Reducing Code Navigation Effort with Differential Code Coverage

Kaitlin Duck Sherwood and Gail C. Murphy
University of British Columbia
Vancouver, British Columbia, Canada

ducky@webfoot.com, murphy@cs.ubc.ca

Abstract

Programmers spend a significant amount of time navigating code. However, few details are known about how this time is spent. To investigate this time, we performed a study of professional programmers performing programming tasks. We found that these professionals frequently needed to follow execution paths in the code, but that they often made faulty assumptions about which code had executed, impeding their progress. Earlier work on software reconnaissance has addressed this problem, but has focused on whether the technique could provide the correct information to a programmer, not on whether the technique reduces or improves navigation. We built a tool, called Tripoli, that provides an approximation to software reconnaissance via differential code coverage and reran a subset of the initial study. We found that Tripoli had a positive effect on code navigation: less experienced programmers with Tripoli were often more successful in less time than experienced programmers without.

1 Introduction

A large percentage of developers' time is spent navigating through code. Ko and colleagues [4] found that developers, working on a small (500 line) program, spent 35% of their code maintenance tasks' time simply navigating through the code. While this percentage is consistent with our own experience, we still wondered where this time goes. Is there a way to get programmers to the right code more quickly or is all of this navigation necessary?

To better understand what is occurring during navigation, we analyzed a set of data we had collected in which six professional programmers worked on four tasks on two medium-sized (tens of thousands of lines) Java software systems. We noticed four significant challenges that multiple programmers faced which consumed a substantial amount of their navigation time, such as difficulty identifying missing code and being misled by badly named identifiers (Section 3). These challenges all involved difficulty in following and understanding an execution path from an identified point of interest in the source forward to the target method of interest.

This problem of providing execution information has been considered many times before. Program slicing encompasses techniques for finding what code is related to a particular variable or line of code (e.g., [14]). Omniscient debugging keeps logs of state changes during an execution and reassembles them in a debugger, allowing users to step through the trace backwards as well as forwards (e.g., [8]). Software reconnaissance extracts and displays the difference between two execution traces as a means of identifying code related to a feature of interest (e.g., [16]). For the navigation challenges we had identified, software reconnaissance appeared the most promising because it is most precise in how it targets code to consider.

Given the long history of investigations into these techniques, we expected to be able to look into the literature to find evidence that software reconnaissance (or slicing or omniscient debugging) helps reduce the effort required to complete a programming task. Optimally, we would have found specific evidence that these techniques reduce navigation effort. Somewhat surprisingly, we found almost no information about user studies for these techniques. For software reconnaissance, the technique of most interest, there are reports of how the technique can narrow down the number of elements identified for a feature of interest in a system, but there was no evidence that programmers could both apply the technique in the context of a particular task and then use the identified elements to perform the task more effectively (Section 2).

To help complete the loop and understand if a software reconnaisance approach could lessen the navigation difficulties we saw, we built Tripoli, a differential code coverage tool, as an Eclipse[1] plugin. Tripoli allows a programmer to:

- collect an execution trace exhibiting some functionality of interest (a fat trace),

[1] `www.eclipse.org`, verified 04/08/09

- collect an execution trace that does not exhibit the functionality of interest (a skinny trace), and to
- see a tree view of the code elements uniquely in the fat trace and also have the code unique to the fat trace coloured when it appears in an editor (Section 4).

Tripoli provides an approximation to the results one would see by applying a software reconnaissance technique. Software reconnaissance retains more information about execution ordering and the number of times various statements are executed. Tripoli focuses on providing the most straightforward bit of information: does this method or line execute in the fat trace and not the skinny trace.

We were interested in answering two questions about Tripoli:

1. Can the provision of differential code coverage information reduce programmers' navigation effort?
2. Is it possible for programmers to set up fat and skinny traces that result in precise enough differential code coverage sets to be useful?

To investigate these questions, we ran a second study (Section 5), involving six non-professional programmers, that used a subset of the tasks from our initial navigation study (Section 3). We found that subjects who used Tripoli were often more successful than their more-experienced counterparts who did not have access to Tripoli. The less experienced subjects frequently found key code faster than the professionals and sometimes found key code that the professionals were unable to find. For tasks involving a GUI application, the less experienced subjects were able to select and compute appropriate differences to drive Tripoli; the subjects were less successful for a non-GUI application.

This paper makes three contributions. First, it describes four navigation challenges we saw professional programmers face on programming tasks of medium-sized systems that help explain how the 35% of a programmer's code exploration time is spent. Second, it is the first to provide empirical data to show that an existing approach, software reconnaissance, can be used to reduce such navigation effort. Third, it shows that having full information about an execution dice – such as the execution order or number of times a method was executed – is not always necessary. Merely knowing if a method was executed or not appears to be highly useful.

