Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks

Amy J. Ko, Htet Htet Aung, and Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh PA, 15213
ajko@cmu.edu, hha@zizawah.com, bam+@cs.cmu.edu

ABSTRACT
Recently, several innovative tools have found their way into mainstream use in modern development environments. However, most of these tools have focused on creating and modifying code, despite evidence that most of programmers' time is spent understanding code as part of maintenance tasks. If new tools were designed to directly support these maintenance tasks, what types would be most helpful? To find out, a study of expert Java programmers using Eclipse was performed. The study suggests that maintenance work consists of three activities: (1) forming a working set of task-relevant code fragments; (2) navigating the dependencies within this working set; and (3) repairing or creating the necessary code. The study identified several trends in these activities, as well as many opportunities for new tools that could save programmers up to 35% of the time they currently spend on maintenance tasks.

Categories and Subject Descriptors
D.2.6 [**Programming Environments**]: Integrated environments.

General Terms
Design, Human Factors.

1. INTRODUCTION
In past decades, it has become increasingly clear that most software undergoes a brief period of rapid development, followed by a much longer period of maintenance, added features, and adaptation to new contexts of use [4]. Thus, an important challenge in software engineering research is to create new and more useful tools for understanding and reshaping software as its requirements change.

Several new tools have been widely adopted, but they typically focused only on the *creation* of code. In particular, Java developers have quickly adopted Eclipse, an integrated development environment (IDE), for its incremental compiling, refactoring support, and "quick fixes" for common errors. This is despite evidence that 60-90% of software development costs involve the *reading* and *navigation* of code as part of programmers' maintenance tasks [8]. While there have been extensive efforts to study these activities, few have assessed the impact of modern IDEs on these tasks.

In this paper, we describe a study of expert Java programmers using the Eclipse IDE to work on five maintenance tasks. Our goal was to discover fundamental activities in maintenance work and use this understanding to elicit design requirements for new tools to support maintenance tasks. The results of our study suggest that maintenance work interleaves three fundamental activities: (1) collecting a group of task-relevant code fragments, which we call a *working set*; (2) navigating these code fragments' dependencies (such as *uses* and *declares* relationships); and (3) repairing or creating the necessary code. We identify many trends, including that programmers spent an average of 35% of their time simply *navigating* between dependencies, and an average of 46% of their time inspecting task-irrelevant code. These observations and many others motivate the design of several new tools.

In the next section, we briefly review prior research on maintenance tasks. In Section 3 we describe the design of our study. In Sections 4 and 5, we describe qualitative and empirical assessments of our data, respectively. We end in Section 6 with a set of design recommendations for maintenance-oriented tools, and a conceptual sketch of a new IDE under development.

2. RELATED WORK
Maintenance work has been studied from many perspectives. For example, many theories of software comprehension include the notion that understanding results in part by the recognition of "beacons," or recurring patterns of code [3, 6]. Corritore and Widenbeck studied the *direction* of programmers' comprehension strategies, finding that object-oriented programmers tend to start top-down, but use an increasingly bottom-up approach as they work [5]. Teasley studied the effects of *naming-style* [17], finding that poorly named program elements can affect novice comprehension, but have little impact on expert comprehension. Green compared the impact of textual and visual languages, finding that visual languages better facilitate the understanding of dataflow, but incur more interactive overhead when editing [11].

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

One limitation of all of these studies is that they have typically had programmers study a program for some (often fixed) period of time and only then have them work on a maintenance task. This is despite evidence that programmers interleave reading and modifying under more realistic conditions [7]. To our knowledge, there have been no studies of the influence of IDEs on maintenance tasks without such artificial constraints.

In addition to studies, there have been several tools designed to support software maintenance. For example, Antoniol et al. describe a system for detecting design patterns [1], which may better help programmers understand the software architecture of large systems. Other maintenance tools, such as the one described by Beyer et al., support queries on specific structures in programs [2]. Müller describes a class of reverse-engineering tools [14] that are designed to help understand legacy systems. While all of these tools have proven beneficial for forming a holistic understanding of software architectures, to our knowledge, there are no tools that directly support the actual *work* of maintaining code, other than the tools offered by existing IDEs.

3. THE ECLIPSE STUDY

The goal of our study was to discover fundamental activities in maintenance work in order to inspire new ideas for more helpful tools. To this end, we employed a methodology [12] that involves recording every detail of programmers' work in full screen-captured videos. The study required programmers to complete five maintenance tasks over a 70-minute period while responding to intermittent interruptions. We included interruptions to reduce the study's artificiality: there is considerable evidence that interruptions are frequent in software engineering workplaces [10, 15] and we wanted to see if IDEs could help handle them.

Our decision to study programmers in the lab instead of "in the large" was driven by our need to compare programmers' strategies. Had we studied programmers working on different code, as would be the case in industry, we would not know if differences in programmers' work were due to variations in strategy or in code. Because this was a lab study, programmers worked alone, had no long-term deadlines, and were motivated only by the monetary incentive that we provided. This places obvious limitations on the study's generality. We will address these issues throughout this section and at the end of this paper.

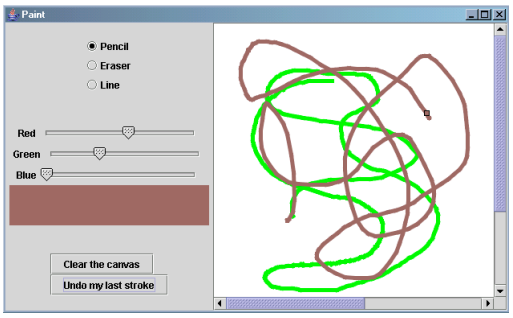

Figure 1. *Paint*, debugged and improved over 70 minutes.

3.1 The Participants

We recruited 10 programmers. In a pre-test survey, 7 claimed to be "Java experts" and 3 claimed "above-average" Java expertise. All claimed to use either Eclipse or Visual Studio "regularly," and reported programming a mean of 24 (±20) hours a week. The group was all male with a mean age of 23 (±3) years, and was comprised of 6 senior and 2 doctoral computer science students, 1 MS in computer engineering, and 1 MS in information systems.

