clarity), and the horizontal axis is time, normalized between the start and end of work on the task. Our categorization was only an approximation: a textual search did not *always* indicate that the developer was looking for task relevant code, because in many instances, developers used textual search as a navigational tool to get from one place in a file to another. Within the plots, there were also several activities interleaved. For example, while determining where to add a declaration for a thickness slider, several developers also

inspected the nearby slider event handler. Once developers had implemented a solution in both of these tasks, they often had to return and correct errors, which involved further navigations and edits later in the task.

Despite the limitations of our categorization, the plots reveal several patterns. For example, there were few early edits for the **YELLOW** task, which was a debugging task. One explanation for this may be that there was little to edit on this task until the developer determined the cause of the problem, whereas for the **THICKNESS** task, there were several things developers could edit before having a complete solution. For example, when we inspected these early **THICKNESS** edits, they were all situations in which the developers inserted a declaration for a new thickness slider, and they knew to do this because they had already worked on the **YELLOW** task and knew that a slider declaration was necessary.

4.3 Searching for Task Relevant Information

For most tasks, developers began by searching: Of the 48 instances of a developer beginning work on a task, 40 began with a textual search for what developers perceived to be a task-relevant identifier in the code, either manually or using one of Eclipse's textual search tools. The remaining 8 began by browsing the files, methods, and fields in the Eclipse package explorer.

For the debugging tasks (**SCROLL**, **YELLOW**, and **UNDO**), developers used *symptoms* and *surface features* of the program's failure to guide their searches. For example, eight of the nine developers who attempted the **SCROLL** task first resized the *Paint* window and noticed that the canvas was only partially painted; thus, they searched for a method with the name "paint" in it, which always resulted in the `paintComponent()` method of the canvas, which was not responsible for the bug. An average of 88 percent ($\pm$ 11) of developers' searches led to nothing of later use in the task. These failed searches were at least partially responsible for the average of 25 ($\pm$ 9) minutes of their time (about 36 percent) spent inspecting irrelevant code. That no one identifier in the code could fully represent the code's purpose is generally called the *vocabulary problem* [21]. The cost of these incorrect guesses in the debugging tasks demonstrates how much the developers' early perceptions of relevance impacted their work.

When developers began the enhancement tasks (**LINE** and **THICKNESS**), their investigations of the source code were driven by a search for *extension points* in the code. For example, five of the developers began the **THICKNESS** task by searching for how the other sliders were implemented, and duplicating the code, three learned how to create an action object for the thickness slider, and two began by searching for how the stroke thickness might be set, investigating the *PaintObject* and *PaintCanvas* classes. Of the eight developers who attempted the **LINE** task, three began by inspecting how the pencil and eraser tools were implemented, eventually copying one of them, two began by investigating how the application created paint objects from the mouse events, two began by investigating the *Action* objects defined by for the pencil and eraser tools, and one began by investigating how to render lines.

4.4 Forming Perceptions of Relevance

The process that developers used to determine the relevance of code or information involved several levels of engagement with information and several types of cues to which developers attended in order to decide whether to continue comprehending some information. For example, a common progression in our observations was as follows: A developer would look at the name of a file in the package explorer in order to judge its relevance. If it looked interesting, he would hover over the file icon with the mouse and possibly select (but not open) the icon. At this point, if he thought the name seemed relevant, he double-clicked on the icon to open the file, or expanded the node in the package explorer in order to inspect its contents. Developers who expanded the explorer node hovered over the names of methods and fields, looking for relevant identifiers, whereas developers who opened the file tended to scroll through the file, skimming the code for identifiers that looked relevant or comments that might explain the intent of the file. If developers found a method or algorithm of interest, they would inspect it more closely, sometimes even selecting the text of the code repeatedly.

The user interfaces that Eclipse provided for summarizing code, such as the package explorer and search tools, determined the structure of these investigations. For example, Eclipse's package explorer allowed developers to consider file names and identifiers before looking at more detailed information. Had these interfaces not been available, developers would have had to open many more files and look at more information before finding what they believed to be relevant code. One problem with these summaries was that they were often misrepresentative of the content. The most glaring examples of this in our data involved misleading names. For example, when developers worked on the **YELLOW** task, half of them first inspected the `PencilPaint` class, but the file that was actually relevant was the generically named `PaintWindow`.

4.5 Types of Relevance

There were several types of relevant information. Developers found code to *edit* and returned to it after referencing other information. In the enhancement tasks, developers found code to *duplicate*, returning to it for reference after they had made changes to their copy of it. The developers also looked for code that helped them *understand* the intent and behavior of some other relevant code. For example, developers sought the documentation on the constructors of the `JSlider` class because they did not know how the various integer arguments would be interpreted. The developers spent time investigating helper classes, such as `PaintObjectConstructor`, to help them understand the purpose of other code they had to duplicate or modify. The developers also looked for code to *reference*, to help determine the appropriate design for some implementation. For example, when working on the **THICKNESS** task, all of the developers examined the way that the author of the *Paint* application had instantiated and added existing user interface components in order to guide their own implementation. Of course, there may be other types of relevance that we did not observe in our study.

TABLE 4
Types of Dependencies Navigated, the Average Percent of Each Type for a Developer, and the Tools that Developers Used to Perform Each

Dependency Type	% of All	Tools
Implicit dependencies	42% (±20)	*Find dialog, tabs*
Class's declaration	10% (±4)	*Open decl., package explorer, tabs*
Uses of a variable	10% (±5)	*Java search, find dialog*
Calls to a method	8% (±8)	*Java search, find dialog*
Variable's declaration	8% (±4)	*Open decl., find dialog*
Uses of var's new value	7% (±4)	*Find dialog*
Method's declaration	6% (±4)	*Open decl.*
Statement assigning var	5% (±5)	*Find dialog*
Uses of this class	4% (±3)	*Java search, find dialog*

4.6 Navigating Dependencies of Relevant Information

After reading a segment of code, developers explored the code's incoming and outgoing dependencies. During this exploration, developers generally followed the static relationships listed in Table 4. Overall, each developer navigated an average of 65 ($\pm$ 18) dependencies over their 70-minute sessions; these were inferred during our transcription process, since few of them used Eclipse's navigation commands. There were two types of dependency navigations that we transcribed. Some were *direct* dependencies that could be determined by static analyses, such as going from a variable's use to its declaration, or from a method's header to an invocation of the method. The other type of navigation was of *indirect* dependencies, such as going from a variable's use to the method that computed its most recent value. These were program elements that were indirectly related by two or more static dependencies. Developers tended to make these indirect navigations later in each task, as seen in the "editing" phases in Fig. 4. Developers' proportions of each type of dependency navigation are given in Table 4.

