

Asking and Answering Questions about the Causes of Software Behavior

Amy J. Ko

May 2008

CMU-HCII-08-106 and CMU-CS-08-122

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Brad A. Myers (Chair)

Bonnie John

Jonathan Aldrich

Gail Murphy (University of British Columbia)

Submitted in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Copyright © 2008 Amy J. Ko. All rights reserved.

This work was supported by the National Science Foundation, under NSF grant IIS-0329090, by the EUSES Consortium under NSF grant ITR CCR-0324770, by an NDSEG Fellowship and by an NSF Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the National Science Foundation.

Keywords: debugging, program understanding, Whyline, natural programming, end-user software engineering, reverse engineering, productivity, defect, fault, Alice, Java, Eclipse, program slicing, execution trace, instrumentation, Crystal

ABSTRACT

Program understanding accounts for the bulk of software development work. Unfortunately, little is known about *why* it is so difficult. To investigate this problem, multiple developer populations were observed debugging. These studies revealed that developers start by asking questions about program behavior, but must answer by speculating about the code responsible. For example, a developer wondering, “Why didn’t this button do anything after I pressed it?” must conceive of a potential explanation such as “Maybe because its event handler wasn’t called” and then use breakpoint debuggers, print statements, and other low-level tools that instrument and analyze code to verify their explanation. The studies showed that not only is this process poorly supported by current tools, but also that developers form valid explanations for only 10-20% of their attempts.

A new kind of program understanding tool called a Whyline was developed, which allows a developer to ask “why did” and “why didn’t” questions directly about a program’s output. In response, the tool determines which parts of the system and its execution are related to the output, also identifying any false assumptions the developer might have about what occurred during the execution of the program. This interaction helps developers avoid speculating about the cause of output, instead allowing the Whyline find code related to the output in question. Three prototypes were developed, supporting Alice (a programming language designed for building interactive 3D worlds), the Java programming language, and a word processing application for end users. In controlled experiments, all three prototypes significantly reduced time to complete debugging tasks (20-800% faster) and significantly increased success rates (by 20-200%).

ACKNOWLEDGMENTS

I would like to thank my advisor, Brad Myers, for his critical eye and foresight. It has been a pleasure learning from his experience and insights for these past six years. I am also deeply indebted to Margaret M. Burnett, who took a chance on me the summer after my freshman year at Oregon State, allowing me to discover the world of research and all its perils before I decided to pursue graduate school. Without her early guidance and encouragement, my years as a Ph.D. student would have been far less productive.

My committee members also deserve thanks for their encouragement and criticism. Bonnie John was especially helpful in looking for confounds in my experimental designs. Jonathan Aldrich shared crucial perspectives on my work from the programming languages and software engineering communities. Gail Murphy has been a strong supporter of my work and she provided invaluable insight into the workings of the software engineering research community.

Every dissertation is a joint effort and mine is no exception. My family deserves special thanks, including my wife Katherine, my daughter Ellen, my father Robert, my mother Judi and my brother Bryan.

I also offer thanks to several students, faculty, staff and friends that have made this work possible, including Rob Adams, Turadg Aleahmad, Sonya Allin, Htet Htet Aung, Lisa Anthony, Danviel Avrahami, Aruna Balakrisnan, Ryan Baker, Aaron Bauer, Laura Beckwith, Andrew Begel, Kenneth Berger, Katie Bessiere, Jake Biel, Moira Burke, Polo Chau, Sarah Culberson, Michael Colbenz, Marian D'Amico, Laura Dabbish, Scott Davidoff, Rob DeLine, Anind Dey, Tawanna Dillahunt, Nathalie Dima, Matt Easterday, Brian Ellis, Martin Erwig, Andrew Faulring, James Fogarty, Jodi Forlizzi, Valentina Grigoreanu, Darren Gergle, Gahgene Gweon, Elspeth Golden, Chris Harrison, Gary Hsieh, Jason Hong, Scott Hudson, Amy Hurst, Sara Kiesler, Miryung Kim Ho, Robert Kraut, Queenie Kravitz, Thomas LaToza, Johnny Lee, Joonhwan Lee, Sandi Lowe, Neema Moravegi, Bilge Mutlu, Nachi Nagappan, Yoko Nakano, Jeff Nichols, Christine Neuwirth, Sue O'Connor, John Pane, Randy Pausch, Madhu Prabaker, Ido Roll, Chris Scaffidi , Peter Scupelli, Mary Shaw, Fleming Seay, Jeff Stylos, Karen Tang, Cristen

Torrey, Angela Wagner, Erin Walker, Jake Wobbrock, Jeff Wong, Ruth Wylie, Thomas Zimmermann, and many others. Each of these individuals helped spark an idea, lifted me out of the Ph.D. doldrums with a kind word, said something that questioned my assumptions, or cast a new perspective.

Thank you also to the corporate and governmental sponsors of this work, including the National Science Foundation, the Department of Defense, Microsoft, and Adobe. I was also generously supported by a NSF Graduate Research Fellowship and a National Defense Science and Engineering Graduate Fellowship.