3.2 The *Paint* Program

The *Paint* program (shown in Figure 1) was a 503-line Java Swing application with 9 Java classes (in 9 files), which allowed users to draw, erase, clear and undo colored strokes on a white canvas. It was based on the concept of a *PaintObjectConstructor*, which created strokes from mouse locations accumulated between mouse down and up events. Because participants were unfamiliar with the code, our results may not generalize to situations in which programmers are closely familiar with the code they maintain. It is also possible that some of our observations are a consequence of the program's size, rather than aspects of maintenance work. Finally, because the code was written without time-constraints, the quality of its design may not be representative of code that is maintained in industry.

3.3 Tasks and Tools

Participants were given the user complaints and requests described in 2nd and 3rd columns in Table 1 and 70 minutes to complete as many of the tasks as they could, in any order.

Table 1. The five maintenance tasks and their solutions. Participants were not shown the solutions or task names.

Task Name	Complaint or Request	Task	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 inside of the scroll pane was not being updated as strokes were created, preventing the scroll bars from appearing when painting outside the window. This also caused the scroll pane to only repaint a fixed region.
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 used twice in the *colorChangeListener*, but the blue slider's not at all.
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 that caused a repaint.
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.	A simple solution involved copying the *PencilPaint* class and revising its paint algorithm to draw a single line between the first and most recent points.
THICKNESS	Users requested control over the stroke thickness of the pencil, eraser, and line tools.	Create a thickness slider with values from values 1 to 50, which controls the thickness of the stroke for all tools.	A simple solution involved copying the initialization and event listener code for one of the color sliders, and calling *setThickness*() instead of *setColor*().

Interruptions came from a server on the experimenter's machine and appeared on the participant's machine as a flashing taskbar item with an audible alert. The interruptions were designed to require programmers' full attention, mimicking real interruptions [15] such as requests from coworkers for help on unrelated projects. Thus, when clicked, a full-screen dialog appeared with a 2-digit multiplication question and text box for the answer. The server sent interruptions randomly, approximately every three minutes. The order of interruptions was fixed and identical. Each question was unique and did not contain 1 and 0 digits.

Participants were given the Eclipse 2.1.2 IDE (released March 2004) and a project with the 9 source files. Participants were allowed to use any resource they desired, including the Internet. The browser's default page was the Java 1.4 API documentation. Participants used a PC and 17" 1024 x 768 LCD. Screen capture videos were recorded at 12 frames per second in 24-bit color and had no discernable impact on the PC's performance.

3.4 The Procedure

Participants worked individually in a private lab. Participants completed a survey on their programming expertise and were then told that they would be given five user requests for an application and would have 70 minutes to complete them. Participants were told they would be paid $10 for each request completed. Participants were then told that a flashing taskbar item would occasionally interrupt them and that they should click it as soon as possible and answer the arithmetic problem presented. Participants were told they would lose $2 for each problem ignored or answered incorrectly. This was used to give the interruptions some cost, but was not actually enforced. Participants were told that their work would be recorded with screen capturing software. Participants were then given the user complaints and requests and told to address as many as possible in the next 70 minutes. Afterwards, the experimenter evaluated the participants' code and paid participants accordingly.

4. A QUALITATIVE ASSESSMENT

Our resulting data set consisted of approximately 12 hours of screen-captured video. The first phase of our analysis focused on uncovering fundamental activities in programmers' work. Our examination of the videos involved several steps:

1. Looking ahead in the video to determine which task the programmer was working on.
2. Returning to the beginning of the task to determine the goal of each of the programmer's individual actions.
3. Generalizing from the goals of programmers' individual actions to more general activities.

The first two authors performed these analyses together over about 40 hours, finding several trends in programmers' work. We give an overview of these trends in this section, and discuss them with empirical evidence in Section 5.

The most apparent trend in programmers' work was their interleaving of three activities: (1) collecting a small *working set* of task-relevant code fragments; (2) navigating dependencies within the working set; and (3) editing and creating the code necessary to complete the maintenance task. The dark gray regions in Figure 2 portrays one such working set.

We noticed many patterns in programmers' formation of working sets. For example, when starting a task, they tended to ask one of two questions:

1. *How does X work?* Programmers asked this when they wanted to integrate new and existing code (as in **LINE** and **THICKNESS** in Table 1).
2. *Why did(n't) X happen?* These were asked when programmers sought the cause of Paint's output (as in **SCROLL**, **YELLOW**, and **UNDO**).

Following these questions, programmers began a process of forming their working set. Because programmers started with no knowledge of *Paint*'s implementation, they were initially biased by what it *seemed* to be doing at runtime. Once they were more familiar with the code, they were biased by seemingly task-relevant names of files, methods, and variables.

Once programmers found task-relevant code, they tended to explore the code's dependencies. During this exploration, programmers looked for answers to questions such as "*What defines this variable's value*?" and "*What uses this variable's definition*?" Programmers' efficiency in answering these questions seemed to be influenced by which tools they used. Some tools seemed to slow programmers' progress by imposing significant interactive overhead (e.g., extra clicking and visual searches), while others provided hidden or inconsistent feedback that caused programmers to overlook important code.

Several things impeded the creation and repair of code. For example, Eclipse often provided hidden or inconsistent error highlighting, causing programmers to act on misinformation. Programmers also made errors when duplicating code via copy and paste, sometimes leaving copied references unchanged, or only copying part of a pattern of code.