An average of 58 percent ($\pm$ 20) of developers' navigations were of *direct* dependencies. Though every developer used Eclipse's support for navigating these direct dependencies (the *Open Declaration* command and *Java Search* dialog) at least once, only two developers used the tools more than once, and only then for an average of 4 ($\pm$ 2) navigations. Instead, they used less sophisticated tools such as the *find and replace* dialog. There are several possible reasons why they chose to use these less accurate tools. Using the *Java Search* dialog required filling in many details and iterating through the search results. Then, in using both the *Java Search* and *Open Declaration* tools, new tabs were often opened, incurring the future cost of visually searching through and closing the new tabs if the files they represented did not contain relevant information. The developers used the *find and replace* dialog for an average of 8 ($\pm$ 6) of their navigations of direct relationships, and spent an average of 9 ($\pm$ 5) seconds iterating through matches before finding a relevant reference. Also, in the six cases of using the dialog, developers did not notice that "wrap search" was unchecked and were led to believe that the file had *no* occurrences of the string. One developer spent six minutes searching for a name elsewhere before finding that there were several uses in the original file.

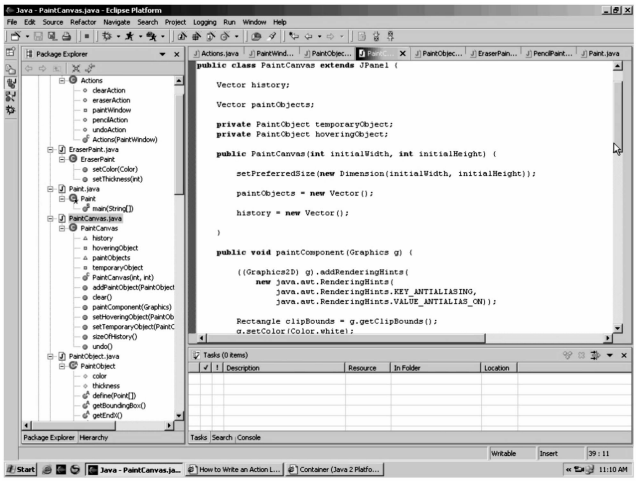


Fig. 5. The package explorer, file tabs, and scroll bars of Eclipse.

Many of the developers' direct navigations involved navigating between multiple code fragments. We inspected each developer's transcript and video and flagged each direct navigation that returned to a recently viewed location. Overall, an average of 27 percent ($\pm$ 13) of the developers' navigations of direct dependencies returned to code recently navigated from. When inspecting these returns in the videos, some were comparisons, in which developers went back and forth between related code multiple times. Of course, since all developers used a single editing window, developers had to navigate back and visually search for their previous location, costing an average of 9 ($\pm$ 7) seconds each time accumulating to 2 ($\pm$ 1) minutes per developer overall. Eclipse support for navigating back to the previous cursor position rarely helped, because developers rarely went directly back to the most recent location, but to some less recent location.

An average of 42 percent ($\pm$ 20) of the developers' navigations were of *indirect* dependencies (this proportion may be even higher, given the difficulty of detecting them in the videos). Because Eclipse's support for navigating direct dependencies was unhelpful for these, the developers used the scroll bars, page up and down keys, the package explorer and the file tabs instead. When navigating *within* a file using the scroll bars, scroll wheel, or page up and down keys, the developers had to perform visual searches for their targets, costing each developer, on average, a total of 10 ($\pm$ 4) minutes. Three developers avoided this overhead by using Eclipse's bookmarks to mark task-relevant code but then always ended up having more than two bookmarks to choose from and could not recall what code each one represented. This required clicking on each bookmark, which was no faster than their average scrolling time. Bookmarks also incurred the "cleanup" costs of their later removal when starting a new task. To navigate indirect relationships that were *between* files, the developers had to use the *package explorer* and the *file tabs*. When several tabs were open (as in Fig. 5), developers could not read the file

names because they were displayed in abbreviated form and many shared the common prefix of "Paint" in their name. If the package explorer had several expanded nodes (as in Fig. 5), the developers had to scroll to find their targets. Overall, this overhead cost each developer 5 ($\pm$ 1) minutes.

An average of 34 percent ($\pm$ 23) of developers' navigations of indirect relationships returned to a code fragment that was recently inspected. When investigating these navigations in the videos, nearly all seemed to be for the purpose of juxtaposing a set of code fragments while editing many dependent fragments. In each of these cases, the developers searched for an average of 10 seconds ($\pm$ 14) before finding their target, costing an average of about 2 ($\pm$ 1) minutes of visual search per developer. Although Eclipse supports viewing multiple files side by side, placing any more than two files side by side would have incurred the interactive overhead of horizontal scrolling within each of the views.

4.7 Representing Task Contexts

The developers kept track of relevant code and information in numerous ways. They used the package explorer and file tabs to keep track of *files* that contained relevant information. Each file's scroll bars and text caret helped temporarily mark the most recent relevant *segment* of code. Two of the 10 developers used bookmarks to mark a *line* of code. In some cases, the developers even used the undo stack in order to access earlier versions of code they had since modified. Outside of Eclipse, the developers also used the Windows taskbar to keep track of running applications, and the Web browser's scroll bars to mark relevant sections of documentation. Two of the developers used paper notes to write down important information, such as method names. These interfaces essentially "cached" the efforts of developers' prior navigations by keeping track of the relevant information they had found, helping a developer to collect a set of relevant information (their task context).

Although these interfaces were helpful, they were far from perfect. The scroll bars only helped developers remember the most recent relevant section of code in a file, and as soon as they moved it, the mark was lost. Five developers temporarily left the **LINE** and **SCROLL** tasks to work on easier tasks, but because part of their task context was represented by the open file tabs and the state of the package explorer (see Fig. 5), they often lost their task context when closing tabs or package explorer nodes to make space for information relevant to the new task. When developers returned to their earlier task, they spent an average of 60 seconds ($\pm$ 28) recovering their task contexts. Furthermore, tabs opened during previous tasks made it more difficult to find relevant tabs, because the tab names were truncated (as seen in Fig. 5). One problem with the package explorer was that developers often found a relevant method or field in a file, but to use the explorer to navigate to it, they had to keep the whole *file* expanded. For example, the developers used the explorer to navigate to `pencilAction` for reference during the **LINE** task (shown in Fig. 5), but in doing so, they also had to show all of the *irrelevant* code in `Action.java`.

4.8 Variations in Developers' Task Contexts

It was unlikely that the developers in the study would all find the same code relevant to a task, since each of the developers did the tasks in different orders and had different levels of experience with Java and the Swing API. This led to several questions:

- What information did *all* developers find relevant, and how did it relate to the code that was *actually* relevant to a task?
- How did developers' task contexts differ?
- How often did developers return to code they perceived as relevant?
- What granularity of information did developers deem relevant?

