Path Exploration during Code Navigation

by

Kaitlin Duck Sherwood

B. Sc., University of Illinois at Urbana-Champaign, 1984

M.Sc., University of Illinois at Urbana-Champaign, 1996

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August, 2008

Abstract

Previous research in computer science shows that developers spend a large fraction of their time navigating through source code. Improving developers' effectiveness in navigating code thus should yield significant productivity improvements. Previous research in a number of fields suggests that a more breadth-first approach to problem solving should be more successful than a more depth-first approach. Unfortunately, modern Integrated Development Environments (IDEs) do not support a breadth-first search well because they do not help developers keep track of exploration paths well.

We implemented an IDE that allows developers to track different exploration paths more easily, and ran a user study with seven subjects. To our surprise, subjects used the tool to mark waypoints instead of to facilitate a more breadth-first search. Intrigued, we examined more closely techniques for finding a starting point and for tracing relationships from there. We describe our findings, including common difficulties our subjects encountered, and propose a novel tool to reduce incorrect search paths.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Working on this research has been one of the most solitary endeavours that I have ever worked on, but even so, I could not have done without the support and assistance of many, many people. It might take a village to raise a child, but it takes a country to grow a MS thesis.

First, I'd like to thank our seven subjects. Without them, I would not have had a user study. Thank you "Bob", "Dave", "Jim", "Mark", "Peter", "Steve", and "Tom".

Thanks to Maciek Chudek, for explaining confirmation bias to me.

I'd like to thank IBM for making the source code to the Eclipse IDE publicly available, so that I had a fine platform to build Weta upon.

I'd like to thank the governments. Dealing with the Canadian federal government – for a study permit for me, a work permit for my husband, study permit extensions, Social Insurance Number, SIN extensions, etc. – has been a stunningly pleasant experience for this U.S. citizen. I'd like to thank the government of British Columbia for creating the University of British Columbia, and to both the federal and provincial governments for supporting it. I also appreciate the government-funded NSERC grant that paid my research assistant salary.

The governments would not be able to support UBC if taxpayers at both the federal and provincial level did not support education. My thanks to the citizens who believe in investing in the future – I hope to repay your investment.

I'd like to thank the people in the Software Practices Lab who made it fun to come in to the office, especially Arjun Singh, Terry Hon, Thomas Fritz, Apple Viriyakattiyaporn, and Brett Canon.

I'd like to thank Andrew Eisenberg, Brian de Alwis, Meghan Allen, and es-

pecially David Shepherd and Beth Simon for productive conversations about my research. Plugging runtime information info Mylyn's Degree of Interest Model was David's idea. Extra thanks to Andrew and Brian for their help with LaTeX!

Living at Green College made my life much more pleasant. In addition to feeding me well and allowing me a very short commute, the vibrant community meant that I wasted very, very few brain cycles on feeling lonely or bored. I would particularly like to thank Mika McKinnon for her role in making the community cohere.

Helen Surkova was our assistant, who watched seven developers do the same tasks over and over again and coded most of the logs. I'd like to thank her for being extremely reliable and cheerfully compliant at everything we asked her to do.

I chose UBC over Stanford in part because I thought the UBC Computer Science department would be more nurturing. I believe that was a correct assessment. I'd like to thank the UBC CS department for providing many resources, a building replete with collaborative spaces, and a congenial working atmosphere.

Gail Murphy, my supervisor, gave cheerful encouragement at every turn. She was particularly good at seeing the core issues and steering me to pay attention to those. (It is a peculiar convention of academic writing that everything needs to be written consistently in either the first person singular or first person plural. Let me say then, that every time "we" screwed up, it was me; every time "we" had a good idea, it was probably Gail.) I also greatly appreciate that she was patient when family issues, the ACM programming team coach duties, Green College Executive Committee duties, or just life in general slowed me down. Thanks, Gail.

And most of all, let me thank my husband, Jim DeLaHunt. Jim was willing to not just join me on this grand international adventure, but to support me enthusiastically. He fed me (when Green College didn't), he did my laundry, he encouraged me when I was down, and he had total faith that I would succeed. Thanks, Jim – for everything.

Chapter 1

Introduction

A large percentage of developers' time is spent navigating through code. Ko et al. [19] found that developers spend 35% of their time on code maintenance tasks simply navigating through the code. This navigation is a graph search problem: the classes, methods and fields are the nodes and various relationships – e.g., *declares*, *uses*, *implements*, and *inherits* – form the edges.

There has been some research on how developers explore the graph. Ko [19] suggests that code navigation consists of collecting a group of task-related code fragments, then navigating along relationships amongst the code fragments. Robillard et al. [30] analyzed five developers' work in great depth, including comparing the average path length between developers.

There have also been tools designed to make exploring the graph easier. Integrated Development Environments (IDEs) support making direct navigation (i.e. single-edge traversals). Mylyn [15] helps developers focus on a reduced set of code, essentially shrinking the graph that developers need to explore. NavTracks [34] and DeLine et al.'s wear-based filtering [8] use information about developers' past path navigation to weight the edges of a navigation graph to allow presenting some edges as probably more interesting than others.

However, we noted that there has been little work done that examines what graph traversal strategies developers use to navigate paths through source code, and which approaches are more successful. Do their searches look more like depth-first search (DFS) or more like breadth-first search (BFS)? Do developers spend too

long on "wild goose chases" if they use a more depth-first search approach? Or do they have trouble keeping track of what to look at if they try a more breadth-first search approach?

There are a number of papers from inside and outside the computer science field which suggest that breadth-first searches through problem spaces are more successful than depth-first searches. These include Schenk [33], Vans [40], Vessey [41], and Newfield [25]. These are in addition to the vast body of work in Psychology showing that having one hypothesis can make it harder to rationally evaluate data and other hypotheses [26].

Armed with that research, we set out to examine how better tools for keeping lines of inquiry distinct and easy to resume would affect search strategies: would more developers use more of a breadth-first search if they had better tools and training? Would they find the code they were looking for faster with BFS?

When looking for an exemplar for good navigational tools, we looked at Web browsers, as Web browsing has many similarities to code navigation. The Web is a graph with pages as nodes and links as edges. There have been a number of papers investigating Web browsing, which we will discuss more in Chapter 2.

We were struck by how modern Web browsers (e.g., Firefox version 2.0) have a one-path-per-tab paradigm while modern IDEs (e.g., Eclipse[1]) have strict one-file-per-tab models. Specifically, when a user requests a new Web page by clicking on a link, modern Web browsers replace the view of the old Web page with a new Web page by default; users must take an extra action to open the new page in a new tab. Modern Web browsers also keep a separate page navigation history for each tab.

Modern IDEs, on the other hand, open each file in a new tab and maintain only one file navigation history. One issue that we (and our colleagues) have noted and complained about was "tab spam" – having so many tabs open that we found it difficult to keep track of all the tabs.

We also wondered if the one-tab-per-file model made developers more likely to use a DFS navigating strategy, and hence potentially get stuck on "wild goose chases" more often. With modern Web browsers, it is easy to open several search

[1] http://www.eclipse.org/, verified 9 July 2008

results in several different tabs, which makes it easier to explore each path in turn. Having the different tabs visible might remind developers that there are other alternatives, which would perhaps encourage them to switch paths sooner. Furthermore, the penalty for switching is low, since it is easy to find exactly where in an exploration the developer had been before they switched – switch back to the previous tab, and you switch back to the previous exploration path.

We ran a user study with seven subject where subjects first performed two tasks with a standard IDE and then performed two tasks with a modified IDE (called Weta) with a more "web-like" tabbing behaviour. Before doing the tasks with Weta, developers were encouraged to use BFS strategies and shown how Weta's tabbing behaviour could facilitate BFS navigation.

We describe Weta in detail in Chapter 3, and the experimental design of the user study in Chapter 4.

Despite the training that encouraged people to use tabs to keep track of places that they should look at in the future, subjects instead used tabs to help them find places that they had visited in the past. Instead of putting places on a stack to explore later, they used the tool to "drop breadcrumbs" to let them remember the location from which they started exploring one path, so they could return to that location and start exploring a path. When coding a search algorithm, "remembering where you were" is a non-issue: returning from one routine automatically takes you to where you were before. Humans, however, need to remember where they chose to take one fork of a branch.

Intrigued, we looked more closely at the strategies and techniques that our subjects used for *finding* a starting point for a path of inquiry, *following* the path into related pieces of code, and *refinding* places that they had been before. We found that our subjects would find a point, then follow that path until they could go no further (a deadend). For finding starting points, our subjects had a variety of strategies which are described more fully in Sections 5.1 and 5.2.

For *following*, each subject used a variety of approaches for each path. It was most common for our subjects to trace statically, following *declares* or *uses* relationships. There was also quite a bit of scrolling through code. Finally, a minority of the paths used dynamic tracing, using breakpoints and the debugger to follow the flow of execution step-by-step. Following strategies are described in more detail in

section 5.3.

Solving the tasks required finding a good starting point and then not getting misled into following unhelpful paths. There were a number of specific lures that tricked developers into heading down unhelpful paths; different following techniques gave differing levels of resistance to the lures. In general, we observed that subjects had more success the more closely they followed the actual execution path. The effectiveness of the various finding and following strategies, as well as which lures tricked developers, is discussed in detail in Section 6.3.

In Chapter 7, we discuss what we believe would be profitable avenues for future research, including what tool support we feel would be useful. We believe that a tool that would present some run-time information passively to the developer would be very helpful. There are also several changes to Eclipse-specific UI elements that we feel would reduce errors. We have already submitted requests for these enhancements, and one has already been implemented. Information about those enhancement requests is in Appendix C.

Chapter 2

Related Work

Many researchers have considered how users interact with Web browsers and navigate through the Web. These efforts may help guide efforts in investigating program navigation because code, like the web, has a complex graph structure. We discuss Web navigation research and its relation to the work described in this thesis in Section 2.1.

Other researchers have also directly considered issues related to code navigation. Some of that research considers navigation processes at a coarse level; little focuses on the *finding* and *following* behaviour that is at the core of this thesis. We review code navigation literature in Section 2.2

2.1 Hypertext

The popularity of Web browsing has led to a number of studies of how people find things on the Web, including how to re-find things. Learning which techniques are successful when Web browsing can potentially inform IDE design.

To understand past research, it is important to understand the context of what capabilities people were using at the time that the papers were written. We thus review a bit of academic and commercial Web browsers' history in 2.1.1.

Web navigation research is a broad area; we focus on what researchers have found out about Web tabbing behaviour, Web bookmarks, and navigation strategies in 2.1.2. In our opinion, these are the areas most likely to inform IDE design.

2.1.1 Development of Tabbing Capabilities

Although many features of Web browsers, including bookmarks, have been relatively stable from the mid-1990s to today, tabbing capabilities have changed enormously. We will describe tabbing features by explaining current capabilities, then trace their development.

Modern Tabbing Capabilities

Modern browsers are implemented with an ordered list of stacks of Web pages that the user has seen. Each stack has a list of pages that the user has previously selected, ordered by when they were first selected. Each stack also keeps track of the current page, which divides the stack into "older" and "newer" pages – ones that were originally selected before and after the current page.

From the user's point of view, each stack is represented by a "tab". Users can select which stack/tab is active by selecting from a (usually) horizontal list with the (possibly partial) name of the current page in that stack, as shown in 2.1. (Each name is displayed inside a GUI element that looks like the tab on a file folder, hence the name.) This row of UI tab elements, the *tab bar*, is usually displayed immediately above the browser's main pane.

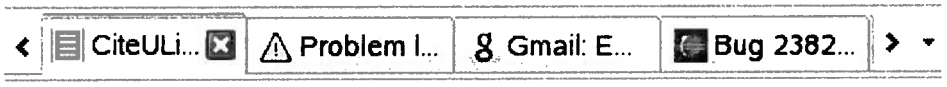

Figure 2.1: Browser tab bar

The Web browser's main pane displays the current page of the currently selected tab/stack. Clicking on a different tab/page name in the GUI changes the selection in the main page to the current page of the new tab/stack.

If there are more tabs/stacks in the list than can fit legibly (or partially legibly) in the tab bar then some of the tabs will be "off-screen" and not displayed. Users can scroll horizontally through the list of tabs (stacks) by clicking on arrow in in the tab bar, or select a tab/stack from a drop-down menu. The items in the menu correspond to the names of all of the current pages, one from each tab/stack.

Users can use a "Backwards" button to display the previous page in the current

tab/stack. The "Forwards" button takes them to the next page in the tab/stack. Browsers also allow users to select from one of two lists: the part of the stack before the current page and the part after the current page.

When users select a hypertext link, they can choose to open the new page "in a new tab" or, by default, "in the same tab". If they choose the latter, the pages that had been higher on the stack than the current page are removed from the stack, and the new page gets pushed onto the stack.

If the users choose to open a new page in a new tab, then a new stack is added to the list of stacks, with the new page as the only element in the new stack. From a UI point of view, the user sees that a new tab-shaped UI element appears in the tab bar.

Historical Tabbing Capabilities

Earlier browsers did not have as many features for keeping track of visited pages. It is not clear which Web browser fully implemented tabbed browsing first, in part because there is disagreement about which early features constituted "tabbed browsing". It appears that the process of coming to tabbed browsing involved independent discovery at several stages.

In 1990, WorldWideWeb, the first graphical Web browser, had only one stack and a Back button, but not a Forward button[1]. Pages seem to have all appeared in different panes.[2]

Mosaic, a very popular early Web browser, also had only one page stack, but users could navigate both forwards and backwards.[3]

Some popular culture assigns credit for the first tabbed browser to the InternetWorks browser in 1994.[4] A description of InternetWorks' tabs is of a simple list (not a list of stacks) of pages, with buttons to select pages from the list.[5] We call this a one-page-per-tab model, and it is very similar to Eclipse's current behaviour.

In 1995, Brown and Shillner published a description of DeckScape, which they describe as being based on a deck of cards metaphor. They explicitly point out that

[1] http://www.w3.org/People/Berners-Lee/WorldWideWeb.html, verified 8 July 2008

[2] as seen in http://www.w3.org/MarkUp/tims_editor, verified 8 July 2008

[3] http://www.wired.com/wired/archive/2.10/mosaic.html?pg=2, verified 8 July 2008

[4] http://en.wikipedia.org/wiki/Tabbed_document_interface#History, verified 9 July 2008

[5] http://adamstiles.com/2005/02/tabbed_browser_/, verified 9 July 2008

"Mosaic and the various Web browsers it has inspired use a depth-first navigational model" [3, p.1097]. DeckScape allows having various sub-panes (decks) inside the application pane, in a multiple-document interface (MDI) style.[6] Each "deck" has its own history stack, and users can have as many 'decks" as they like. Thus it has multiple selectable stacks of pages, but users select the stack by selecting a pane, not by pressing a button. They did not perform a user study, nor was their idea validated in the court of consumer opinion.

In 1998, Newfield et al. developed a browser, Scratchpad, with explicit support for breadth-first navigation [25] . Scratchpad kept a FIFO list of pending links; it placed links at the end of the FIFO when the user clicked on them, and presented the front link to the user when the user indicated he or she was done reading the current page. (A Back button is visible in the screenshot, but its use is not described [25, p.6].) Scratchpad's function is similar to Mosaic, but with one queue instead of one stack. The authors admit that there are use cases that are difficult with the Scratchpad approach.

The NetCaptor plug-in for Microsoft's Internet Explorer indisputably had what we consider modern tabbed browsing: multiple tabs, each capable of containing multiple files, and a per-tab history. NetCaptor was released on 3 Jan 1998[7].

Many people believe that version 4 of the Opera Web browser, released on 28 June 2000[8], was the first mass-market browser to have tabbed browsing built in. This is disputed[9] [10]. If you expanded one MDI sub-window to fill the entire MDI window, then you would have something equivalent to modern IDE tabbing – one pane, multiple tabs, but only one file per tab.

The MultiZilla plug-in for Mozilla, released April 2001[11] had all the features of modern tabbed browsing. Modern tabbed browsing appeared built in to the popular

[6]MDI applications have one main window, and independent sub-windows inside the main window, as described in http://en.wikipedia.org/wiki/Multiple_document_interface, verified 9 July 2008.

[7]http://www.netcaptor.com/, verified 9 July 2008

[8]http://www.opera.com/pressreleases/en/2000/06/20000627.dml, verified 8 July 2008

[9]http://www.hyperborea.org/journal/archives/2005/07/04/tabs-vs-mdi/, verified 9 July 2008, and http://weblogs.mozillazine.org/asa/archives/008433.html, verified 9 July 2008

[10]Our interpretation of the evidence is that Opera had an MDI interface, and each MDI sub-window could have a number of tabs associated with it, each with one file per tab.

[11]http://multizilla.mozdev.org/, verified 8 July 2008

Mozilla/Firefox line of browsers in October 2001.[12]

In 2004, Aula et al. described a tool, Session Highlights, which allows people to put thumbnails of Web pages into a chronological list. They said that "when using Session Highlights, the use of multiple windows or tabs is no longer necessary" [2, p.590] This statement does not reflect any appreciation for users wanting to keep track of paths of inquiry, nor for users wanting to go back to places that they did not know they would want to come back to. They did not report any user evaluation of Session Highlights.

Internet Explorer implemented tabbed browsing with version 7, which was released on 18 October 2006.[13] As of 2006, essentially all consumer-grade browsers had modern tabbing capabilities.

2.1.2 Web Navigation

Web surfers use tabs to find their way back to previously-seen locations, but use other strategies as well, e.g., bookmarks. Bookmarks would seem well-suited for finding their way back to previously-seen locations, yet research shows that Web surfers rarely do use bookmarks to return to locations, as we discuss below.

It would be nice to be able to discuss bookmarks, tabs, and navigational strategies independently. Unfortunately, the research tends to look at all of navigation together.

Refinding Locations

Briefly, the literature tends to show that people know about bookmarks, set bookmarks frequently, but only rarely navigate using bookmarks. There are many additional techniques that users employ to note Web pages for future reference.

In 1994, Catledge and Pitkow instrumented a browser, XMosaic, to capture user interactions [5] from all consenting users on 250 academic computers for a three week period. They found that following hyperlinks accounted for 52% of all document requests; 41% of requests were via the "Back" command. Only 2% of

[12] http://www.mozilla.org/releases/mozilla0.9.5/, verified 9 July 2008, and http://www.mozillazine.org/talkback.html?article=2054, verified 9 July 2008.

[13] http://www.microsoft.com/technet/abouttn/subscriptions/flash/archive/2006/10-18-06.htm, verified 9 July 2008

requests were via the hotlist (bookmarks).

In 1998, Abrams et al. [1] reported collecting surveys from 322 people and found that 94% of the users had at least one bookmark, and most (60%) had over 25. Users reported that bookmarks were easy to create, but difficult to manage. Some pointed out that if they did not organize their bookmarks, then their bookmarks list gave a time-ordered list of interesting sites they visited. They found the time-ordering useful for helping to keep track of what they were working on recently, but if they organized their bookmarks, they would lose that temporal information.

Newfield and Shillner implemented *dogears*, a form of temporary bookmark, in Scratchpad in 1998. Their rationale was that bookmarks "are often too persistent and place an undesirable management burden on the user" [25, p.2]. They did not evaluate Scratchpad.

In 2001, Jones et al. [13] noted a number of different strategies that 24 subjects used to find Web pages again. Bookmarks were one of many strategies, including sending email to themselves, printing out the Web page, saving the page as a file, pasting the URL into a file, adding a hyperlink to a personal Web site, re-doing a Web search, and typing in the Web address, perhaps accepting the browser's auto-completion suggestion. Bookmarks, however, were rarely used, in part because they were reportedly not easy to synchronize between computers, the page referred to might not persist, they could not provide context for the bookmark, they are poor reminders (since users did not check them frequently), and do not fit easily into existing organizational schemes.

In a 2004 follow-up study[4], researchers found that when nine subjects were asked to re-find Web pages that they had looked at a year earlier, 93% were able to successfully find the site on their first try, taking under a minute on average. Of the subjects who were successful on the first try, 18% used bookmarks. Of the first-time-successful respondents, at least 78% used techniques that did not require marking the location and keeping track of it for later: 42% did direct entry of a URL, 18% used a search site (e.g., Google), and 18% accessed the site via a different Web site.

2.1.3 Effect of Tabbing Capabilities on Browsing Behaviour

In 1995-6, Tauscher and Greenberg also instrumented XMosaic to record 23 users' browsing behaviour, then analyzed the logs [39]. Their major focus was on revisitation frequency, but they also identified seven browsing patterns, including "hub-and-spoke" – which they explicitly likened to a breadth-first search – and depth-first search. They suggested that "the excessive backtracking that results from hub-and-spoke and depth-first navigation styles could be reduced by a graphical overview map, or by retaining the index page within the browser window" [39, p.401]. Like Catledge and Pitkow, they found that the Back button accounted for a large fraction (31.7%) of all logged navigation actions, and the hotlist (bookmarks) and history list were used infrequently – 3% and less than 1% respectively.

In 2004, Aula et al. [2] collected questionnaires from 236 experienced Web users asking about finding and re-finding strategies. Their subjects self-reported, using a Likert scale, that they almost always used multiple tabs and multiple windows when browsing. For re-accessing information, they reported that they used a search engine often and very rarely wrote down a URL. Note that Aula et al. did not report on the use of tabs for re-finding. A clear majority of their respondents *set* bookmarks, but the respondents had a number of complaints about bookmarks, including that they never knew if they would use the bookmark in the future, they had difficulty transporting bookmarks across multiple computers, and it was difficult to organize bookmarks. Aula et al. did not report on how many respondents used bookmarks to *return* to pages.

Mindful of earlier studies with instrumented browsers ([5, 39]), Obendorf et al. did a followup study in 2004 to see how changes in the Web and browsers, such as more web applications, better search, and tabbed browsing, affected navigation [27]. They found that users used the Back button only about half as often as they did in the Tauscher and Greenberg study – only 14.3% of navigation events. Although they did not break out bookmarks separately, they said that the total of all "Direct Access" navigation styles, which include bookmarks, was somewhat less than in earlier studies. This paper strongly influenced our thinking early in this project.

In 2007, Obendorf et al. reported evidence that the reduced Back button use

was because many of the back-and-forths inherent in the older "hub-and-spoke" navigations were replaced by switching between tabs or windows. Instead of going back and forth between the hub and a sequence of different spoke pages, users would open many windows or tabs to navigate from the hub page. They found a strong inverse correlation between tab actions (open, select, and close) and Back button pushes. In Obendorf's study, 25 subjects used a Web proxy which collected logs of page requests, the triggering user actions, and central page characteristics. Of those subjects, 15 also consented to use a highly instrumented version of the Firefox Web browser which logged the use of 76 UI widgets.

2.2 Code Navigation

In order to do a more breadth-first search of the graph that is a code base, developers need to be able to reliably find locations where they branched, where they chose to explore one sub-graph of the code instead of another. There are a large number of tools designed to suggest to developers the next code element that they consider, such as [8, 34] and Rigi [37], but far less research has been done to help the developers keep track of interesting points where they have been. Even less has been done on keeping track of *path* histories.