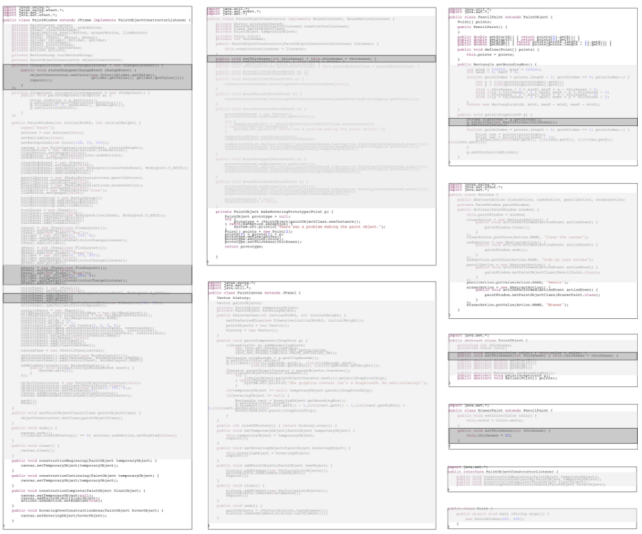

Figure 2. The 503 lines of the *Paint* program. Each box represents a single Java class file. The white regions portray the code that one programmer scrolled through, but did not stop to read, while working on the THICKNESS task. The light gray regions portray the code that he navigated to and read. The dark gray regions portray the code that he included in his working set, as indicated by his frequent navigation of dependencies between these regions.

Table 2. The types of events transcribed from the screen-captured videos of programmers' work and what details were recorded for each.

Reading a segment of code, identified by the movement of the text caret through a code fragment or the hovering of the mouse over code.
Editing code, and the tool that was used (keyboard, copy and paste, refactoring, quick fixes, etc.).
Navigating between dependencies, the tool that was used to perform the navigation, and whether the navigation returned to code recently navigated from.
Searching for names, and the Eclipse tool that was used.
Testing *Paint*. The duration of testing was recorded from the time of execution to the time of returning to Eclipse.
Reading the Java API documentation and whether Eclipse or a web browser was used to view it.
Switching environments (between Eclipse, *Paint* and the web browser).
Reading the task descriptions.

5. AN EMPIRICAL ASSESSMENT

In order to empirically assess the trends described in the previous section, the first two authors transcribed the events listed in Table 2 from participants' videos, along with the start and stop times for each. They also recorded switches between the 5 maintenance tasks, situations in which programmers asked *how* and *why* questions, and any errors that programmers introduced. The first two authors enumerated *Paint*'s static and dynamic dependencies prior to transcription so that they could detect navigations of these dependencies in the videos. The first two authors transcribed each participant's video together, taking about 3 hours per programmer.

In this section, we provide empirical evidence for the trends described in the previous section. We report per programmer averages as *average* (± *standard deviation*) (space prohibits a thorough discussion of any skew, and/or floor or ceiling effects in this distributions). All time proportions we report *exclude* any time spent on handling interruptions, which accounted for an average of 22% (±6) of programmers' time.

5.1 Division of Labor

Table 3 lists the number of programmers completing each task and the average time spent on each task (independent of whether the task was completed). Overall, each programmer finished an average of 3.5 (±0.8) tasks in 70 minutes. The pie chart shown in Figure 3 portrays programmers' average division of labor in terms of the events in Table 2 (read counter-clockwise).

Table 3. The number of programmers completing each task and the average time spent on each.

Task	# of Programmers that Completing the Task	Average Time Spent on Task
SCROLL	1 of 10	17 (±13) min.
YELLOW	10 of 10	10 (±8) min.
UNDO	9 of 10	6 (±5) min.
LINE	6 of 10	22 (±12) min.
THICKNESS	10 of 10	17 (±8) min.

5.2 Forming a Working Set

Based on our qualitative assessments, the programmers' central goal for each maintenance task was to collect a *working set* of task-relevant code fragments. We considered any code fragment that a programmer modified or followed static or dynamic dependencies on to be part of the programmer's working set for a task. These code fragments were individual statements or expressions, lists of statements, or complete methods, and consisted of an average of 18 (±11) lines of Java code per task.

Programmers began each task by searching for task-relevant code. Of the 48 instances of a programmer beginning work on a task, 40 began with a search for a task-relevant *name* of a program element. For example, when beginning the **LINE** task, programmers often searched for the text string "line," "drag," or "pencil." Among these searches, only $^1/_2$ led to task-relevant code; in the other instances, the name led to task irrelevant code. Programmers used Eclipse's *find and replace* dialog for only 6 of these searches; in all other instances, they scrolled through the code, visually searching for the task-relevant names. In the 8 instances where programmers did not search for a name, they navigated to code that they were familiar with from previous tasks; only 2 such navigations led to relevant code.

Surface features of *Paint*'s output influenced programmers' choice of names to search for or familiar code to inspect. For example, 8 of the 9 programmers who attempted the **SCROLL** task first resized the *Paint* window and noticed that the canvas was only partially painted; thus, they decided to find a method name with the name "paint" in it, which typically led them to the *paintComponent*() method of the canvas, which was irrelevant. Overall, an average of 88% (±11) of programmers' hypotheses about the cause of runtime failures were false, causing each programmer to spend an average of 25 (±9) minutes of their time inspecting task-*irrelevant* code. This data is comparable to our prior studies of false hypotheses in novice programming [13].

Five programmers temporarily abandoned the more difficult tasks (**LINE** and **SCROLL**) to work on easier tasks. Because part of their working sets were represented by the open file tabs (as in Figure 4) and the state of the package explorer (as in Figure 5), they often lost their working sets when closing tabs or package explorer nodes. When programmers returned to these tasks, they spent an average of 60 seconds (±28) recovering their working sets.

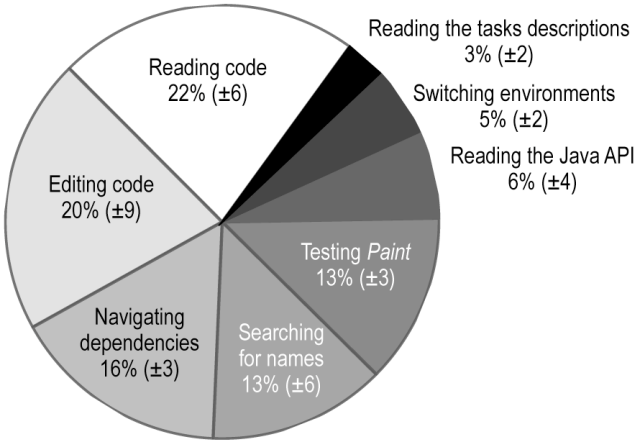

Figure 3. Programmers' division of labor (excluding interruptions) in terms of the events in listed in Table 2.