TABLE OF CONTENTS

Abstract	iii
Acknowledgements	iv
Table of Contents	vi
List of Figures and Tables	x
1. Introduction	1
1.1. The Problem.....	2
1.2. A Solution.....	2
1.3. An Approach.....	3
1.4. Definitions.....	5
1.5. Contributions.....	6
1.6. Outline.....	6
2. Related Work	8
2.1. Studies of Program Understanding.....	9
2.2. Causes of Software Errors.....	13
2.3. Program Understanding Tools.....	24
2.4. Summary.....	33
3. Breakdowns in Alice	34
3.1. The Alice programming environment.....	35
3.2. Alice in the Field.....	36
3.3. Alice in the Lab.....	39
3.4. An Analysis of Breakdowns.....	41
3.5. Limitations.....	60
3.6. Summary.....	60
4. Learning Barriers in VB.NET	62
4.1. Method.....	63
4.2. Six Learning Barriers.....	65
4.3. Discussion.....	69
4.4. Limitations.....	71
4.5. Summary.....	71

5.	Exploring Java Code in Eclipse	73
5.1.	Method.....	74
5.2.	Results.....	79
5.3.	Limitations.....	96
5.4.	Implications for Theory.....	99
5.5.	Implications for Tools.....	103
5.6.	Summary.....	109
6.	Information Needs at Microsoft	110
6.1.	Method.....	111
6.2.	Task Structure.....	112
6.3.	Information Needs.....	113
6.4.	Quantifying Information Needs.....	122
6.5.	Most Common Information Needs.....	124
6.6.	Rating Information Needs.....	125
6.7.	Discussion.....	126
6.8.	Summary.....	130
7.	The Whyline Concept	131
7.1.	Asking about Output.....	132
7.2.	Questions from Code.....	133
7.3.	Explaining Causality.....	134
7.4.	Summary.....	135
8.	A Whyline for Alice	136
8.1.	An Example.....	137
8.2.	User Interface.....	140
8.3.	Implementation.....	141
8.4.	Evaluation.....	145
8.5.	Discussion.....	148
8.6.	Limitations.....	150
8.7.	Summary.....	151
9.	A Whyline for Applications	152
9.1.	An Example.....	153
9.2.	User Interface.....	154
9.3.	Implementation.....	159
9.4.	Evaluation.....	162
9.5.	Discussion.....	166

9.6.	Summary.....	167
10.	A Whyline for Java	168
10.1.	An Example.....	169
10.2.	User Interface.....	171
10.3.	Implementation.....	185
10.4.	Evaluation.....	210
10.5.	Discussion.....	223
10.6.	Summary.....	225
11.	Limitations and Future Work	227
11.1.	Limitations.....	227
11.2.	Future Work.....	234
11.3.	Summary.....	240
12.	Conclusions	241
	Appendix	245
	Bibliography	304

FIGURES

Figure 1.1. On the top, a question about the color of a rectangle using the Whyline for Java. On the bottom, the corresponding answer, showing the source of the black color and the causes of its creation (<code>gSlider</code> was used twice instead of <code>bSlider</code>).	4
Figure 1.2. A historical map of the contributions in this dissertation and their corresponding chapters. The arrows represent the approximate flow of ideas from studies to tools and tools to studies. The height of the boxes approximate the time spent on each portion of the research.	7
Figure 2.1. Dynamics of software error production, based on Reason's systemic view of failure. Each layer has latent errors (the holes), predisposing certain types of failures. Layers also have defenses against failures (where there are no holes). Several layers of failure must go unchecked before software errors are introduced into code.	17
Figure 3.1. The Alice 2 programming environment.	35
Figure 3.2. A ‘Building Virtual Worlds’ programmer fine-tuning an animation.	37
Figure 3.3. An image from the recording of a study participants’ work on the Pac-Man task.	40
Figure 3.4. A framework for describing the causes of software errors based on chains of cognitive breakdowns. Breakdowns occur in specification, implementation, and runtime activities. A single breakdown is read from left to right and consists of one component from each column within an activity. The cause of a single software error can be thought of as a path through these various types of breakdowns, by following the “can cause” arrows between and within the activities.	43
Figure 3.5. An example of a chain of cognitive breakdowns, where a programmer has several breakdowns while implementing a recursive sorting algorithm.	46
Figure 3.6. Deductively reconstructing the causal chain of breakdowns represented in Figure 3.5, using a programmer’s actions and verbal utterances.	49