2.2.1 Keeping Track of Interesting Points

Most of the research on keeping track of interesting points has been about bookmarks. Like in Web browsing, bookmarks would seem like an appropriate way to keep track of points, yet most of the research on code bookmarks has shown that developers very rarely use bookmarks. As Storey at al. said, "From our research with software engineers, we have observed that typical annotation mechanisms (notably bookmarks and tasks) tend to be ineffective at supporting software navigation, despite the intuition that they should be sufficient" [36, p.265].

Storey et al. [36] reported that only .02% of user view selections were bookmarks. The Eclipse Usage Data Collector,[14] which collected 1,829,889 usage data events from 1228 developers over 14 days found only 18 bookmark events from six developers – .001% of all events. Murphy et al. [24] reported that only five

[14] http://www.eclipse.org/org/usagedata/results.php, verified 9 July 2008

developers out of seventy-four (6%) in a long-term study used bookmarks.

Ko and Myers [20] found that three of their ten subjects used bookmarks, but ended up with multiple bookmarks, and had difficulty remembering what code the bookmarks represented.

There have not been many efforts to improve upon the bookmarking experience. Two of the few efforts to improve bookmarking explicitly are Zhang's IDE-Waypoint [43] and Storey et al.'s TagSEA [36].

IDEWaypoint attempts more than bookmarking. It combines some features of bookmarks with some of the task-management features of Mylyn.[15] "Waypoints" capture and store the author, the optional waypoint name, a hierarchical tag, an optional comment, the creation time, the set of perspectives, the set of views, the set of files open. Restoring a waypoint allows a developer to return exactly to the state and location of a particular task. In this manner, it has perhaps a closer relationship to Mylyn than to bookmarks. Zhang did not evaluate IDEWaypoint.

With TagSEA, developers can set "waypoints" by inserting special text similar to the Javadoc codes into code comments. The TagSEA tool creates a Tags Viewer that allows the developer to then re-find those locations. The TagSEA waypoints have more metadata associated with them than standard Eclipse bookmarks, including author, date, and a "tag" that can help to group related waypoints. Note that because the waypoints are put into the code comments, they are accessible to colleagues.

In a user study about TagSEA, Storey et al. found two of the three users mentioned using waypoints "to support navigation" and to "explore complex code paths" [36, p.270]. The quantitative data from the study did not reveal how often the subjects used waypoints for navigation aids versus for setting reminders of things to examine or fix later or for clarifying code function. Some of the developers in the study chose to continue using TagSEA after the study ended, which suggests that TagSEA is more useful than standard Eclipse bookmarks.

[15] We discuss Mylyn extensively in 4.6.1.

2.2.2 Keeping Track of Code Exploration Paths

In this section, we discuss two efforts to represent the range of approaches to keeping track of code exploration paths.

Developers can use the JQuery plugin for Eclipse to keep track of exploration paths as they make queries about elements in code [12]. JQuery displays a history of the queries selected in a separate pane, and suggests that the "explicit, unbroken, representation of the exploration paths... helps a developer to retain a sense of orientation within the context of an exploration task" [12, p.179].

A study evaluating JQuery's usefulness (and two other tools) was inconclusive, possibly due to large differences between task difficulties and/or subjects' skill levels) [6]. However, four of five subjects who used JQuery reported JQuery to be useful.

Sextant sets representing the exploration path as an explicit goal [32]. Like JQuery, the developer explores paths in a separate window from the code, in this case on a graphical representation of the past exploration path. When Sextant, JQuery, and Creole (an Eclipse plug-in version of SHriMP [22]) were compared in a study with five subjects working on four tasks, the authors reported that Sextant's representation of the exploration path as a graph worked better than JQuery's hierarchical representation, and that Creole users frequently got lost [32].

2.2.3 Breadth-First Strategies

There has been some research from several domains that suggests that people find solutions faster if they use a more breadth-first search strategy than depth-first search, or at least if they keep multiple hypotheses in play.

Schenk et al. [33] examined 25 systems analysts and found that experienced subjects who were rated highly by their management verbalized hypotheses more often, and explicitly discarded hypotheses more often than those who were less highly rated by their management or who were less experienced. Discarding hypotheses frequently corresponds to exploring to a shorter depth in the solution-space graph, while holding on to hypotheses until they are proved false epitomizes depth-first search.

Vans also found experienced programmers discarding hypotheses. Vans found

that among four experienced programmers, 45% of hypotheses were abandoned, 40% were confirmed, and 14% failed [40].

Vessey reports, based on verbal protocol analysis of sixteen programmers, that programmers who are not as good at solving a problem make worse first guesses than those who do well, but also that the poor performers "seize on a hypothesis and find great difficulty in rejecting it; that is, they continue to search for support for it long after they have sufficient knowledge to discard it" [41, p627]. This suggests a DFS approach has drawbacks.

Newfield et al. also imply that DFS has limitations. They mention that "most current web navigation techniques are based on the paradigm of depth-first traversal" [25, p.1], in part because "backtracking is often difficult since the context in which that page was being viewed has been forgotten. Frequently, entire sub-trees are accidentally skipped because users do not remember what their intentions had been when an earlier hub was last being viewed" [25, p.1].

Klahr and Dunbar [16] asked ten subjects to figure out what a specific button on a programmable toy did, giving them the toy to try their guesses out. They found that subjects who spent some time generating as many hypotheses as they could, before attempting the task, were vastly faster (6.2 minutes on average) at correctly figuring out the function of the button than those who did not have a dedicated hypothesis-generation period (19.4 minutes on average). Although the paper did not explicitly say how long the hypothesis generation stage took, a request for clarification revealed that that stage "took only a couple of minutes"[16]

This is consistent with the phenomenon known in Psychology called *confirmation bias* [26]. Essentially, if a subject has a hypothesis about something, he or she will trust data more if it agrees with that hypothesis than if it disagrees with the hypothesis. It might be that generating multiple hypotheses (as in the Klahr and Dunbar study [16]) reduces confirmation bias.

In contrast, Gugerty and Olson [9] found that experienced programmers found bugs faster than novice programmers merely by generating better hypotheses than the novices. They found that novices tended to add bugs to code in the course of debugging while the experts did not. Otherwise, Gugerty and Olson found no

[16]Private email message from David Klahr to Kaitlin Duck Sherwood, 28 March 2008.

difference between the experts' processes and the novices' processes. Their study had 18 novices and six experts.

2.2.4 Meta-behaviours

Solloway et al. [35] found that developers who undertook a "systematic" program understanding strategy, namely reading the entire source code line-by-line, were more successful. Unfortunately, given the size of modern software projects, that would now be a massive undertaking.

Robillard et al. [30] found that successful developers did more focused searches, recorded their findings, perpared a modification plan, and kept to the plan while implementing the change. Unsuccessful subjects used a more opportunistic, ad-hoc approach.

2.2.5 Omniscient Debuggers

One way for developers to navigate through code is via dynamic tracing, using the debugger to step, line-by-line, through an actual execution trace. *Omniscient debuggers* extend that ability to allow developers to step *backwards* along a trace as well as forwards. Omniscient debuggers have appeared in both academic literature [11, 17, 21, 28] and as commercial products (e.g., UndoDB[17] and CodeGuide[18]). Omniscient debuggers provide a powerful tool for navigating through code, but must record every variable state change, along with its location in the code. They thus can require collecting a very large amount of data.

[17] http://undo-software.com/, verified 9 July 2008

[18] http://www.omnicore.com/en/, verified 9 July 2008

Chapter 3

Weta

As explained in Chapters 1 and 2, we believed that developers would profit from an IDE that used a one-path-per-tab model instead of a one-file-per-tab model. To that end, we developed a modified version of the popular Eclipse IDE, giving it a different tabbing model.

In this chapter, we explain how modern IDEs and Web browsers implement tabs, discuss some problems with current tabbing behaviour, and describe the new tabbing features we put into our version of Eclipse (called Weta, for WEblike TAbbing). We discuss the UI choices that we faced and how we chose to resolve them. Finally, we discuss briefly some of the implementation issues.

3.1 Modern IDE Tabbing Behaviour

Eclipse is representative of common IDEs in its tabbing behaviour, using a one-file-per-tab model. (Although we have not examined every single IDE in existence, we do not know of any that use a different tabbing model.) If the user directs Eclipse to open a file, Eclipse will open the file in a new tab. Switching tabs switches the file that is open in the main (source) view. If the user requests something in a file that has already been opened, the tab corresponding to that file is given focus, and that file's contents are displayed in the source view.

Figure 3.1 shows a view of Eclipse after the user opened four files – `A.java`, `B.java`, `C.java`, and `D.java` – in alphabetical order. The large central view

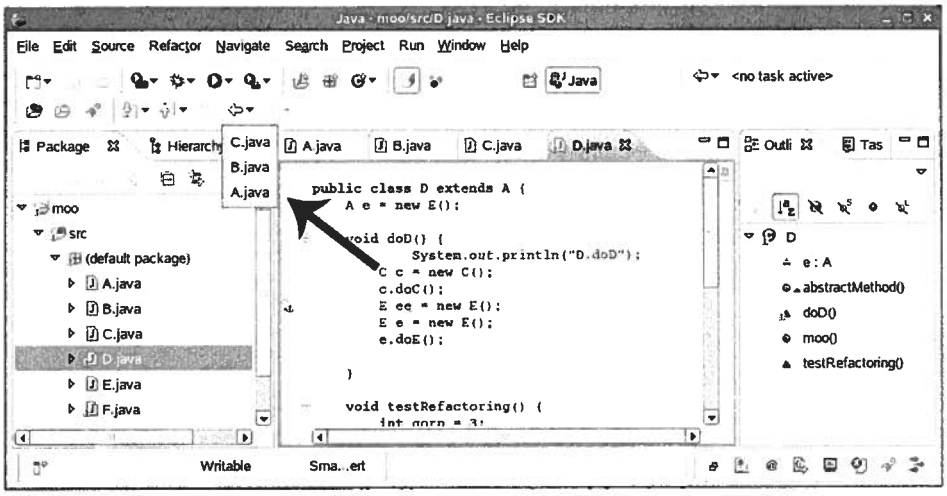

Figure 3.1: Standard Eclipse tabs with global history

shows the most recently opened file (`D.java`). Four tabs associated with four files sit just above the `D.java` source. The tab for `D.java` is highlighted, indicating that `D.java` is the file currently in the source view.

Selecting a different file from the drop-down history de-highlights the current tab, highlights the new tab, and replaces the source in the central view with the source of the new file. For example, if the user selects `C.java` from the history list in Figure 3.1 the `C.java` tab will be colored, the `D.java` tab will lose its coloring, and the `D.java` source in the center view will be replaced by the contents of `C.java`.

The navigation history is global – navigating forwards and backwards through the history causes different tabs to become active, and all the locations that were navigated to are in the history, even files that were subsequently closed. (Selecting a closed file from the history, either implicitly by using the Back button or explicitly by selecting directly from the history list, will reopen the file in a new tab.)

The history list (indicated by the arrow) shows the files that were opened but are not in focus. `D.java` does not show up in the history list because it is the current file.

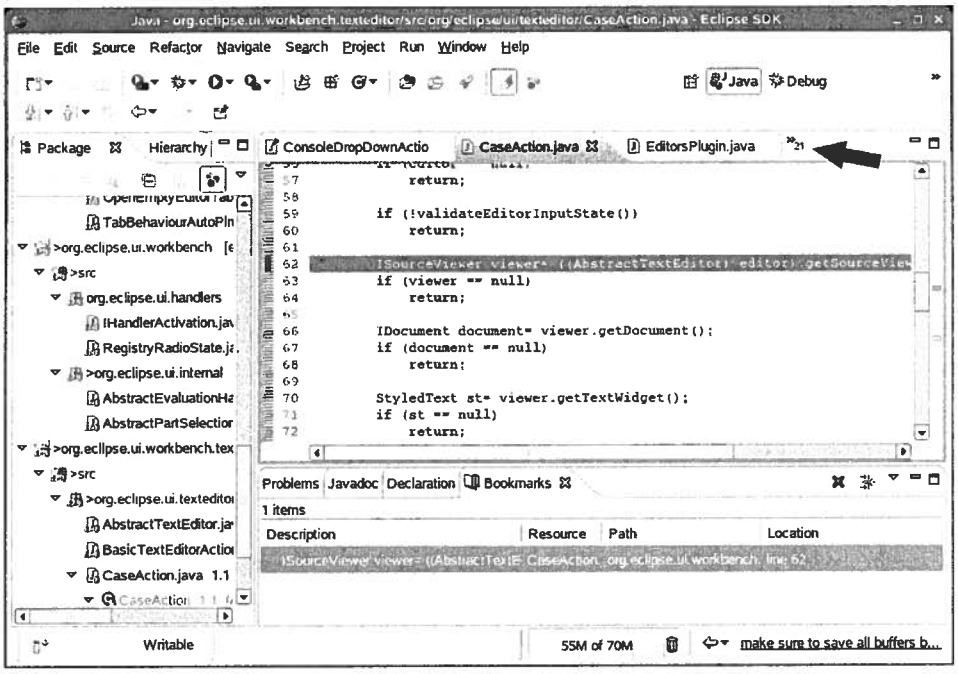

Figure 3.2: Standard Eclipse "tab spam" - 21 files hidden

3.1.1 "Tab Spam" in Eclipse

Even before we did our user study, we had seen ourselves and heard from others that developers frequently got overwhelmed by an overabundance of tabs in Eclipse. The tab width is a function of the length of the name of the file. If the user opens a lot of files, then their tabs can "overflow" the width allotted to the tabs. In Eclipse, if there are too many tabs to fit in the width of the source window, only a subset will be visible, with a double-arrow to the right of the tabs with a number indicating the number of files that are not visible. (The "21" in Figure 3.2 means that there are twenty-one files that are not visible.)

If a user explicitly requests a file that is open but not visible (e.g., by performing a search), then Eclipse will rearrange the tabs, making one of the visible tabs invisible and making the tab for the requested file visible. The user can also click on the >> button to get a list of all the open files, with the invisible open files listed in bold.

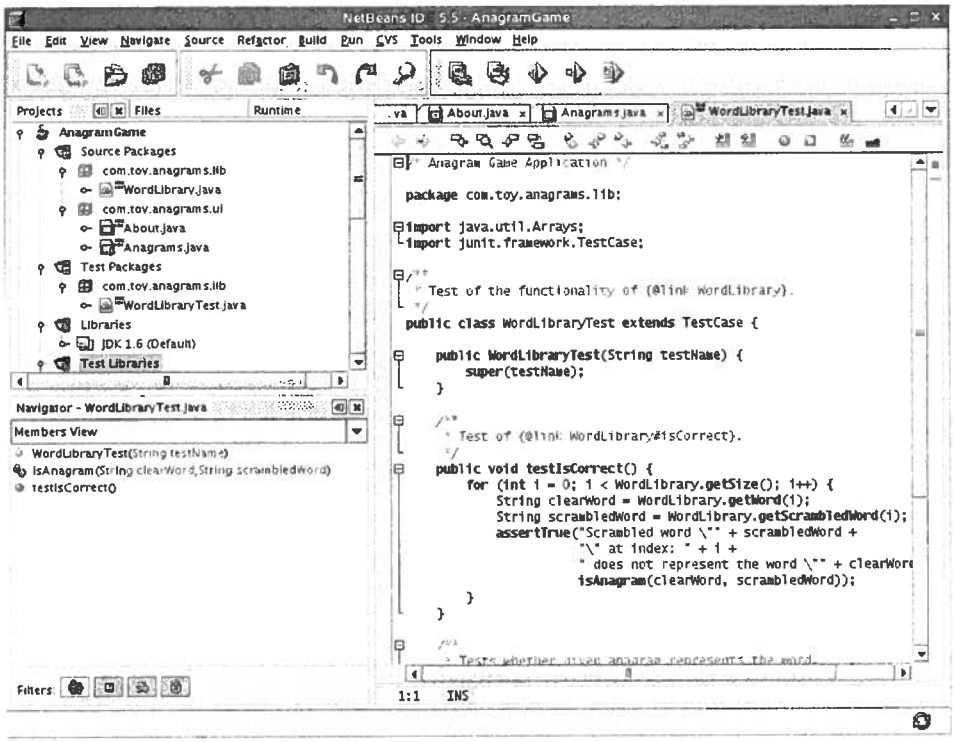

Figure 3.3: Tabs in Netbeans

3.1.2 Other IDEs' Tabbing Behaviour

Netbeans,[1] another popular Java IDE, does something similar to Eclipse when there are more tabs than will fit on one line. As seen in Figure 3.3, Netbeans also shows only as many tabs as will fit. It has a "down arrow" (visible as the black triangle near the upper right) that is similar to Eclipse's double-arrow, and has left/right arrows to scroll the tab bar horizontally. Because its interface is functionally identical to Eclipse's, presumably it has problems with "tab spam" as well.

The IntelliJ IDEA IDE[2] has a slightly different behaviour. IntelliJ IDEA starts another parallel row of tabs when the tabs "overflow" a row. This keeps all tabs visible, but at the expense of screen real estate. IntelliJ also has a preferences

[1] http://www.netbeans.org/

[2] http://www.jetbrains.com/idea/

option to only show one row of tabs, so presumably there are developers who dislike having multiple rows of tabs.

3.2 Web Browser Tabbing Behaviour

Currently-available Web browsers like Firefox 2 and Internet Explorer 8 have a different tabbing paradigm than IDEs, what we will call a one-tab-per-path model instead of a one-tab-per-file model. If the user requests the browser to open a new page, then by default the browser will open that page "in the same tab": the new page replaces the old page and the title of the new page replaces the title of the old page. The position of the highlighted tab stays the same.

The user can also issue a slightly more complex command to specifically request that the file be opened in a new tab. In the case of Firefox 2 and Internet Explorer 8, a single left-click will open a page in the same tab; either a single Control-click or a single middle-click will open a page in a new tab.

In those Web browsers, there are "active elements" that the user can select to switch views of Web pages. If a user clicks on an active element (a hyperlink, bookmark, or history element), then a view of the new page replaces the old page in the current tab. If a user Control-clicks on an active element, then the new page loads in a new tab. In Web browsers, each tab has its own history, and users can navigate forward and backwards through pages that have appeared in that tab and that tab only.

3.3 Weta Tabbing Behaviour

Weta's tabbing behaviour is much like Web browsers' tabbing behaviour.

Like Web browsers, Eclipse and Weta have active elements – Java classes, methods, and fields – that the user can select to switch views of source code. Using Weta, if a user double-clicks on an active element, then a view of that element's declaration replaces the old source in the current tab. If a user Control-clicks on an active element, then the new declaration loads in a new tab. In Weta, each tab has its own history, and developers can navigate forwards and backwards through files that have appeared in that tab and that tab only.

Figure 3.4 the result of opening `A.java`, then `B.java` in one tab, then `C.java`

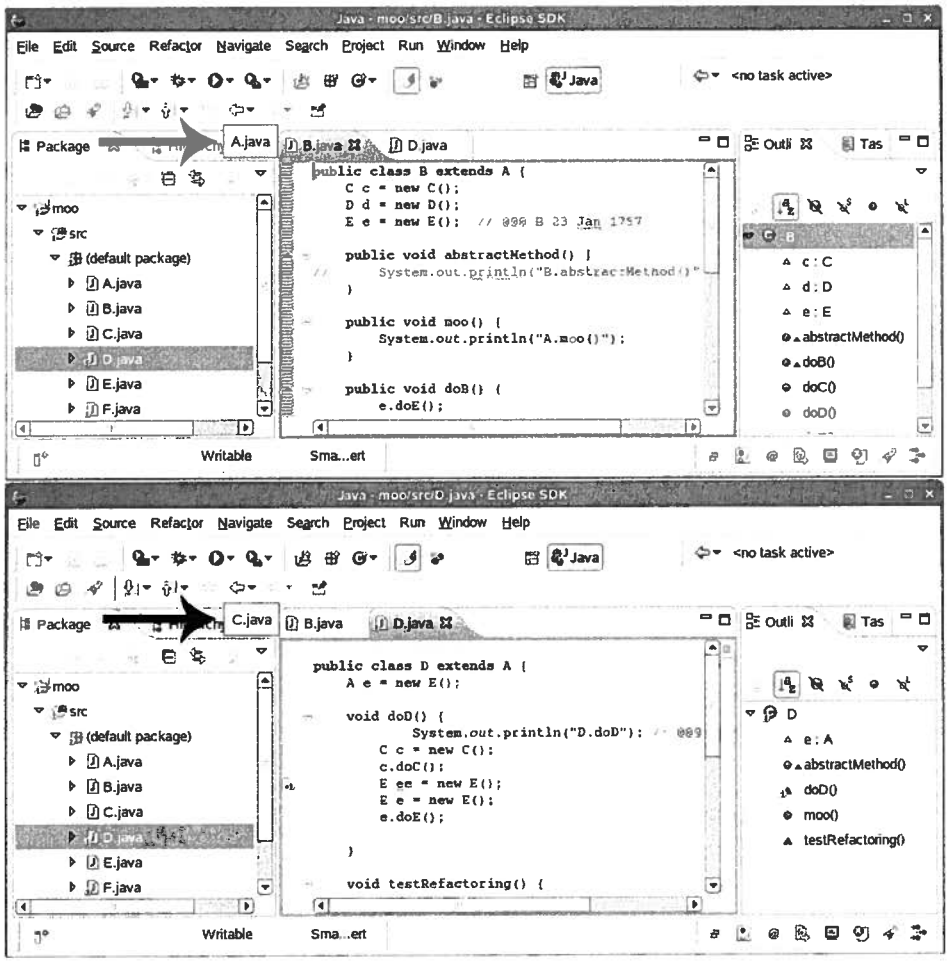

Figure 3.4: Weta tabs with per-tab history

and `D.java` in another tab. When the first tab (which has `B.java` in it) has focus, then the only file in the drop-down history list is `A.java`. When the other tab has focus, then only `C.java` is in the drop-down history list.

3.3.1 Hyperlinking Behaviour

In the standard Eclipse, Java elements (fields, methods, and classes) in the source become underlined and blue if the user holds down the control key. Clicking on

the underlined blue element causes Eclipse to open that element's declaration in a new tab.

In Weta, Java elements are always underlined and in blue. If the user double-clicks on the Java element, then Eclipse opens that element's declaration in the *same* tab. To open an element's declaration in a new tab, the user needs to Control-click on the hyperlinked Java element.

3.4 Design Decisions

There were a number of choices that we had to make regarding Weta's behaviour where we could not look to existing paradigms for guidance. We do not pretend to believe that we got all of the choices right, but we include our thought processes here to help future researchers work through the same issues.

3.4.1 Dirty Files

One issue that Web browsers could not give us guidance on was how to handle dirty (i.e. changed but not yet saved) files, since Web pages do not get dirty. (At least not how the term is used in Computer Science.) There are a number of instances where this comes up.