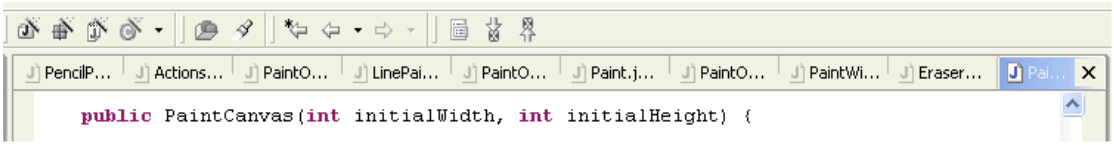

Figure 4. The file tabs, which represented part of a programmer's working set of task-relevant code. Because file names were truncated and many had identical prefixes, programmers spent considerable time searching through the tabs for a particular file.

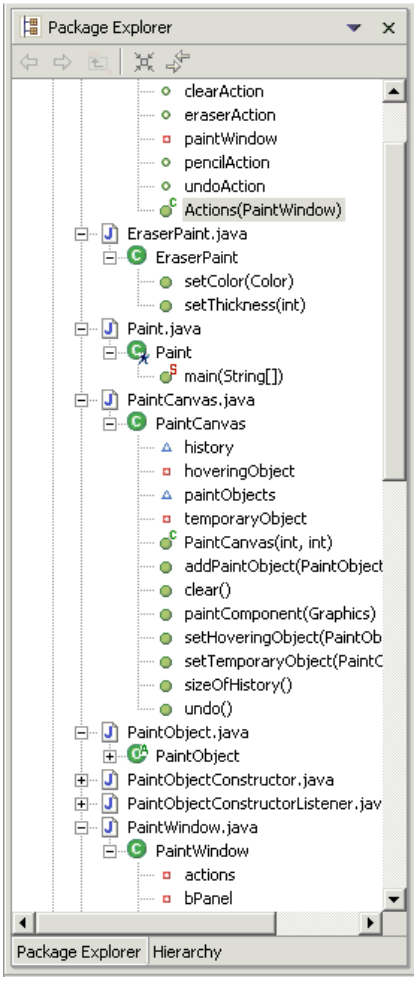

Figure 5. The package explorer, which represented part of programmers' working sets.

Table 4. Types of dependencies that programmers navigated, the percent of each type among all programmers' navigations, and the Eclipse tool that directly supported the navigation.

Type of Dependency	% of All	Eclipse Tool
Indirect	42% (±20)	-
This class's declaration	10% (±4)	*Open declaration*
Uses of this variable	10% (±5)	*Java search*
Calls to this method	8% (±8)	*Java search*
This variable's type	8% (±4)	*Open declaration*
Uses of this variable's new value	7% (±4)	-
This method's declaration	6% (±4)	*Open declaration*
Statement that set this variable	5% (±5)	-
Uses of this class	4% (±3)	*Java search*

5.3 Navigating the Working Set

Each programmer navigated an average of 65 (±18) dependencies over their 70-minutes. A close inspection of these navigations revealed two types. Some were of *direct* dependencies, such as going from a variable's use to its declaration, or from a method's signature to a call on 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. Programmers tended to make these indirect navigations once they understood the intermediate direct dependencies. The proportions of each type of navigation are given in Table 4.

5.3.1 Navigations of Direct Dependencies

An average of 58% (±20) of programmers' navigations were of *direct* dependencies. Though every programmer used Eclipse's support for navigating these direct dependencies at least once (when available), only 2 programmers used the tools more than once, and only then for an average of 4 (±2) navigations. Instead, they used less sophisticated tools such as the *find and replace* dialog. There are several reasons why they may have preferred other tools. For example, programmers had to set up the *Java Search* dialog and then iterate through its results. Then, in using both the *Java Search* and *Open Declaration* tools, new tabs were often opened, incurring the future cost of searching through and eventually closing the new tabs if the files they represented were not relevant.

Programmers used the *find and replace* dialog for an average of 8 (±6) of their navigations of direct relationships. Programmers spent an average of 9 (±5) seconds iterating through matches before finding a relevant reference and frequently had to reposition the dialog to uncover concealed code. Also, in 5 instances of using the dialog, programmers did not notice that "wrap search" was unchecked, and as a result, were led to believe that the file had *no* occurrences of the string. One programmer spent as much as 6 minutes searching for a name elsewhere before discovering that there were in fact several uses in the original file.

Overall, an average of 27% (±13) of programmers' navigations of direct dependencies returned to code just navigated from. This suggests that over half of programmers' navigations of direct relationships were part of "glances" (there and back). Programmers searched for an average of 9 (±7) seconds before finding their previous location, costing an average of over 2 (±1) minutes per programmer over all of their direct navigations.

5.3.2 Navigations of Indirect Dependencies

An average of 42% (±20) of programmers' navigations were of *indirect* dependencies. Because Eclipse's support for navigating direct dependencies was unhelpful for these navigations, programmers had to use 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 or page up and down keys, programmers had to perform lengthy visual searches for their targets, costing each programmer, on average, a total of 10 (±4) minutes. Three programmers tried to avoid 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.

Programmers had to use the *package explorer* and the *file tabs* to navigate indirect relationships that were *between* files. When several tabs were open (as in Figure 4), programmers could not read the file names. If the package explorer had several expanded nodes (as in Figure 5), programmers had to scroll to find their targets. Overall, this overhead cost each programmer an average of 5 (±1) minutes.