Figure 3.7. A segment of one of P2's cognitive breakdown chains. The last breakdown shown here did not cause further breakdowns until 20 minutes later, after the camera position made it apparent that Pac was still jumping.	54
Figure 3.8. A model of the major causes of software errors during programmers' use of Alice. Each line represents a causal link between one type of breakdown and another; the number on the line represents the proportion of the type of link out of all links in all chains. (This does not include numbers for the links between software errors, runtime faults, and runtime failures, since it was only possible to identify errors that led to failures; also, the numbers do not add to 100% because not all types of breakdowns are shown).	59
Figure 4.1. In overcoming barriers, learners risk making invalid assumptions that often lead to error.	63
Figure 4.2. Learning barriers overcome with invalid assumptions often led to insurmountable barriers of a different type.	64
Figure 4.3. For surmountable barriers, the percent of each type overcome with invalid assumptions, and the type of barrier to which the assumptions led.	70
Figure 5.1. The Paint application.	75
Figure 5.2. The flashing taskbar notification (top) and one of the arithmetic interruption tasks (bottom).	78
Figure 5.3. Developers' division of labor in terms of time on activities. The vertical bars represent one standard deviation above and below the mean.	83
Figure 5.4. The developers' actions for the Thickness and Yellow tasks.	84
Figure 5.5. The package explorer, file tabs, and scroll bars of Eclipse 2.0.	90
Figure 5.6. A model of program understanding in which developers search for relevant information and relate it to other relevant information while collecting information necessary for eventually implementing a solution.	101

Figure 5.7. The 50 lines of code and other information that developer B indicated as relevant, portrayed in a mockup of a workspace that help developers collect relevant information for a task in one place, independent of the structure of a program.	106
Figure 5.8. Jasper, an Eclipse plug-in that allows developers to gather arbitrary fragments of Java code in a single view.	108
Figure 6.1. An excerpt from J's observation log.	112
Figure 6.2. The backgrounds and task structures of the 17 observed developers. The right edge of each task block indicates the reason for the task switch (Tasks are labeled time blocks with coded ends (thin line for done, thick line for blocked, jagged line for interrupted). When a task gets broke up by interruptions or blocking, we draw its fragments at the same vertical level to show resumption.	114
Figure 6.3. Types of information developers sought, with search times in minutes; perceptions of the information's importance, availability, and inaccuracy; frequencies and outcomes of searches; and sources, with the most common in boldface.	123
Figure 6.4. Information needs per participant.	124
Figure 7.1. Asking about precursors to output presumes that the fault did occurred before the precursor, which is not always the case.	133
Figure 8.1. The Alice programming environment, before the world has been played: (1) the object list, (2) The 3D world view, (3) the event list, (4) the currently selected object's properties, methods, and questions tabs, and (5) the code area.	137
Figure 8.2. Ellen expected Pac to resize, but he did not.	138
Figure 8.3. Ellen explores the questions and decides to ask "Why didn't Pac resize .5?" which highlights the code.	138
Figure 8.4. The Whyline's answer shows a visualization of the runtime actions preventing Pac from resizing. Ellen uses the time cursor to "scrub" the execution history, and realizes that Pac did not resize because isEaten was true.	139
Figure 8.5. A false proposition answer, which explains to the developer that the code they thought did not execute, actually did execute.	144

Figure 8.6. An invariant answer, which explains to the developer that the code can never be reached.	145
Figure 9.1. The answer for why “Teh” was changed to “The” The “?” in the upper left shows where the F1 key was pressed.	154
Figure 9.2. Menus resulting from typing F1, showing sub menus for character and paragraph properties.	156
Figure 9.3. A question menu about whitespace.	156
Figure 9.4. The “global” why menu, showing recent commands that did and did not execute.	157
Figure 9.5. The user typed “g” in Figure 5 while “helpful” was selected, so “helpful” was deleted. Crystal inserts an invisible marker in the text so a question will appear about the deleted object.	157
Figure 9.6. The answer to “Why is the ‘p’ bold?”, explaining that the user set the property using the toolbar button.	158
Figure 9.7. The answer shown for when a property’s value, in this case the font size, is inherited from a style.	158
Figure 9.8. Percent of people in each group that completed the tasks and the overall average. Taller bars are better.	164
Figure 9.9. For the participants who could complete the task, the average time they took, with bars showing the standard error of the mean. Shorter bars are better.	165
Figure 10.1. Using the Whyline: (a) The developer demonstrates the behavior; (b) after the trace loads, the developer finds the output of interest by scrubbing the I/O history; (c) the developer clicks on the output and chooses a question; (d) the Whyline provides an answer, which the developer navigates (e) in order to understand the cause of the behavior (f).	170
Figure 10.2. The main Whyline window, showing the user’s launch configurations on the left and saved recordings on the right.	172
Figure 10.3. The launch configuration window.	172
Figure 10.4. The loading progress bar in the Whyline window.	173