In Eclipse, there is a "Save File" command which saves the file in the tab that has focus. We chose to make the "Save File" command save all of the dirty files in a tab – including those that are not visible. Thus if you open file A, edit A, open file B in the same tab, edit B, and then issue a "Save File" command, both A and B are saved.

There are other choices that we could have made. We could have chosen to only save B. We could have also chosen to force the user to save or close A before opening B. We did not study the effects of that choice in our study, as that did not seem germane to the study of navigational strategies.

As it happened, none of our subjects ever tried to open a new file "over" a dirty one, so the issue never came up.

One might suppose that it would be tricky to decide what to do when closing a tab if some files in a tab were clean and some were dirty. Fortunately, the standard Eclipse has a metaphor for that, which it uses if you close all the tabs at once

(e.g., if you exit Eclipse). If you close many files at once, Eclipse puts up a dialog box asking which files you wanted to save and which you did not. We used this metaphor.

3.4.2 Tab History

It was not clear whether opening a new file would only add to the tab's history or whether it could displace another file. For example, if the user opens file A, then file B, then went back to A, then opened file C, should the tab's history show the history of all the files opened in the order that they were opened (**A, B, A, C**), or should it use a stack model (**A, C**)? Since Internet Explorer 7 and Firefox 2 both use a stack-based model, presumably the general populace is able to deal with a stack-based model, so we chose a stack-based model. It is reasonable to expect that computer programmers would be equally adept at a stack-based model, but that was not explicitly tested in this study.

3.4.3 Click Choices

All of the choices we had for how to distinguish between clicks to open in a new tab versus opening in the same tab had drawbacks. No matter what modifier keys we assigned to what actions, we would be inconsistent with either Web browsers or Eclipse or both. We had to choose the least bad alternative.

Eclipse is even inconsistent with itself – the behaviour is different depending upon whether the user is clicking on a hyperlinked Java element in a source view or on an element in one of the navigation views. Single-clicking on something in the Package Explorer, for example, opens it in a new tab; to do the same in a source view requires Control-clicking.

Table 3.1 shows how modifier keys affect clicks on hyperlinks in Web browsers, standard Eclipse, and in Weta. As mentioned above, standard Eclipse is inconsistent between navigation view and source hyperlinks.

We settled on using a Control-click to open in a new tab (like Firefox) and a double-click to open a file in the same tab. This meant that our subjects had to use a single-click to select a word. There is a slight but noticable delay between clicking on the word and the whole word becoming selected. It is slower than double-

Table 3.1: Comparison of click behaviours in Web browsers, Eclipse, and Weta.

	single-click	**double-click**	**Control-click**
Web browsers	normal open	normal open	open in new tab
Eclipse - source views	delayed select	select word	normal open
Eclipse - navigation views	select	normal open	multi-select
Weta - source views	select	normal open	open in a new tab
Weta - navigation views	select	normal open	open in a new tab

clicking, perhaps because Eclipse spends some time to bring up the javadocs for that element when you single-click, but not if you double-click.

Using Control-click for open files in a new tab meant that our subjects could not use multi-select. (For example, the subjects could not select multiple lines in search results to delete.) We did not observe our subjects attempting to do multi-select in this study, even on the non-Weta tasks.

3.4.4 Discarded Click Choice Alternatives

There were many choices that we could have made; this section describes why we discarded those choices.

We originally wanted to use Shift-Control-click to open a file in a new tab. Unfortunately, to distinguish between Control-click and Shift-Control-click would have required modifying the C code in the underlying graphics framework (e.g., gtk on Linux). We decided that was an excessively challenging task for the incremental value we would get from it.

We flirted briefly with using a single-click to open a Java element's declaration in the same tab (like Web browsers), but discovered quickly that this made editing individual characters in a Java element difficult. This is different from Web browsing, where users do not ever have the opportunity to edit a hyperlink.

We considered using the middle mouse button to open in a new tab. Unfortunately, the middle mouse button is used to paste in Linux. Furthermore, in an informal poll around our lab, many people did not even know how to click the middle mouse button on their mice. (Most people had a mouse with two buttons

and a scroll wheel in between them, the scroll wheel acting as the middle button.) Several who we asked were not even sure that they would be able to click the scroll wheel without scrolling.

We considered using Alt-click, but alt-click keystrokes are normally captured by Linux desktop managers. (If you hold down the Alt key and click anywhere in a window, you can move the window around.)

3.5 Implementation Difficulties

Modifying Eclipse presented some challenges. The code base is very, very large, as mentioned in the introduction to this chapter. Because we were tampering with very fundamental features of Eclipse, our code changes were very broad and far-reaching. Our changes were very much cross-cutting concerns, touching forty-nine classes in nine packages. This section lists the major changes required.

Every single place that could open a tab needed to bifurcate the tab opening code to allow both open-in-new-tab *and* open-in-same-tab actions. Tab-opening code was found in code related to the hyperlinked Java elements, the hierarchical package/class/method browser, the type hierarchy viewer, the call hierarchy viewer, and the search results.

Eclipse has a very strong model that there is only one file per tab. Eclipse does have a feature where the user can restrict the number of tabs that Eclipse will create, but it does not allow more open files than there are tabs. It closes the old file before it opens the new file in the tab. This simplifies the reference counting of dirty files, but means that you cannot open a new file in a tab "over" a dirty file. This meant that if the user was not scrupulous about saving files, Eclipse would pop up dialogs on a regular basis that would distract from the job at hand. In order to coerce Eclipse to allow opening new files "over" dirty ones, we had to augment the reference counting of dirty files.

When there are more files than tabs, Eclipse uses a Most Recently Used strategy to assign tabs, again with one file per tab. We had to write code to specify explicitly which tab to put the new file into, wresting control away from the MRU strategy.

Because we chose to use Control-click (for reasons given above) to open files in a new tab, we needed to disable multi-select in a number of classes.

There was a somewhat obscure bug[3] in Eclipse: scrolling does not set the file position properly in the state that is saved when a tab is reused. This means that if someone opens file `A.java`, scrolls to the end of the file (but does not click anywhere), opens some large number of other files, and then goes back to `A.java`, then the user will see the cursor at the *beginning* of `A.java` and not at the end. Although it is a bug in Eclipse, it come up much more frequently in Weta than Eclipse, so we needed to fix it.

There were a number of cases where, because one package could not see information in another package, we needed to move methods up into an interface that could be seen. This meant writing stubs for that method in all the classes implementing that interface.

There were a number of cases where in the stock Eclipse, a class did not need to reveal its inner working, but Weta did need to know about the inner workings. In several cases, it was expedient to change private or protected methods to protected or public methods.

As mentioned above, some of the code that we wanted to modify lived deep in the guts of the underlying GUI framework. The code that dealt with multiple selection is in the GTK framework, for example.

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=238202

Chapter 4

User Study

We did a qualitative user study of seven professional programmers working on four tasks in medium-sized, unfamiliar Java code bases, two tasks with Eclipse and two tasks with Weta. Our initial intention was to focus on how breadth-first navigation, made possible by Weta, compared to stock Eclipse. Our study was structured towards that end, but because our subjects used depth-first strategies even with Weta, our study ended up being much more qualitative.

4.1 Subjects

We recruited seven subjects by posting an invitation to participate on a local Java users' group mailing list and by requesting introductions to potential subjects from people known to us.

To be eligible for the study, a subject needed to be a professional Java programmer with at least six months of experience with Java and Eclipse or an Eclipse-based IDE (e.g., WebSphere), be at least 19 years old, and be proficient in the English language.

The subjects had a minimum of nine months and a maximum of seven years experience with Eclipse according to self-reports. All were male, and all were Caucasian. All lived and worked in the greater Vancouver, British Columbia area. The subjects also reported between nine months and seven years of experience with Java, and between nine months and twenty years of programming experience, as

shown in Table 4.1. (All names are pseudonyms.)

Table 4.1: 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
Jim	0.75	0.75	0.75
Mark	>10	5	6
Peter	20	7	6
Steve	12	6	6
Tom	5	4	4

We compensated the subjects with a gift certificate to an online retailer, valued at CDN$20. They did not need to complete the study to receive the certificate.

4.1.1 Subject Differences

The individual characteristics of the subjects are important for interpreting the results. It would be convenient if all the subjects had the same professional backgrounds, personalities, strengths, and weaknesses, but that would not be realistic, nor would it capture the richness and nuance of the human experience of coding.

Dave's workplace used a lot of automatically-generated code that makes interpreting results from searching for references to Java elements difficult. As a result, Dave was unaccustomed to the References command (described in Section 5.3.1). Thus, he tended to do searches where the other subjects used the References command.

Jim had more than the minimum number of months of professional programming – nine months while we required six. However, he did not have any academic computer science background, had never programmed (in any language) before getting thrown into a programming job assignment nine months earlier. He had never seen a non-GUI program, and had only used his company's customized version of Eclipse. In short, he had a very limited experience base compared to the more diverse coding and environment terrain that the other subjects had explored.

He got stuck in unusual ways.

We debated at length whether we should include or exclude Jim's results from the study, since he was such an outlier. On the one hand, he was inexperienced and probably not representative of professional software developers. On the other hand, it was illuminating to see how his mental models and strategies diverged from the more experienced subjects. He also attempted strategies that other subjects tended to avoid (especially searching). His difficulties with those strategies highlighted why more experienced subjects had learned to avoid them. We ultimately elected to include his navigation data sometimes, but to note when we excluded it. Because he was such an outlier, we did not transcribe or use his interview.

Steve had more than the minimum number of months of experience with Eclipse, but they were not all *recent* months. He also seemed to have particular difficulty using our installation of Eclipse. For example, he had difficulty several times finding the menu command for search, a command he performed frequently. However, of all the subjects, Steve seemed to have the strongest ability to reason about a problem. He frequently verbalized hypotheses which indicated an understanding of the deep structure of the program code. More than any of the other subjects, he had a tendency to stop, think about the problem and the information he had gathered so far, and realize what the solution must be. This strength allowed him to succeed using navigational strategies that did not lead to success when other subjects tried them.

Peter was also a strong developer, working slowly and surely and tooking the time to understand how the code worked. When time was called, he was always deliberately making progress towards the goal, not flailing around aimlessly. We had the sense that he *would* finish if given a little more time. In fact, he did finish the Output task (which only Steve finished in time), though we later discovered we had accidentally given Peter an extra minute.

Bob, Mark, and Tom, were all experienced developers. They did not have any noteworthy characteristics germane to this study.

4.2 Procedure

Each session consisted of eight events.

1. We introduced the subject to the study, and went through the consent form.

2. We checked that the subject understood that one can control-click on a Java element (type, method, or class) to open the declaration of that element. We checked to make sure the subject was familiar with that functionality.

3. We asked the subject to work on two brief programming tasks with an unmodified version of Eclipse. (We describe all the study's tasks in Section 4.5.) The subject was allowed twenty minutes for each task.

4. We briefed the subject on how some research suggested that breadth-first-ish search techniques were better than depth-first-ish searches. We worked from a script that is shown in Appendix B.5.

5. We introduced the subject to Weta and trained him on how it could be used to effect a more breath-first-ish search strategy. We worked from a script that is shown in Appendix B.6.

6. We asked the subject to work on two tasks using Weta, again each for twenty minutes.

7. We asked the subject to fill out a brief questionnaire which asked about prior experience with Java and various tools, and opinions on Eclipse behaviour vs. Weta behaviour, BFS vs. DFS, etc. The questionnaire is given in Appendix B.9.

8. We interviewed the subject for approximately 20 minutes.

All of the documents relating to the tasks, the training materials, and the questionnaire are included in Appendix B.

For all four tasks, the subject worked with a hired undergraduate. Each subject was asked to treat the undergraduate like a new hire at the subject's company who was good at programming from scratch, knew Eclipse, but "was really bad at finding things in other people's code". We asked the subject to demonstrate to the assistant how to find things in an unfamiliar code base.

The assistant was instructed to give very little programming assistance, but to frequently prompt the subject to verbalize what they were doing and why. We discuss using the assistant in Section 4.3.

4.3 Assistant Interactions

Using an assistant was our second choice for eliciting verbalizations from our subjects. Originally, we had planned to have pairs of subjects do pair-programming. However, this would have required finding twice as many subjects. It was difficult to find just the seven subjects we did find. We therefore hired an undergraduate to assist us.

We instructed the assistant to be friendly but to act "a little bit stupid". We asked her to avoid revealing any knowledge of the code base, and to not give substantive help. We explicitly granted her permission to point out syntax errors and to give gentle guidance if they took a task that was just completely in the wrong direction. We asked her to ask the subject from time to time what he was doing if he did not explain it, particularly when it came to navigation choices.

4.3.1 Data Gathering

In addition to the questionnaire and interview, we recorded a video (screen capture) of the display and audio of the subject, assistant, and researcher. We started gathering this data as soon as the consent form was signed, and stopped it after the interview was done.

In addition, the researcher observed and took notes. The observer positioned herself behind the subject and the assistant in a manner where she could see the monitor.

For the screen capture and audio, we used commercial software. We used the Mylyn UI Usage Reporting Plug-in to log user interactions, as described in 4.6.2.

4.4 Code Bases

We wanted to see how well our subjects navigated with unfamiliar code, so we deliberately chose medium-sized code bases that our subjects were not likely to

have seen before. We also did not want learning effects to taint the comparison of the two tools, so we used two very different code bases: JHotDraw[1] and ProGuard.[2]

Both JHotDraw and ProGuard used the English language for element names, comments, package names, output, etc.

4.4.1 JHotDraw

Two of the tasks used the JHotDraw 6.0b1 code base, and were based on tasks used in Safer's work [31]. JHotDraw is a Swing-based drawing package, that – as modified – has 19,928 standard lines of code as measured by SLOCCount.[3] JHotDraw is as much a drawing framework as it is an editor; it has two different graphics editors bundled with it. One, NetApp, allows users to rubber-band connections between nodes. The other, JavaDrawApp, is a drawing application similar to MacPaint.

Unlike in Safer's experiment, we did not give any training on how the JHotDraw code worked because we were interested in seeing how our subjects coped with completely new code bases.

Some of the subjects said that they had seen Swing before, but not for a long time. While we did not ask the participants if they had seen ProGuard or JHotDraw before, none of the subjects showed any signs of recognition of either of them.

4.4.2 ProGuard

Two of the tasks used the ProGuard 3.8 code base, and are new for this user study. ProGuard is a byte-code obfuscator: its main function is to make it more difficult for third parties to decompile and understand a Java bytecode file. Unlike JHotDraw, ProGuard's GUI is not integral to the application; it is merely used for configuration. We stripped out the GUI for this user study, using only the textual command line/configuration file interface. As-modified, ProGuard has 29,602 standard lines of code as measured using SLOCCount.

Although we assumed that everybody would have seen a drawing program before, we presumed that they would be unfamiliar with byte-code obfuscators in general or ProGuard in particular. We thus gave a some background on what Pro-

[1] http://www.jhotdraw.org/, verified 9 July 2008

[2] http://proguard.sourceforge.net/, verified 9 July 2008

[3] http://www.dwheeler.com/sloccount/, verified 9 July 2008

Guard was designed to do. Furthermore, because ProGuard does some things that are unrelated to byte-code obfuscation, we told subjects that code having to do with those areas, optimization, shrinking, and pre-verification, would not be involved in the tasks.

4.5 Task Details

Users always did two tasks from one code base using the stock Eclipse, and then two tasks from the other code base using Weta. Four subjects did the JHotDraw tasks first; three did the ProGuard tasks first. We also varied which task of each pair the subject did first. Table 4.2 shows the order of the tasks used.

Table 4.2: Task order

Name	Arrows	Size	Output	Obfuscate
Bob	1	2	3	4
Dave	2	1	4	3
Jim	4	3	2	1
Mark	3	4	1	2
Peter	3	4	2	1
Steve	1	2	4	3
Tom	1	2	4	3

We did not counterbalance Eclipse and Weta because we were more interested in BFS versus DFS navigation, as opposed to Eclipse vs. Weta. If some subjects used Weta first, then some of the subjects would have the BFS training when using Eclipse. We did not want to confuse the results in that manner.

The subjects were allowed to work on each task for twenty minutes. In a few cases, where the subject was obviously very close to a solution, we allowed the subject to continue working for a little longer (less than two minutes).

Arrows Task

The Arrows Task revolved around a bug in one of the JHotDraw drawing applications. Executing a menu command to remove the arrow tip from the end of an

arrow (improperly) did not remove the arrow tip. Subjects were asked to fix this bug.

Fixing the bug required finding the method `PolyLineFigure.setAttribute`, recognizing that code for setting the ending arrowtip was missing, and adding a small amount of code, using the start arrowtip code as an exemplar.

Everyone but Jim eventually figured out that code was missing. The subjects' strategies and effectiveness on the Arrows task are discussed in detail in Section 6.3.5.

Size Task

For the Size Task, subjects were asked to finish a partially-completed upgrade in a JHotDraw application. In a multi-pane drawing application, the text "Active View Size:" appeared in the application (outermost) frame's status bar, but no dimensions appeared. They were asked to add in the numerical dimension of the resized pane.

Subjects needed to find a place in the code that got triggered every time a resize event took place and that could pass the size information to the application.

None of our subjects fixed the bug. Many had difficulty stemming from getting started in a superclass and never realizing that the important class was a subclass. This aspect of the Size task is discussed in more detail in Section 6.3.1.

Obfuscate Task

For the Obfuscate Task, subjects were asked to add the ability to use a different old-name-to-new-name scheme. The class for generating a new name was provided; the subject needed to add code to three different methods in three different classes in order to recognize the command-line argument, plus modify another method in another class to conditionally use the provided name factory class.

Most subjects found the right place(s) to parse and store the command line argument, and were on their way to figuring out how to connect up the name factory. Subjects did not seem to have particular trouble on this task aside from needing more time to find and edit four different files.

Output Task

For the Output Task, subjects were asked to fix a bug where the number of fields in a class was being written out to the output .jar file as a four-byte integer, where it should be written out as a two-byte integer. This required changing a single line of code in a very straightforward manner. Because the change was so straightforward and obvious, once they found the right spot, a comment in the code that they did not need to actually make the change. All they needed to do was to find the location.

Finding the line turned out to be extremely difficult because of misleading naming of code elements, as described in 6.3.4. One might be concerned that this is not representative. However, this misleading language is in ProGuard source as downloaded, and was only discovered *after* we modified the code to insert the bug.

4.6 Technical Specifications

The Eclipse used in the first two tasks was Eclipse 3.3.0 with Mylyn 2.1 and the Mylyn UI Usage Reporting Plug-in 2.0. Weta was built on the same base, but was customized to allow subjects to associate more than one file with a tab, as discussed in Chapter 3. The source from the Auto-pin tweaklet[4] was used in the customization, as per Section 4.6.3.

All sessions used an IBM T42 ThinkPad laptop running Windows XP Service Pack 2. The laptop had a 1.73 GHz Pentium M processor and 1.5G of RAM.

For the audio and screen capture, we used Camtasia Studio version 4.0.0. [5]

All but one session took place in a somewhat secluded spot in our laboratory, using a Microsoft Digital Media Keyboard 1.0A keyboard, Microsoft Optical Mouse 1.1A mouse, and 24" Dell 2407WFP TFT active matrix display with a resolution of 1920 by 1200 pixels at 60 Hz.

One subject (Jim) worked in a conference room at his workplace, using his own keyboard and mouse, and our laptop's built-in LCD display (1024 by 768 pixel resolution).

For the first two subjects (Bob and Mark), the Eclipse window filled the screen (except for the taskbar, which contained the clock), but because of the controls that

[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=168379#c32, verified 9 July 2008

[5] http://www.techsmith.com/camtasia.asp, verified 9 July 2008

the Camtasia replayer used, we could not see both the top part of the Eclipse window (where most of the action was) and the clock when replaying. For subsequent subjects, we limited the size of the Eclipse window to approximately 1900 by 1100 pixels.

For the first two subjects (Bob and Mark), we used the built-in Windows clock in the taskbar. We then discovered how difficult transcribing and coding the data was without having seconds displayed on the clock. We thus installed Free Desktop Clock version 2.2[6] for the subsequent sessions.

4.6.1 Mylyn

Mylyn, formerly called Mylar [14, 15] is an Eclipse plug-in that throttles the amount of information that is presented to the developer. By tracking the developer's interactions with programming elements (types, methods, and fields) associated with a task, it calculates a number representing the *degree of interest* (DOI) for each element. When Mylyn features are enabled, Mylyn modifies which elements are displayed or how they are displayed based on the degree of interest. For example, with Mylyn features enabled, the Package Explorer can be set to only display program elements whose degree of interest exceeds some threshold. As another example, in source editors, Mylyn can preferentially expand methods with a high DOI and collapse methods with a low DOI.

Mylyn also highlights elements with a particularly large DOI. Such high-DOI elements are called Landmarks.

Mylyn calculates and stores the DOI values on a per-task basis; it also stores which classes are open on a per-task basis. Creating a new Mylyn task reinitializes the DOIs and closes all open files; returning to a previous task restores the DOI values and the and re-opens the files that had been open. This allows developers to keep better track of the *context* of a task when switching between tasks.

Mylyn users can, to a limited extent, override Mylyn's automatic DOI calculations. For example, users can easily increase the DOI value significantly by setting a Mylyn Landmark via a context menu or hotkey.

Four of the subjects (Bob, Mark, Tom, and Peter) use Mylyn habitually at their

[6] http://www.drive-software.com/freedesktopclock.html, verified 9 July 2008

job. Bob elected to not use Mylyn for the study, but the others carefully created a new Mylyn Task for each task in the study.

4.6.2 Mylyn UI Usage Reporting Plug-in

We used the Mylyn UI Usage Reporting Plug-in[15] to collect logs of user interactions. It captured many – but not all – of the interactions that the subject had with Eclipse in an XML log file. We used the Usage Reporting log as a starting point for annotating log files.

4.6.3 Auto-pin Tweaklet

The Auto-pin tweaklet[7] is a plug-in for Eclipse that gives developers some control over whether files open in a new tab or not. With the tweaklet installed and configured, files always open in a new tab unless the file is dirty or if the user has executed an action to mark a tab ("pinning") to not allow any more files to use that tab. This is slightly different than what we wanted Weta to do.

In particular, using the tweaklet, it is somewhat cumbersome to deal with the use case where a user looking at `A.java` sees `B`, `C`, and `D` classes to explore, and wants to open all three in new tabs. In Eclipse with the tweaklet installed, the user would need to do something like pin `A.java`, control-click on `B`, pin `B.java`, return focus to `A.java`, control-click on `C`, pin `C.java`, return focus to `A.java`, and control-click on on `D`. If the user then wanted to explore the "`B`" path, keeping all the files from that path in the "`B`" tab, then the user would need to *un*pin `B.java`.

Although the Auto-pin tweaklet did not do exactly what we wanted, it did have some code that was useful, so we incorporated it Weta.

4.7 Analysis

We analyzed the data in several steps. We converted the UI log data to a terser, more manageable form. We annotated those files with a rough transcription and comments about what the subject was doing, and coded the tasks' navigation actions. We transcribed most of the interviews.