An average of 34% (±23) of programmers' navigations of indirect relationships returned to a code fragment that was recently inspected. Programmers were likely performing these navigations in order to juxtapose and compare a *set* of code fragments. In each of these navigations, programmers searched for an average of 10 seconds (±14) before finding their target, costing an average of about 2 (±1) minutes per programmer overall. 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 since so little code would have been visible.

5.4 Hidden and Inconsistent Feedback

After programmers sufficiently understood their working set of code, they began to create or repair the code necessary to complete their task. They faced several obstacles in obtaining reliable feedback from Eclipse in the process.

Many obstacles were simple compiler errors. Eclipse is different from other modern IDEs because it incrementally compiles code while it is being edited, allowing more immediate feedback about compiler errors. While this was frequently helpful for identifying common errors, there were 24 instances across the 10 programmers' sessions where Eclipse marked valid syntax as invalid. For example, when programmers forgot syntactic delimiters such as semi-colons or curly braces, Eclipse marked the valid syntax just *after* the missing delimiter as incorrect (as in Figure 6). In these situations, programmers knew that the highlighted code was valid, but they had to spend time searching for the invalid code. Also, because Eclipse's incremental compiler was often invoked only after a file was save, code that programmers thought they had repaired, and in fact did repair, often remained marked as invalid (as in Figure 7). If programmers were interrupted before they had saved, they often returned from interruptions, not realizing they had not saved, and tried to repair their already valid code. Overall, each instance where Eclipse inconsistently marked code cost an average of 38 seconds (and at least 10 seconds) before programmers realized the inconsistency.

Three programmers overlooked the off-screen error in Figure 8 an average of 3 times before noticing it. This cost each of these programmers an average of 6 (±1) minutes of unnecessary debugging. In many situations, programmers quickly scrolled to right to glance at the code off-screen, and quickly scrolled back.

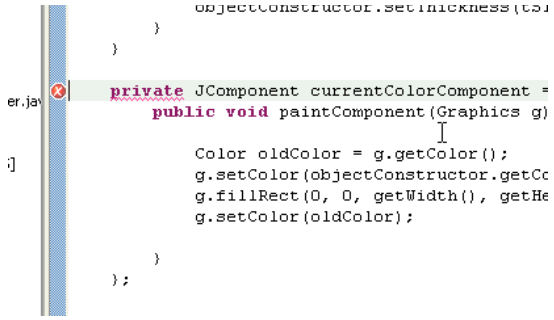

Figure 6. Eclipse frequently marked syntactically valid code as invalid because of syntax errors above.

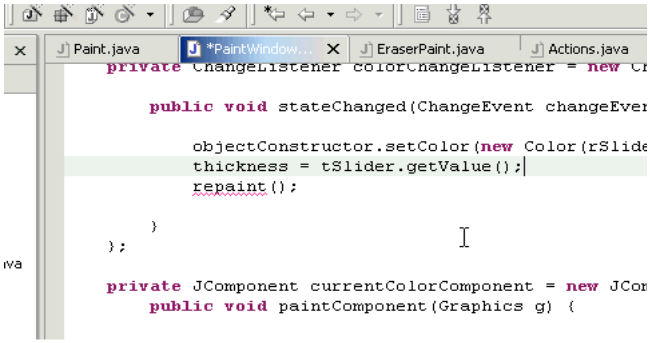

Figure 7. Eclipse frequently marked valid code as invalid because programmers had not invoked the incremental compiler by saving the file. As a result, programmers spent time trying to repair valid syntax.

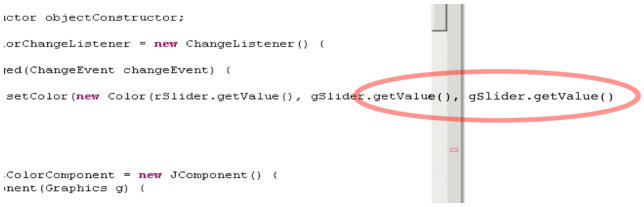

Figure 8. The off-screen duplication of a reference to *gSlider* (the solution to the YELLOW task), which was frequently overlooked because programmers only glanced at it.

Figure 9. Two copy and paste errors caused by an interruption. At (1), *tSlider*'s listener should be changed from *colorChangeListener* to *thicknessChangeListener*. In (2), the programmer neglected to add *tPanel* to the *colorPanel*.

5.5 Copy and Paste Errors

Every programmer used copy and paste for the **LINE** and **THICKNESS** tasks to copy code and then modify it to perform a similar function. Programmers copied code an average of 4 (±3) times during their work. This behavior was previously documented by Rosson and Carroll in a study of Smalltalk [16].

In 10% of the copies, the programmer left one or more visually indistinct or off-screen references unchanged (one such error is portrayed in point 1 of Figure 9). Because these references were syntactically valid, the compiler did not complain, and because programmers believed that their copied code was correct, it was the *last* place they looked for errors. Programmers spent an average of 3 (±1) minutes testing false hypotheses before finding these errors, except in one case (not included in the average), where a programmer worked for over 18 minutes.

In 12% of the copies, the programmer only copied part of a pattern of code that had to be distributed within and/or between files. These partially copied patterns often led to *dead-end data*: variables that were assigned some value that was not subsequently used (one such error is shown in 2 of Figure 9, where the new *tPanel* was supposed to be added to the *colorPanel*). When these errors led to runtime failures, programmers did not think to look for these unused definitions, because they did not know they were unused. Programmers spent an average of 4 (±1) minutes testing false hypotheses before finding such dead end data.

5.6 Overall Navigational Overhead

While no single problem in the previous sections incurred dramatic overhead, overall, navigation was a significant bottleneck. Adding the navigational costs of recovering working sets, iterating through search results, returning from navigations, and navigating between indirect dependencies within and between files, programmers spent an average of 19 minutes, or 35% of their time not spent answering interruptions, simply navigating.