Figure 10.5. Hovering over graphical, textual, and exception output. The graphical and textual output both include pop-ups indicating the temporal context of the output (such as “after this was printed” and “after this window repainted”).	174
Figure 10.6. Questions about properties of a rectangle.	175
Figure 10.7. Questions about fields and methods that indirectly affect output.	175
Figure 10.8. The time slider, showing only mouse (the left-most icon is selected, indicating this filter).	175
Figure 10.9. Time relatively for positively and negatively phrased questions.	176
Figure 10.10. Token level highlighting in the Java Whyline source viewer and crosshatching over an unfamiliar source file.	177
Figure 10.11. Three types of selections (tokens, lines, and methods) and the menus for each shown upon clicking.	178
Figure 10.12. Some event selections will show multiple files, if multiple files are relevant to the selection. The example above shows both the use of the field color and the assignment to the field color, because the user has selected a question about why the field color had its current value.	179
Figure 10.13. Threads separated along the y-axis.	180
Figure 10.14. The meaning of various colors in the Java Whyline.	180
Figure 10.15. Followup questions about the selected execution event.	182
Figure 10.16. An answer showing (1) a collapsed invocation, (2) a hidden call context, (3) several instructions not executed and (4) a conditional that evaluated in the wrong direction, preventing the desired instruction from executing.	183
Figure 10.17. The source file outline and search field.	184
Figure 10.18. The simplified output history and time controller, which shows the position of the currently selected event in the visualization.	184
Figure 10.19. The object watch window (top) and the threads, call stacks, locals and objects (bottom).	185
Figure 10.20. Algorithm <code>getSources</code> , which gathers instructions that could produce a value for a given instruction’s argument, and <code>getArraySources</code> , which gathers instructions that could produce values for an array.	195

Figure 10.21. The textual output question user interface.	199
Figure 10.22. Algorithms <code>markAffectors</code> and <code>markInvokers</code> , which mark methods and fields that affect or invoke output (the two algorithms do not invoke each other).	202
Figure 10.23. All supported questions for a graphical output event in the Java Whyline prototype, showing six types of questions currently supported by the prototype (numbered 1-6) and three types of menus. For each, the content on the left lists the meaning of the question (items in []'s represented nested menus of the specified type) and the content on the right gives an example screen shot.	203
Figure 10.24. Algorithm <code>whynotvalue</code> , which explains why a certain dynamic data dependency did not occur.	210
Figure 10.25. ArgoUML bug 3121, titled “Remove ‘Report Usage Statistics’ since it does not do anything.”	216
Figure 10.26. ArgoUML bug 3128, titled “Problems with two classes with the same name in different packages”.	217
Figure 10.27. For task 1, the number of successful participants and the time on task.	218
Figure 10.28. For task 2, the number of successful participants and the time on task.	219

TABLES

Table 2.1. Studies classifying “bugs”, “errors” and “problems” in various languages, expertise, and programming contexts.	13
Table 2.2. Actions performed during programming activity.	16
Table 2.3. Types of skill breakdowns, adapted from [Reason 1990]. The → means “causes.”	19
Table 2.4. Types of rule breakdowns, adapted from [Reason 1990].	21
Table 2.5. Types of knowledge breakdowns, adapted from Reason [1990].	23
Table 3.1. For the BVW study, programmers’ self-rated programming language expertise, their total observed work time, and the tasks that they worked on during observations.	37
Table 3.2. Details about the four participants of the Pac Man study.	40
Table 3.3. A summary of skill, rule, and knowledge breakdowns, which can be used to answer deductive questions from observations.	51
Table 3.4. Programming and debugging time, and the number of software errors, breakdowns, and chains, as well as chain length, by programmer.	55
Table 3.5. Breakdowns split by activity and type.	56
Table 3.6. Frequency and percent of breakdowns and software errors by type of information and the average debugging time for software errors in each type of information.	56
Table 3.7. Software errors and debugging time by cognitive breakdown type and action. Only actions causing software errors are shown.	57
Table 4.1. The seven Visual Basic.NET tasks.	64
Table 5.1. The five maintenance tasks.	77
Table 5.2. Developer actions transcribed from the screen-captured videos.	80
Table 5.3. Task completion statistics for the ten developers, including the average time spent on each task and the number of actions per task per developer.	82
Table 5.4. Types of dependencies navigated, the average percent of each type for a developer, and the tools that developers used to perform each.	87

Table 5.5. An approximation of developers' task contexts for THICKNESS and LINE, derived from edits, dependency navigations, and searches.	93
Table 8.1. Frequency of question/answer types in each study and times the Whyline was found useful for each.	146
Table 9.1. Fields and methods of the command objects in Crystal. Properties in bold are novel.	159
Table 10.1. All of the supported keyboard commands in the Whyline visualization.	182
Table 10.2. The 55 different kinds of events recorded by the Java Whyline. The constant, value, and argument categories contain 8 events each, to cover each of the 8 primitive types in Java. The group of six events at the bottom are custom instrumentation to capture certain I/O events.	191
Table 10.3. The file hierarchy of a recorded Whyline trace.	193
Table 10.4. Statistics about tracing slow down, trace size with and without compression, and trace loading time, on five open source Java programs, averaged over ten runs. The profiling times were computed using the YourKit Java profiler with tracing mode on (rather than sampling). Lines of code for each program were computed omitting whitespace lines.	211
Table 10.5. Nine bug reports and the Whyline questions that could be asked.	212
Table 12.1. Knowledge contributions from empirical studies of developers.	242
Table 12.2. Technical contributions across three Whyline prototypes.	243

1.