[7] https://bugs.eclipse.org/bugs/show_bug.cgi?id=168379#c32, verified 9 July 2008

4.7.1 Data Coding and Annotation

We converted the log data for all twenty-eight of the tasks from the original XML to a terser format in which there is only one (very long) line per interaction. We removed some non-navigation interactions in the process.

We then coded the files in conjunction with the videotapes, assigning codes to user interactions. Although the terse log file was a good starting point and saved us the headache of filling in time codes, there was a lot of information that we cared about that the Mylyn UI Usage Reporting Plug-in did not give us. For example, if the reporting plug-in logged that the subject opened the Search Results window, then opened a specific file, we could not be certain that the subject selected the file from the search results. The log would be the same if the subject selected a search result or if the subject did not see any search results he wanted and clicked on a tab instead.

We also added information about what the subject was doing. These included things like what search strings the subject typed in (including what he typed when he changed his mind), prosodic elements, and sometimes mouse motions. It was very common for the subject to use deictic words whose referent was entirely unclear from the naked text. For example, Dave once said, "yeah, just for the sake that this method is called before it closes this writer." (What is "this"? "It"? "This writer"?) However, with information from tone of voice, delays, emphasis, and especially the movement of the cursor, it was usually possible to figure out referents and annotate the file appropriately.

For thirteen of the twenty-eight tasks, an assistant did the first pass of the coding and annotating. One of the researchers (Sherwood) followed up and did the transcription plus a double-check of the assistant's coding. For fifteen of the tasks, Sherwood did all of the coding and annotating.

4.7.2 Interviews

We transcribed six of the interviews. (Jim was omitted for not being representative.) Sherwood did all of these transcriptions.

4.7.3 Data Exploration

After the coding and annotation was done, we examined the data in detail. In many cases, this consisted of merely counting occurrences (frequently with a simple script), but sometimes there was more involved data markup.

We were very interested in finding starting points of exploration paths, in distinguishing between finding novel starting points and going back to previously-seen points. This meant that we had to look carefully at finding-type actions and what the subject said in order to figure out the beginning (and hence ending) points of paths, and whether or not it was a search for a novel or previously-seen location. We frequently referred to the video in order to get the full prosodic richness when making our determinations.

We also were interested in distinguishing *why* subjects returned to previous locations. When we located revisits in the logs, we classified it as being because he hit a deadend, because he was looking something up briefly, explaining something to the assistant, or unclear. (This was difficult. Some subjects were very quiet.) Note that we classified something as a revisit only when they were making a deliberate effort to go there. Imagine, for example, a subject said, "Let's go back to where X is set", started at A, passed through previously-seen B, ended up at at C where X is set, and said, "Well, that wasn't it." It would be pretty clear that he was not making a deliberate effort to revisit B. He was, however, intentionally trying to revisit C.

We also annotated *how* they returned to previous locations. Frequently it would only be one step, so we could say definitively that they got there, for instance via an Open Type search. Sometimes there were sequences of the same actions, such as five presses of the Back button in a row. Those were coded with that action. Occasionally, they would retrace their steps: go to point A, then look for things that looked familiar, and trace from there until they got to their new starting point. Those were coded as a retrace. Rarer still were combinations of different actions that were not tracing. Usually when there were several different types of actions, the subject tried one approach unsuccessfully, then tried another until they found what they were looking for. We annotated those revisits as having the last (i.e., successful) type of action, or on rare occasions noted the last two.

```
@@@     H       12:09:01        FOCUS_ON_FILE       \
o.e.jdt.ui.CompilationUnitEditor        selection   \
=proguard3.8/src<proguard.io{\
DataEntryWriter.java[DataEntryWriter
# DEADEND DataEntryWriter was not interesting.
# P: Oh, it's just an interface.  (reads?)
# (looks at several options in DataEntryWriter hierarchy,
#   hovering on JarWriter)
# P: So there are a couple of classes that implement this
# interface DataEntryWriter and I'm not really sure what it
# needs to be; let's see what this factory will do
@@@     TSrd    12:09:25        FOCUS_ON_FILE      \
o.e.jdt.ui.CompilationUnitEditor  selection     \
=proguard3.8/src<proguard{OutputWriter.java[\
OutputWriter~writeOutput~QClassPool;~QClassPath;~I~I~I
# REVISIT DEADEND TS: DataEntryWriter was not interesting
@@@     H       12:09:38        FOCUS_ON_FILE   \
o.e.jdt.ui.CompilationUnitEditor                \
selection                                       \
=proguard3.8/src<proguard{DataEntryWriterFactory.java[\
DataEntryWriterFactory~createDataEntryWriter~QClassPath;~I~I
@@@     TSre    12:10:18        FOCUS_ON_FILE   \
o.e.jdt.ui.CompilationUnitEditor selection      \
=proguard3.8/src<proguard.io{\
DataEntryWriter.java[DataEntryWriter
```

Figure 4.1: Sample of annotated log file

Figure 4.1 shows a snippet from a final annotated log file. Lines that have been given navigation codes start with "@@@" and Lines that start with "#" are transcription or explanation. Spacing and line breaks have been modified slightly to allow the transcription to fit in the Graduate Studies-mandated line length. A " " specifies that there is no line break in the original. Inline explanations are in parentheses.

Chapter 5

Navigation Observations

Initially, we were interested in how our subjects would change their navigation behaviour when given better tools for keeping track of exploration paths. Did they use a more breath-first-ish search strategy: did they explore more different paths more shallowly? However, we found that they did not really change their search strategy. Instead of using Weta's tabs to allow them to keep track of more potential exploration paths, they used Weta's tabs to help them re-find previously seen locations. Intrigued, we looked more carefully at the overall navigation process. Note that some aspects of the navigation were very similar for Eclipse and Weta and so are combined in this analysis.

Our subjects would *find* a starting location, then *follow* relationships (like *uses* or *inherits*). We found that they used similar but not identical techniques for finding novel points and re-finding previously-seen points. Section 5.1 discusses techniques they used for finding novel points, and section 5.2 discusses how they found previously-seen points.

Our subjects mostly used static tracing to follow relationships from one piece of code to another. Using dynamic tracing was rare, and usually interleaved with static tracing. Section 5.3.1 discusses static tracing techniques that we observed; section 5.3.3 discusses our observations of dynamic tracing.

5.1 Discovering a Novel Starting Location

To be successful in exploring code, developers must find a good starting point. Most of the time, that starting point will not be the ending point, so they must follow relationships (what Ko calls *dependencies* [19]) through the code to (hopefully) find the point of interest.

From observing our subjects, we determined a number of interesting things about finding a novel starting point. This section describes and classifies the various techniques that we saw our subjects employ to find places that they had never been before.

We saw two basic ways that our subjects discovered novel starting points: *searching* (i.e., entering alphanumeric text into a box and having Eclipse show where it found that string) and *browsing* (i.e., where the subject scanned through a list of candidate types by eye) for an interesting-sounding class. They searched and browsed in roughly equal amounts, although the searches were dominated by Dave and Jim. Excluding Dave and Jim, subjects browsed much more than searched.

Ko et al. reported that in 40 of 48 (83%) instances, their subjects started tasks by searching, browsing only eight of 48 (17%) times [20]. However, when our subjects first started a task, they searched only 18 of 28 times (64%) and browsed 10 of 28 times (36%). Excluding Jim, they searched only 14 of 24 times (58%). This suggests that what strategy subjects choose is strongly dependent upon either the subjects or the tasks.

We discuss searching in Section 5.1.1 and browsing in Section 5.1.2. We found that our subjects almost always used File Search to find user-visible strings.

Although there are some pitfalls of using browsing to find a starting point (that we will discuss in Section 6.3.5), we found that developers are quite good judges of which classes and interfaces are germane to the problem at hand.

5.1.1 Searching for Novel Locations

There are two different search tools for types in Eclipse: the Open Type Dialog and the Search Dialog.

In Eclipse, a menu command or a shortcut sequence (shift-control-T) brings up the Open Type dialog. As the developer enters a string, the list dynamically (and

immediately) changes to present all of the types (and only those types) that start with the given string. Developers can use wildcards to match patterns in the class names. For instance, "*Tip" finds the class `ArrowTip`, while "Tip" does not.

The Search Dialog has four variations, of which our subjects used two: File Search and Java Search. Java Search lets developers search for one kind of Java Element – e.g., types, methods, or fields – through all of the code (but not comments) in the developer's workspace. File Search is similar to the Unix grep command, and will find a text string no matter where it appears – comments, types, methods, etc. Users may also search for regular expressions or use wildcards.

Our subjects used Java search to find a novel class exactly once, used File Search 31 times, and used the Open Type search exactly once. Interestingly, they almost always used File Search to find user-visible strings, i.e., text that appeared in the GUI or that was printed out to the console.

Note that we logged a total of 1189 navigation events. This number includes accidental and unsuccessful navigation events like Open Declaration and Open Type, but does not include scrolls or non-navigation interaction events like edits, saves, closes, or perspective switches. (Appendix E gives a list of all of the codes we logged.) File Searches thus accounted for only 2.7% of all navigation actions.

File Search

We saw 31 (2.6%) cases where subjects did a File Search to find a novel location, but Jim did 19 of them – 61% of the total number of file searches.

Excluding Jim's searches (because he was clearly an outlier), there were eleven searches for novel locations, all successful. (We use "successful" here to mean that the search found code that the subject felt was worthy of further exploration.) In eleven of those fourteen searches, the subjects used File Search to locate a string that was visible in the running application (either the GUI for the JHotDraw tasks or the console for the ProGuard tasks).

Dave performed the remaining four File Searches. He used File Search successfully once to find implementors of an interface, once to find references to a field, and once to find a non-user-visible string ("resize"). He also unsuccessfully searched once for "main".

5.1.2 Browsing for Novel Locations

Subjects frequently browsed for novel locations by scanning through the Package Explorer for interesting-looking types. Altogether, we identified twenty-four instances of browsing for an unknown class with an unknown name (2.0% of all navigation events). Of those 24 browsing efforts, our subjects abandoned their effort three times without selecting anything.

In the 21 remaining cases, thirteen types were selected once apiece. (One type was selected three times, and two types were selected twice each.) The fact that most of the classes were only selected once might make you think that their selections would be haphazard and not ultimately useful. However, the choices that they made when they did this type of non-directed browsing were sensible and productive ones. All of the types selected were ultimately important in the solution except for three. The subjects who chose those three types figured out relatively quickly that they were not interesting, so choosing the wrong class was not a significant problem.

Jim was the only one who never browsed at all; one other (Bob) browsed once but did not end up selecting a class.

The other subjects browsed for an unknown class much more often than they searched. While Jim and Dave searched frequently, the other five searched for a novel location only twice.

5.2 Re-finding Locations

The strategies that the subjects used when going back to locations that they had previously seen were similar to but not identical to those for finding novel types.

This is due in part to having more options. Once a developer has seen a location, they can navigate there with the Navigation History, an ordered list of locations the developer has been to, which is described in more detail in Section 5.2.6. Eclipse also keeps the set of open files easily accessible, with the most-recently accessed ones accessible via a row of tabs above the source pane, as described in Chapter 3. Finally, developers also have various tools for "waypointing" – marking a spot to come back to later – which we will discuss in detail in the sections below.

Unfortunately, the waypointing tools are poor. Our subjects almost never used

waypointing tools. The Bookmarks feature, which seems tailor-made for helping developers revisit locations, was almost never used. This is consistent with previous research, as described in Chapter 2. We saw a very small number of subjects use Breakpoints and Mylyn Landmarks to either find their way back to a known place or to mark a known place, but never both. (We did not consider setting a breakpoint in order to force execution to halt at that point to be marking behaviour.) We saw very little other marking and returning behaviour.

In Eclipse, both the Navigation History and the visible tabs, to a good first approximation, show *all* locations that the user visited instead of ones that the developer thought were important. True, the visible tabs bar does not always contain all the open files (if there are more tabs than will fit), and closed files will not show in the tabs bar. However, because the study's tasks were small, there were not very many occasions where there were more tabs than would fit, and our subjects very rarely closed files.

Users mostly used the Navigation History features or used the collection of visible tabs to find places to go back to. They never used Java Search or File Search to find their way back.

Subjects did sometimes use Weta's ability to open files in a new tab to explicitly mark locations to make them easier to find later. Given that we had given them a small amount of training that encouraged them to use the tabs to create a set of reminders of places that they should explore later, we were surprised that marking for refinding was how they chose to use Weta's capabilities. We discuss the tabbing usage in Section 5.2.7.

5.2.1 Bookmark

Eclipse has a "bookmark" feature that works in a fashion similar to bookmarks in modern Web browsers and which is specifically designed to allow developers to return to certain locations. To set a bookmark, the developer needs to right-click in the left gutter of a source view, then enter an identifying string. (The default is to use the entire text of the line which is at the selection point.) A little icon of a bookmark is left in the gutter. To return to a bookmark, the developer can open the Bookmark View find the specific bookmark in the list, and click on that line. Figure

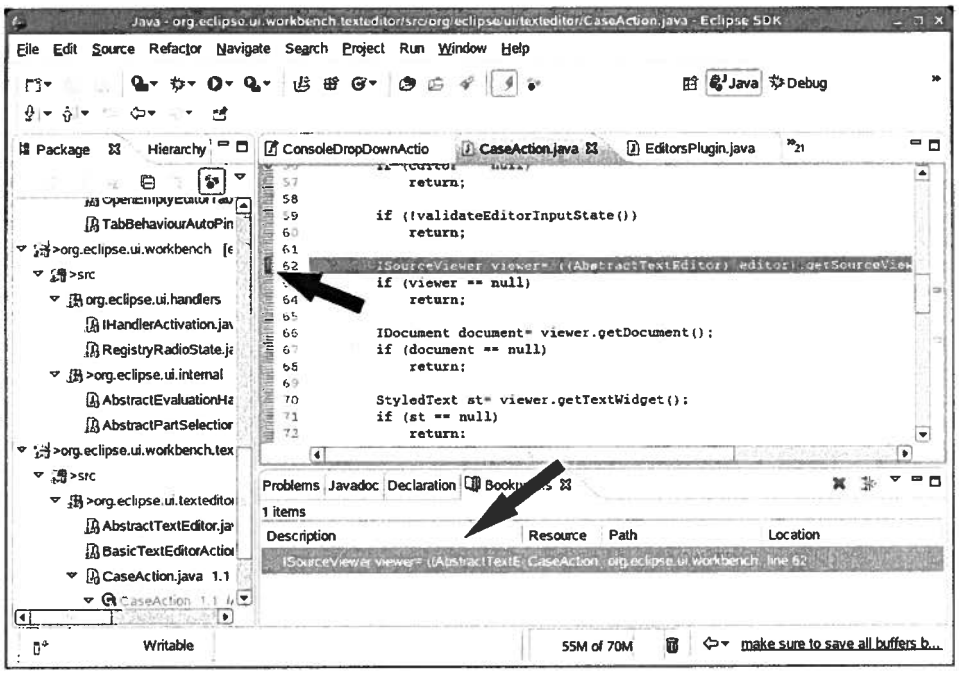

Figure 5.1: A bookmark in left gutter and in Bookmark View

5.1 shows one bookmark in the left gutter of the source view, and a reference to that bookmark in the Bookmark View.

We saw exactly one person (Bob) set a bookmark exactly once, and he returned to it less than one minute later by looking for the bookmark icon as he scrolled. However, in his day-to-day work, where he uses Mylyn, even Bob would not have used a Bookmark. In the interview, Bob said that he would normally have set a Mylyn Landmark. Bob did not set a Landmark only because he had decided not to use Mylyn in the course of the user study. (Mylyn Landmarks are discussed in Section 5.2.3.)

When asked about bookmarks, Peter said that he did not like how bookmarks accumulated, Peter said:

> I tend not to use bookmarks very much because it's just an extra view I have to go look for, and then I have this long list of stuff sometimes.

Bookmarks by their nature are meant to be left for some potential future point that might never come. Users thus do not have an obvious trigger in Eclipse to remind them that a bookmark should be removed.

5.2.2 Breakpoints

Breakpoints have some similarities to bookmarks. Like bookmarks, they are set via a context menu in the left gutter, they are listed in their own view, and clicking on a breakpoint in Breakpoints View takes the user to the line with the breakpoint. We saw two subjects (Jim and Peter) use them as navigational aids, one with forethought and one opportunistically.

One subject (Jim) set a breakpoint exactly once with foresight, verbalizing that he wanted to come back to it, saying

> I will not take note of it, but I will try to remember. I'll put a breakpoint here, and I'll come back to the breakpoint, that's my note taken.

Another subject (Peter) twice opportunistically looked for a breakpoint he had set earlier. He did not say anything about wanting to come back to them when he set them; he set those breakpoints to make the debugger stop there.

No other subjects ever directed Eclipse or Weta to take them back to a specific breakpoint location. They only went to a breakpoint location when they set the location, when the debugger took them there, or got there by other means.

5.2.3 Mylyn Landmarks

One subject (Mark) set a Mylyn landmark exactly once to indicate that a code location was interesting, but never took advantage of the Landmark. Although he did turn on Mylyn restrictions for the Package Explorer immediately, he never went back to the Landmark method via the Package Explorer.

Nobody besides Mark set a Mylyn Landmark that we could tell. However, since keystrokes are not visible on the screen capture video, and Mylyn operations are not captured in the interaction log, we cannot be entirely certain that Mark was the only one to do so.

As mentioned in Section 5.2.1, Bob indicated that if he had been using Mylyn for this study, he would have set a Mylyn Landmark instead of using a bookmark.

5.2.4 Search

Our subjects never used Java Search or File Search to return to places that they already knew of. (In two arguable exceptions, one subject (Dave) used File Search once and Java Search once to find implementors of and references to the interface NameFactory. However, we classify those as *following* techniques and not *finding* techniques.)

Four subjects used the Open Type dialog nine times to find known types (or types that they thought they knew). One – a search for "main" – was unsuccessful (since he was using the Type search to look for the method `main()`!), and the others were successful.

5.2.5 Browsing Package Explorer for Known Class Names

Subjects searched for known locations twenty-two times by looking for the class in the Package Explorer. All twenty-two attempts were successful. Of the twenty-two selections, ten of them were for the class with the `main()` method. Finding the `main()` method was something our subjects did often and had some trouble with, as we will discuss in Section 6.2.1.

Using Mylyn to Aid Re-finding

One of the key features of Mylyn is that it can reduce the number of items shown in the Package Explorer, making it easier to see elements that the developer has seen before. Four of the subjects used Mylyn in their workplaces, and three (Mark, Tom, and Peter) set up Mylyn carefully at the beginning of each new user task.

Tom used Mylyn to restrict what he saw in the Package Explorer for each of the six times that he looked in the Package Explorer for known locations. Mark normally did not have Mylyn's restrictions turned on in the Package Explorer, but he did once turn Mylyn restrictions on immediately before selecting a method from the Package Explorer.

Bob uses Mylyn at his workplace, but did not use Mylyn for any of the tasks in

the user study. At one point he expressed regret that he had not done so:

> I was just trying to figure out how to get back to where I.. Ah, if only I had Mylyn!

5.2.6 Navigation History

Subjects used the Navigation History features extensively to return to places that they had been before. The Navigation History features all operate on a stack of locations that the developer has visited. Developers can go backwards and forwards in the stack, or can select from the navigation history. There are forwards and backwards toolbar buttons, forwards and backwards hotkeys, a drop-down navigation menu, and a keyboard-selectable menu. In Eclipse, this history is global. In Weta (like in current Web browsers), this history is constrained to show the files opened in the current tab.

Backwards/Forwards

Eclipse keeps track of the global navigation history as a stack. Users can select locations from a drop-down history bar, or move forwards and backwards in that stack by using Backwards and Forwards commands. They can execute the Backwards and Forwards commands by pressing buttons (displayed as left-facing and right-facing arrows), or via a hotkey. (In our data analysis, we did not distinguish between hotkey and button invocations.)

Five of the subjects did use Backwards and Forwards commands in Eclipse to go back to a previously-seen location frequently. They returned to a previously-seen location via Backwards/Forwards commands(s) on 25 separate occasions, for 2.1% of all navigation events. It frequently took them multiple clicks to get back to the location they wanted, and they had to visually examine each step to see if it was the location they wanted. Dave and Jim did not ever use the Backwards and Forwards commands in Eclipse.

Back-to-last-edited

In addition to having a button for going to a previous location on a stack, there is a button (and hotkey) for going to the previous line that the developer edited. A

Figure 5.2: Comparison of Backwards/Forwards and Back-to-last-edited buttons

comparison is in Figure 5.2; the Back-to-last-edited button is on the leftmost and looks very similar to the Backwards button – like the Backwards button but with an asterisk.

Navigation History Menu

Subjects did not use the Navigation History menu very often. They only used it to get back to previous locations three times (though once was after unsuccessfully trying to use the Backwards command).

5.2.7 Tab Select

Eclipse Tabbing

Our subjects used tabs heavily to return to locations that they had been to previously. In the tasks done with Eclipse, they used tab selection to return to locations about as often (23 times, 3.9% of navigation events done with Eclipse) as they used the Backwards command (25 times, or 4.3% of all 582 navigation events done with Eclipse). Like the Navigation History features, to a first approximation tabs show all the places that a user visits, not the ones that a user thinks are important. Although it is possible in theory to mark all the interesting locations by closing all the *un*interesting tabs, in practice we did not see subjects do this. The subjects only closed tabs seven times, compared to opening 82 files.

Eclipse uses a Most Recently Used policy to decide which to keep, which means that tabs can disappear from any location on the tab bar (as opposed to always disappearing from the far left or far right). Although none of the subjects verbalized confusion about why tabs disappeared, there is evidence that finding the

right tab has a high cognitive load[1]. Four of the six interviewees (excluding Jim) volunteered frustration at trying to keep track of many tabs, for example, Tom said:

> If you start navigating with F3 or control-click for any length of time, you end up with so many tabs that it's kind of useless, you cannot use the tabs to go back to where you were or find something relevant.

Tabs in Eclipse thus appear to be barely adequate for storing previously seen locations, and very poorly suited for distinguishing interesting locations from uninteresting ones.

Only once did we see a subject consciously open a tab for later examination in Eclipse. The subject (Peter) intentionally opened a file and stated explicitly that he opened it so that he could come back to it later.

> I also want to see there is this thing called OutputWriter so open it just to have a look at it for later.

This was during a task performed with stock Eclipse, so he opened the tab knowing that the tab/file would have to compete for his attention with all the other files that he would open afterwards. In this case, he came upon `OutputWriter` in the course of exploring, so did not need to come back to `OutputWriter` via the tab.

Weta Tabbing

We expected our subjects to use Weta to open tabs of places that they wanted to explore in the future. We expected to observe our subjects say things like, "Well, I'm not sure which of these three methods I should explore, so I'm just going to open the declarations of these two in new tabs but not look at them yet. I'll explore this third method, and maybe come back to those other two later."