6. DESIGN REQUIREMENTS AND IDEAS

The central goal of our study was to elicit design requirements for tools to help with maintenance tasks. To this end, we present a summary of our empirical findings and corresponding design requirements in Table 5. It is important to note that the requirements listed in Table 5 do *not* include requirements from our study already satisfied by Eclipse. For example, the results of our study suggest that Eclipse's "open declaration" and other Java search tools are essential features for *every* maintenance-oriented IDE. Here, we limit our discussion to requirements that have yet to be satisfied by a modern IDE.

Given these requirements, what type of environment would best satisfy them? There are several possibilities. For example, since many of the bottlenecks identified in our study were partially due to a lack of screen real estate, it is possible that simply giving programmers a larger screen, but the same environment, might mitigate many of these inefficiencies. However, while more space would, for example, leave more room for file tabs and result in fewer off-screen code fragments, it would not make direct or indirect dependencies easier to identify or navigate, nor would it help any of the other fundamental difficulties discussed in Table 5. Furthermore, more space might even introduce issues with screen real estate *management*, removing one interactive bottleneck, while introducing another.

Rather than further discuss incremental improvements to Eclipse, in the rest of this section we discuss several new ideas for maintenance-oriented IDEs. We will describe these ideas relative to the conceptual sketch of a new kind of maintenance-oriented IDE shown in Figure 10. We will discuss the features of the IDE by the requirements in Table 5 that the features satisfy, and use one programmer's working set for the **THICKNESS** task, which is portrayed in Figure 2, to illustrate our points.

Table 5. Design requirements for maintenance-oriented tools, elicited from the empirical trends in the study.

#	Empirical Result	Design Requirement for Maintenance-Oriented Tools
R1	Programmers formed working sets of task-relevant methods and statements.	Provide a working set interface that supports the quick addition and removal of task-relevant code fragments.
R2	Because programmers had to store their working sets in the interactive state of file tabs and package explorer, when they changed tasks, they lost their working set.	Automatically save and recover of working sets of task-relevant code fragments, ensuring that the tools used to *navigate* working sets are distinct from the tools used to *represent* working sets.
R3	When programmers found task-relevant code, they tended to glance at its dependencies. Also, more than 60% of navigations of *indirect* relationships were for the purpose of comparison. All of these incurred significant visual search costs.	When programmers add code to a working set interface, automatically add its direct and indirect dependencies. Then, directly or indirectly related code could be placed side-by-side, avoiding the interactive overhead of opening and closing file tabs.
R4	When copying code, programmers often left indistinct or off-screen references unchanged. Because they believed the copied code was correct, it was the last place they checked for errors.	Copied code should maintain a dependency with its "original" so that unchanged references can be marked as "suspect" until verified. These markers should be apparent even when off-screen.
R5	When copying a pattern of code that was distributed within or between files, programmers often duplicated only part of the pattern, leading to *dead-end data*.	When programmers copy code, the IDE should check if the programmer is neglecting any dependencies in the copied code and offer to help collect them.
R6	Programmers searched for task-relevant *names*, but only half of such searches led to task-relevant code. Programmers also used surface features of *Paint*'s output to deduce the cause of failures, but only a 1/4 of such features correlated with the cause.	Let programmers ask about program output of interest and have the IDE gather all of the code that was directly responsible for the output in question. This way, the correct working set can be built *automatically* by the IDE.

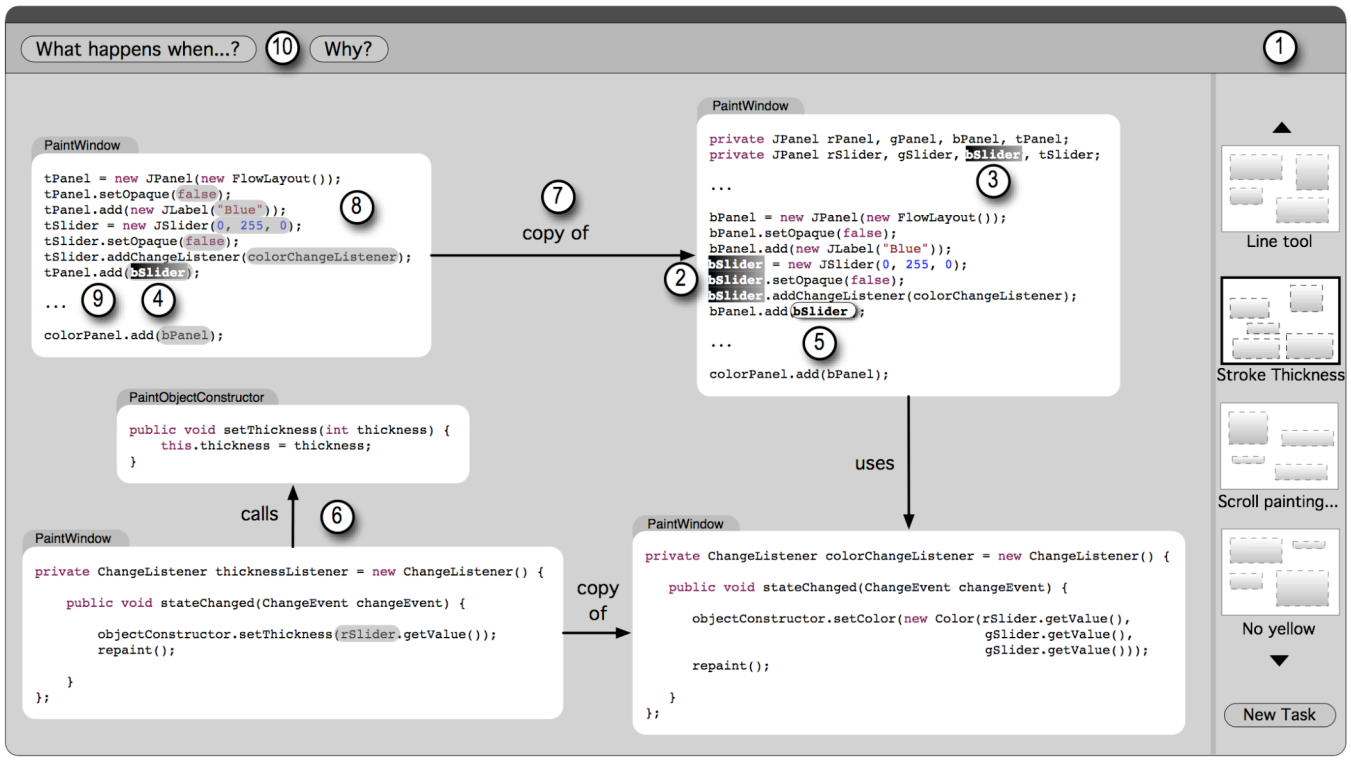

Figure 10. A conceptual sketch of one possible maintenance-oriented development environment that could satisfy the requirements listed in Table 5. Each of the circled numbers is discussed throughout Section 6.