2 Related Work

The studies and tool described in this paper relate most closely to two bodies of previous work: studies of code navigation and studies of software reconnaissance. There are many other related efforts. In particular, we have chosen not to survey program comprehension studies as these typically focus on how programmers gain cognitive understanding of code as opposed to where they are having problems with activities associated with gaining comprehension, such as code navigation. We have also chosen not to include details on other approaches for providing execution-based information, such as slicing, as our focus is on the empirical study of the use of software reconnaissance to reduce navigation effort; future work might compare the effectiveness of various techniques for providing needed execution information.

2.1 Code Navigation Studies

Early studies in program comprehension described the presence of non-localized code that required visiting multiple locations when performing a task to a system [7]. Although such needs for code navigation were identified many years ago, it is only recently that more attention has been specifically paid to code navigation.

Ko and colleagues quantified the degree to which code navigation activities contribute to the performance of maintenance tasks, finding that developers spend 35% of their time on these tasks simply navigating through the code [4]. They report that programmers began tasks searching for task-relevant code, but that their first search found relevant code only half of the time, and irrelevant code the other half of the time. Overall, programmers in their study spent an average of 25 minutes (out of an average of 72 minutes) inspecting task-irrelevant code.

Robillard and colleagues also studied navigation, focusing on how subjects successful at a programming task performed their navigation [10]. From their study of a small number of developers, they report that successful subjects did more focused searches, recorded their findings, prepared a modification plan, and kept to the plan while implementing the change. Unsuccessful subjects used a more opportunistic, ad-hoc approach.

The study we report on in this paper of six professional programmers differs in producing results about what programmers find challenging during the large amounts of time they spend navigating.

2.2 Software Reconnaissance Studies

A number of papers have discussed *software reconnaissance* (also called *dynamic program dicing* or *simultaneous dynamic program slicing*) where differences in execution slices are used to locate features, including [1, 2, 3, 9, 13, 15]. With the exception noted below, all of the validation in these papers (when done at all) focuses on how well their tool generates slices that includes the correct element(s) (e.g., on the precision and recall), and not on whether the tool reduces navigation time in practice. The only paper we

found that examined subjects using a tool was by Simmons and colleagues, who compared two different software reconnaissance tools, CodeTEST and Recon3, as used by two subjects. In contrast to the study we report on in this paper, they did not compare developers using software reconnaissance with those who did not [13].

3 Experiment #1: Weta

We conducted a user study of professional programmers performing four programming tasks on two separate software systems. The main goal of this experiment was to understand the effect of code navigation strategies on the performance of programming tasks. For half of the tasks, subjects used a regular release of the Eclipse IDE; for the other half, subjects used an Eclipse we modified, which we refer to as Eclipse/Weta, that provided better support for a breadth-first navigation via a one-tab-per-navigation-path (instead of one-tab-per-file) editing behaviour. From this study, we did not find any conclusive differences between support for different navigation strategies. In fact, subjects largely used similar navigation strategies with both Eclipse and Eclipse/Weta. From the study, we did observe a number of common navigation problems that accounted for a substantial amount of the programmers' time spent on navigation. We focus on these aspects of the experiment in this paper; full details are available elsewhere [12].

3.1 Subjects

We recruited professional programmers to participate in the study who had at least six months experience with Java and Eclipse. Seven programmers agreed to participate as subjects. Although one subject met the base requirements of previous experience by having nine months of experience, this was the limit of his experience; he had no prior training or classes in programming. His behaviours were sufficiently different enough from the other six that we have omitted his data from this paper as not being representative of a more expert level of programming.

The remaining six subjects had between five and twenty years of overall programming experience and between four and seven years of Java experience, as seen in Table 1. All were male, and all but one spoke English natively. The non-native English speaker spoke an Indo-European language natively and had an outstanding, idiomatic grasp of English. We refer to the subjects by pseudonyms in this paper.

3.2 Method

Each subject performed two programming tasks that involved debugging and changing code in an existing system with Eclipse 3.3, Mylyn 2.1[2], which provides a task-focused programming interface and the Mylyn UI Usage Reporting Plug-in 2.0, which supports logging of a programmer's interactions with the IDE. For each task, a subject was given a textual description of the task. For one of the systems, a byte-code obfuscator, they were also given a short textual description of the purpose of byte-code obfuscation. Each subject was then given training on Eclipse/Weta's navigation support, which provides similar tabbing functionality for the Eclipse Java editor to a web browser; for instance, at a call site, users can open the the definition of the call either in a new tab (as is the standard behaviour in the stock Eclipse), or in the same tab (as is the default in current Web browsers). Since Eclipse/Weta augments Eclipse, subjects could still navigate as in a regular Eclipse, which many did. The differences in usage between Eclipse and Eclipse/Weta are sufficiently small that we consider navigation patterns across all traces in this paper and do not differentiate between tasks performed with Eclipse and tasks performed with Eclipse/Weta any further in this paper. Each subject was given twenty minutes to work on each task. Each subject worked with a 24" monitor set to a resolution of 1920 by 1200 pixels.