We expected this behaviour for two reasons. In casual, day-to-day life, we observed this behaviour among many of our friends and colleagues when browsing the Web. We also explicitly encouraged them to use this type of behaviour during their Weta training.

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=68684, verified 9 July 2008. Also see e.g., Eclipse bugs 68684, 106708, 106282, and 223201.

Instead, we were surprised to see our subjects instead use Weta to keep track of places they had already been. If they saw several methods that looked interesting, they would open a new tab for the declaration of the first method, and immediately switch to that new tab and explore that path. If that path did not look promising, they'd return to the first tab and *then* look for other paths to explore. In almost every case (thirty out of the thirty-two) where subjects used Weta to perform an Open Declaration in a new tab intentionally, the subjects immediately switched to the new tab and started exploring from there. (For new tabs generated with Open Declaration, Weta keeps the focus on the original tab, so our subjects had to explicitly, consciously switch.) Note that in one of the two cases where the subject did not switch, we suspect that the subject did not intend to open in a new tab. In the other, the subject opened a tab for the same class he was looking at (effectively cloning his current tab). He then continued exploring the path in the original tab, but he would have gotten the same effect as if he had switched.

One might argue that the way our subjects used tabs is functionally equivalent to what we expected. We argue that they are not. First, opening tabs in quick succession for all the places that look interesting ensures that the developer will not forget any of those places. If a developer only opens one new tab, then they must use working memory to remember what in the old tab they thought was interesting. We saw several instances of subjects forgetting to explore a branch that had looked interesting earlier.

Second, making the explicit decision to open multiple tabs in quick succession forces the developer to recognize multiple possibilities explicitly. This might help reduce confirmation bias. If the developer merely marks the old location, they are not forced to explicitly recognize what other paths are possible, so would not get the benefit of reducing confirmation bias.

Third, one of the important elements of the Web-style tabbing is that it allows the developer to "suspnd" exploring a path instead of "abandoning" the path. We never observed a subject declaring that he wanted to suspend exploring that path but keep it around because it might turn out to be useful later. Subjects always abandoned paths completely.

With Weta, they had far fewer open tabs than with Eclipse, so less "tab spam",

Table 5.1: Maximum number of open tabs

Name	Eclipse total max tabs	Weta total max tabs
Bob	8	10
Dave	9	8
Mark	13	4
Peter	16	3
Steve	14	11
Tom	13	9
Average	12.2	7.5

as shown in Table 5.1.[2]

the second column has the maximum number of tabs opened at any one time in the first task added to the maximum number of tabs opened in the second task. Similarly, the third column is the sum of the maximum number of tabs for the two Weta tasks. This gives a concise figure to compare Eclipse and Weta behaviour.

Based on his comments and behaviour, at least eight of Steve's 11 Weta open tabs were misfires. If those 8 are removed from the data, then the sum of the maximum number of tabs in the two Weta tasks drops to 6.2 – barely over half of the Eclipse total.

For the display size that we had, generally the subjects could see about seven tabs before tabs started getting evicted. The subjects very rarely had more files open than tabs visible. The vehemence of the subjects about their dislike of "tab spam", suggests that in their day-to-day professional life they have more tabs than these twenty-minute tasks generated.

Note that the lower number of tabs visible in Weta cannot be due to subjects opening fewer files; they opened 82 files when using Eclipse and 101 files when using Weta.

Despite there being fewer tabs open, subjects closed tabs much more often when using Weta than when using Eclipse. During Eclipse tasks, the subjects closed tabs seven times, while they closed tabs 21 times in the Weta tasks.

[2]Jim was excluded from this data because his monitor was smaller, so did not have as much room for tabs as everyone else did.

Table 5.2: Comparison of Weta and Eclipse usage

Tool	Back	Tab	Files opened	Tabs closed
Eclipse	25	23	82	7
Weta	37	13	101	21

It seemed like the subjects frequently wanted to go back to places that they had not realized earlier that they would want to come back to. In addition to opening significantly more files when using Weta than when using Eclipse, they used the Back button more. In cases where a subject deliberately returned after a deadend to a location they had seen previously with Eclipse, they used a sequence of Back commands a total of 25 times (4.3% of events done using Eclipse), and switched by selecting a tab 23 times (3.9%). With Weta, they used a sequence of Back commands 37 times (6.1% of all Weta events) and tab select only 13 times (2.1%). The fact that they used the Back commands so much more when using Weta than when using Eclipse might indicate that developers are not very good at marking which locations they will want to come back to, either because they forgot to mark it or didn't realize it was worth marking.

Interestingly, Dave never used the Back button at all in tasks using Eclipse, though he pressed the Back button 34 individual times in tasks using Weta and used Back sequences five times in deadend situations. In the interview, Dave said,

> I never use Back, I don't even know what Back does in Eclipse to be perfectly honest.

5.2.8 Other Methods for Marking Locations

We did not see subjects using any other creative ways to mark locations for later perusal. We did not see our subjects inserting deliberate errors to mark locations, nor did we see our subjects deliberately dirtying a file to make it more obvious, nor did we see our subjects inserting easily-searched-for-strings, nor did we see our subjects insert special Eclipse-recognized strings into the source[3].

[3]For example, if the string "TODO" is inserted into a comment, Eclipse will mark those locations in the right-hand gutter.

We provided scratch paper and pens, and were careful to point them out to every subject, yet there was only one occasion when anyone wrote anything down. Even then, it was the name of a field, not a location in code that he could navigate back to. We never saw any other subject make any form of note: not in a text file, not in a comment, not in a Bookmark label. Note that this is at odds with Ko et al.'s (implicit) assertion that developers "rely extensively on external memory sources, such as digital or paper notes and whiteboards" [?]Ko2006Exploratory.

Ko et al. [20] reported that some subjects used the undo stack to go back to previous locations. We never saw that behaviour.

5.2.9 Other Methods for Revisiting Locations

Sometimes the subjects would retrace their steps to find their way back to a location. They would browse or search for a class, then statically trace through the code to find the place they were looking for.

5.3 Following Techniques

Users used either *static tracing* or *dynamic tracing*, or a hybrid of the two. Static tracing uses no information about any actual execution paths of the program; dynamic tracing only uses information about the actual execution path of a program currently running.

5.3.1 Static Tracing

In static tracing, developers read code, select interesting Java elements in that code to focus on, and then follow a relationship. (Usually our subjects examined either the callers (references) of that element or the declaration of that element.)

Open Declaration

One of the relationships between code elements is *declares*. The Open Declaration command makes Eclipse replace the contents of the source pane with the contents of the declaration of that element. Users can execute the Open Declaration command in Eclipse by selecting a Java element and pressing F3, selecting a Java element and performing the menu command Navigate->Open, or control-clicking

on a Java element. Users can trace statically through source in much the same way that they navigate through the World Wide Web, jumping from declaration to declaration to declaration.

Jumping from declaration to declaration to declaration approximates the behaviour of an actual execution of the program. It is only approximate because the execution path is not entirely certain and because the type of the object is not completely certain. Which branch of a conditional should be taken is not known; which class a method belongs to is not completely specified (the object could be a subclass of the type determined statically). This type of navigating can be thought of as going "forwards" in time, since the order that the developer reads lines of code can approximate the order which the virtual machine executes those lines.

This form of navigating "forwards" through code was extremely common in our study, perhaps the most common form of navigating. Even removing misclicks and recoveries from misfires, the subjects performed an Open Declaration 219 times (18.4% of all navigation actions).

Note that this is inconsistent with Ko et al.'s finding that only two developers used tools for "direct dependencies" (Open Declaration and Java Search) more than once, and then only for an average of four navigations [20]. This discrepancy might be due to their code base being much smaller than ours, because they used a different version of Eclipse (2.1.2) than we did, or perhaps because they distinguished between the Open Declaration being selected from the menu and being selected via a control-click. (We did not distinguish between the two.)

References Search

The References search command in Eclipse presents a list of all of the locations that can call the selected Java element. Users can then click on one of the locations to make Eclipse replace the contents of the source pane with the selected location's source. If tracing by hopping from declaration to declaration (as described in 5.3.1 above) can be thought of as approximating tracing "forwards", then opening references approximates tracing "backwards" in time along the execution path. Tracing statically backwards in this manner has the same uncertainties that tracing statically forwards does.

Eclipse also has a Call Hierarchy View which allows developers to look at the calling tree in a hierarchical form. Although developers can use Call Hierarchy view to view a tree of what methods are called ("forwards" in time), we never saw any of our subjects use it except to view who called a method ("backwards" in time).

Subjects traced "forwards" a little more than twice as often as they traced "backwards". They used the References command 71 times (6.0%) and the Call Hierarchy viewer 25 times (2.1%). In addition, from context, two of Dave's File Searches were clearly searches for references. There was one misfire when a subject attempted a References command when no Java Element was selected.

Subjects almost always found worthwhile results from a References command; subjects selected one of the References search results in 60 of the 70 cases (or 62 if you count Dave's two File Searches).

Inheritance Trees

Another, less common way to trace statically is to look examine inheritance relationships. Only four of the subjects (Peter, Steve, Tom, and Mark) looked for inheritance relationships.

5.3.2 Navigation History

We never observed the Back-to-last-edited feature being helpful, only harmful. We never saw anyone use the Back-to-last-edited feature on purpose. Two subjects accidentally pressed the Back-to-last-edited button instead of the Backwards button. One did not realize what happened and got confused.

It is possible that better training would allow developers to make better use of the Back-to-last-edited feature. Users could dirty a file with something harmless like adding and deleting a space character to mark that it was worth returning to. However, making files dirty that are not *actually* dirty can potentially interfere with figuring out which files should be checked in after making a fix.

Navigation History Menu

Subjects did not use the Navigation History menu very often. They used it to get back to previous locations three times (though once was after trying to use the Backwards command was not successful). Although it was not unhelpful, it did not appear to be very helpful.

Note, however, that these tasks were short and there were very rarely so many files open that some of their tabs would not be visible.

5.3.3 Dynamic Tracing

In dynamic tracing, the developer uses the debugger to step through interesting code as it is executing. At any point, developers have much more limited choices about what relationships to explore: they can step into a method (similar to the static examination of the declaration of a method), step over the method (similar to continuing to read code), or step-return to skip tracing of the rest of the method and return to the calling method. (This is perhaps most similar to pressing the Back button while doing static tracing.)

Unlike in static tracing, developers never make choices about what path they think the execution takes. In some senses, they do not *follow* an exploration path as much as they are *led down* a path.

Five of the seven subjects used dynamic tracing in at least one task. 10 of the 28 individual tasks featured dynamic tracing, and in an additional two individual tasks the subject set a breakpoint which was never hit.

One subject (Jim) used dynamic tracing on all four of his tasks. The other four subjects used dynamic tracing on six individual tasks. Of those, there was only one "purely" dynamic trace. In the other five cases, the subjects switched back to static tracing at least once before resuming the dynamic trace.

5.4 Summary

In summary, we saw that the experienced subjects avoided searching to find places, using it almost exclusively to find user-visible strings. They usually browsed using the Package Explorer and selected files from tabs. They almost never set markers of any type to help them return to a known point when using Eclipse. When using

Weta, they frequently used Weta to "leave behind" a location that they thought they might be interested in later. However, it seems that they were not very good at figuring out what points they would later want, as they ended up using the Back button extensively to find earlier points.

Our subjects mostly used static tracing instead of dynamic tracing. They also traced "forwards" almost twice as often as they traced "backwards".

Chapter 6

Discussion

Our initial overall goal with this research was to understand whether a breadth-first navigation strategy would be more productive than a depth-first approach. In this chapter, we discuss what the results of our experiment say about navigation strategies. Despite providing subjects in our experiment with tool support for breadth-first navigation, we found that subjects used Weta for marking waypoints and not for using breadth-first navigation. We evaluate Weta, and note common difficulties that neither Weta nor Eclipse solve, and evaluate threats to the validity of our findings.

6.1 Weta

Weta was not an obvious, immediate, total success. Some of this might have been acclimation; some of the developers clearly struggled to get used to the new way of doing things. (In the interviews, several subjects mentioned difficulty acclimating.) There were also some bugs in Weta that might have interfered with their ability to use the tool successfully.

We developed a set of metrics for each task to give a measure of how successful the subjects were. Each task had four actions subjects could get a point each for, like finding an important method, recognizing that code needed to be added to a specific method, or adding correct code. The success metrics are given in Appendix F.

There was not an obvious difference in how successful the subjects were with Eclipse vs. Weta, as shown in Table 6.1.

Table 6.1: Success measures with and without Weta

Name	Eclipse tasks score	Weta tasks score	total score
Jim	0	0	0
Mark	1	3	4
Dave	4	0	4
Peter	3	4	7
Tom	5	2	7
Bob	4	4	8
Steve	4	8	12
Average	3.0	3.0	6

The Eclipse tasks score is the sum of the score on the two tasks the subjects did using Eclipse. The Weta tasks score is the sum of the scores when using Weta. The total score is the sum of the subject's score on all four tasks during the time allotted.

(Note that Peter scored an additional three points on an Eclipse task 55 seconds after time ran out, and Dave scored an additional one point 37 seconds after time ran out on an Eclipse task. Adding those four points to the Eclipse total in Table 6.1 would give the Eclipse tasks an average score of 3.57 vs. 3.0 for Weta.)

Learning effects were clearly visible at times. For example, in Steve's first Weta task, he opened a file in a new tab eight times, but seven of those were unquestionably mistakes. He would verbalize that he made a mistake (e.g., "Oh yeah, double-click"[1]), then ignore the new tab, or open the same file a moment later but in the same tab. Once he opened a file in a new tab where we feel it was *probably* a mistake, but we can not be certain.

By the time Steve got to his second Weta task, he only opened a file in a new tab once. We suspect that he got better at remembering to do double-clicks instead of control-clicks and that his one opening in a new tab was a mistake.

[1] In Weta, double-click opens in the same tab, while control-click opens in a new tab.

In interviews and the questionnaires, the response to Weta was generally positive.

Subjects were asked on the questionnaire, "Overall, do you prefer Eclipse's behaviour or Weta's behaviour?", with an answer key of a five-point Likert scale. Three of the seven gave Weta the highest mark possible, as seen in Table 6.2. For the subject's preference, 1 means "I liked Eclipse's way much more" and 5 means "I liked Weta's way much more". The average Likert rating was 3.71.

Table 6.2: Subject's tabbing preference

Person	Likert rating	Total success score	Adjusted success score
Jim	1	0	0
Mark	5	4	4
Dave	5	4	5
Tom	5	7	7
Bob	4	8	8
Peter	3	7	10
Steve	3	12	12

Interestingly, the three who gave Weta the lowest three marks (Dave, Steve, and Peter) were at the ends of the spectrum: two of the most successful subjects and the least successful subject. It might be that the most successful had developed strategies that worked well enough that they hesitated to change, and the least successful did not have adequate experience to appreciate it.

However, even Steve and Peter said in the interviews that they liked using Weta, and gave specific things that they liked about it. Peter said:

> What I liked the best was if I am exploring and I see something that might be interesting, but is essentially a tangent off to what I am exploring right now, I can essentially open a tab on it and have that as a reminder to go back and look at it, but still not have it interfere in any way with the hypothesis I'm exploring right at that point.

Steve mentioned less tab spam as an attraction, and also thought he would use tabs for task management, as he notes in the following statement:

> I definitely would be using it for multi-tasking, so that if my boss wants me to do one thing and my co-worker wants me to do another thing, I can have two investigations open at the same time and I can actually keep track of what's going on, so it would be definitely useful on a day-to-day basis.

What Steve and Peter did not like was different. Steve seemed to dislike having to get used to a new user interface, while Peter said he did not have as good an overview of where he was coming from and going to.

Bob, Dave, Mark, and Tom were quite enthusiastic in the interview about Weta. Bob would have liked to be able to open a blank tab to start a new path, Dave mentioned a known bug, Mark said that selection was slightly problematic, and Tom had no complaints.

Some of the things that we agonized over when implementing Weta turned out to be non-issues. In particular, we never saw a subject trying to do multiple-selection (either in Eclipse or in Weta), and we never saw someone try to open a new file in a tab that had a dirty file in it already.

6.1.1 BFS, DFS, or Hypotheses?

We started this research influenced by literature that seemed to show that a more breadth-first search (BFS) approach – tracing a number of paths to a shallower depth before deciding on a path to explore in more depth – would be more successful than a DFS-ish approach [26, 33, 40, 41]. Prior research also seemed to suggest that better tabbing facilities might lead to a more BFS approach [3, 27]. We thus were interested whether enhancing the tabbing facilities might lead to more of a BFS approach, and thus more successful navigation.

Through the study we conducted, we were unable to verify if a BFS-ish approach was more successful than a DFS-ish approach, as subjects used DFS-ish approaches both with Eclipse and with Weta. They consistently would follow a path until they could not get any farther, look for a new starting point, follow that path, and repeat.

Given the prior research that seemed to indicate that BFS-ish navigation would be more successful, why did our subjects use a very DFS-ish approach? Given

the changes to Eclipse that were similar to the changes in Web browsers' tabbing between 1996 and 2007, why didn't we see the kinds of navigational changes that Obendorf et al. [27] saw?

First, it might just take longer for a subject to acclimate to a new way of navigating than the time they had in this study. We saw clear evidence that some subjects took about one task just to get used to the mechanical actions of Weta, as discussed in 6.1. By contrast, Obendorf's subjects had literally years to get used to tabbed browsing.

Second, it is not clear what researchers actually meant by "BFS". Most of the researchers who talked about "BFS" navigation (ourselves included) did not specify well what the term meant. Did they literally mean that locations should be put on a FIFO as in the Scratchpad project [25]? Given how comprehensively the Scratchpad ideas have failed to penetrate either the academic literature or the consumer browser market, we would suggest no. We ourselves did not believe that developers should operate in a strict FIFO model, but "BFS" was a quick, easy term to describe *both* forming multiple hypotheses before exploring any one hypothesis in detail *and* being willing to suspend exploration on a given path more quickly.

Third, the cognitive load required to parse a Web page and judge which relationship to follow is probably much lower than the cognitive load required to understand a Java method. Web pages do not have conditionals, they are written in something more closely approximating a natural language, and the density of hyperlinked elements (links) is usually very low compared to the density of Java elements in most Java program code.

Fourth, the Web has penurious inter-page relationships. Where code has calls, called by, uses, inherits, and overrides relationships, Web pages only have links. A developer needs to keep in mind all of the different types of relationships that he or she could follow, while a Web surfer has no choice in the type of relationship to follow. The higher amount of choice probably adds an additional cognitive load.

Fifth, Web browsers might have a shorter hierarchy of goals. While a Web surfer's only goal for a Web session might be "find the time that *Revenge of the Tropical Fish IX* is playing at the Orderlyville Megaplex 35", we suspect that developers have more complex goal hierarchies. A developer might have the goal "find where libraryClassCount is declared" as a subordinate goal to "find out what

the names of variables related to the Java bytecode format are" as a subordinate goal to "find the name of the variable that stores the number of member variables" as a subordinate goal to "find why the number of member variables is being displayed wrong". An increased goal hierarchy might add even further cognitive load.

Sixth, if there is a high cognitive load associated with each path, then that might make the costs of switching from one path to another higher than the costs of sticking with one path for too long.

Seventh, it might be that there is a hierarchy of needs associated with developing code, and that the need for a useful bookmarking feature is greater than the need for a navigation aid. Perhaps our subjects would have used Weta's tabs for navigation assistance if Eclipse's bookmarking feature were more useful. (We discuss ways to make bookmarks more useful in Section 7.3.1.

6.2 Common Pitfalls During Finding Actions

There were a few common pitfalls we saw our subjects making while attempting to find a starting point. The Search feature seems very difficult to use correctly, and `main()` was surprisingly hard to find.

6.2.1 Hard to Find main()

We were surprised to see how much trouble some of our subjects had finding `main()`, the starting point for code execution. Although everyone but one looking for `main()` eventually found it, it sometimes took a few tries. The subjects tried eighteen times to find nine instances of `main()`, i.e., they failed nine times and had try again.

Using the Java Search or File Search was unsuccessful nine out of eleven times. The two subjects who tried (Jim and Dave) usually did the search incorrectly, or they got so many results that they did not try to wade through the results. One subject (Jim) waded through 226 results twice. A different subject (Bob) erred by doing an Open Type search (which finds types, not methods) for `main`.

Going to the Run Dialog to find the name of the file was a very successful approach. In three of three tries, subjects found the name of `main()`'s class very quickly via the Run Dialog, then found the file either by browsing (Dave, twice) or

via the Open Type search (Peter, once).

Subjects also had success guessing at the name of `main()`'s class in the Package Explorer and by tracking backwards with the Call Hierarchy browser.

6.2.2 Search Difficulties

You might have noticed that in Section 6.2.1, where we discussed the difficulty that subjects had in finding `main()`, many of the difficulties encountered pertained to the Search features, especially the Java Search.

We saw seven cases in the 28 tasks where the subjects attempted a Java Search. Only one of them was successful. In four of the six unsuccessful Java Search attempts, the subjects made an mistake in setting up the search. In the two cases where they did not make an error in the search, they did not select any results.

We saw 31 attempts at File Search, nine of which were unsuccessful. In six unsuccessful cases, the subject got discouraged by the number of search results, and in three cases there were no results.

Jim and Dave did almost all of the searching; the other five only did one search that was not for a user-visible string. In interviews, we asked two of the subjects (Peter and Tom) about search, and they both said that they did not like Java search because of how fragile it was. They said that it was easy to make a mistake when setting up the search, and thus believe that there were no instances of what they were looking for when really there were (false negatives). As Tom said:

> I almost exclusively just use the File Search. ... [with] the Java Search, I'm always worried that it's going to not find what I'm looking for because I haven't selected the right radio button on it. ... [It's] usually not a concern if I get too many results using the File Search ...

Note that Jim did not have difficulties using Eclipse in general. He was quite fluid and proficient in using the Eclipse features that he chose. However, Jim was very inexperienced. Meanwhile, Dave mentioned that because of his company's environment, some of the tools that other subjects used heavily (like Find References) were not useful in his normal work environment. (See Section 4.1.1 for

more discussion on how Jim and Dave differed from the other subjects.) We believe that Jim was not experienced enough to learn what the majority reported in interviews: that search was dangerous to use; Dave's environment had trained him that he had few other options.

6.2.3 Navigation History

We never observed the Back-to-last-edited feature being helpful, only harmful. (See Section 5.2.6 for a discussion of this feature.) We never saw anyone use the Back-to-last-edited feature on purpose. Two subjects accidentally pressed the Back-to-last-edited button instead of the Backwards button. One did not realize what happened and got confused.

It is possible that better training would allow developers to make better use of the Back-to-last-edited feature. Users could dirty a file with something harmless like adding and deleting a space character to mark that it was worth returning to. However, making files dirty that are not *actually* dirty can potentially interfere with figuring out which files should be checked in after making a fix.

Navigation History Menu

Subjects did not use the Navigation History menu very often. (See Section 5.2.6 for a discussion of the Navigation History menu.) They used it to get back to previous locations three times. While it was not *un*helpful, it did not appear to be very helpful.