INTRODUCTION

Software is buggy for many reasons. Developers write programs purposefully and carefully, but despite their best efforts at vigilance, they miss things, they skip edge cases, they overlook exceptional circumstances. Sometimes conditions and requirements change and the original assumptions in a program's design are violated. In other cases, companies sacrifice quality for time-to-market in order to survive.

Of course, software errors are only a problem when they result in software *failures*. When they do, there are often long periods of disarray as developers scramble to diagnose the problems. For example, in 2007, the Canadian Revenue Agency's electronic tax filing system experienced an undiagnosed failure for almost a week, causing mass confusion over temporary filing guidelines due to the bug¹. On January 24th, 2004, the Mars rover Spirit ceased communications with mission control. Engineers worked for nearly a week on the theory that the rover was in a reboot loop caused by hardware failure, only to find that the problem was due to a file system beyond capacity².

What makes these incidents so problematic is not the initial failure, but the *repeated* failures that occur as engineers work tirelessly to diagnose the problems. Unfortunately, finding the causes of these software failures is no simple task. In 2002, the National Institute of Standards and Technology estimated that 30-90% of the costs in successfully developing software are in testing and debugging; they

¹ <http://www.cbc.ca/canada/story/2007/03/08/tax-computer.html>

² <http://www.planetary.org/blog/article/00000702>

further found that the average error takes 17.4 hours to find and fix [Tassey 2002]. According to the respondents of the study, the top reason for these numbers is the lack of effective tools. Every day, millions of software developers work to improve the world's software infrastructure, but must do so with little more than a breakpoint and a print statement.

1.1. THE PROBLEM

An essential part of knowing what kinds of diagnostic tools could help these developers, is knowing precisely *why* this diagnostic activity is so difficult. I performed several studies that examined this question and a common explanation emerged. For any given software failure, there must be some observable symptom of the failure (if there were not, we would not know of the failure). This might be a wrong value on a display, the lack of response on the other end of a network connection, or any other variety of unexpected behavior. When a developer sees such a failure, they must *guess* about its cause. This guess determines what breakpoints they might set, what print statements they might write, and what tools they might use to explore the code. The problem is that novices and experts alike usually guess wrong the first time. They search and test and eventually discover some other fact about the program that leads them closer to the cause, but only after exploring other unrelated code. To make matters worse, many failures are a *lack* of feedback. For example, a developer might press a button and expect it to have some effect, but see nothing. In these situations, there is little feedback that the developer can use to speculate about the cause of the problem.

1.2. A SOLUTION

What if instead of speculating about the cause of a failure, developers could analyze the observable symptoms of a failure *directly*? For example, if a developer saw an incorrect value in a user interface, what if the developer could click on that value and ask why it was produced? If they did not see a value but they expected one, what if they could ask why it did not appear? If such a tool existed, the tool could take the developer from the faulty *output* to the code responsible, eliminating much of the guesswork required by today's tools. This leads to my thesis:

A tool that allows developers to ask questions explicitly about a program’s output and behavior can significantly improve developers’ productivity and solutions with debugging and software maintenance tasks, relative to conventional program understanding tools.

The goal of this dissertation work is to investigate this claim, inventing new ways of understanding program execution by asking questions about program output. These inventions center around the concept of a *Whyline*, which is a kind of tool that allows a developer to ask “why did” and “why didn’t” questions about program behavior. The tool derives questions directly from the program itself by performing a number of static and dynamic program analyses based on a recording of the program’s execution. It then answers the developer’s questions using additional analyses, helping the developer to explore the causal relationships between the queried output and the program’s execution.

To illustrate, consider the question in Figure 1.1. In this example, the user expected the black rectangle to be blue, since the blue color slider was at its maximum value. The user can choose a question about the color of the rectangle and the system responds with an answer that shows where the black color originated. The developer can then follow up with other questions, such as where the individual color components of the color originated. This is only a glimpse of the range of interactions with Whyline tools. As shown in later chapters, these types of questions, which allow developers to work backwards from output, can dramatically reduce the time required to diagnose a program failure.

1.3. AN APPROACH

The approach throughout this work is classic Human-Computer Interaction research. This dissertation explores program understanding and debugging as *human activities*, observing different populations of programmers and different kinds of tools, looking for commonalities. These studies are each grounded in a different type of methodology and gather a different type of data, leading to *convergent validity*, a triangulation of evidence. The results of these studies and of the process itself was the Whyline concept. The process of designing each Whyline prototype was grounded in empirical evidence, both from the original studies and from many iterations of usability testing and redesign.