Table 1. Self-reported experience programming, using Java, and using Eclipse (all figures in years)

Pseudonym	Programming	Java	Eclipse
Bob	10	4	3
Dave	7	7	7
Mark	>10	5	6
Peter	20	7	6
Steve	12	6	6
Tom	5	4	4

Two of the tasks (SIZE and ARROWS) used JHotDraw 6.0b1[3] and were based on a previous study conducted in our research group [11]. The other two tasks (OUTPUT and OBFUSCATE) used a modified ProGuard 3.8 code base,[4] in which the GUI-based configuration was stripped out so that subjects worked with a command-line version of the system. Table 2 provides brief descriptions of each task. Subjects did not perform the tasks in the same order; however, each subject performed two tasks from one code base and then two tasks from the other code base.

Each subject worked with a hired undergraduate assistant. The purpose of this assistant was to try to encourage

[2] `www.eclipse.org/mylyn`, verified 4/09/08
[3] `www.jhotdraw.org`, verified 4/09/08
[4] `proguard.sourceforge.net`, verified 4/09/08

Table 2. Description of tasks

Task	Description
Arrows	Figure out why menu commands that should alter arrow tips were not; fix.
Size	Add code to write the active pane's height and width to status bar.
Output	Change the output for number of fields to be two bytes instead of four.

verbalizations of the subjects' thought processes. Each subject was told to assume that the assistant was familiar with Eclipse, was good at programming, but was really bad at finding things in other people's code. We asked each subject to teach the assistant how to find things in other people's code. The assistant was given permission to point out syntax errors and to gently redirect the subjects if they went too far off track, but was asked to not give any help otherwise. The assistant also was instructed to ask the subject for clarifications if it was not clear why they chose a particular navigational step.

We gathered data from several sources as each subject worked. We recorded the screen and the audio of the conversation with the assistant using Camtasia Studio 4.0.0.[5] We captured a trace of the programmer's interactions with Eclipse using the Mylyn UI Usage Reporting Plug-in. We reviewed the video carefully in order to add some information to the interaction trace that the Mylyn usage plugin did not capture; for example, we annotated the traces with what terms were used in searches. We also generated a rough transcription that interposed the think-aloud statements with important events from the trace.

3.3 Results

From our analysis of the sessions, we observed that the same navigation problems occurred over and over again for the subjects. We classified these issues as:

1. lost in superclass
2. finding where GUI actions were handled
3. finding missing code
4. misleading language

3.3.1 Lost in the Superclass

In the SIZE task for JHotDraw, the subjects were asked to display the height and width of the application's active pane in the application's status line next to the text "Active View Size". All six of the subjects began by immediately searching for the string "Active View Size", which took each of them quickly to the method `DrawApplication.showSizeInformation()`[6]. At this point, all of the subjects ceased to make progress, iteratively looking at parts of `DrawApplication` without making progress. A problem the subjects encountered was that `DrawApplication` is a *superclass of a superclass* of the application class with the `main()` method being executed. When the subjects traced statically forward in the code—for instance, asking Eclipse to show them the declarations of various methods in `DrawApplication`—Eclipse took them to where those methods were declared in `DrawApplication` (or its superclasses), but never to methods in `DrawApplication`'s subclasses. As a result, the code that the application was executing differed from the code that the subjects saw when attempting to statically navigate through the code. Stating this another way, the execution was using `DrawApplication`'s subclasses' overriding methods, but subjects only saw methods at or above `DrawApplication`.

Three of the six subjects never realized that there were subclasses of `DrawApplication` that they should be considering.

3.3.2 Finding where GUI actions were handled

In the SIZE task, while the subjects were quick to find code that should be called (`showSizeInformation()`), each stalled trying to find where they should put the call. A problem seemed to be that the subjects could not find a starting point from which to start exploring. There was no text in the GUI associated with resizing for which they could search. They also did not know what the active pane was called: was it a window, a canvas, a drawing, a frame, or a view? Only one of the subjects ever passed their eyes over the key method `MDIDesktopPane.endResizingFrame()`, where `showSizeInformation()` should be inserted, but even he did not recognize it as an important spot.