To answer these questions, we needed to know what code developers thought was relevant and what code they did not. Because we did not have this information —and developers may not have even explicitly formed this knowledge—we decided to approximate it by looking for developer actions that might indicate a developer's decision that some code or information was relevant. Because of evidence that there were several stages involved in forming perceptions of relevance, we chose to ignore the more preliminary decisions of relevance, such as opening a file or reading a method, and instead focus on more final indicators: 1) editing a code fragment, 2) navigating a dependency from a particular line of code, and 3) dwelling on API documentation found as a result of reading a particular identifier in a particular line of code. Although each of these indicators are not without uncertainty, they allowed us to approximate the set of code fragments that each developer may have thought relevant. Unfortunately, we do not know what developers would have actually chosen, and so we cannot assess the error in our approximation.

We used these three indicators to select a subset of actions from developers' **THICKNESS** and **LINE** tasks that suggested decisions of relevance. We chose these two tasks because they required the greatest amount of code to write and modify in order to successfully complete, but also because most developers finished them. By looking for indicators such as the text caret and mouse cursor movement and text selections in the video, we determined the source code lines or other information that the developer may have decided were relevant. In most cases, this information was just a single line, but others were sequences of lines and, more rarely, a whole method. After this analysis, we had approximations of each developers' task contexts for the **THICKNESS** and **LINE** tasks, and the sequence of their formation.

The resulting sets of relevant information are shown for all 10 developers in Table 5. The first column is the developer ID, and the second and fourth columns contain the number of relevant lines for the **THICKNESS** and **LINE** tasks, respectively, the files they were in (in order of decreasing number of relevant lines), and also the number of times relevant code was returned to. The third and fifth columns list the amount of time each developer spent on the

TABLE 5
An approximation of Developers' Task Contexts for **THICKNESS** and **LINE**, Derived from Edits, Dependency Navigations, and Searches for Text Strings

ID	Relevant Information for THICKNESS	Time	Relevant Information for LINE	Time
A	**36** lines from `PaintWindow`, `EraserPaint`, and `PencilPaint`, and the `JSlider` documentation. Returned to relevant code 14 times across 3 files.	17 min	(*failed*) **34** lines from `PencilPaint`, `PaintObjectConstructor`, `PaintWindow`, `PaintObject`, `PaintCanvas`, the `LinePaint` class created and documentation on `JRadioButton` and `Graphics`. Returned to relevant code 6 times across 3 files.	26 min
B	**50** lines from the `PaintWindow`, `PencilPaint`, `PaintObject`, and `EraserPaint`, the `JSlider` documentation, the `SliderDemo.java` example code, and JavaDocs on `ChangeListener`. Returned to relevant code 39 times across 3 files.	29 min	(*didn't attempt*)	-
C	**36** lines from `PaintWindow`, `PencilPaint`, `Actions`, and `PaintObjectConstructor`. Returned to relevant code 15 times across 2 files.	16 min	**13** lines from `Actions`, `PaintWindow`, `PaintObject`, and `PaintObjectConstructor`, and the `LinePaint` class created. Returned to relevant code 15 times across 3 files.	25 min
D	**41** lines from `PaintWindow`, `PaintObjectConstuctor`, and the new `LineThickness` class created. Returned to relevant code 23 times across 2 files.	11 min	**12** lines from `Actions` and `PaintWindow`, and the `LinePaint` class created. Returned to relevant code 11 times across 3 files.	16 min
E	**18** lines from `PaintWindow` and `EraserPaint`, and the `JSlider` documentation. Returned to relevant code 10 times across 1 file.	8 min	**16** lines from `PaintWindow`, `PaintObjectConstructor`, `PaintCanvas`, and `Actions`, and the `LinePaint` class created. Returned to relevant code 6 times across 2 files.	30 min
F	**27** lines from `PaintWindow`, `PaintCanvas`, and `PaintObjectConstructor`. Returned to relevant code 19 times across 1 file.	21 min	(*didn't attempt*)	-
G	**29** lines from `PaintWindow`, `PaintObjectConstructor`, and `EraserPaint`, and the `JSlider` documentation. Returned to relevant code 13 times across 3 files.	10 min	(*failed*) **19** lines from `Actions`, `PaintCanvas`, `PaintObjectController`, `PaintWindow`, `PaintObjectConstructor` and `PaintObject`; the `LinePaint` class created; and the `Rectangle2D.Double` and `Rectangle` classes. Returned to relevant code 32 times across 4 files.	47 min
H	**31** lines from `PaintWindow`, `EraserPaint`, `LinePaint` (created in a previous task), and `PaintObjectConstructor`. Returned to relevant code 16 times across 3 files.	35 min	**11** lines from `Actions`, `PaintWindow`, `PaintObject`, and the `LinePaint` class created. Returned to relevant code 3 times across 1 files.	12 min
I	**26** lines from `PaintWindow` and `EraserPaint`. Returned to relevant code 7 times across 1 file.	11 min	**13** lines from `Actions`, `PaintWindow`, and `PaintObjectConstructor`, and the `LinePaint` class created. Returned to relevant code 15 times across 3 files.	20 min
J	**33** lines from `PaintWindow` and `PaintObjectConstructor`. Returned to relevant code 22 times across 1 files.	11 min	**18** lines from `Actions`, `PaintWindow`, `PaintObjectConstructor`, `PaintObject`, and `PencilPaint`, and the `LinePaint` class created. Returned to relevant code 9 times across 3 files.	13 min

tasks. The minimum number of lines to successfully complete each task was 12 for **THICKNESS** and 15 for **LINE**.

What information did all developers find relevant, and how did it relate to the code that was actually relevant to a task? In general, successful developers' relevant information included the parts of the program that were part of the solutions described in Table 1. For example, everyone found similar segments of the `PaintWindow` class's constructor method relevant, because that was the place where the user interface was constructed and, thus, where the thickness slider would be added. Everyone who was successful at creating a line tool found the `Actions` class relevant, because that class had to be modified to include an action for the line tool radio button.

How did the developers' task contexts differ? One way was in how much information they deemed relevant. For the **THICKNESS** task, the developers deemed an average of 33 ($\pm$ 9) lines relevant, and for **LINE**, a median of 14 lines, not including the `LinePaint` class that each developer wrote, which was generally about 20 lines. For both of these tasks, this was about 7 percent of the 508 lines in the whole program. Note that this is less than the standard 40 lines visible in an Eclipse editor or other standard code editor, but in none of these editors is it possible to show these exact lines together in a single view. The developers also differed in the kind of information they found relevant. For example, many developers consulted documentation on the `JSlider` class, and many looked for examples on how to use the class. Other differences seemed due to strategy. For example, the developers differed on which lines of `PencilPaint` were relevant to the **THICKNESS** task because some noticed the `setThickness()` method, but those that did not changed the rendering algorithm instead. Other differences were due to prior understanding of the program; for example, some developers on the **THICKNESS** task only looked at the few files necessary to modify, possibly because they learned about other information in earlier tasks. Others indicated part of almost every file relevant, possibly because they needed or wanted to understand more about how the strokes were being generated and rendered.

How often did developers return to code they perceived as relevant? For the **THICKNESS** task, developers returned an average of 18 ($\pm$ 9) times to code that we transcribed as relevant, and for the **LINE** task, an average of 12 ($\pm$ 9) times.