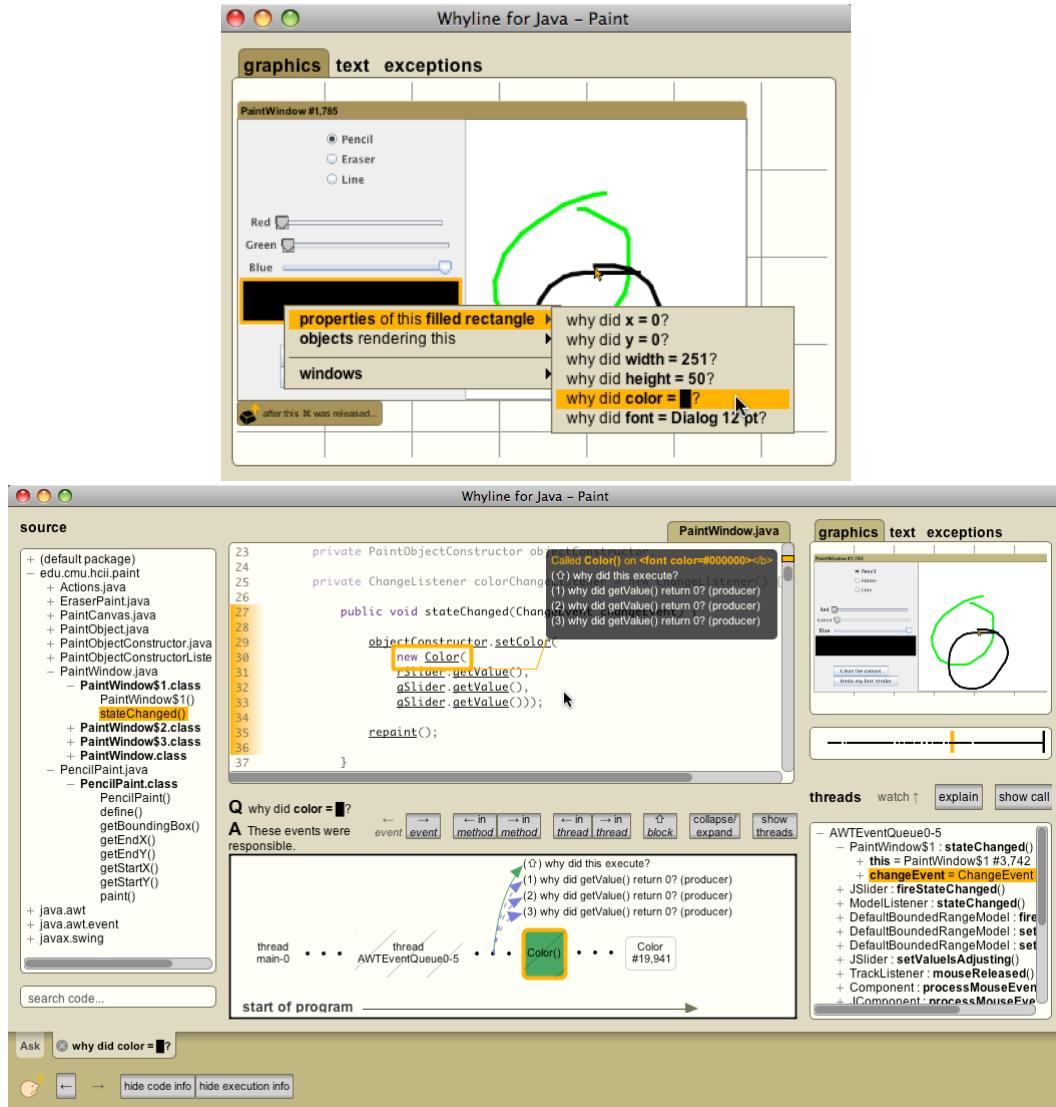


Figure 1.1. On the top, a question about the color of a rectangle using the Whyline for Java. On the bottom, the corresponding answer, showing the source of the black color and the causes of its creation (`gSlider` was used twice instead of `bSlider`).

The final step in the process was to assess the Whyline's impact on developers' work. If program understanding is a hypothesis-driven task that fails because developers must guess, then a tool that avoids speculation about the causes of output should help people be more successful than those using traditional debugging tools. The final chapters of this dissertation test this claim across three studies and show it to be true for a variety of program understanding tasks.

1.4. DEFINITIONS

There are a number of terms and phrases used in this dissertation that deserve some consistent use (despite their inconsistent use in HCI and software engineering literature). First and most importantly are various classifications of people who write code. In this dissertation, *developer* is used to refer to anyone who writes computer code in order to have a computer take some later action. *Novice developers* are those who have little practice at this activity; *skilled developers* have more experience. *Professional developers* get paid to write code. *End-user programmers* write code to support some other work task or hobby (writing code is not their primary goal). This dissertation uses the generic term of *developer* to encompass all of these categories and does not assume any particular experience or context.

There are many types of software development activities mentioned in this document. *Program understanding* is any process a developer undergoes to develop an explanation of how a program executed or will execute. There are many named activities of this kind that are distinguished by their goals. *Debugging* is program understanding with the goal of understanding a particular class of program executions that exhibit an undesirable behavior. *Reverse engineering* is program understanding with the goal of understanding the larger architectural relationships between program elements. *Enhancement tasks* are reverse engineering tasks that have the goal of finding an appropriate way to add new features into an existing program design.