In the ARROWS task, subjects needed to figure out why menu commands to change the arrow tips had no effect on the ending arrow tip. In this case, the subjects were able to find the target method (`PolyLineFigure.setAttribute()`), but it took a substantial amount of time—an average of 8:46 min. The key problem encountered during this navigation was implicit invocation: a menu listener gets set up at one point in the code and executed at a later point, but it was difficult to trace directly from the menu action creation to the code that handled the action.

[5] `www.techsmith.com`, verified 04/08/09

[6] We elide method arguments throughout this paper.

Table 3 reports the time spent navigating until they found `setAttribute()`, measured from the time they finished reading the instructions and reproducing the problem to the time when they first opened `setAttribute()`.

Table 3. Time to find `setAttribute()` in minutes:seconds

Subject	Time
Tom	4:43
Mark	5:43
Bob	6:58
Dave	11:17
Peter	11:57
Steve	12:02

3.3.3 Finding Missing Code

Subjects also experienced a lot of trouble recognizing when code was missing. During the ARROWS task, only two of the six subjects realized right away that the code to set the ending arrow tip was missing from `PolyLineFigure.setAttribute()`. The other four recognized that `setAttribute()` set the starting arrow tip, but then spent a fair amount of time trying to find the (non-existent) code where the ending arrow tip code was set.

They appeared to make the faulty assumption that `setAttribute()` was not executed. (If the subject did not trace the execution path explicitly, they in fact had no assurance that the method would execute in response to the user requesting a change in the ending arrow tip.) Evidence to support the hypothesis that they made faulty assumptions about `setAttribute()` not executing is that while all of the subjects eventually figured out that code was missing, they usually figured it out *after* using a different navigation strategy that gave more information about the actual execution path.

Table 4 shows how long it took subjects after first seeing `PolyLineFigure.setAttribute()` to realizing that the method was missing code for the ending arrow tip. In the table, the first letter is an "M" if they traced from the menu action creation to `PolyLineFigure.setAttribute()` and "I" if they started tracing from a class that looked "interesting". The next characters are "S" for static tracing, "D" for dynamic tracing, or "S/D" for a combination of the two. "M" and "D" follow the execution path more closely than "I" and "S", and the table below shows that the successful strategies were more likely to be "M" and "D" than the unsuccessful strategies.

Table 4. Time after seeing `setAttribute` to realizing code was missing

Subject	time to recognize	first strategy	successful strategy
Tom	0:00	MS	MS
Peter	0:00	IS/D	IS/D
Steve	2:17	IS	IS
Dave	2:50	MS	MD
Bob	5:09	IS	MS
Mark	10:15	IS	MS

3.3.4 Misleading Language

As seen in Table 5, only two subjects found the key method (`ProgramClassFile.write()`) in the OUTPUT task[7]. Six of the methods and classes on the execution path the subjects needed to follow to solve the task by tracing had `read` and/or `input` in their names; this was disconcerting to subjects who wanted to find code related to *output* or *writing*. Frequently and repeatedly, subjects traced to one of the misleadingly-named methods and stopped. One subject found `ProgramClassFile.write()` by tracing dynamically; the other browsed to the class of the penultimate method because the class name looked interesting, and read far enough in the file that he spotted a call to `write()`.

Table 5. Time to find `ProgramClassFile.write()`

Subject	Time
Tom	DNF
Peter	17:11
Steve	14:35
Dave	DNF
Bob	DNF
Mark	DNF

The subjects made the understandable but faulty assumption that the methods with names including "input" or "read" were not executed during output processing.

[7] "DNF" means "Did Not Finish"

3.3.5 Summary

What is striking about these navigation issues that the subjects encountered is that each involves difficulties in following an execution path of the program. As a result of the difficulties, subjects tended to make false assumptions about how the program worked, become confused about how it could work under such an assumption, would wander through the code or end up revisiting paths they had visited before.

An obvious question is that if the subjects had such difficulties following execution paths statically, why did they not use the debugger? Only seven times across eighteen tasks did the subjects invoke the debugger. Two of the subjects never used the debugger at all. A difficulty we observed when subjects did use the debugger was the tedious nature of stepping into and over code. The navigation support in the debugger is primitive compared to the navigation support available in the static code browsing tools.

We hypothesize that the navigation difficulties we observed could be lessened if the execution path of interest was more apparent as subjects navigated the code.