Not only does the magnitude of these numbers give some validity to our measurements of perceived relevance, but it also reinforces our earlier findings of navigational bottlenecks in the Eclipse user interface (Section 4.6). There was no linear relationship between the time that a developer spent on a task and the number of relevant lines of code deemed relevant. There was a linear relationship between time on task and the number of returns on the **THICKNESS** task ($R^2 = .65$, $F = 17.8$, $p < .005$), but not for **LINE**. This difference may be explained by that fact that the most of the new code required for the **LINE** task was contained within a single method, unlike the **THICKNESS** task.

Although our method of deciding what code a developer deemed relevant biased the granularity of the relevant information, we can infer from the videos the higher level structures that developers may have thought relevant. Most of the developers' relevant information were single statements or pairs of statements; however, there were also several small subsections of the `PaintWindow` constructor that developers indicated as relevant. The developers indicated several whole methods as relevant, but these were generally getters, setters, and other simple methods. The developers rarely indicated the whole body of more complicated methods as relevant.

4.9 Impact of Interruptions

In order to understand the impact of interruptions, we analyzed all of the situations in which developers were interrupted and compared what the developers were doing before the interruptions to what they were doing after. Interruptions had an impact on the developers' work only when two conditions were true: 1) an important task state was not externalized into the environment at the time of acknowledging the interruption and 2) developers could not recall the state after returning from the interruption. The developers were very careful to externalize important task states before acknowledging the interruption. For example, in *every* case where a developer was interrupted while he was performing an edit, whether large or small, the developer *always* completed the edit before acknowledging the interruption. This was even the case when one developer had just copied the entire `PencilPaint` class in an effort to convert it into a new `LinePaint` class: Before acknowledging the interruption, he modified the class name, constructor name, commented out the old rendering algorithm, and wrote a comment about an implementation strategy for the new algorithm. In other cases where the task state was stored implicitly in Eclipse, developers forgot to externalize the state. For example, seven developers were interrupted just after repairing a syntax error but just before saving the changes. If they had saved, it would have caused Eclipse to incrementally compile and remove the syntax error feedback. Because they did not save, when they returned from interruptions, they often saw the underlined syntax error, and tried to repair the already valid code. In these seven cases, developers spent an average of 38 seconds before they realized that Eclipse's feedback about the syntax errors was out of date because they had not invoked the incremental compiler (more recent versions of Eclipse have rectified this problem).

5 Limitations

5.1 Measurement Error

Many of our findings were based on subjective interpretations of the developers' behaviors, so it is important to characterize the sources of error in our measurements and their impact on our findings. All of the data reported in this paper was based on our transcription of developers' actions. To assess the error rate in this transcription, we randomly sampled three task sequences from different developers and carefully compared them to the videos. This revealed three omitted dependency navigation actions and three transcription errors out of 108 actions. Therefore, one estimate of the error rate in our data would be about 5 percent. In general, these errors were not due to disagreements about whether an action had occurred or what type of action it was, but rather to the high level of difficulty of coding particular actions. For example, it was easy to identify application switching and code editing, but more difficult to identify dependency navigations.

This error rate affects a number of our results. The statistics in Fig. 3 would likely be impacted. For example, if we missed 5 percent of dependency navigations, there would probably be an average of 68 navigations per participant instead of 65. This would then impact the estimate of the time spent reading code, which was generally the default category when we observed no other actions. Our transcription error rate also impacts our proportion estimates of kinds of navigations (Table 4), and likely increases the number of navigations that were coded as returns (Section 4.6), which would increase the amount of time we estimated developers spent doing visual searches and scrolling. Because our transcription errors were omissions, most of the raw numbers we reported would simply increase, so our interpretations remain the same.

Another source of error are the time measurements, which were coded from time stamps in the videos, which were recorded by the second. Therefore, the time spent on each task could change slightly, and the durations that we reported would then have an error of ± 2 seconds, which could cause minor changes in our estimates. Despite this source of error, the error is likely to be distributed throughout our measurements, and so it likely impacts all of our data equally.

The estimates of code that developers deemed relevant in Section 4.8 are another potential source of error. These estimates were conservative in the sense that they were only based on explicit actions taken by the developers; we did not attempt to identify code that developers may have thought relevant but made no explicit action to indicate. Therefore, developers may have deemed many more lines relevant, but probably not fewer. Furthermore, relevance is not necessarily discrete; if asked, developers may just indicate a general area of a source file.

5.2 External Validity

Because this was a lab study, there are obvious limitations on our study's external validity. First, the size and complexity of the *Paint* program is not representative of most heavily maintained software systems. This may have

led us to observe work strategies that are dramatically different from those when developers face hundreds of thousands of lines of code, rather than hundreds. Our hypothesis is that the general strategies that developers used in our study would still be present in these situations, but particular activities, such as searching for relevant code and navigating dependencies, might require different tools and occur at different time scales (for example, developers would probably not use the package explorer to navigate among thousands of items). This is partially confirmed by some of the findings of Robillard et al.'s study of the medium-sized *JEdit* application [45], but it should be further investigated in larger systems. In addition to the program itself, the tasks that we designed could be called "quick fix" maintenance tasks, which may have a different character than other methods of software maintenance, such as agile methods or methods based on impact analysis. The tasks were also primarily GUI tasks and it is unknown whether developers' strategies differ for less visual programs. The lack of comments in the source code in our study are another limitation, especially given the importance of information cues suggested by our results. Had these cues existed, we may have seen more successful searches and less navigation.

The limited size of our sample and the limited experience of the developers in the sample both limit our study's generalizability. It may be the case that even within this population, there are variations in strategies that we did not observe because of our small sample. The developers with more experience in industry may be different because of their work context and depth of experience. The developers who work in teams may have different strategies for maintaining code that do not involve the sequence of high-level activities that we observed in our study. For example, it may be the case that, rather than searching to find relevant code, developers seek out a colleague they know to be more experienced for a particular aspect of the software and obtain hints about relevant code. Furthermore, because developers were unfamiliar with the code, our results may not generalize to collaborative situations in which developers are quite familiar with the code they are responsible for maintaining. Further studies of maintenance work are required to verify the generalizability of our findings in these contexts.

Our investigation only considered one programming language and one development environment. Some of our findings would obviously be different if other languages and environments were used. For example, the dependencies that developers navigated depended on the types of dependencies expressible in Java (although most widely used languages are quite similar). The user interfaces that developers used to represent their task context would likely differ in command-line environments. For example, perhaps developers who use command-line tools are better able to keep their task context in memory and are less reliant on their tools. Or perhaps they use different tools as memory aids, which have a different influence on the developers' work.