Because this dissertation is largely about debugging, it is also important to clarify the meaning of *bug* and related terms. *Error* and *defect* are used interchangeably to refer to some program code that results in a fault and/or failure. A *fault* is a program state that can cause a failure. A *failure* is some pattern of program output that is inconsistent with a program's intended behavior. *Bug* can refer to any one or a combination of error, fault, or failure. Regarding executions of programs, an *execution event* (or *event* for short), refers to a particular execution of a particular instruction in a program's code. A *cause* of an execution event is any other execution event that was necessary for an event to occur. Causes are typically *control events* (branches and other conditional logic) and *data events* (uses and assignments of variables and other memory).

1.5. CONTRIBUTIONS

Throughout this dissertation, there are a number of major contributions:

- A framework for modeling the cognitive causes of software errors.
- A methodology for reconstructing the causes of software errors from video and verbal data.
- Evidence that developers verbalize “why did” and “why didn’t” questions in response to program failures, but form false hypotheses about the causes of program failures on almost every first attempt.
- Evidence that developers of all levels of expertise often investigate problems that do not exist, because they misinterpret or misperceive program output.
- Evidence that guessing incorrectly about the causes of a program failure leads developers of all expertise to spend more than half their time (on average) investigating irrelevant code.
- Evidence that users’ questions about program failures tend to specify, to varying degrees of clarity, both *what* went wrong and *when* by referring to visible entities, physical devices, or user actions (and rarely code).
- Three Whyline prototypes: for the Alice programming environment, for a word processor, and for graphical and textual Java programs.
- Algorithms for extracting “why” questions from source code and execution history.
- Algorithms for answering “why” questions using both modified existing techniques and new techniques.
- Timeline visualizations representing answers that help developers explore control and data dependencies related to the subject of their questions.
- Evidence for all three prototypes that people are significantly more productive and successful when given access to Whyline tools.

1.6. OUTLINE

As an aid to the reader, Figure 1.2 provides a map for the content in this dissertation and depicts the flow of ideas over time between studies and tools.

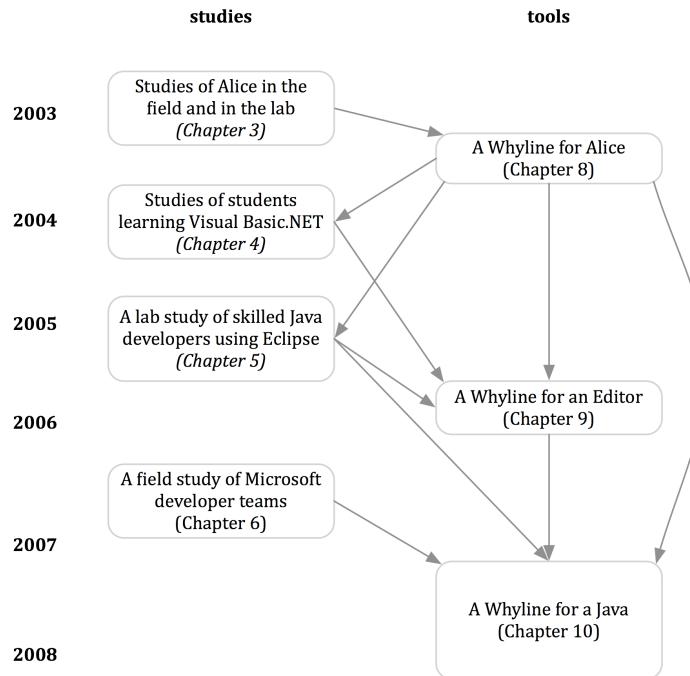


Figure 1.2. A historical map of the contributions in this dissertation and their corresponding chapters. The arrows represent the approximate flow of ideas from studies to tools and tools to studies. The height of the boxes approximate the time spent on each portion of the research.

This document itself has two major parts. The first explores program understanding tools and how they are used today. Chapter 2 discusses the history of studies of program understanding in a variety of disciplines and then illustrates the design space of debugging and program understanding tools. Chapters 3 through 7 discuss several of my own studies, exploring the challenges of program understanding.

The rest of the dissertation discusses the Whyline concept. Chapter 8 is the philosophical core of the dissertation, summarizing both the findings of the studies of program understanding and illustrating the Whyline concept, which embodies these findings. Chapters 9, 10, and 11 explore the Whyline concept in three domains of software development and use, including a simple 3D programming environment, a word processing application, and Java development by experienced programmers. Chapter 12 discusses limitations and future research directions of the Whyline approach and Chapter 13 concludes.

2.

RELATED WORK³

Researchers have approached the problem of program understanding from a variety of perspectives, studying the people who do it and their strategies, but also inventing a range of tools and techniques to support it. This chapter explores both of these perspectives and attempts to describe a conceptual foundation in which they can be understood. To do this, this chapter explores two major areas of research: the *history* of studies of programmers understanding software and the *design space* of program understanding tools (as opposed to a chronology).