4 Tripoli

To investigate our hypothesis, we built the Tripoli Eclipse plugin. Tripoli is essentially a software reconnaissance [15] tool built into an IDE. The idea, as with software reconnaissance, is that the difference between two executions of a system can be an effective way to identify code related to some specific functionality of interest. With software reconnaissance (and Tripoli), a programmer collects an initial trace when executing the system demonstrating a behaviour of interest and then collects a second trace when executing the system without the behaviour. The difference between the two traces is intended to help pinpoint the code of interest. An advantage of the differencing approach is that it removes all code not unique to the functionality of interest; in particular, this can help remove voluminous amounts of irrelevant code, e.g., initialization code.

Tripoli is built on an Eclipse code coverage plugin called EclEmma[8], which is itself based on the EMMA code coverage tool.[9]. EclEmma provides an ability to colour lines of Java code in Eclipse editors based on whether the line was executed or not in the previous program run.

In Tripoli, we refer to the first run that will be input to a differencing run as a *fat* run and the second run as a *skinny* run. A programmer using Tripoli must explicitly capture a fat and skinny run. Tripoli then computes the difference and (in an editor) colours green all lines of code that are executed only in the fat run. All other executable lines are coloured red. Tripoli also provides a tree view of the computed difference that shows the program elements that were executed only in the fat run.

With Tripoli, it is possible to experiment with what happens to code navigation when execution path differences are delivered in the context of existing static code browsing features.

[8] `www.eclemma.org` (version 1.2.2), verified 04/09/08

[9] `emma.sourceforge.net`, version 2.0.5312, verified 04/09/08

5 Experiment #2: Tripoli

To investigate our hypothesis that code navigation difficulties encountered during programming tasks could be lessened with execution path information, we reran a subset of Experiment #1, encouraging subjects to use Tripoli. We were interested in two specific questions:

- Would the occurrence of the navigation challenges listed in Section 3.3 be reduced with the use of Tripoli?
- Would subjects be able to compute (an) appropriate difference(s) with Tripoli to aid the completion of a given task?

5.1 Subjects

The six subjects who participated in this study were drawn from our colleagues in the Software Practices Laboratory at the University of British Columbia. Five were graduate students and one was an undergraduate. All but one had less than one year of full-time professional (non-academic) experience. SubjectC had seven years of development experience, including three years of hands-on coding. When self-reporting their experience, most of the subjects commented that they did not program full-time and asked how they should answer the questions about experience. We advised them to state how long it had been since they *first* started programming, using Java, and using Eclipse. As a result, their experience is likely greatly inflated in comparison to Experiment #1's subjects.

None of the subjects were native English speakers. Two subjects were raised speaking an Indo-European language and four subjects speaking an Asian language. Three of the subjects were women and three were men.

5.2 Method

The experimental setup for Experiment #2 was very similar to the setup for Experiment #1. This section describes the differences only.

The CPUs used were slightly different. Experiment #1 used an 1.73GHz IBM ThinkPad T42; Experiment #2 used a 2GHz Lenovo T61 ThinkPad. We do not think this difference resulted in a qualitatively different user experience.

Table 6. Self-reported experience programming, using Java, and using Eclipse (all figures in years)

Subject	Programming	Java	Eclipse
SubjectA	7	3	3
SubjectB	10	.75	.5
SubjectC	12	7	4
SubjectD	5	5	4
SubjectE	1	1	1
SubjectF	11	7	6

Before a subject's first task, we gave some instruction about Tripoli and demoed how to use the tool. We also allowed the subjects to experiment with Tripoli on some demo code that was not the target of any task in the study. As all tasks were to be performed with Eclipse and Tripoli, we did not provide any instruction on Eclipse/Weta or on the idea of using breadth-first navigation.

The subjects always did the two JHotDraw tasks first and then were given the option to do the OUTPUT task for ProGuard. Everyone except for SubjectA did the ARROWS task first and the SIZE task second. (We made this change because we realized that Arrows would give a better learning experience, and thus would be better as a first task.) If a subject chose to do the OUTPUT task, we provided the same training about ProGuard that we gave the Experiment #1 subjects. Four subjects elected to do the OUTPUT task.

During Experiment #1, we stopped subjects after they had been working for 20 minutes (or occasionally a minute or so over). During Experiment #2, we allowed the subjects to continue working as long as they liked. At the 20 minute mark, we told them that they had hit the 20 minute mark and that they could stop if they chose. The Experiment #2 subjects almost always elected to continue if they had not already finished the task, continuing fourteen times out of the eighteen total tasks.

There was no assistant in this study; the subjects were asked to think-aloud without aid. One researcher, the developer of Tripoli, sat beside the subject, took notes and occasionally asked questions like those that the assistant asked. If Tripoli bugs appeared, she would point them out. The researcher would also answer questions about Eclipse for those who were less familiar with Eclipse.