6.3 Common Pitfalls During Following Actions

We observed some areas where many of the subjects had similar difficulties while following relationships between code elements. These difficulties were not only common, but significant. When our subjects went astray because of these issues, we observed them losing a lot of time.

Subjects had a difficult time

- getting lost in the superclass,
- figuring out if a method was executed,

- "crossing the GUI divide",
- not getting diverted by misleading language, and
- recognizing when code was missing (as opposed to broken).

A common feature of these common pitfalls was subjects not knowing which code was actually executed when they reproduced the bug. Sometimes they spent time tracing branches of code which were not executed in reproducing the bug and hence not germane. Sometimes they mistakenly believed that code was not germane when it was. Sometimes they tried to find a method by visual inspection, but because they did not know which code was actually executed, had the entire code base as their search space. Dynamic tracing would have given them information about which code actually ran, but it was relatively rare for our subjects to use dynamic tracing.

6.3.1 Lost in the Superclass

One of the most striking and consistent difficulties that subjects had was what we termed "getting lost in the superclass". In the Size Task, every single subject started by searching for the user-visible string "Active View Size", and landed in the class `DrawApplication`. Unfortunately, `DrawApplication` was a *superclass of a superclass* of the class that was run (i.e., whose `main()` was executed). When the subjects traced statically forwards in the code – i.e., asked Eclipse to show them the declarations of various methods in `DrawApplication` – Eclipse took them to where those were declared in `DrawApplication` (or its superclasses), but never to methods in `DrawApplication`'s subclasses. This meant that the code that the subjects executed was different from the code that they saw when statically tracing: execution would use `DrawApplication`'s subclasses' overriding methods, but subjects would only see methods at or above `DrawApplication`.

Subjects expressed quite a bit of confusion as to why various methods in `DrawApplication` were not executed. Even with the clues that code was not executing when they thought it should, four of the seven subjects (Bob, Steve, Peter, and Jim) never realized that `DrawApplication` had subclasses.

At one point, Eclipse passively mislead one of the subjects (Steve). He was

exploring the `DrawApplication` when he examined the hierarchy of `setSize` using the Quick Type Hierarchy View. Unfortunately, the Quick Type Hierarchy shows the *method* hierarchy, not the *class* hierarchy (as Type Hierarchy does). Because it happened that no subclass overrode the method Steve selected, he did not get the message that there were subclasses, as demonstrated by what he said at the time:

> Do I have any subclasses of this that I'm following instead, that's forgetting to call super? So let's have a look at the type hierarchy for `DrawApplication.setSize`. Nope, that's the bottom, so there is not that.

Because he was actively investigating `setSize` at the time when he made that comment, for his immediate question, perhaps the Quick Type Hierarchy View gave him the correct answer. However, it would be easy to imagine that seeing no subclasses when looking for `setSize` could have predisposed him to assume that there were no subclasses at all of `DrawApplication`. His verbalization does not *prove* that he believed that there were no subclasses, but if "this" meant `DrawApplication` and not `setSize`, then his verbalization certainly implies that he was misled.

6.3.2 Is This Method Executed?

There were a number of times, particularly in the Size Task, when subjects wanted to know if a specific method was ever called. For example, five of the seven subjects ran experiments to see if the `showSizeInformation` method was ever called. Two subjects (Dave and Steve) assumed that it was not called, based on inspection of `showSizeInformation` and observation of the GUI behaviour. Two subjects (Bob and Mark) changed the method to display an obvious string, reran and observed the output. Three subjects (Peter, Tom, and Jim) set a breakpoint in `showSizeInformation` and reran.

Although Dave, Steve, Bob, and Mark's interpretation that the method was never called happened to be correct, their experiments and observations did not definitively prove that `showSizeInformation` was not called. It could have

been that `showSizeInformation` was called and then another method was called that immediately changed the display text. In that case, the display update might have been so fast that they would not be able to see the text that `showSizeInformation` displayed. Thus it would *appear* that the method was never called when in fact it was executed.

On the other hand, by setting a breakpoint, Peter, Tom, and Jim showed conclusively that `showSizeInformation` was not ever called.

6.3.3 Crossing the GUI Divide

Subjects had some difficulty "crossing the GUI divide" – finding code that was executed in response to a GUI event. This is difficult because the menu is set up during the initialization phase of the GUI, and there is an extensive event-handling sequence between where the subject performs an action and the code that actually responds to the specific user event.

Subjects used several strategies for crossing the GUI divide. In the Arrow task, they examined constants/parameters used in setting up the menu, inspected the class used in in setting up the menu, did a dynamic trace through the event handling code, and looked for classes that had interesting names and hoped.

All of these techniques worked eventually in the Arrows Task. Examining the references to the constants used in setting up the menu worked well in this particular case, although that is not always guaranteed. There might have been lots of uses of the constants. Tracing the parameters took longer than tracing the constants, but had there been lots of uses of the constants, it might have been faster and/or more reliable. Dynamic tracing worked, but the subject who did so (Peter) expressed some trepidation:

> I usually do not like putting breakpoints into draw methods because they get called a lot.

Inspecting the class used in setting up the menu worked for some subjects, but some were not able to recognize the germane calls.

In the Size Task, almost nobody tried to follow the execution trace from the menu, perhaps because the germane user-visible strings were the exceptionally common strings "File" and "new". Furthermore, if they did trace from File->New,

what they found was only part of what they needed, the window creation code. They also needed to find the code that handled the window *resizing*, and they were no user-visible strings that they could use to find that.

6.3.4 Misleading Language

In the Output task, where the subjects were asked to fix an output formatting problem, subjects were easily derailed by language in a number of places. To solve the Output task, subjects needed to fix where the number of fields in a class was output.

Bob, Dave, and Steve successfully traced statically from `main()` to the method `OutputWriter.writeOutput`, where at least six possible exploration paths presented themselves, as seen in Figure 6.1. (We modified the spacing and line breaks slightly to accomodate thesis margin requirement.)

Although it is possible to get to the method that needs fixing from `writeOutput()`, the subjects needed to traverse five different methods which have lots of references to reading or input and none to writing or output. The subject would need to open

1. `InputReader.readInput` with one signature, which calls
2. `InputReader.readInput` with a different signature, which calls
3. `DirectoryPump.pumpDataEntries`, which calls
4. `DirectoryPump.readFiles`, which calls
5. `DataEntryReader.read` (an interface), which is implemented by
6. `ClassFileRewriter.read`. The subject would need to read through to the penultimate line of
7. `ClassFileRewriter.read`, which calls
8. `ProgramClassFile.write`.

It is perhaps not surprising that none of those three (Bob, Dave, and Steve) managed to suspend disbelief long enough to get through all the cues telling them they were doing input.

```
// Construct the writer that can write jars, wars, ears,
// zips, and directories, cascading over the specified
// output entries.
DataEntryWriter writer =
    DataEntryWriterFactory.createDataEntryWriter(
                                        classPath,
                                        fromOutputIndex,
                                        toOutputIndex);

// Create the reader that can write classes and copy
// resource files to the above writer.
DataEntryReader reader =
    new ClassFileFilter(new ClassFileRewriter(
                                        programClassPool,
                                        writer),
                        new DataEntryCopier(writer));

// Go over the specified input entries and write
// their processed versions.
new InputReader(configuration).readInput("
                Copying resources from program ",
                classPath,
                fromInputIndex,
                fromOutputIndex,
                reader);

// Close all output entries.
writer.close();
```

Figure 6.1: Code from OutputWriter.writeOutput

Bob, Tom, and Steve all found `ClassFileRewriter` – the penultimate class in the chain above – by browsing through class names in the Package Explorer. However, only Steve managed to notice that the penultimate line of `ClassFileRewriter` had a call to `ProgramClassFile.write`. Steve was the only one who solved the problem in the allotted time.

Peter also solved the problem, although he was 55 seconds over the allowed

time. (He was given the extra time by mistake). Peter traced through the program statically from `main()` to `OutputWriter.writeOutput`, then traced dynamically. He traced 90 steps, carefully taking the time to understand what was happening and occasionally breaking into static tracing to explore side paths. By the time the dynamic trace brought him to `OutputWriter.writeOutput`, he understood that the important code was *re*writing, so was not frightened off by the methods having "input" in their names. He traced statically to `ClassFileRewriter`, then traced dynamically through to `ProgramClassFile`, where he discovered the errant line of code.

6.3.5 Missing Code

Subjects tended to spend a lot of time looking for missing code before realizing that it was missing. They happily dealt with missing code when they knew it was missing, either by figuring it out or being told that they needed to add code, as in a feature request. However, when we gave them a bug report, it was much easier for them to find code that was present but incorrect.

The closer the subject followed the execution path, the easier it appeared to be to recognize that code was missing. When subjects browsed for a class that "looked interesting" and followed relationships to the appropriate method, they had no confidence that that method would be executed when reproducing the bug. If they traced statically from the specific action that revealed the bug, then they could be more certain that that method was executed. If they traced dynamically to that method, they had proof that the method was executed.

In the Arrows task, everyone but Jim found the method, `setAttribute`, that had the correct code to manage arrow tips at the start of a line, but was missing the code to manage arrow tips at the ends of lines. However, most subjects then spent time working on finding out where the ending arrow tip attributes were set. Only Tom and Mark realized immediately that they were at the right place.

Table 6.3 shows the time that subjects first saw the `setAttribute` method, and the time that they recognized that the `setAttribute` method was the place where the end arrow tips were supposed to be set. Those who did not immediately recognize that `setAttribute` was missing code spent an average of 223 seconds

before they realized their error.

Table 6.3 also shows the strategy that they used to find `setAttribute` the first time, and what strategy they were using when they realized that setAttribute was the right place.

When subjects hit a dead end, they usually (eventually) switched to a strategy that followed the execution trace more closely. Steve was the only one who did *not* come to `setAttribute` via a directed trace. Steve saw `setAttribute` when scrolling through the entire `PolyLineFigure` class, then later reasoned his way to understanding that there was code missing. As mentioned in Section 4.1.1, Steve had particularly good reasoning abilities.

Table 6.3 below uses codes to represent the navigation strategy the subjects used. The first letter refers to how they found a starting point:

- T: found an interesting-looking class by using the Open Type dialog. Bob looked for classes that started with "arrow".
- B: found an interesting-looking class by browsing the Package Explorer
- M: found the place where the menu was created by searching for the user-visible menu item, e.g., "none" or "at Start"

The next letters refer to how the subject explored (followed) from the starting point:

- S: statically, by using References and Open Declaration commands
- D: dynamically, but stepping through line-by-line in the debugger
- S/D: hybrid, stepping through lines in the debugger but occasionally interleaving static traces
- R: reasoning

The next column shows the time when `setAttribute` first passed their eyeballs (regardless of whether or not they noticed it). The time is counted from when they finished reading the problem and/or reproducing the problem.

This table shows both that tracing closer to the execution path helps to recognize that code is missing, and how much time can be lost by not getting information about the dynamic trace.

Table 6.3: Seeing vs. recognizing missing code

Subject	first seeing	recognizing	first found	recognized missing code
Tom	4:43	4:43	MS	got first time
Peter	11:57	11:57	BS/D	got first time
Bob	6:58	12:07	TS	MS
Dave	11:17	14:07	MS	MD
Steve	12:02	14:19	BS	BR
Mark	5:43	15:58	BS	MS

6.4 Why Didn't Subjects Use Dynamic Tracing?

Using dynamic tracing avoids a number of the common following pitfalls discussed in Section 6.3. Why then, did our subjects use dynamic tracing so rarely? Excluding Jim, who was a bit of an outlier, subjects used dynamic tracing on only six of twenty-four tasks.

The fact that subjects so frequently broke out of dynamic tracing to do a little static tracing before resuming suggests that subjects find dynamic tracing to have some cost associated with it. During Peter's second task, he traced dynamically, then continued tracing the same exploration path statically, set a breakpoint, and resumed dynamic tracing at that point. Peter would have gotten to the same point if he had traced dynamically, but clearly felt there was an advantage to tracing statically.

We hazard a guess that subjects find dynamic tracing slow and somewhat tedious (as there is minimal cognitive effort involved in pressing the Step Over or Step Into buttons). Even though dynamic tracing was used rarely, stepping actions in the debugger still accounted for 17.9% of all navigation actions. (15.3% if you exclude Jim.)

While omniscient debuggers (discussed in Section 2.2.5) have appeal, tracing backwards dynamically in time might equally dull as tracing forwards.

In the interview, Bob admitted that he did not use the debugger because he felt that using the debugger was looked down upon. Bob said:

> [Other developers have implied] you should be able to just kind

> of look at it and remember interactions with libraries and stuff and quickly navigate through things and it should be obvious to you what's being used, the value of things.

Bob also implied that looking through code without using the debugger was part of the learning process.

None of the other subjects reported feeling this way (or even hearing others give that opinion), so we are not sure if this feeling was unique to Bob or if it is more widespread.

Even among those who used the debugger, there was widespread dissatisfaction with the layout in Eclipse's Debug Perspective. (A Perspective is a grouping of window panes and how they are arranged on the screen.) One subject (Peter) immediately customized the Debug Perspective to be more to his liking when he first hit a breakpoint; two others (Tom and Mark) both elected to use the Java Perspective instead of the Debug Perspective when debugging. When asked why they did not use the Debug Perspective, Mark and Tom said they did not like the disorientation that came from the Debug Perspective being different from the Java perspective. Peter and Tom said that they did not like the choice of window placement and sizing.

In the interview, Steve said that he only used the debugger about once per year, but did not explain why he did not use it.

6.5 Being "Systematic" or "Methodical"

Our observations did not completely support what other researchers have seen, but our observations did not completely reject them either. Soloway et al. [35] suggested that developers who read line-by-line through code were more successful at tasks, and we did not see that. Peter was definitely the most systematic of all of the coders, yet based on the scoring system described in 6.1 he was right at the median. Steve was by far the most successful at these tasks, yet he very much used a skim-and-guess approach.

We suspect our findings were different than the Soloway et al. findings because the source code was very large in comparison to the size of the task. We feel our subjects just did not have time to take a systematic approach and complete in time.

We did note that in the Arrows task, subjects who did not have success with one approach would find success with an approach that more closely followed the execution path. Skim-and-guess was not as effective as following statically from the menu selection, which in turn was not as effective as dynamic tracing.

Robillard et al. [30] found that developers who made a plan and stuck to it were more successful. Alas, none of our subjects recorded their findings, and they very rarely verbalized (and never wrote down) plans or even hypotheses.

It might be that the one task in Robillard et al., which the subjects had up to 125 minutes to complete, was complex enough that the subjects realized that they had to plan. Our tasks, at twenty minutes, were short enough and simple enough that our subjects might have been able to keep all their plans in their heads.

6.6 Threats to Validity

There were many threats to the validity of our findings.

6.6.1 Assistant

We had an assistant asking questions but not helping, or providing minimal help, during the tasks, as described in Section 4.3. This situation is unrealistic. We believe that most software developers work either by themselves, or pair-programming with a peer. We do not know how having an audience changed the way they worked, and whether any such changes were material in the conclusions drawn by this study.

We also do not know how consistent the assistant was in her interactions with the subjects. She was cleared to give the subjects some guidance if they went totally off-track, and for syntax errors. She varied in how much guidance she gave. For example, she reminded Dave that the task documentation said that he would not need to examine code related to shrinking, potentially preventing him from going down a dead end. Although we were not particularly interested in whether or not they finished the tasks, we did draw some conclusions about how successful a given technique was; their success might have been influenced by her comments.

It is also possible that she might have made some subjects less successful. It is possible that her occasional questions broke subjects' concentration. One argument

for her breaking subjects' concentration is that during the last three minutes of his last task, Mark was almost completely unresponsive to the assistant's six questions because he was too focused on what he was doing. In retrospect, it would have been interesting to ask the subjects how they felt about the assistant, but we did not.

In retrospect, it would have been interesting to ask the subjects how they felt about the use of the assistant, but we did not.

6.6.2 Code Base

We invested a lot of effort to find code bases that were medium-sized, not well-known, and had both GUI and non-GUI components. However, there is still the danger that the code base was not representative of modern Java projects. JHotDraw and ProGuard do not use technologies common in modern applications such as databases, object persistence, or any sort of remote method invocation. The types of problems encountered in this study and the strategies for solving them thus might not reflect actual problems and strategies found in professional software development.

It is also possible that we overestimated how much developers work with unfamiliar code. Although developers who are new to a company will usually face a completely unfamiliar code base, they might develop enough familiarity with a code base over time that they will have an easier time finding starting points.

Dave showed how the choice of code base might affect the strategies used. Dave said that at his workplace, they used a lot of computer-generated code, which meant that some tools (like Find References) were unhelpful in his workplace, essentially training Dave that he should not use them. As a result, Dave did not use them in the study.

6.6.3 Subjects

We had a very small sample size: seven men in total, six if you ignore Jim. All were from the greater Vancouver, British Columbia region. Four of the subjects reported six or more years of experience with Eclipse. Given that Eclipse was not

released until approximately 6.4 years before the user study,[2] this suggests a group of extreme early adopters of Eclipse.

It is certain that this handful of men living in Western Canada does not reflect the richness and diversity of all global software developers; we do not know how this affects the validity of the study.

6.6.4 Working in Isolation

Most code is just one piece of a much richer development ecosystem. There are multiple information sources to draw upon, including colleagues in a number of roles and a number of different textual information sources, including source code repositories, mailing lists, bug reports, and internal and external Web sites [18]. The task ecosystem of our user study was very barren compared to real-world software projects. Although our subjects did have our assistant at their elbows, she had been instructed to give minimal help. Although they had a task description at their fingertips, that was the only textual information they had. (Although we did not tell them not to access the Web, none of the subjects did.)

6.6.5 Short Tasks

The tasks needed to be short enough that we would be able to recruit volunteers, which limited how complex the tasks could be. This meant that the tasks were probably smaller than tasks developers are likely to encounter in the field.

In particular, while in interviews our subjects reported really disliking "tab spam", our subjects almost always could see all of the tabs that were open. It was rare that tabs were hidden. In addition, most people's working memory is able to keep track of seven plus or minus two things [23], and most of the time, the number of open files and tabs was within that range. As described in Table 6.4, in 21 of the 24 individual tasks (four tasks times seven subjects), subjects looked at nine or fewer files total. In 22 of 24 tasks, subjects never had more than nine tabs open at once, as seen in Table 6.4.

[2] http://www.eclipse.org/org/press-release/20061025cb_eclipsebirthday5.php, verified 9 July 2008

Table 6.4: File and tab count

Number of tasks with	Total files opened	Max tabs simultaneously open
Less than five	3	14
Seven plus or minus two	18	8
More than nine	3	2

6.6.6 Coding

As mentioned in Section 4.7.1, although we had two researchers coding the activity log for thirteen of the twenty-eight individual tasks, only Sherwood coded fifteen of the tasks. Sherwood also had the ultimate say on questions where there was contention. In addition, Sherwood did all of the transcription, all of the revisit codings, and most of the analysis. Ultimately, this study relies on her.

6.6.7 Learning Effects

The subjects had very little time to get used to Weta. This clearly was a factor for some of the subjects, as discussed in Section 6.1. In the interviews, four of six subjects mentioned some difficulty getting the hang of Weta's interface. It might be that we would see more benefit to Weta after developers used it for longer.

6.6.8 Weta Bugs

Weta had some bugs, as discussed in Appendix D. Although we specifically asked in the interviews about difficulties using Weta, only one person mentioned a bug. Those bugs might have interfered with subjects' ability to get the hang of Weta. In five of six interviews, with some prompting, the subject was able to think of a bug that he had seen. However, none of them made any indication that the bug was a serious issue for them.

Chapter 7

Future Work

This research raises many questions about how people act and suggests a number of interesting tools and modifications to Eclipse.

7.1 Navigation: Tools and Studies

As discussed in Section 6.1, at least some of the subjects took some time to get used to Weta. To better understand how a developer's navigation behaviour changes with changes in tabbing behaviour, it would be useful to distribute a plug-in with Weta-style tabbing, and collect and compare navigation behaviour with and without[1] the plug-in.

We believe that it would be useful to conduct a study that considers different ways of collecting information and working to assess the impact, both positive and negative, of different methods of eliciting information from the subject. As mentioned in 6.6.1, we are concerned the assistant might have interfered with the subjects' abilities, yet we do not know of any research papers that rigorously study such environmental effects on coders. Note that some forms of eliciting information *might improve* productivity. For example, some research suggests that certain forms of self-explanation can improve learning effectiveness [10]. We suggest a study that quantitatively measures large numbers of subjects doing the same task, but divided into groups that program in a variety of environments designed to cap-

[1] A large amount of archival data is available by request to software engineering researchers from murphy@cs.ubc.ca.

ture information: pair-programming, talking to a tape recorder about what they are doing/thinking, talking to a human being about what they are thinking, talking to a teddy bear about what they are doing/thinking, working with a human being sitting silently behind them, etc.

As discussed in Section 6.1.1, there is some evidence that having a short period devoted exclusively to generating many hypotheses before trying to solve a problem leads to faster problem-solving. It would be interesting to run a user study to measure the effectivenesss of a hypothesis-generation period on programming task success.

Omniscient debuggers (discussed in Section 2.2.5) might lead to different navigation styles. It would be interesting to compare navigational strategies between developers using an omnisicent debugger and those using a normal debugger.

7.2 Dynamic Information Visualization Tool

We believe that there is an opportunity for a tool to drastically reduce the difficulties associated with not knowing which code was actually executed when reproducing a bug, as discussed in Section 6.3.

There already exist code coverage tools that will, as a side effect, colour all the lines in the source based upon how fully they were executed. EclEmma, for example, colours lines green if they were fully executed, red if they were not executed, and yellow if they were partially executed.[2]

Unfortunately, EclEmma pays attention to the entire execution of the program. There is no way to specify that it should collect information only for part that is specific to a bug.

If a developer could instruct a code colouring tool when to start and stop gathering execution trace data – either via a type of breakpoint or by **Start gathering** and **Stop gathering** buttons – then the developer could see exactly which code was executed when executing the feature of interest. The developer might then not get misled into thinking that irrelevant code, such as initialization/teardown code, was relevant to the bug at hand. For the purposes of this discussion, presume that we have such a source colouring tool "DIVOT", named loosely for Dynamic Informa-

[2]http://www.eclemma.org/, verified 9 July 2008

tion Visualization Tool.

DIVOT would have some features that are similar but not identical to program slicing. Where a backwards slice contains the statements that *might* have an effect on the value of the slicing criterion [42], DIVOT would show which methods were *actually* executed.

There is some research that has investigated combining static and dynamic information. Richner and Ducasse lay the theoretical framework for combining static and dynamic information in query results. [29]. De Alwis and Murphy extend the theoretical framework, and report on a user study of a tool, Ferret, which, among other things, allows developers to restrict query results to those elements involved in an application's actual execution[7]. Only one of the Ferret subjects combined static and dynamic information, but he reported that "what was *actually* called was useful because it eliminated spurious calls made from other tests" (emphasis of the authors) [7, p.8]. The Shimba tool creates sequence diagrams that combine static and dynamic information [38]. Shimba requires the developer to switch from normal *following* behaviour to viewing diagrams in a separate tool.