6.1 Fragments Instead of Files

Programmers' working sets consisted of code fragments containing individual statements and methods, as opposed to whole files (**R1** in Table 5). Thus, IDEs should allow programmers to organize and view their work in terms of these fragments. For example, the tool in Figure 10 portrays *all* of the code fragments that were part of one programmers' final working set for the **THICKNESS** task on a single screen. Compare this view of the code to the extent of his navigations portrayed in Figure 2.

To add code fragments, programmers could simply select a region of code in a traditional editor and drag it to the environment shown in Figure 10. When doing so, the IDE could automatically add all of the direct and indirect dependencies in the code. Programmers could then "prune away" the task-irrelevant dependencies by simply dragging them out. Having the system automatically include dependencies, rather than having the programmer collect them manually, would ensure that no dependencies were overlooked, possibly preventing future errors.

6.2 Sets of Working Sets

Because programmers' working sets were represented by the interactive state of the package explorer and file tabs, programmers frequently lost their working sets when changing tasks (**R2**) because they had to close tabs and collapse nodes to make room for others. Instead, programming environments could allow programmers to save explicit representations of their working sets and help maintain a list of unfinished maintenance tasks, as in point (1) in Figure 10. This way, if a programmer was working on several independent maintenance tasks, all of the relevant state could be preserved and recovered by the environment, rather than manually by the programmer. In collaborative software development situations, the iconic representations of a working set could become highlighted when other programmers modify code fragments that are part of the working set. Not only would this give programmers a notification about a potentially important changes in their tasks, but it would also create the opportunity for collaboration on the task, or help avoid the overhead of a duplicated effort.

6.3 Supporting Reading

Nearly 1/5th of programmers' time was spent reading code within a fixed view in the Eclipse editor (as in Figure 3). This reading likely consisted of visual searches for local dependencies in the code, by looking for similar names. Thus, it could be very helpful to highlight dependencies in the code automatically (points 2, 3 and 4) based on the current text caret position or text selection (point 5). This way, programmers could inspect a program element's dependencies by simply clicking and moving the text caret, as opposed to using Eclipse's cumbersome commands and dialogs. This would incur virtually no interactive overhead, but would reveal nearly all of the important relationships for which programmers in our study had to carefully read (and re-read) to discover, even those that would have been off-screen.

In addition to reducing the interactive cost of finding dependencies, this selection-based dependency highlighting could also help prevent and find errors. For example, as soon as the text caret moves over the reference to *bSlider* at point (5) on the right in Figure 10, the invalid reference to *bSlider* at point (4) on the left becomes immediately obvious, especially given its relative distance from the other *valid* references.

6.4 Visualizing Dependencies

Programmers spent considerable time *glancing* at dependencies as well as navigating between indirect dependencies for the purposes of comparison (**R3**). Instead of requiring programmers to *navigate* such relationships, programming environments could explicitly visualize these dependencies side-by-side, as in point (6) in Figure 10. If all of the dependencies were on a single screen, there would be virtually no interactive overhead to compare a set of code fragments, since both direct and indirect relationships could be viewed without navigating. This would have saved each programmer an average of 9.2 minutes in our 70-minute study.

In addition to more traditional relationships such as "*uses"* and "*declares*," "*copy of*" relationships could also be shown, as in point (7), to help programmers compare copied code to its original. Unchanged references in the copied code could be highlighted, as shown in point (8), which would have prevented the costly errors in our study (**R4**). Also, to help avoid errors when copying code that is distributed within a file (**R5**), the IDE could hide unrelated code in order to bring the indirectly dependent code closer, as in point (9).

6.5 Context In-Place

One potential tradeoff of the view in Figure 10 would be that programmers would not see the code surrounding each code fragments. While our study suggests that such context is mainly useful only for navigation, there may be other situations context is important. One way to support this would be to show the surrounding code *in-place*. For example, programmers could hold a meta key in order to temporarily see the surrounding code around the text caret, without having to navigate to it. This could serve as a quick reminder of the purpose of the surrounding code, but would incur very little interactive overhead to invoke.

6.6 Working Sets From Questions

When starting a task, programmers typically asked a *how* or *why* question about the program's output (**R6**). To answer these questions, programmers essentially had to guess an answer to the question and then verify it by inspecting the code. Not only did this cost time, but programmers also frequently made risky assumptions about the program's runtime behavior in the process, often leading to errors. Therefore, instead of requiring programmers to answer these *why* and *how* questions themselves, programming environments could provide an interface for asking directly about program output and have the system *automatically* build a working set of task-relevant code with a precise dynamic slice [18] on the output in question. This is in fact precisely what the *Whyline* [13] does for *why did* and *why didn't* questions, which was shown to reduce novices' debugging time by a factor of 8, and help them complete 40% more tasks compared to novices without the Whyline.

When using a Whyline-like tool to ask "why" questions (point 10), programmers questions could be checked for invalid assumptions about the program's runtime behavior, and the interface could reveal such assumptions by comparing the question against what actually happened in the program's runtime history. For example, the programmers in our study could have asked, "Why wasn't *undoButton*'s *action* executed?" and the tool could have replied, "*undoButton's action* was executed; maybe its execution didn't change anything on-screen?" This would prevent programmers from acting on false assumptions about a program's behavior, saving time, and potentially preventing errors from being introduced due to these false assumptions. Furthermore, the tool could automatically build a working set of the code executed as a result of the action being executed, helping the programmer to find out whether and why nothing changed on-screen. We are currently developing a Whyline prototype for Java.