5.3 Results

The subjects in Experiment #2 (with Tripoli) were frequently more successful than the Experiment #1 subjects, despite the subjects in Experiment #2 having vastly less experience than the professionals in Experiment #1.

Table 7. Time to opening of `setAttribute()`

Name	From end of repro	From start of fat
SubjectA	DNF	DNF
SubjectB	N/A, 24:32	N/A, 3:12
SubjectC	2:40	2:23
SubjectD	3:58	3:22
SubjectE	3:31	7:19
SubjectF	N/A	N/A

5.3.1 ARROWS Task

In the ARROWS task, two of the less-experienced subjects were able to find the `PolyLineFigure.setAttribute()` method faster with Tripoli than the fastest professional programmer. The three who unambiguously found `setAttribute()` using Tripoli spent an average of 4:27, compared to the professional's average of 8:46.

Almost all subjects in both studies started by reproducing the problem, following the instructions given in the task description. Sometimes Experiment #2 subjects used this run as their fat run, but usually they generated a fat run a second time. The second column in Table 7 gives the time from the end of the reproduction run to the time the Experiment #2 subjects first opened `setAttribute()`. This is the same method of calculating the time as used in Experiment #1. However, it does not always give a good reflection of how fast Tripoli gives answers, because subjects sometimes attempted to solve the problem without first using Tripoli to generate the difference and focus their efforts. The third column gives the time from the start of the first Tripoli attempt to when they opened `setAttribute()`.

SubjectA never found `setAttribute()`. He spent a very long time exploring the first method shown in the list of program elements in the difference; he mistakenly thought that the results were ranked by relevance. SubjectB spent a long time navigating using more conventional means before trying Tripoli. She saw `setAttribute()` first before trying Tripoli, so her first run in Table 7 is marked as "Not Appropriate" ("N/A") in Table 7. She found `setAttribute()` again 3:12 after starting her first Tripoli run. SubjectE took a long time to reproduce the problem, and SubjectF solved the problem without using Tripoli at all.

Most subjects were able to execute a skinny run that gave a small number of methods to explore. Table 8 shows the number of methods in each subject's difference. The me-

Table 8. Number of methods in the Arrows diff

Name	Number of methods
SubjectA	144
SubjectB	36
SubjectC	1
SubjectD	13
SubjectE	31
SubjectF	N/A

Table 9. Time after seeing `setAttribute()` to editing

Name	Time
SubjectA	N/A
SubjectB	6:12
SubjectC	2:44
SubjectD	2:32
SubjectE	DNF
SubjectF	N/A

dian number of methods in a difference was 31. By contrast, when subjects followed the written instructions for reproducing the problem, 692 methods were executed.

It is difficult to compare how quickly the Experiment #2 subjects realized that code was missing versus the Experiment #1 subjects. The Experiment #2 subjects verbalized significantly less than the Experiment #1 subjects, so it was difficult to tell when they decided code was missing. Absent a verbalization, we used the time when they started editing `setAttribute()`. This might be overly conservative.

Even so, the Experiment #2 subjects still compared favorably to the more-experienced Experiment #1 subjects. The average of the Experiment #2 subjects who finished was only 24 seconds slower than the Experiment #1 subjects, 3:25 versus 3:49, as can be seen in Table 4 and Table 9.

As shown in Table 9, SubjectA never saw `setAttribute()` and SubjectF never used Tripoli, so they are both marked as Not Applicable. SubjectE saw `setAttribute()`, but never edited `setAttribute()` or verbalized, so is marked as Did Not Finish.

5.4. SIZE Task

As mentioned earlier, the SIZE task requires finding code that is always called when the pane is resized, but which is never called otherwise. There is only one possible location

Table 10. Time from start of "fat" to opening of `endResizingFrame()`

Name	Time
SubjectA	1:55*
SubjectB	1:07
SubjectC	13:32, 4:44
SubjectD	18:25
SubjectE	13:46
SubjectF	12:50

Table 11. Number of methods in the Size diff

Name	Number of methods
SubjectA	11*
SubjectB	1
SubjectC	0,?
SubjectD	11
SubjectE	16
SubjectF	0,11

for that code, in the method `endResizingFrame()`. In Experiment #1, only one of the the professional programmers ever saw `endResizingFrame()` and it is not clear if he *noticed* it. By contrast, all of the subjects in Experiment #2 were able to find `endResizingFrame()` by using Tripoli, as seen in Table 10. Not all of the Experiment #2 subjects opened `endResizingFrame()` within the nominal twenty-minute task time, but sometimes they spent a long time before they realized that they should try Tripoli.