For GUI applications, it is also possible that DIVOT could automatically split the data into initialization and post-initialization pieces, then display them separately. The tool could perhaps recognize when the waiting-for-user-input event loop started by looking for the name of a class/method that starts the event loop. The tool could perhaps be pre-loaded with the class/method name for common GUI frameworks like Swing, AWT, or SWT. The tool could also look for changes in the idle patterns.

Another limitation of current code coverage tools' integration with IDEs is that they only give cues about what is executed in the source pane, typically via colour-coding the source lines. When doing searches, or browsing for classes, the developer cannot tell if a given class or method was ever executed. If DIVOT showed only programming elements that were executed between the **Start gathering** and **Stop gathering** points, it might be far easier for the developer to browse or search for germane code. One way to achieve this would be to use Mylyn, adding execution information to Mylyn's Degree Of Interest model. (Mylyn is described in section 4.6.1.)

We believe that DIVOT would have helped our subjects avoid the common

following pitfalls mentioned in Section 6.3, as the subjects would no longer need to take an extra step to see if a method was executed. If a developer statically traced and found themselves in a "red" method, that would give them a strong clue that they should look for a superclass or subclass. Even if the code elements had misleading names, following green lines would lead them through the code that actually executed. With fewer elements to look through, browsing through the Package Explorer might be faster and less error-prone.

7.3 Eclipse Modifications

There are a few modifications/enhancements to Eclipse that we feel would be helpful. Improving bookmark management, developing a search for user-visible strings, and making it easier to find `main()` might be useful in any IDE. Redoing the JavaSearch UI is probably specific to Eclipse, however.

7.3.1 Bookmarks

One thing that was clear from how subjects used Weta is that developers will use a tool to track of locations in code if they have it. Given that one might imagine that bookmarks would be the appropriate way to do so, yet developers don't use bookmarks, suggests that there is room for improvement in bookmarks.

We asked for and received an enhancement to Mylyn that hides all bookmarks that were set when Mylyn tasks other than the current Mylyn task were active, so the bookmark list should stay smaller and more manageable.[3]

However, bookmark tools' limitations might not be the principal reason why developers do not use bookmarks: it might be that they just do not recognize well enough when they should be marking something for future reference. The fact that our subjects used the Back button much more often with Weta than with Eclipse is perhaps evidence that subjects do not know when to mark something for future use. On the other hand, developers seemed to find value in the TagSEA waypoints [36], which would suggest that developers are able to mark points well. Comparing usage logs from before the enhancement request was filed to logs gleaned after developers learned to use the enhancement might give insight into the utility of

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=201144, verified 9 July 2008

preventing runaway growth of bookmark lists is. (This could potentially be useful information for the Web-browser manufacturers as well.)

Abrams et al. [1] noted that some of their survey respondents reported liking to keep a time-ordering of their bookmarks, to help them remember what they had been working on recently. This might also be useful in code browsing, so we filed an enhancement request asking for the Bookmarks View to show the creation date of the bookmark.[4]

7.3.2 User-visible String Search

As mentioned in 5.1.1, most of our subjects used File Search only to look for user-visible strings. One could imagine a tool that would allow the user to search only for user-visible strings was displayed. We suspect that it would be difficult to write such a tool. We suspect that it would not be terribly useful to developers who were very familiar with the code base in question, nor to developers who worked on "well-behaved" code bases with the user-visible strings separated into resource files. However, this could be useful to people who need to quickly get up to speed on a lot of different code bases. Thus, we submitted a enhancement request for such a tool.[5]

7.3.3 Finding main()

We were surprised at how difficult it was for some of our subjects to find main(). It might be worthwhile to have a button or command somewhere that takes the user to the main() of the immediately preceeding run. This would probably not be hugely helpful, but it is probably also very easy to implement. We filed a enhancement request for this feature.[6]

7.3.4 Java Search

As mentioned in Section 6.2.2 Eclipse's Java Search feature is practically unusable. Four out of six times Jim and Dave tried to use it, they did it incorrectly. Steve used

[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=239519, verified 9 July 2008
[5] https://bugs.eclipse.org/bugs/show_bug.cgi?id=234019, verified 9 July 2008
[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=238036, verified 9 July 2008

it once successfully, and the other four subjects avoided it completely.

We believe that Java Search could be made much more useful with minor changes to the interface. Specifically, Java Search has radio buttons for which type of program element (class, method, interface, package name) the user wants to search for. It is easy to press the wrong button and get false negatives.

We filed a enhancement request asking for the radio buttons to be turned into check buttons, so that developers could easily search for the target string anywhere except the comments.[7] This would increase the false positives, but could potentially dramatically reduce the false negatives. This enhancement request has been assigned to someone (meaning that it is likely to be implemented), but has not been implemented at the time of this writing.

It might be interesting to compare data from the field on search usage from before this enhancement request is implemented and after. It might give an interesting quantitative piece of evidence on just how much of a difference UI makes.

[7] https://bugs.eclipse.org/bugs/show_bug.cgi?id=221081, verified 9 July 2008

Chapter 8

Conclusions

We cannot draw conclusions about the value of Weta-style tabbing for improving search strategies, for reasons we discuss in detail in Section 6.1.1.

However, there are a number of conclusions that we *can* draw from this study.

Developers want to mark locations in the code to come back to. Our subjects surprised us by not marking locations for future exploaration, but for places that they had already explored. They did this despite receiving training and encouragement to use tabs to mark places for future exploration. This tells us that developers have an unmet demand for bookmarking functionality.

Bookmarking, as currently implemented, is not useful. One's intuition might lead one to expect that developers would use the Bookmarks feature to mark locations in the code to come back to, but developers almost never used bookmarks. Many studies in both the Web browsing field and in computer science have also found that people set bookmarks, but have difficulty finding the right bookmark among a sea of bookmarks. We suggest that hiding all bookmarks but the ones set for the task at hand will ameliorate the problem. We requested, and already received, an enhancement to Mylyn that will do this.[1]

There are a number of common pitfalls that developers can avoid with better information about the runtime behaviour of the code. Developers had trouble crossing the GUI divide, getting lost in the superclass, getting sidetracked by misleading language, and recognizing when code was missing. Subjects did bet-

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=201144, verified 9 July 2008

ter when they more closely followed the execution path than when they skimmed and guessed. We propose a new tool, DIVOT, to help users better visualize runtime information while they statically trace, as discussed in Section 7.2.

Search in Eclipse is difficult to use correctly. The more experienced subjects avoided using search. When subjects did use search, they frequently made mistakes. We submitted a feature request to improve the Search Dialog.[2]

[2]https://bugs.eclipse.org/bugs/show_bug.cgi?id221081

Bibliography

[1] Abrams, D., Baecker, R., and Chignell, M. (1998). Information archiving with bookmarks: personal web space construction and organization. In *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 41–48, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. → pages 10, 86

[2] Aula, A., Jhaveri, N., and Käki, M. (2005). Information search and re-access strategies of experienced web users. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 583–592, New York, NY, USA. ACM Press. → pages 9, 11

[3] Brown, M. H. and Shillner, R. A. (1995). Deckscape: an experimental web browser. *Comput. Netw. ISDN Syst.*, 27(6):1097–1104. → pages 8, 64

[4] Bruce, H., Jones, W., and Dumais, S. (2004). Keeping and re-finding information on the web: What do people do and what do they need? *Proceedings of the American Society for Information Science and Technology*, 41(1):129–137. → pages 10

[5] Catledge, L. D. and Pitkow, J. E. (1995). Characterizing browsing strategies in the world-wide web. *Computer Networks and ISDN Systems*, 27(6):1065–1073. → pages 9, 11

[6] de Alwis, B., Murphy, G., and Robillard, M. (2007). A comparative study of three program exploration tools. In *15th IEEE International Conference on Program Comprehension*, pages 103–112. → pages 14

[7] de Alwis, B. and Murphy, G. C. (2008). Answering conceptual queries with ferret. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 21–30, New York, NY, USA. ACM. → pages 84

[8] Deline, R., Czerwinski, M., and Robertson, G. (2005). Easing program comprehension by sharing navigation data. In *VLHCC '05: Proceedings of the*

2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), pages 241–248, Washington, DC, USA. IEEE Computer Society. → pages 1, 12

[9] Gugerty, L. and Olson, G. M. (1986). Comprehension differences in debugging by skilled and novice programmers. In Soloway, E. and Iyengar, S., editors, *Empirical Studies of Programmers*, Human/Computer Interaction, Norwood, New Jersey. Ablex Publishing Corporation. → pages 15

[10] Hausmann, R. G. (2001). Can a computer interface support self-explaining? Master's thesis, University of Pittsburgh. → pages 82

[11] Hofer, C., Denker, M., and Ducasse, S. (2006). Design and implementation of a backward-in-time debugger. In *Proceedings of NODe 2006*, volume P-88, pages 17–32. → pages 16

[12] Janzen, D. and De Volder, K. (2003). Navigating and querying code without getting lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA. ACM Press. → pages 14

[13] Jones, W. P., Bruce, H., and Dumais, S. T. (2001). Keeping found things found on the web. In *Tenth International Conference on Information and Knowledge Management*, pages 119–126. → pages 10

[14] Kersten, M., Elves, R., and Murphy, G. C. (2006). Wysiwyn: Using task focus to ease collaboration. In *Supporting the Social Side of Large-Scale Software Development - CSCW Workshop 2006*, pages 19–22. → pages 37

[15] Kersten, M. and Murphy, G. C. (2006). Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 13th ACM SIGSOFT 14th international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA. ACM Press. → pages 1, 37, 38

[16] Klahr, D. and Dunbar, K. (1988). Dual space search during scientific reasoning. *Cognitive Science*, 12:1–55. → pages 15

[17] Ko, A. (2006). Debugging by asking questions about program output. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 989–992, New York, NY, USA. ACM Press. → pages 16

[18] Ko, A., Deline, R., and Venolia, G. (2007). Information needs in collocated software development teams. *International Conference on Software Engineering*. → pages 80

[19] Ko, A. J., Aung, H., and Myers, B. A. (2005). 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 Software engineering*, pages 126–135, New York, NY, USA. ACM Press. → pages 1, 43

[20] Ko, A. J., Myers, B. A., Coblenz, M. J., and Aung, H. H. (2006). An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987. → pages 13, 43, 56, 57

[21] Lewis, B. (2003). Debugging backwards in time. In Ronsse, editor, *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*. → pages 16

[22] Lintern, R., Michaud, J., Storey, M.-A., and Wu, X. (2003). Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–ff, New York, NY, USA. ACM. → pages 14

[23] MILLER, G. A. (1956). The magical number seven plus or minus two: some limits on our capacity for processing information. *Psychol Rev*, 63(2):81–97. → pages 80

[24] Murphy, G. C., Kersten, M., and Findlater, L. (2006). How are java software developers using the eclipse ide? *IEEE Software*. → pages 12

[25] Newfield, D., Sethi, B. S., and Ryall, K. (1998). Scratchpad: mechanisms for better navigation in directed web searching. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 1–8, New York, NY, USA. ACM Press. → pages 2, 8, 10, 15, 65

[26] Nickerson, R. S. (1998). Confirmation bias: a ubiquitous phenomenon in many guises. *Review of General Psychology*, 2(2):175–220. → pages 2, 15, 64

[27] Obendorf, H., Weinreich, H., Herder, E., and Mayer, M. (2007). Web page revisitation revisited: implications of a long-term click-stream study of browser usage. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 597–606, New York, NY, USA. ACM. → pages 11, 64, 65

[28] Pothier, G., Tanter, E., and Piquer, J. (2007). Scalable omniscient debugging. In *OOPSLA 2007*. → pages 16

[29] Richner, T. and Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 13–22. → pages 84

[30] Robillard, M., Coelho, W., and Murphy, G. (2004). How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903. → pages 1, 16, 78

[31] Safer, I. and Murphy, G. C. (2007). 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. ACM. → pages 33

[32] Schafer, T., Eichberg, M., Haupt, M., and Mezini, M. (2006). The sextant software exploration tool. *Software Engineering, IEEE Transactions on*, 32(9):753–768. → pages 14

[33] Schenk, K. D., Vitalari, N. P., and Davis, S. K. (1998). Differences between novice and expert systems analysts: what do we know and what do we do? *J. Manage. Inf. Syst.*, 15(1):9–50. → pages 2, 14, 64

[34] Singer, J., Elves, R., and Storey, M.-A. (2005). Navtracks: Supporting navigation in software. In *13th International Workshop on Program Comprehension (IWPC'05) pp. 173-175*, pages 173–175. → pages 1, 12

[35] Soloway, E., Lampert, R., Letovsky, S., Littman, D., and Pinto, J. (1988). Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267. → pages 16, 77

[36] Storey, M. A., Cheng, L. T., Singer, J., Muller, M., Myers, D., and Ryall, J. (2007). How programmers can turn comments into waypoints for code navigation. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 265–274. → pages 12, 13, 85

[37] Storey, M. A. D., Wong, K., and Muller, H. A. (1997). Rigi: A visualization environment for reverse engineering. In *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on*, pages 606–607. → pages 12

[38] Systä, T., Koskimies, K., and Müller, H. (2001). Shimba – an environment for reverse engineering java software systems. *Softw. Pract. Exper.*, 31(4):371–394. → pages 84

[39] Tauscher, L. and Greenberg, S. (1997). Revisitation patterns in world wide web navigation. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 399–406, New York, NY, USA. ACM Press. → pages 11

[40] Vans, M. A., von Mayrhauser, A., and Somlo, G. (1999). Program understanding behavior during corrective maintenance of large-scale software. *International Journal of Human-Computer Studies*, 51:31–70. → pages 2, 15, 64

[41] Vessey, I. (1986). Expertise in debugging computer programs: An analysis of the content of verbal protocols. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(5):621–637. → pages 2, 15, 64

[42] Weiser, M. (1981). Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA. IEEE Press. → pages 84

[43] Zhang, J. (2006). Idewaypoint: Support task-oriented ide navigation. Master's thesis, University of Victoria. → pages 13

Appendix A

Ethics Approval Certificate

The University of British Columbia
Office of Research Services
Behavioural Research Ethics Board
Suite 102, 6190 Agronomy Road, Vancouver, B.C. V6T 1Z3

CERTIFICATE OF APPROVAL - MINIMAL RISK

PRINCIPAL INVESTIGATOR:	INSTITUTION / DEPARTMENT:	UBC BREB NUMBER:
Gail C. Murphy	UBC/Science/Computer Science	H07-01637

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:

Institution	Site
UBC	Point Grey Site

Other locations where the research will be conducted:
Study participants will have the option of coming to UBC-Pt Grey, a public space, or doing the study at the participants' workplaces. Aside from the UBC sites, this study might be administered at up to ten software development companies in BC and/or Washington State. The study will use software on a desktop or laptop that we will provide, and we will give the participants the option of using their own keyboards, mice, and monitors. No software or proprietary information from the company will be involved in the study. This study will involve pairs of programmers. Each member of the pair may come from a separate company (given the difficulty of recruiting subjects in this area) In such cases, the researchers will work with the pool of solos to arrange mutually agreeable sites for pairs to meet

CO-INVESTIGATOR(S):

Kaitlin Sherwood

SPONSORING AGENCIES:

Natural Sciences and Engineering Research Council of Canada (NSERC)

PROJECT TITLE:
Evaluating tools for improving code navigation

CERTIFICATE EXPIRY DATE: July 28, 2008

DOCUMENTS INCLUDED IN THIS APPROVAL:	DATE APPROVED:
	July 28, 2007

Document Name	Version	Date
Consent Forms:		
Code Navigation Consent Form - main	1.4	July 25, 2007
Code Navigation Consent Form - main	1.5	July 26, 2007
Code Navigation Consent Form - extended	1.2	July 25, 2007
Advertisements:		
Code Navigation Recruitment Form	1.4	July 25, 2007
Questionnaire, Questionnaire Cover Letter, Tests:		
Code Navigation Questionnaire	1.3	July 15, 2007
Code Navigation Interview Script Guidelines	1.1	July 16, 2007
Letter of Initial Contact:		
Letter to company	1.0	July 17, 2007
Initial Contact Letter	1.1	July 25, 2007

The application for ethical review and the document(s) listed above have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board and signed electronically by one of the following:

Dr. Peter Suedfeld, Chair
Dr. Jim Rupert, Associate Chair
Dr. Arminee Kazanjian, Associate Chair
Dr. M. Judith Lynam, Associate Chair
Dr. Laurie Ford, Associate Chair

Figure A.1: Ethics Certificate

Appendix B

User Study Documents

This chapter has all the documents that the subjects received. Because of this thesis' formatting requirements, these are not exactly visual representations of what the subjects received, but the text is identical, and elements of the formatting were preserved where possible (e.g., italics, bulleted lists, etc). We have preserved typographical errors, but removed footers that gave the date and revision number.

B.1 About the study

Thank you for participating in this study!

In this study, we are investigating how developers navigate through code. You will do two programming tasks with regular, unmodified Eclipse and then do two tasks with our modified version of Eclipse (called "Weta").

B.1.1 Tasks

Two of the tasks will involve source from Proguard, a byte-code obfuscator. (You will get more information on Proguard later.) The other two tasks involve JHotDraw, a graphics editor. You will have 20 minutes to work on each task.

We are interested in how you navigate, not in whether you finish or not. Some of the tasks are difficult enough that you probably will not finish in 20 minutes. Do not be concerned: we are not trying to judge your abilities.

We are very, very interested in your thought processes, which is where Helen comes in. Please treat Helen like a new hire at your company who you are training. Assume:

- Helen just graduated with a BS CS, is bright, is good at programming from scratch, and knows Eclipse.
- Because of how her courses were taught, she has never had to understand other people's code.
- She is really bad at finding things in code.

You're going to demonstrate to Helen how you navigate through code, so she can improve her own code navigation skills.

We are very interested in the hypotheses you form about

- where points of interest might be
- what you think are good ways to search for them

Please tell Helen what places you are trying to find, what steps you are taking to get there, and when you change your mind.

B.1.2 Reminders about Eclipse

We want to make sure that you are aware of one of Eclipse's features: hyperlinking. If you hold down the control key, Java elements – fields, classes, and methods – become hyperlinks. Control-clicking on a Java element takes you to the definition of that element. This is very similar to pressing F3.

B.2 Proguard

B.2.1 What is Proguard?

Proguard's primary function is to make Java code harder to reverse-engineer. From the Proguard documentation:

> "By default, compiled bytecode still contains a lot of debugging information: source file names, line numbers, field names, method names, argument names, variable names, etc. This information makes it straightforward to decompile the bytecode and reverse-engineer entire programs. Sometimes, this is not desirable. Obfuscators such as ProGuard can remove the debugging information and replace all names by meaningless character sequences, making it much harder to reverse-engineer the code. It further compacts the code as a bonus. The program remains functionally equivalent, except for the class names, method names, and line numbers given in exception stack traces."

Proguard can do a few other things that you will not need to worry about for these two tasks, but you might see references to them in the code you explore. Proguard can:

- remove unused code ("shrinking")
- do some analysis to make sure that the bytecode cannot do "unsafe" things. (This is usually done by the Java compiler, but some compilers split this "preverification" into a separate step.)
- do a number of code optimizations

You do not need to worry about shrinking, preverification, or optimization.

B.3 Obfuscation Task

ProGuard currently changes symbol names to meaningless strings like "b" and "az". It uses a class SimpleNameFactory to come up with the simple strings that it uses for the symbols.

It would be even better to change symbol names to things like "import", "private", and "return". The VM doesn't know anything about reserved words, so it would still run the bytecode just fine. However, if anyone tried to decompile the code, it would be a mess to understand (and most compilers would choke). A line like this:

```
return foo(a[i]+b);
```

is much easier for the compiler to understand than something like this:

```
return for(return[private] + import);
```

B.3.1 What you need to do

In this task, you will add the capability to specify a file containing words to use for the symbol names.

You need to add a command line flag

```
-obfuscationdictionary \textit{filename}
```

which will cause ProGaurd to read words in the input file *filename* and use them as the symbol names. If there are more symbols than words in the file, then the rest of the symbols should use the standard "meaningless strings" names as before.

There is a class named DictionaryNameFactory which reads the file and generates the reserved-word names properly. You don't need to write this class yourself; you just need to plug it into the command line. You may alter DictionaryNameFactory if you choose, but there are solutions that do not require changing it.

B.3.2 Config file

The run configuration *with Dictionary* points to a config file, which contains all the command line flags, including `-obfuscationdictionary keywords.txt`. If you decide you want to look at the configuration file, ask Ducky to open it for you.

B.3.3 How to see the bug

1. From the menu, select Run->Open Run Dialog
2. Select the run configuration named *without Dictionary*
3. Click on the Run button
4. Wait for a little bit (it takes a while to run). You can monitor the progress in the Console.
5. Open the results folder.
6. Examine the file `withoutDictionary.map`

The .map file has lines like

```
boolean isTokenChar(char) -$>$ b
```

This means that the symbol for the method `isTokenChar(char)` in the original is now `b`. If you look at `goodWithDictionary withDictionary.map`, however, you will see lines like this:

```
boolean isTokenChar(char) -$>$ if
```

meaning that the symbol for that method is now the reserved word `if` (which is what we want).

B.3.4 How to know when you're done

1. From the menu, select Run->Open Run Dialog
2. Select the run configuration named *with Dictionary*
3. Click on the Run button
4. Wait for a little bit (it takes a while to run). You can monitor the progress in the Console.
5. Open the results folder.
6. Examine the file `withDictionary.map`

If you see mappings that use reserved words, you are done.

B.4 "Output" Task - fix output

The output of Proguard is wrong. The output .jar file is supposed to contain the number of fields in a class as a two-byte integer, but Proguard is currently writing it out as a four-byte integer.

B.4.1 What you need to do

Please fix the code so that that the number of fields in a class is written out to the .jar file as a two-byte integer.

B.4.2 How to reproduce the bug

1. From the menu, select Run->Open Run Dialog.
2. Select the run configuration named *without Dictionary*.
3. Click on the Run button.
4. Wait for a little bit (it takes a while to run). You can monitor the progress in the Console.
5. Open the folder "results" (which should be in the taskbar). If not, get Ducky to open that folder for you.
6. Right-click on `withoutDictionary.jar` and select `Properties`. The size should be 48,917 bytes.
7. Click OK.
8. Open the folder `goodWithoutDictionary`.
9. Get the size of `withoutDictionary.jar`. The size should be 48,902 – different from 48,917!

B.4.3 How to tell when you have fixed it

Compare the size of the output jar you create with the good version and see if they match.

It will be pretty obvious if you have gotten it right.

B.4.4 Config file

The run configuration *without Dictionary* points to a config file, which contains all the command line flags. If you decide you want to look at the configuration file, ask Ducky to open it for you.