The time constraints we imposed were also somewhat artificial. For example, there may have been little incentive for developers to deeply investigate any problem on the Web or with a debugger because they may have felt pressured to complete *all* of the tasks in the 70 minutes. Furthermore, they may have felt unable to leave their work and focus on some other nondevelopment task, such as learning about the Swing API for future use; this may have impacted their problem solving efficacy, given evidence that time away from difficult problems can help people change their mindsets and conceive of new solutions [2]. The interruptions in the study were also artificial in several ways. No interruption was more or less *useful* to acknowledge than another; in reality, some interruptions are beneficial [22]. Furthermore, no interruption was more or less valuable socially; interruptions by friends and family may have caused developers to acknowledge interruptions without first externalizing important task state, possibly leading to errors. Finally, all of the interruptions took a similar amount of time; in the real world, some interruptions can be hours or days long.

Our limitation of the screen resolution to $1{,}024 \times 768$ could have been the source of many of the interactive bottlenecks that we observed in our data. It is possible that with a larger screen resolution, many of these effects would disappear or be lessened. However, while more space would leave more room for file tabs and result in fewer offscreen code fragments, this could have easily introduced issues with screen real estate *management*, replacing one interactive bottleneck with another.

6 Implications for Theory

We believe that our findings about developers' search strategies (Section 4.3), the importance of the user interface in perceptions of relevance (Section 4.4), the frequency with which developers returned to certain fragments of code (Section 4.6), and the variation in developers' task contexts (Section 4.8) call for a new model of program understanding. Our model describes program understanding as a process of *searching*, *relating*, and *collecting* relevant information, all by forming perceptions of relevance from cues in the programming environment.

To help explain this model, suppose we consider a program and its metadata such as comments and documentation as a graph consisting of individual fragments of information as nodes, and all possible relationships between information (*calls*, *uses*, *declares*, *defines*, etc.) as edges. In this representation, the code relevant to a particular task will be one of many possible subgraphs of this graph, with the particular subgraph for a developer depending on the implementation strategy, the developer's experience and expertise, and other factors. Using this representation, we can think of a developer's program understanding process as described in Fig. 6. A developer begins a task by looking for a node in the graph that seems relevant (*searching*). To do so, they use cues throughout the development environment, such as identifier names, comments, and documentation, to form perceptions about the relevance of information. Once a developer has found what is perceived to be a relevant node, the developer attempts to understand the node by relating it to dependent nodes (*relating*). Because each node in the graph could have a vast

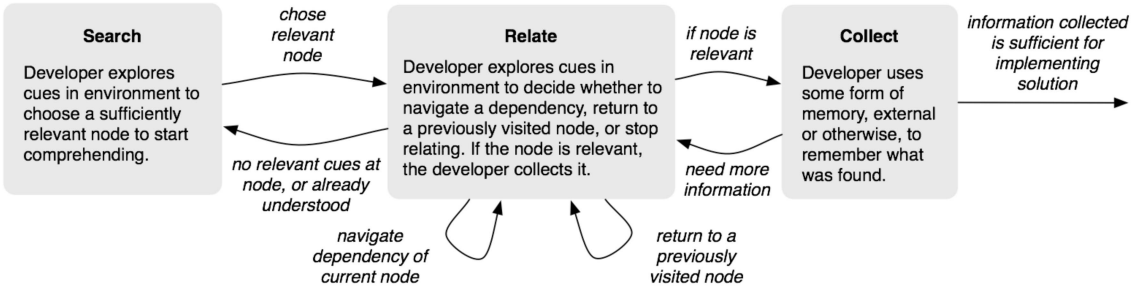


Fig. 6. A model of program understanding in which developers search for relevant information and relate it to other relevant information while collecting information necessary for eventually implementing a solution.

number of related nodes, the developer uses cues in the programming environment to determine which relationship seems most relevant. After choosing and navigating a relationship, the developer may investigate nodes related to the new node, and so on, or return to a previous node. If, at any point in this cycle of relating, the developer believes there are no more relevant cues, the developer drops out of the relating cycle and goes back to searching for a new node to comprehend. As this searching and relating continues, the developer gathers any nodes that seem necessary for completing the task, whether for editing, reference, or other purposes (*collecting*). If, at any point, the developer believes that the nodes that have been collected are sufficient to implement a solution for the task, the developer drops out of this understanding process altogether and focuses on the information collected to implement a solution. Problems during this implementation process then may lead to further search, relate, and collect activities.

Within this model, two factors are central to a developer's success: 1) the environment must provide clear and representative cues for the developer to judge the relevance of information, and 2) the environment must also provide a reliable way to collect the information the developer deems relevant. If an environment does not provide good cues, it may lead to fruitless investigations; if an environment does not provide an effective way to collect information, the developer will have to retrace his steps to locate information that has already been found.

Our model of program understanding is directly informed by *information foraging* theory [40], which posits that people adapt their strategies and environment to maximize gains of valuable information per unit cost. It proposes that a central mechanism of this adaptation is *information scent*: the imperfect "perception of the value, cost, or access path of information sources obtained from proximal cues." In general, these cues include artifacts such as hyperlinks on a Web page or graphical icons in a toolbar. In software development environments, they include the names of program elements, comments, the source file names, and so on. Information foraging theory may suggest more rigorous explanations of how developers might form their perceptions of relevance, so future work should further investigate its relationship to our model.

With regard to existing models of program understanding, our model is largely consistent with their predictions; the difference is that our model suggests a lower-level explanation of developers' actions than prior work. For example, many models have argued that developers begin with questions and form hypotheses [8], [27], [49]; this corresponds to the *searching* part of our model, in which developers ask "What is relevant?" and use cues to both form and test hypotheses about what is relevant. Other models have focused on high-level strategic differences, such as whether developers understand programs from the top down or bottom up [12], [32], [49], and whether they use systematic or as-needed strategies [29]; recent work on these issues tend to suggest that developers do all of these [45]. Under our model, a top-down strategy involves choosing a high-level node and following more specific dependencies; a bottom-up strategy is just the reverse. An as-needed strategy might involve many short paths through this graph, whereas a systematic strategy would likely involve longer and more consistent paths. Our model allows for all of these possibilities and predicts that the particular strategy chosen depends on the cues provided in the environment. Models of knowledge formation during program understanding [35], [51], which have suggested that a developer's mental model consists of relationships between code elements and the purpose and intent of these elements, are consistent with our description of knowledge as the combination of paths that a developer has traversed in a program over time and their existing knowledge. Finally, because our model describes a pattern of activity that is fundamentally driven by cues offered by the environment and the developers' perceptions of their relevance, it is also consistent with research on the influence of the visual representation of code on program understanding [3], [23], [33], [48].

7 Implications for Tools

While no single navigational problem in any of the developers' activities incurred dramatic overhead, overall, navigation was a significant component of developers' time. The total time developers spent recovering task contexts, iterating through search results, returning from navigations, and navigating between indirect dependencies within and between files was, on average, 19 minutes (35 percent of the time not spent answering interruptions). While much of this navigation was a necessary part of the developers' work, some of it was simply overhead, and, as we have seen, many of the navigations were repeated navigations that might have been avoided had more helpful tools been available. Although tools are only part of the complex

nature of software engineering work, it is worthwhile to discuss how they might impact developers' day-to-day efforts.