SubjectA (marked with an * in Table 11) turned out to have a fundamental misunderstanding about how Tripoli operated, so we gave him a little bit of coaching and encouragement before he tried doing his first diff. (SubjectA was the only one who did the SIZE task first and as a result, the only one who had significant issues learning Tripoli on the SIZE task. However, once he got the idea, he was very fast to open `endResizingFrame()` from the results.)

SubjectC's first attempt had no methods in it, as mentioned above. She explored a number of things without using Tripoli. After some time, we asked her a leading question, which prompted her to try again. On the second try, it took only 4:44 from the start of the fat run until she opened `endResizingFrame()`.

All of the Experiment #2 subjects were able to use the tool to dramatically narrow the list of methods to examine (from 582), as shown in Table 11.

SubjectC's screencapture was unusable, so we had to rely on notes and log information. The first time she tried

to use Tripoli, she (presumably accidentally) did the exact same actions in her fat and skinny runs, so got no methods in the difference view. Our notes do not mention her second Tripoli run yielding an unusually high or low number of methods, so her second run is marked with a ? in Table 11.

SubjectF also got zero results on his first try. (In his "fat" run, he resized the pane. In the "skinny" run, he clicked on the corner of the pane; while that was not enough to resize the pane, it was enough to trigger the same methods that resizing triggered.) He immediately tried again and got a more manageable eleven methods in his diff.

5.4.1 OUTPUT Task

The Experiment #2 subjects frequently outperformed the Experiment #1 subjects on the JHotDraw tasks. Their comparative performance on the Proguard OUTPUT task was not as impressive.

The OUTPUT task was not a GUI task. Instead of giving different user inputs to generate the fat and skinny runs, subjects needed to modify the source code to generate the two different runs. The two subjects who managed to create an appropriate Tripoli difference were successful in finding the key method, although we had to give each a hint or a prod to create the difference.

SubjectB found the important method (`ProgramClassFile.write()`) without doing a Tripoli diff, although she did have information (from EclEmma) about which classes and methods were executed. She did encounter a bug where the source lines were not always coloured to distinguish executed code from non-executed code.

SubjectD looked through the code that handled input for a long time, including doing a Tripoli difference that isolated the input section of the code. At 17:52, we gave her a hint that the professional programmers had looked at the output code. She took 37 seconds to find a place where she could isolate the output section of the code, and another 17 seconds to yield a diff with 58 methods in it. 4:33 later, she found the target method, `ProgramClassFile.write()`.

SubjectF reproduced the bug, then browsed the 230 files and 1874 methods visible in the program element view for that trace. After 13:48, we asked "Could the tool help you?, and then 36 seconds later, "What are you trying to find?" 2:32 later, SubjectF had found the necessary method, `ProgramClassFile.write()`, after generating a difference with 29 files and 56 methods.

SubjectA and SubjectC declined to do the Output task, while SubjectE did not finish.

5.4.2 Tool Issues

The subjects did have some difficulties using Tripoli. A number of times, the subjects got mixed up about the order of fat and skinny traces. We corrected them before they went too far astray.

In addition, subjects sometimes had difficulty because the red colouring was ambiguous, meaning all of "neither fat nor skinny executed this line", "both executed this line", and "only the skinny run executed this line". We believe that Tripoli would be easier to use if it represented all four execution cases with four different colours.

There was a bug such that sometimes the source code was not coloured properly. There was also a bug such that occasionally the code coverage output would not be written out. Time spent dealing with bugs was subtracted from all times reported above.

5.4.3 Summary

The results of Experiment #2 provide evidence to support the hypothesis that the navigation challenges we saw during Experiment #1 could be reduced by providing execution path information. While not all Experiment #2 subjects did better than all Experiment #1 subjects on all measures, many of the Experiment #2 subjects were more successful on a number of significant measures, despite having less experience. The results of this experiment suggest that the programmers were able to produce an appropriate difference from a selected fat and skinny trace to aid their navigation. The results are not as positive for a non-GUI application where each subject had difficulty producing an appropriate trace. Subjects might do better at non-GUI tasks with more familiarity with Tripoli.

6 Threats to Validity

The generality of the results from each of the studies we report on in this paper are dependent upon the degree to which the tasks and systems we chose are representative of the problems faced by professional developers in their everyday work. As the tasks varied in their cause—one involved adding code, one involved adding a call to existing code, and one involved modifying code—we believe at least some of the tasks overlap with problems encountered in the field.

Another threat to the results of Experiment #1 is our conclusion based on an analysis of the data that subjects did not perform significantly differently when using Weta than when using Eclipse. If such differences are in fact significant, this would cast doubt on the Experiment #1 results.