B.5 Breadth-first- vs. depth-first-search navigation

There is a lot of research in problem-solving that indicates that people who do a more breadth-first-ish search find a solution faster than those who do a more depth-first-ish search. By breadth-first-ish, we mean that they examine a lot of possible solutions shallowly. By depth-first-ish, we mean that they examine one hypothesis at a time, and follow it until they can't go any further with it.

One suggestion for why this is true has to do with "confirmation bias":

> a tendency to search for or interpret new information in a way that confirms one's preconceptions and avoids information and interpretations which contradict prior beliefs. It is a type of cognitive bias and represents an error of inductive inference, or as a form of selection bias toward confirmation of the hypothesis under study or disconfirmation of an alternative hypothesis.

Basically, if you only have one hypothesis in your head, that can block you from seeing evidence that your hypothesis is incorrect.

Some things that can help you to do a more BFS-ish search:

- Make yourself think of several different possible answers to questions that you come up with. It might help to verbalize them and/or to write them down.
- Mark where you stopped working on a particular hypothesis. If it is easy to pick up "where you left off", it won't be as scary to "leave off". You can "pause" or "hibernate" a particular exploration path instead of "abandoning" it.
- Think consciously about what question you are trying to find the answer for. Framing your exploration as *a question with an answer* can make it easier to come up with hypotheses for what the answer might be.

B.6 About Weta

Unlike regular, "out of the box", Eclipse, Weta lets you keep track of multiple files per tab. This is similar to how modern tabbed Web browsers work. For example, in Firefox:

- URLs are underlined and in blue.
- Selecting the backwards/forwards arrows takes you to web pages that you opened in that tab, but you will not see pages that you opened in different tabs.
- If one tab is active and you click on something "live" the new page will open in the same tab. "Live" elements include:
  - Hyperlinks
  - Bookmarks (in the toolbar)
  - Bookmarks (in the sidebar)
  - History elements (in the sidebar)
- If you control-click on a "live" element, Firefox will open a new tab for the page.

Similarly, using Weta

- Weta turns Java elements – classes, methods, and fields – underlined and blue when you roll over them.
- Selecting the backwards/forwards arrows takes you to files that you opened in that tab, but you will not see files that you opened in different tabs.
- If one tab is active and you double-click on something "live", Weta will open the definition of that element in the same tab. "Live" elements include classes, methods, or fields in:
  - Java editor hyperlinks
  - Package Explorer

  - Call Hierarchy
  - Search results

- If you control-click on a "live" element, Weta open the definition in a new tab.

To reiterate and simplify broadly, with Weta:

- Double-click to open an element's definition in the same tab

and¡br¿

- Control-click to open an element's definition in a new tab.

B.6.1 BFS with Weta

Weta can help you use a more BFS strategy in much the same ways that tabs help BFS search for information on the Web.

Before tabs in Web browsers, you would need to search down one path, decide it wasn't working out, then use the Back button to retreat to a root location (e.g. Yahoo or Google). It was difficult to keep different exploration paths separate. (You could open a new browser window, but that was an expensive operation.)

With tabbed Web browsing, you probably go to a root location, then open a bunch of tabs in the background, and explore each one shallowly, quickly, before going too deep down any one of the paths.

Similarly, with stock Eclipse, you would need to search down one path, decide it wasn't working out, then use the Back button to retreat to a root location (i.e. the spot in the code where you decided to take execution path A instead of B). It was difficult to keep different exploration paths separate. (You could open a new Eclipse window, but that is an expensive operation.)

With Weta, you can explore until you get to a point where there are multiple things that might be worth exploring, then open a bunch of tabs, and explore each one shallowly, quickly, before going too deep down any one of the paths.

B.6.2 Implications

It is worthwhile to familiarize yourself with some of the implications of Weta's behaviour.

- **Single-click to select.** You might be used to double-clicking on a Java element to select it (for example, to highlight it everywhere in the file). Double-clicking in Weta will open that element's declaration! Instead, single-click on the element and wait approximately one second for Weta to highlight that element.

- **Weta keeps track of all dirty files in a tab, even if the file is no longer in the tab "stack".** If you open A, then B, use the back button to go to A, and then go to C, B will no longer be in the forward/backward history. (This is just like how Firefox and IE work.) However, if you have edited B, when you close the tab, it will remember that B is dirty and ask if you want to save it.

- **You can't select more than one thing at once** (like control-click or shift-click normally do in the Package Explorer or Search).

- **Just because an element is underlined and blue does *not* mean that it is selected.** Hovering over an element and then hitting F3, for example, is probably not going to give you what you want, alas. You need to click (once) on an element in order to select it.

B.6.3 Bugs

There are a few bugs in Weta:

- ***Save* and *Save all* aren't working right.** They don't always remember all of the files that are dirty but "hidden", i.e. not at the "top" of a tab's stack.

  - **Be careful about saving files before a run.** When you Run or Debug, Weta will not ask you if you want to save files that are not visible (i.e. at the "top" of the tab).

B.7 Arrows Task

When you create a connecting line between two shapes or nodes in the *NetApp* application, the connecting line shows arrows pointing in both directions (both shapes or nodes).

The style of the arrow can then be changed from the *Attributes -> Arrow* menu. You can slect where there should be arrowheads on the line:

- none
- at Start
- at End
- at Both

Unfortunately, this doesn't work properly. Regardless of what arrow style is chosen, NetApp will always draw an arrowhead at the end of the connecting line.

B.7.1 How to recreate the problem

- Select the menu option *Run ->Open Run Dialog->NetApp*
- Click the Run button.
- In NetApp, click on the square in the toolbar, then click elsewhere in the canvas to create a node.
- Repeat the previous step to create a second node.
- Select the arrow tool from the toolbar
- Hover the cursor over one of the nodes; four blue circles will appear.
- Drag a line from one of those circles to a circle on the other node.
- Select the line. (This is harder than it ought to be; if you have trouble, try clicking near an arrowhead.)
- Select the menu item *Attributes->Arrow->none*

You will see that the line still has an arrowhead at one side.

B.7.2 What we want

Please fix the JHotDraw framework so that all the attributes in the *Attributes -¿ Arrow* menu work correctly. In particular, if *none* or *at Start* are selected, there should be no arrowhead at the end of the connecting line.

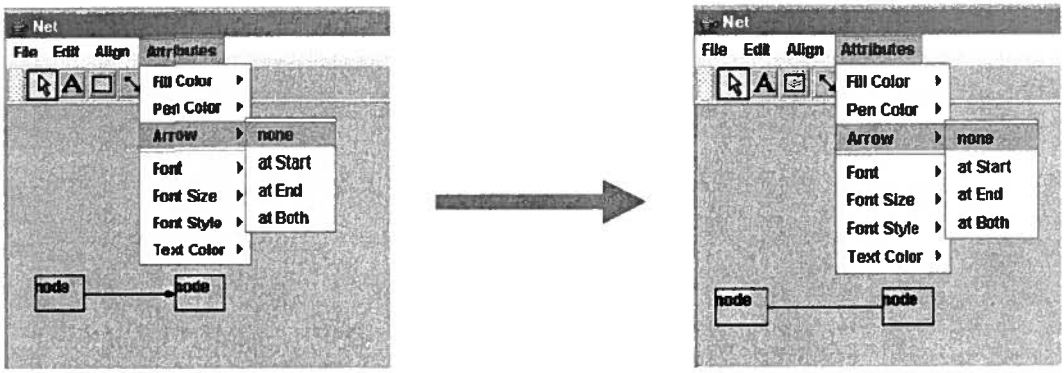

Figure B.1: Arrowheads

B.8 Size Status Line Task

Your coworker Joe got fired, leaving the JavaDraw application in a somewhat jumbled state. He had been working on a feature to display the size of the active drawing canvas in the application's frame. Joe's code puts the text "Active View Size:" in the application's trim, but doesn't actually put the actual height and width there. Please finish Joe's job.

B.8.1 How to recreate the problem

- Select the menu option Run ->Open Run Dialog->JavaDrawApp
- Click the Run button
- In JavaDrawApp, select the menu option File->New

Note that in the lower-right-hand of the frame, there is text "Active View Size", but no height and width.

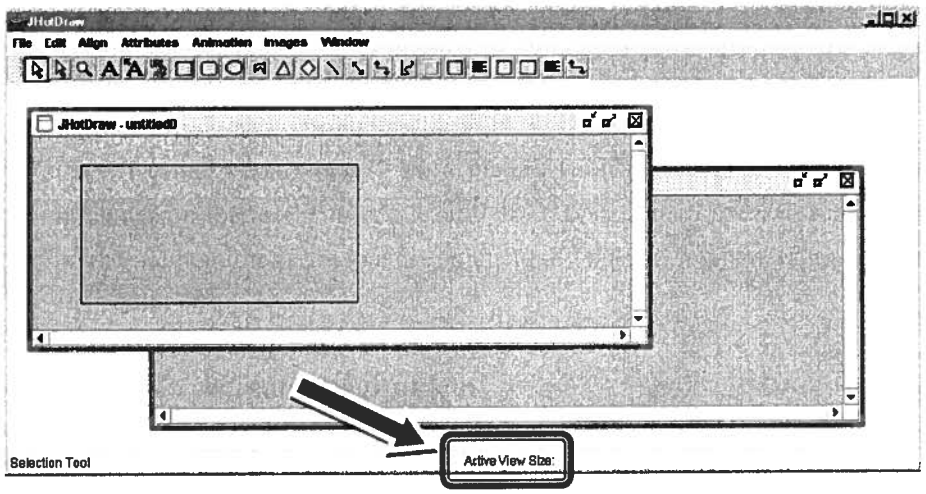

Figure B.2: Status bar with no numbers

B.8.2 What we want

We want the height and width to be displayed, as in the following picture:

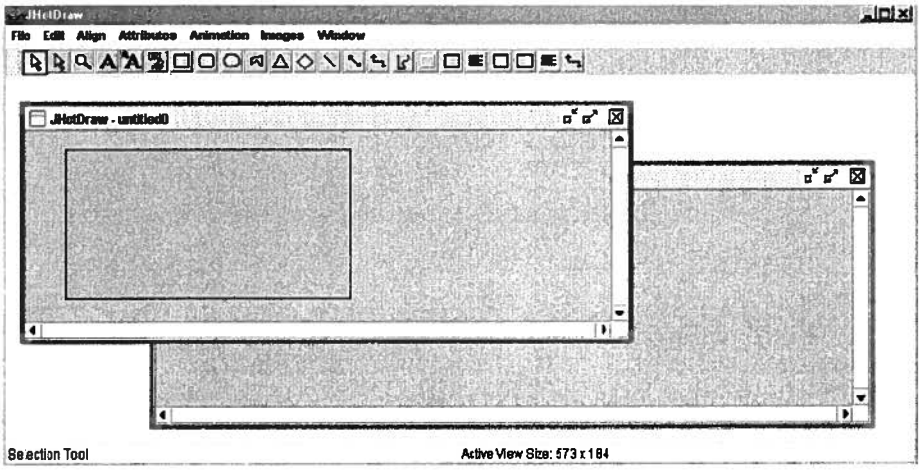

Figure B.3: Status bar with numbers

B.9 Code Navigation Questionnaire

Participant #:____

We are interested in how you feel Weta compares to the unmodified Eclipse in a number of different areas. If you didn't use a feature enough to have an opinion, draw a line through the entire row of answers.

We will collect the questionnaires as soon as you finish them (before the interview) but we will not look at them until after the interview.

Note: When we say Eclipse below, we mean the unmodified version of Eclipse.

What IDE do you use in your day-to-day work? _____

How long have you been using your IDE? _____

What fraction of the time do you program in Java currently (vs. another language?) _____

How long have you been using Java? _____

How long have you been programming? _____

What operating system do you normally use? _____

Comments on the above (optional):

_____

1. I liked Eclipse's way much more

2. I liked Eclipse's way a little more

3. I didn't really prefer one or the other

4. I liked Weta's way a little more

5. I liked Weta's way much more

By default, Eclipse opens a file in a new tab, while Weta opens in the same tab by default. Which do you prefer?

Tabbing behaviour 1 2 3 4 5

In Eclipse the Forward/Backward buttons do the same thing regardless of which tab you are in. In other words, they use a global history. When using Weta, the Forward/Backward buttons only take you to places that have been displayed in that tab. In other words, the Forward/Backward buttons use a per-tab history in Weta.

Which behaviour do you prefer?

Global / per-tab Forwad / Back history 1 2 3 4 5

Overall, do you prefer Eclipse's behaviour or Weta's behaviour?

overall preference 1 2 3 4 5

Comments on the above (optional):

---

1. Never

2. Less than once per day

3. About once per day

4. About five times per day

5. At least twenty times per day

How much did you use control-click (to jump to the declaration) before this session?

Prior use of control-click 1 2 3 4 5

How much do you expect to use control-click (to jump to the declaration) after this session?

Future use of control-click 1 2 3 4 5

Comments on the above (optional):

---

1. I have wanted this feature for a long time
2. I would probably use it multiple times per day
3. I might use it once per week
4. I would use it less than once per week
5. I don't think I would ever use it

How useful is it to have the same file in two different tabs?

usefulness of one-file-twoplaces 1 2 3 4 5

Comments on the above (optional):

---

1. Never had a problem with it
2. It happened a few times but I'm sure I'd get used to it
3. It only happened a few times
4. It happened a few times and really bugged me
5. I had the problem a lot

When using Weta, hovering over a Java element (field, class, or method) underlines it and turns it blue. Did you ever mistake that for the element being selected?

hovering 1 2 3 4 5

In Weta, you had to double-click to open the declaration in the same tab. Did you ever click once when you meant to open the declaration in the same tab?

double-click 1 2 3 4 5

Comments on the above (optional):

---

1. I only used DFS search strategies.

2.

3.

4.

5. I only used BFS strategies.

We talked about using breadth-first search vs. depth-first search. How would you characterize your strategies with the first two tasks (in Eclipse)?

BFS vs DFS - Eclipse 1 2 3 4 5

How would you characterize your search strategies with the last two tasks (in Weta)?

BFS vs. DFS - Weta 1 2 3 4 5

Comments on the above (optional):

---

1. Weta's tabbing made no difference on what strategy I used.

2.

3.

4.

5. Weta's tabbing strategy made a big difference on what strategy I used.

How much do you think the tabbing behaviour of Weta affected your choice of strategy?

effects of tools 1 2 3 4 5

Comments on the above (optional):

What other things did you really like/dislike?

Appendix C

Enhancement Requests

In the course of this research, we found a number of Eclipse features that we feel could be improved. We saw subjects draw incorrect conclusions on a number of occasions due to either misleading output or overly-difficult UI. We also saw subjects fail to make use of features that did exist because of perceived difficulty.

As a result, we logged a number of enhancement requests in the Eclipse bug-tracking system.[1] These include:

- #201144 – request bookmarks be filtered by Mylyn task
- #221081 – request Java search default to widest scope
- #221082 – request history drop-down menu should give method names, not just file names
- #221086 – request to allow sharing perspective configurations
- #221990 – request for a navigation history panel (like Photoshop's)
- #229185 – request to put all the limit options of Find dialog together, which includes changing the sense of "Wrap Search".
- #228541 – show subclasses for control-T

[1] http://bugs.eclipse.org/bugs, verified 9 July 2008

- #234009 – request for allowing setting "home" location and jumping to "home"
- #234019 – request for search for user-visible strings
- #238036 – request for button from Run Dialog to jump to main()
- #239519 – request for Bookmarks View to show creation date of bookmarks

We also filed one bug against Eclipse, #238202. We describe bug #238202 in detail in Section D.2.

Appendix D

Weta Bugs

Weta had a few bugs when we started the user study. We felt that these would not be significant issues for the user study. In fact, the only Weta bug that we saw anyone encouter is described in Section D.1, having to do with .class files.

Subjects also encountered two bugs that were also in Eclipse, but which appeared more often in Weta. The first had to do with losing one's place when scrolling, which we describe in Section D.2. It was significant enough that we fixed it after the second trial of the user study. The second was that the double- or control- click on a link itermittently would not be recognized, as described in Section D.3.

D.1 Switching between .class and .java Files

If the subject was looking at a .class file and opened a .java file in a new, tab, then the new tab would open to the *left* of the old tab instead of to the *right* of the old tab. The same would happen if they were looking at a .java file and opened a .class file in a new tab.

D.2 Losing Place

There is a bug in the stock Eclipse where the navigation does not get properly updated.[1] If you open a new file at the top of the file (location number one), scroll

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=238202, verified 9 July 2008

to location number two, switch to location number three in a manner that causes the file containing location number three to be hidden, and then attempt to go back to location number two, you will actually get taken to location number one. When using the standard Eclipse, it is relatively rare to encounter this bug. (If there are more tabs open than will fit on the tab rank, standard Eclipse uses a Most Recently Used policy to decide which tabs files in the tab rank and which get hidden. Thus, if there is room for five tabs across the tab rank, to hide a tab requires opening five more files. It is rare in Eclipse for developers go backwards enough times to discover that the location is not what they were expecting.) However, because of how Weta was implemented, it was much more common for our subjects to encounter this problem.

As a result, some of our early subjects encountered difficulties with the back button when using Weta.

We fixed that bug in the middle of the user study for both Eclipse and Weta.

D.3 Intermittant Loss of Clicks

Sometimes, control-clicking or double-clicking on a hyperlinked Java element would not open the element's declaration. We saw this with both our Eclipse setup and our Weta setup, but it was more common with Weta. Frequently, clicking on a different place in the same element would make Eclipse/Weta notice the event. Other times, subjects would use an alternative method for opening the declaration.

D.4 Open Blank Tab

With stock Eclipse, right-clicking on a tab and selecting "New Editor" will open a new tab containing the same file. That feature did not work in Weta, but we never observed anyone trying to use it.

D.5 Save All

Save All – invoked either via Shift-Control-S or via closing a running Weta application with open tabs – did not save dirty files that were not visible, i.e. were in the stack of files but not the current file. We never observed anyone trying to do either

a Shift-Control-S or close a Weta window.

Appendix E

Navigation codes

Table E.1 shows the codes used to annotate the interaction logs with navigation information.

Additional information would get attached to the code if it was only a few characters long (e.g., BPEK would be for browsing the Package Explorer to a known location) and appended to the end of the line if it was long (e.g., search patterns or method names).

There were some navigation codes that we tried to take, but which just proved too difficult. For example, if someone started scrolling and the assistant asked a question, and he paused for a moment, then resumed scrolling, was that one scroll or two? It was also very difficult to tell when someone hovered deliberately and when someone hovered accidentally.

For any operations that opens a file, in Eclipse, the default code is always used. In Weta, the default code is used if the same tab is used for the new file. (This is what happens when the subject double-clicks on something.) However, if the subject opens a file in a new tab (i.e., uses a control-click), then the "N" suffix is added.

Cancelled operation means that the subject did not complete the action for whatever reason. There are a number of search-like actions that have two steps: first typing in some text, and then selecting a Java element from the results. If they do not type something in or if they do not select from the list of results, both are considered a cancelled operation.

Table E.1: Navigation Codes

Action	Code	Additional information
Browse Package Explorer	BPE	+K=known location, U=unknown location
Browse Outline View	BOV	+Q=Quick Outline View
Open hyperlink	H	
Call hierarchy forward	CHF	
Call hierarchy backward	CHB	
Quick Call hierarchy	CHQ	
Type hierarchy	TH	+Q=Quick type hierarchy
Open references	R	
Open declarations (via menu)	D	
Save	S	name of file saved
Close tab	C	name of file closed
Tab select	TS	tab codes (see Table E.2)
Tab menu	TM	
History menu	HM	
Find	F	
Java Search	JS	search pattern and search code (see Table E.3
File Search	FS	search pattern
File Search – documents	FSd	
Open Type	OT	search pattern
Open Resource	OR	search pattern
Mylyn on	Mon	where: +PE=Package explorer, OV=outline view
Mylyn off	Moff	where, as above
Set breakpoint	BP	classname.methodname:line
Disable bp	BP-	classname.methodname:line
Enable existing bp	BP	classname.methodname:line
Run run	RR	
Run debug	RD	
Debug resume	DR	
Step over	SO	*classname.methodname:line*
Step into	SI	*classname.methodname:line*
Step return	SR	*classname.methodname:line*
Select from stack trace	SR	*classname.methodname:line*
Forward	FW	
Backward	BW	
Hover - show source	Hs	

Table E.2: Tab codes

Action	Code
Revisit - reference	rf
Revisit - resume	rr
Revisit - explain	re
Revisit - deadend/done	rd
Wandering aimlessly	v
Bring focus to newly-opened tab - o	
Close (bring focus to in order to close) - c	

Table E.3: Search codes (two letter)

Unknown (do not know what going to get)	U
Known location	K
User-visible string	U
Ordinary String (non-user-visible)	S

Misfired means that there was a disconnect between the command that the subject meant to execute and the one that was executed. For example, if the subject meant to open the Search Dialog, but opened the Open Type Dialog instead, that was a misfire.

Wrong means that the subject executed and/or interpreted the command wrong, e.g., wanted the references of a method but left the Search Dialog radio button set to search for declarations. A list of all the suffixes is in Table E.4.

Table E.4: Suffixes

Opened in a new tab	N
Cancelled operation	x
Misfired	m
Wrong	w

Note that we attempted to log scrolling actions, but it was very difficult to determine which methods were viewed on each time, and it was easy to forget to log a scrolling action. We do not put a high confidence in those codes, and did not use them in our analysis. Hovering over elements to see the associated Javadocs information was also extremely common, and it was extremely difficult to determine when it was accidental and when it was intentional. We did not use analyze anything having to do with those annotations. The scrolling and annotation codes are listed in Table E.5.

Table E.5: Ignored Navigation Codes

Action	Code	Additional information
Scroll	Sc	
Hover - show Javadocs)	Hd	A for Accidental, ? for unsure

Appendix F

Success measures

Table 6.1 gives the metrics used to determine how successful subjects were, as used in Section 6.1. Each subtask that the subject achieved was worth one point. Each task had a maximum of four points.

Arrow:

- Subject navigated from menu construction to event handling.
- Subject found `PolyLineFigure.setAttribute`.
- Subject understood that code was missing.
- Subject added correct code.

Size:

- Subject recognized that there were subclasses.
- Subject recognized that `setSize(int, int)` is the interesting method, not `setSize(Dim)`.
- Subject found either
  - `StandardDrawingView.setSize` or
  - `ZoomDrawingView.setSize`.

- Subject recognized that `ZoomDrawingView` was used when reproducing the bug.

Output:

- Subject found to `ClassFileRewriter`.
- Subject found to `ProgramClassFile`.
- Subject found to `ProgramClassFile.write`.
- Subject found `u2fieldsCount`.

Obfuscate:

- Subject added plausible code to the right places:
  - `Configuration`,
  - `ConfigurationConstants`, and
  - `ConfigurationParser.parse`.
- Subject used the right parse method (`parseFile`, not `parseOptionalFile`).
- Subject connected `DictionaryNameFactory`.
- Subject connected `DictionaryNameFactory properly`.