7.1 Helping Developers Search More Effectively

Much of the navigational overhead in our study was caused by the developers' use of inadequate or misrepresentative cues in the development environment to guide searches (Section 4.3). One approach to alleviating this is to provide better relevance cues. Some studies suggest that the most important information in understanding code is its *purpose* and *intent* [30]. For example, a method named `paintComponent` likely does something close to what it describes. Although our data suggests that names are an important way to indicate purpose, names can also be misleading, causing developers to form invalid perceptions of relevance. Comments are a common means of conveying intent and, perhaps, if written well and kept up to date, would be more indicative of purpose and intent. Imagine, for example, if the Eclipse package explorer annotated each file icon with a brief description of its purpose, extracted from Javadoc documentation. Another idea would be to annotate the icons with the number of other files using the code in the file to help a developer know how "important" the code is to a project, much like ranking is computed in search engines. Future work should investigate other types of information that correlate with the relevance of information.

Another approach is to provide more *layers* of cues before a developer has to read the code or information in full. For example, rather than having to double-click an icon representing a method in the package explorer in order to see its code, hovering over the icon might show its header comments or highlight all of the files in the project that use the file being hovered over. These extra layers would help developers decide that information was irrelevant earlier, without having to inspect the information in full.

Tools such as Hipikat have tried to *automatically* find potentially relevant code for developers to inspect [13]. However, because the recommendations are based on a developers' investigation activities or their current location in the code, when the developer is investigating irrelevant code, these tools may recommend more *irrelevant* code. Furthermore, the relevance cues in these recommendations can be misleading, since these systems present the names of program elements. Recommender systems such as these should be further studied in order to understand their impact on the developers' perceptions of relevance.

7.2 Helping Developers Relate Information More Efficiently

Once developers found relevant code, our observations suggest that they began navigating its dependencies, using relevance cues in the programming environment to decide whether to continue investigating and, if so, what dependency to navigate. One reason that developers did not use Eclipse navigation commands to perform these navigations is the overhead that they incurred by opening new tabs and requiring a return navigation. One way to reduce this overhead is to make dependency navigations more *provisional*. For example, nearly one-fifth of developers' time was spent reading code within a fixed view in the Eclipse editor (Section 4.1), so it could be helpful to highlight dependencies in the code automatically based on the current text caret position or text selection (Eclipse does have basic support for this, but it must be invoked). The developers also spent a lot of time going back and forth between related code (Section 4.6), so interaction techniques that allow developers to *glance* at related code could be helpful. Tools such as FEAT [42] might be a good place to start, by replacing the context menus used to inspect dependencies with something that requires fewer steps.

7.3 Helping Developers Collect Information More Reliably

Collecting information was central to the developers' success, but developers currently have a limited number of ways to do it. Each has its own flaws: Memorizing information tends to be unreliable [1]; writing it down is reliable, slow, imprecise, and requires developers to renavigate to relevant code; and, finally, using the interactive state in Eclipse was precise, but unreliable (Section 4.7). None of these approaches help developers compare information side by side, which our study suggests was quite important (Section 4.6).

The mockup in Fig. 7 illustrates one way that these fragments could be collected and viewed. In this hypothetical situation, the developer has already found all of the code he thinks is relevant, and he has just copied the `rSlider` setup code in order to create code to add a thickness slider. He is in the middle of changing `rSlider` to `tSlider` in the duplicated code. The basic concept of the workspace is to provide a single place for developers to view *all* relevant information for a single maintenance task side by side, in order to eliminate much of the navigational overhead that we observed in our study. In the rest of this section, we will describe the features portrayed in our conceptual workspace.

One issue with collection tools is how the workspace *refers* to code and metadata internally. For example, some tools have used a *virtual file*, which allows developers to combine a set of line ranges [41] or methods [9] and edit it as if it were a single file. Robillard proposed the concept of a *concern graph* [43], which represents information as a set of relationships between program elements (such as *declares* and *subclasses* relationships). This approach is more robust to changes to a program, but the tradeoff is that arbitrary subparts of the smallest granularity element (such as parts of a method) cannot be referenced (FEAT can reference the *calls* and *uses* of a method, but not arbitrary lines of code). The representation we envision in the workspace in Fig. 7 would involve *regions* of code that one could imagine a developer circling on a code printout; this approach might better match the granularity in which developers think about code.

The interaction technique that developers might use to *add* information to this workspace, once found, largely depends on the representation used to refer to the information (if information is collected as lines of code, for example, the interaction technique must allow developers to select lines). However, it must also be simple enough that developers can quickly specify the relevant