To use the "What happens when...?" tool to answer *how* questions (point 10), programmers could execute a program, click the "What happens when...?" button, and then perform some action on a user interface in the program to show the tool what dynamic behavior to analyze. For example, to determine what happens when a button is pressed in a program with a graphical user interface, rather than manually inspect a program's code for code that *seems* related, the programmer could ask the environment to automatically collect all of the static and dynamic dependencies related to the click event on the button of interest by simpling clicking on the button. The environment would then build a working set of code that was executed as a result of clicking the button, determine all of the direct and indirect static and dynamic dependencies, and present them to the programmer.

7. CONCLUSIONS

We have presented a study that suggests that maintenance work consists of three fundamental activities focused on forming, navigating, and manipulating a working set of task-relevant code fragments. Our findings are largely consistent with previous studies of program comprehension: programmers' understanding is facilitated by recognizable patterns in code [3, 6]; expert programmers often start understanding code top-down, but finish bottom-up [5]; and programmers follow an as-needed strategy for navigating dependencies [9]. Our study augments these findings with a higher-level account of maintenance work, and specific data on the impact of the Eclipse IDE on programmers' maintenance task performance. In particular, our study found that on average, programmers spent about 35% of their time with the mechanics of navigating between dependencies.

Of course, our study also suffers from several limitations. The 10 programmers in our study are not likely to be representative of programmers in industry. The size of the *Paint* program is certainly not representative of heavily maintained software systems. Programmers that work on teams may have different strategies for maintaining code that do not involve forming working sets of task-relevant code; for example, they may be given a set of code to maintain and have to collaborate with other maintainers if their tasks take them outside of this set. Further studies of maintenance work are required to verify the generalizability of our findings.

Despite such limitations, our study's findings have directly inspired several novel ideas for maintenance-oriented IDEs that deserve further elaboration and development. We are currently building an environment like the one portrayed in Figure 10 to support Java programs that we hope will both eliminate the interactive overhead identified by our study, as well as come closer to solving some of the fundamental difficulties of maintenance activity that we observed. We intend to evaluate its utility both in the lab and in the large to see how well the findings in our study generalize in practice. We hope that others will find the results presented in this paper useful for similar inspirations.

8. ACKNOWLEDGEMENTS

We would like to thank Scott Hudson, James Fogarty, Elsabeth Golden, Santosh Mathan, and Karen Tang for helping with the experiment design and execution, and we also thank the study participants for their efforts.

This work was funded in part by the National Science Foundation, under NSF grant IIS-0329090, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. The first author is also supported under a National Defense Science and Engineering Graduate Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

9. REFERENCES

[1] Antoniol, G., Fiutem, R., and Cristoforetti, L., Design Pattern Recovery in Object-Oriented Software, *6th IEEE International Workshop on Program Understanding*, 153–160, 1998.

[2] Beyer, D., Noack, A., and Lewerentz, C., Simple and Efficient Relational Querying of Software Structures, *10th IEEE Working Conference on Reverse Engineering*, 2003.

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

[4] Brooks, F. P., The Mythical Man-Month: Essays on Software Engineering, 20th anniversary edition ed: Addison-Wesley, 1995.

[5] Corritore, C. L. and Wiedenbeck, S., An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers, *International Journal of Human-Computer Studies*, 54, 1-23, 2001.

[6] Crosby, M. E., Scholtz, J., and Widenbeck, S., The Roles Beacons Play in Comprehension for Novice and Expert Programmers, *14th Workshop of the Psychology of Programming Interest Group*, Brunel University, 58-73, 2002.

[7] Davies, S. P., Models and Theories of Programming Strategy, *International Journal of Man-Machine Studies*, 39, 236-267, 1993.

[8] Erlikh, L., "Leveraging Legacy System Dollars for E-Business," in IT Pro, vol. May/June, 2000, pp. 17-23.

[9] Fry, C., Programming on an Already Full Brain, *Communications of the ACM*, 40, 4, 55-64, 1997.

[10] Gonzalez, V. M. and Mark, G., "Constant, Constant, Multi-Tasking Craziness": Managing Multiple Working Spheres, *CHI 2004*, Vienna, Austria, 113-120, 2004.

[11] Green, T. R. G., Petre, M., and Bellamy, R. K. E., Comprehensibility of Visual and Textual Programs: A Test of Superlativism against the 'Match-Mismatch' Conjecture, *Empirical Studies of Programmers, 4th Workshop*, 121-146, 1991.

[12] Ko, A. J. and Myers, B. A., A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *To appear in the Journal of Visual Languages and Computing*, 2004.

[13] Ko, A. J. and Myers, B. A., Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, *CHI 2004*, Vienna, Austria, 151-158, 2004.

[14] Müller, H. A., Orgun, M. A., Tilley, S. R., and Uhl, J. S., A Reverse Engineering Approach to Subsystem Structure Identification, *Journal of Software Maintenance: Research and Practice*, 5, 4, 181-204, 1993.

[15] Perlow, L., The Time Famine: Toward a Sociology of Work Time, *Administrative Science Quarterly*, 44, 57-81, 1999.

[16] Rosson, M. B. and Carroll, J. M., The Reuses of Uses in Smalltalk Programming, *ACM Transactions on Computer-Human Interaction*, 3, 3, 219-253, 1996.

[17] Teasley, B. E., The Effects of Naming Style and Expertise on Program Comprehension, *International Journal of Human-Computer Studies*, 40, 757-770, 1994.

[18] Zhang, X. and Zhang, Y., Precise Dynamic Slicing Algorithms, *International Conference on Software Engineering*, Portland, OR, 319-329, 2003.