A major threat to the results from Experiment #2 was our choice of subjects. The subjects in this experiment are our

collaborators, so it is possible that they heard things about the tool or tasks that might have affected their ability to use the tool or to complete the task. Hearing about the tool beforehand might render the experiment less reproducible, but more representative of how the tool would be used in the field after people got used to it. Judging by the difficulties the subjects encountered when working on the tasks, it seems unlikely that anything they had heard about the tasks was significantly useful to them.

The difference in demographics of the subjects between the two experiments strengthens our argument that Tripoli is likely a useful tool. The subject pool in the two experiments differed considerably in experience, age, native language, and even gender. The experience and language differences would suggest that Experiment #1 subjects would be more skilled than the Experiment #2 subjects. The fact that the Experiment #2 subjects did well with Tripoli, despite those challenges, strengthens the argument that Tripoli was a factor in producing better performance.

It is possible that the questions that the researcher asked during Experiment #2 helped the subjects. If the subject spent a long time without using Tripoli, the researcher asked questions like, "What are you looking for?" and "Could the tool help you?". In two cases (SubjectA's ARROWS task and SubjectF's OUTPUT task), when it became clear that the subjects were not going to finish by themselves, the researcher asked questions that perhaps crossed the line between "leading question" and "helping question", but we carefully noted the assistance. In SubjectA's case, we marked his attempt as "Did Not Finish". We also did some training and coaching for SubjectA in the SIZE task, as he had a very fundamental misunderstanding of the intent of Tripoli. We did give SubjectD a hint about what area of the code the professionals had examined, but not about how to use Tripoli.

We feel that the minimal assistance we gave still did not bring these less-experienced users who were unfamiliar with Tripoli up to the skill level that professional programmers who used Tripoli regularly would achieve.

7 Discussion

We chose to provide the execution information we thought programmers needed using differential code coverage rather than a slicing [14] or omniscient debugging approach [8]. We made this choice because differential code coverage provided a balance by easing the determination of a starting point of interest for the task and understanding the execution flow forward from that point. Program slicing applies only after an initial starting point is determined. Perhaps the best example of omniscient debugging is Ko and colleagues' Whyline approach [5, 6]. While initial user studies show great promise, to query omnisicent debuggers about what happened also requires first finding an interesting point *or* it requires a query interface that contains a lot of semantic knowledge about the particular program being run. For example, to be able to answer the question, "Why did the number of fields print out as a four-byte integer instead of a two-byte integer?" would require that the tool understood "the number of fields", "print out", "four-byte", and "two-byte".

As a side benefit, we showed that even having information only about which code implemented a feature was helpful; it was not necessary to know the order of execution or how many times a statement was executed. This suggests that differential code coverage is a valuable approximation to execution slicing-based software reconnaissance.

8. Conclusions

In one study, we cataloged some difficulties that we observed professional programmers repeatedly having when attempting to navigate code as part of a programming task. We theorized that these difficulties stemmed in large part from faulty assumptions about the actual execution path of a run that they were attempting to retrace in the source code. To investigate this hypothesis in a second study, we built a tool called Tripoli that provides execution information to developers as an integrated part of an IDE. We did so using differential code coverage to approximate a software reconnaissance approach, colouring code and listing elements in the difference between two code coverage traces. In the second study, we found that less experienced programmers with Tripoli were frequently more successful than more experienced subjects without Tripoli. Moreover, for GUI programs, the less experienced programmers were able to select appropriate execution paths to generate differences, lending support that programmers may be able to use the approach effectively.

References

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151, 1995.

[2] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. In *Software Maintenance ,1993. CSM-93, Proceedings., Conference on*, pages 378–385, 1993.

[3] R. J. Hall. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automated Software Engineering*, 2(1):33–53, March 1995.

[4] 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. In *ICSE '05: Proceedings of the 27th international conference on Soft-*

ware engineering, pages 126–135, New York, NY, USA, 2005. ACM Press.

[5] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, New York, NY, USA, 2004. ACM Press.

[6] A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 301–310, New York, NY, USA, 2008. ACM.

[7] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, 1986.

[8] B. Lewis. Debugging backwards in time. In Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, September 2003.

[9] J. Li, D. Weiss, E. W. Wong, and X. Ma. An integrated solution for testing and analyzing java applications in an industrial setting. Technical report, Avaya Labs, 2005.

[10] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903, 2004.

[11] I. Safer and G. C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 229–243, New York, NY, USA, 2007. ACM.

[12] K. D. Sherwood. Path exploration during code navigation. Master's thesis, University of British Columbia, August 2008.

[13] S. Simmons, D. Edwards, N. Wilde, J. Homan, and M. Groble. Using industrial tools for software feature location and understanding. Technical report, Software Engineering Research Center, 2005.

[14] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[15] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 312–318, 1996.

[16] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.