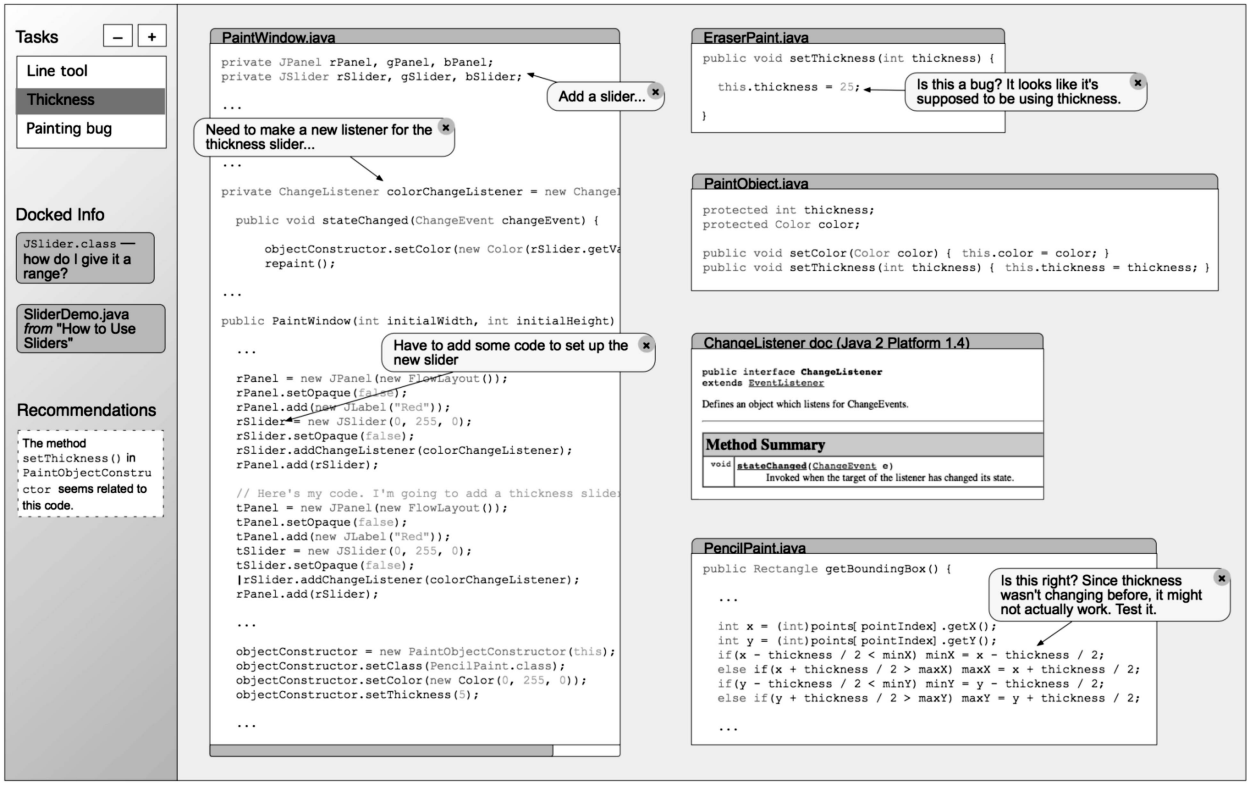


Fig. 7. The 50 lines of code and other information that developer B indicated as relevant, portrayed in a mockup of a workspace that help developers collect relevant information for a task in one place, independent of the structure of a program.

information and continue with their task, uninhibited. Tools proposed in the past have tended to be heavyweight. For example, Desert [41] requires users to know the line numbers beforehand and enter them manually. FEAT [42] requires users to navigate a program by its relationships and add "relations" to a concern graph through a contextual menu, which is a somewhat oblique way for a developer to say "this code is important to me." To add information to our workspace, developers could use a keyboard shortcut to add a single line, and possibly a gesturing technique with the mouse to circle the relevant code and its surrounding context. This would be quite similar to the snapshots of code that developers claim they see when trying to recall the shape and location of familiar code [39].

Given that nearly all of the navigational overhead we saw was due to the way code and information was organized on-screen, the *visual representation* of information in collection tools is also an important issue. Desert [41] presents relevant lines of code as a single integrated file; FEAT [42] represents concern graphs as a hierarchical tree and requires developers to select an element in this hierarchy, and use a pop-up menu to request the source file; Concern Highlight [36] highlights relevance code in a conventional editor. There are several problems with these approaches: 1) they treat all information as if it had the same role in the task, 2) they do not support side-by-side comparison of information, and 3) they incur much of the same interactive overhead we observed in relying on file tabs and scroll bars. Our proposal in Fig. 7 represents the code and information that a developer deems relevant concretely, rather than using abstract icons or names. Not only does this avoid the navigational overhead of navigating to the information, but it affords other advantages: Code can be placed side by side in order to aid comparison and editing, views can be collapsed to the bar on the left of Fig. 7 in order to allow developers to focus on the subset of information that is necessary for the current task, and code and other information can be directly annotated, as seen throughout Fig. 7.

Of course, there are some limitations to representing code concretely, rather than summarizing it and allowing users to navigate to it. One obvious concern is whether such a workspace would be able to fit *all* of a developer's relevant information on a single screen. To consider a lower bound, developers in our study found about 30 lines of code relevant on average. To consider an upper bound, a study of the CVS repository of GNOME, which is over a million lines of code, found that the average check-in was about 28 lines of code, with a standard deviation of 38 and a maximum of 237 [28]. This suggests that an average task, even with a few lines of surrounding context for contiguous fragment, would be likely to fit inside a full-screen window. We are now constructing such a tool [10] and we will be testing how well the tool scales.

8 Conclusions

In order to successfully complete a modification task, developers must locate and understand the parts of the software system relevant to the desired change. The exploratory study presented in this paper, which investigated

the strategies of 10 Java developers using Eclipse to perform five maintenance tasks, inspired a new model of this process that is based on searching, relating, and collecting, driven by the developers' perceptions of the relevance of information cues throughout a software development environment. This model is an extension of the more general theory of *information foraging* [40] but applied to software development. Our model is consistent with prior models of program understanding, while accounting for developers' actions at a level of detail that prior models have not. We also demonstrate how developers utilized Eclipse to complete their tasks, and we show a number of ways in which developers relied on valid but misleading information cues in Eclipse. These findings lead to a number of design ideas for more streamlined environments that may better help developers to find, relate, and collect task relevant code more quickly and effectively.

ACKNOWLEDGMENTS

The authors would like to thank Scott Hudson, James Fogarty, Elspeth Golden, Santosh Mathan, Karen Tang, and Duen Horng Chau for helping with the experiment design, execution, and analyses. The authors also thank the developers who participated in the study for their time and efforts, and the reviewers for their thorough and insightful comments about earlier drafts of this paper. This work was funded in part by the US National Science Foundation (NSF) under grant IIS-0329090 and part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. The first author was also supported by a US National Defense Science and Engineering Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the NSF.

REFERENCES

[1] E.M. Altmann, "Near-Term Memory in Programming: A Simulation-Based Analysis," *Int'l J. Human-Computer Studies,* vol. 54, pp. 189-210, 2001.

[2] J.R. Anderson, "Problem Solving," *Cognitive Psychology and Its Implications,* fifth ed., pp. 239-278. Worth, 2000.

[3] R.M. Baecker and A. Marcus, *Human Factors and Typography for More Readable Programs.* Addison-Wesley, 1990.

[4] E.L.A. Baniassad, G.C. Murphy, C. Schwanniger, and M. Kircher, "Managing Crosscutting Concerns during Software Evolution Tasks," *An Inquisitive Study, Aspect-Oriented Software Development,* pp. 120-126, Enschede, 2002.

[5] L.M. Berlin, "Beyond Program Understanding," *A Look at Programming Expertise in Industry, Empirical Studies of Programmers, Fifth Workshop,* pp. 6-25, 1993.

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

[7] D.A. Boehm-Davis, J.E. Fox, and B.H. Philips, "Techniques for Exploring Program Comprehension," *Empirical Studies of Programmers Conf. Report,* pp. 3-37, 1996.

[8] R. Brooks, "Towards a Theory of the Cognitive Processes in Computer Programming," *Int'l J. Human-Computer Studies,* vol. 51, pp. 197-211, 1999.

[9] M. Chu-Carroll, J. Wright, and D. Shields, "Supporting Aggregation in Fine Grained Software Configuration Management," *Proc. ACM SIGSOFT Int'l Symp. Foundations of Software Eng.,* pp. 99-108, 2002.

[10] M. Coblenz, *JASPER: Facilitating Software Maintenance Activities with Explicit Task Representations,* Carnegie Mellon Univ., Pittsburgh, Penn., CMU-HCII-06-107, 2006.

[11] C.L. Corritore and S. Wiedenbeck, "Mental Representations 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, 1999.

[12] 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, pp. 1-23, 2001.

[13] D. Cubranic and G. Murphy, "Hipikat: Recommending Pertinent Software Development Artifacts," *Proc. Int'l Conf. Software Eng.,* pp. 408-418, 2000.

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

[15] S.P. Davies, "Models and Theories of Programming Strategy," *Int'l J. Man-Machine Studies,* vol. 39, pp. 236-267, 1993.

[16] S.P. Davies, "Knowledge Restructuring and the Acquisition of Programming Expertise," *Int'l J. Human-Computer Studies,* vol. 40, no. 4, pp. 703-726, 1994.

[17] C. Douce, "Long Term Comprehension of Software Systems: A Methodology for Study," *Proc. Psychology of Programming Interest Group,* 2001.

[18] 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, Jan. 2001.

[19] J. Fogarty, A.J. Ko, H.H. Aung, E. Golden, K.P. Tang, and S.E. Hudson, "Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility," *Proc. ACM Conf. Human Factors in Computing Systems,* pp. 331-340, 2005.

[20] M.A. Francel and S. Rugaber, "The Value of Slicing while Debugging," *Science of Computer Programming,* vol. 40, nos. 2-3, pp. 151-169, 2001.

[21] G.W. Furnas, T.K. Landauer, L.M. Gomez, and S.T. Dumais, "The Vocabulary Problem in Human-System Communication," *Comm. ACM,* vol. 30, pp. 964-971, 1987.

[22] V.M. Gonzalez and G. Mark, "Constant, Constant, Multi-Tasking Craziness: Managing Multiple Working Spheres," *Proc. Conf. Human Factors in Computer Systems (CHI '04),* pp. 113-120, 2004.

[23] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Visual Languages and Computing,* vol. 7, pp. 131-174, 1996.

[24] A.J. Ko and B.A. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior," *Proc. Conf. Human Factors in Computing Systems,* pp. 151-158, 2004.

[25] A.J. Ko, H. Aung, and B.A. Myers, "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," *Proc. ACM Conf. Human Factors in Computing Systems,* pp. 1557-1560, 2005.

[26] A.J. Ko, H. Aung, and B.A. Myers, "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks," *Proc. Int'l Conf. Software Eng.,* pp. 126-135, 2005.

[27] A.J. Ko and B.A. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," *J. Visual Languages and Computing,* vol. 16, no. 1-2, pp. 41-84, 2005.

[28] S. Koch and G. Schneider, *Results from Software Engineering Research into Open Source Development Projects Using Public Data,* Wirtschaftsuniversität, p. 22, 2000.

[29] J. Koenemann and S.P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," *Proc. Conf. Human Factors and Computing Systems,* pp. 125-130, 1991.

[30] T. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," *Proc. Int'l Conf. Software Eng.,* pp. 492-501, 2006.

[31] M.M. Lehman and L. Belady, *Software Evolution—Processes of Software Change.* Academic, 1985.

[32] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance," *Proc. First Workshop Empirical Studies of Programmers,* pp. 80-98, 1986.

[33] J.R. Miara, J.A. Musselman, J.A. Navarro, and B. Shneiderman, "Program Indentation and Comprehensibility," *Comm. ACM,* vol. 26, no. 11, pp. 861-867, 1983.

[34] G.C. Murphy, M. Kersten, M.P. Robillard, and D. Cubranic, "The Emergent Structure of Development Tasks," *Proc. European Conf. Object-Oriented Programming,* pp. 34-48, 2005.
[35] R. Navarro-Prieto and J.J. Canas, "Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension," *Int'l J. Human-Computer Studies,* vol. 54, pp. 799-829, 2001.
[36] E.C. Nistor and A. van der Hoek, "Concern Highlight: A Tool for Concern Exploration and Visualization," *Proc. Workshop Linking Aspect Technology and Evolution,* 2006.
[37] N. Pennington, "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology,* vol. 19, pp. 295-341, 1987.
[38] L. Perlow, "The Time Famine: Toward a Sociology of Work Time," *Administrative Science Quarterly,* vol. 44, pp. 57-81, 1999.
[39] M. Petre and A.F. Blackwell, "A Glimpse of Expert Programmers' Mental Imagery," *Proc. Seventh Workshop Empirical Studies of Programmers,* pp. 109-128, 1997.
[40] P. Pirolli and S.K. Card, "Information Foraging," *Psychological Rev.,* vol. 106, no. 4, pp. 643-675, 1999.
[41] S.P. Reiss, "The Design of the Desert Software Development Environment," *Proc. Int'l Conf. Software Eng.,* pp. 398-407, 1996.
[42] M.P. Robillard and G.C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," *Proc. Int'l Conf. Software Eng.,* pp. 406-416, 2002.
[43] M.P. Robillard, "Representing Concerns in Source Code," PhD thesis, Dept. of Computer Science, Univ. of British Columbia 2003.
[44] M.P. Robillard and G.C. Murphy, "Automatically Inferring Concern Code from Program Investigation Activities," *Int'l Conf. Automated Software Eng.,* pp. 225-234, 2003.
[45] M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Trans. Software Eng.,* vol. 30, no. 12, pp. 889-903, Dec. 2004.
[46] M.P. Robillard, "Automatic Generation of Suggestions for Program Investigation," *Proc. ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 11-20, 2005.
[47] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," *Proc. Conf. Centre for Advanced Studies in Collaborative Research,* pp. 209-223, 1997.
[48] B.E. Teasley, "The Effects of Naming Style and Expertise on Program Comprehension," *Int'l J. Human-Computer Studies,* vol. 40, pp. 757-770, 1994.
[49] A. Vans and A. von Mayrhauser, "Program Understanding Behavior during Corrective Maintenance of Large-Scale Software," *Int'l J. Human-Computer Studies,* vol. 51, no. 1, pp. 31-70, 1999.
[50] M. Weiser, "Programmers Use Slices When Debugging," *Comm. ACM,* vol. 26, pp. 446-452, 1982.
[51] 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.

Amy J. Ko received the BS degrees in computer science and psychology from Oregon State University in 2002. She is currently a PhD student at the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University. Her research interests are in social and cognitive factors in software development.

Brad A. Myers received the PhD degree in computer science from the University of Toronto. He is a professor in the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University. His research interests include user interface development systems, user interfaces, handheld computers, programming by example, programming languages for kids, visual programming, interaction techniques, window management, and programming environments. He is an ACM fellow, a senior member of the IEEE and a member of the CHI Academy, the IEEE Computer Society, ACM SIGCHI, and Computer Professionals for Social Responsibility.

Michael J. Coblenz received the BS and MS degrees from Carnegie Mellon University in 2005 and 2006, respectively. He is interested in improving the user experience for both end-users and professional programmers and has developed the SLATE and JASPER systems to help users create software more quickly, easily, and reliably. He currently works as a software engineer in industry.

Htet Htet Aung received the MS degree in human computer interaction from Carnegie Mellon University in 2003. She was a research associate with the Pebbles Project at the Human Computer Interaction Institute of Carnegie Mellon University from 2003 to 2004. Later, she was a UI designer/usability Engineer at QuadraMed Corporation from 2005 to 2006. She is currently a senior product analyst at Siemens Medical Solutions. She is a member of ACM SIGCHI.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.