To frame this discussion, it is important to note that software is a unique kind of artifact, with particular dynamic characteristics. The execution of a program has inherent causality, determined by a program's code and other particular factors such as input and hardware behavior. This means that program execution is often easily reproducible and can be understood systematically. We can think of the execution of a program resulting in a history of *execution events* (variable assignments, control passing from function to function). We can think of this sequence of events as an *execution history*. Of course, computers execute software at millions of instructions per second and so these execution histories can be quite large.

Within this conception, the difficult thing about program understanding is that of all the events in an execution history, a developer is typically interested in only a few. In this sense, program understanding is like a *search task*. The goal of *debugging* is to find one or more sets of the code fragments in a program that can be modified to

³ Portions of this chapter appear in [Ko 2005c].

prevent a specific program failure. In a *reverse engineering* task, the goal is to find a set of code fragments related to some architectural feature. In order to successfully determine any such set, the programmer must search a space of execution events, code, all of the dependencies between the two, and perhaps even different versions of the program as they change over time. This is no simple search.

2.1. STUDIES OF PROGRAM UNDERSTANDING

Given these definitions, let us discuss various factors that influence this search, including cognitive factors (in a developer’s head), social factors (in the developer’s community or team), and technical factors (in the program’s code and its changes over time). The goal of this section is not to cover all of the studies that have been done on this topic, but instead highlight historical milestones in the findings of such studies, as they relate to debugging and program understanding.

2.1.1. COGNITIVE FACTORS

There is a long history of empirical research on debugging and program understanding, dating back to the 1950’s (called *psychology of programming* and *empirical studies of programming*, among other names). These bodies of work largely focus on forming predictive theories of developer behavior and providing insight into the fundamental difficulties of program understanding. Since the mid-seventies, researchers have categorized the various types of “bugs” that people insert into programs, leading to a variety of insights. For example, Eisenberg studied novice bugs in APL and proposed categories such as “Gestalt bug,” which occurred when a programmer did not foresee the side effects of a command [Eisenberg and Peele 1983]. Subsequent studies focused on novice mistakes in other languages, but as software became more ubiquitous, the focus moved to more skilled developers. For example, Knuth recorded all of the debugging he performed in the development of TeX [Knuth 1989], revealing that the majority of his mistakes were due to oversights, which he labeled “surprise scenarios.” Eisenstadt interviewed industry developers and found that 50% of the debugging difficulties were attributable to two sources: large temporal or spatial chasms between the root cause and the symptom, and bugs that rendered debugging tools useless [Eisenstadt 1997]. Of course, these results are biased by the fact that Eisenstadt only interviewed people about *memorable* bugs.

Researchers also studied program understanding from a theoretical perspective, performing controlled studies to investigate how developers approached the task of understanding or debugging a program. In one of the earliest investigations into the cognitive processes of software development, Brooks found that debugging and other understanding activities were primarily hypothesis-driven [Brooks 1972]: to explain how a program performs a particular function, a developer generates and tests a hypothetical explanation of the program’s behavior using both cognitive and external resources. Studies by Littmann et al. [1986] and Gugerty and Olson [1986] found that skilled programmers tended to form more accurate hypotheses about the causes of program behavior than novices, and that novices often inserted errors into their programs while debugging because of their inaccurate hypotheses. Gilmore [1992] studied existing models of programmers’ debugging strategies, which had primarily described debugging as only a fault localization activity, and proposed that hypothesis formation and testing is central not only to program understanding tasks, but also to implementation and design activities. Vans and von Mayrhauser replicated many of these findings in a study of a larger system [1999].

In addition to studying hypothesis formation in program understanding, a number of studies characterized developers’ strategies for hypothesis *testing*. For example, Koenemann and Robertson [1991] argued that developers follow primarily an “as-needed” strategy for understanding programs, in which developers’ process was unplanned and opportunistic. This contrasts with the findings of Littman et al. [1986], who argued that skilled programmers followed a more systematic strategy than novices, characterized by concrete plans and guided navigations of a program’s dependencies. It has since been shown that both skilled and novice developers use a combination [Baniassad 2002, Robillard 2004], but that systematic strategies are generally more productive than “as-needed” strategies [Boehm 1976, Pennington 1987, Robillard 2004]. Katz and Anderson [1988] identified other less common strategies for hypothesis testing, including hand-simulation of a program’s execution and more rigorous causal reasoning. Few of these studies investigate how developers actually form their hypotheses, nor what factors influence their formation. This is a central issue, given that many of the difficulties that developers had in these studies were due to false hypotheses.

More recently, studies have considered the questions that programmers ask during understanding tasks. Sillito et al. identified 44 questions that center around the concept of a *focus point*, which are places in source code related to the developer’s

goal. They group these questions around finding initial focus points, building on focus points, connecting focus points, and integrating focus points, but the specific questions generally focus on code specific questions such as "Where are instances of this class created?" or "Is there an exemplar for this?" LaToza et al. describe a similar study, framing program understanding tasks as "fact finding" missions, driven by developers' efforts to discover properties of the program at varying levels of abstraction [LaToza 2007]. These newer studies are largely consistent with those in past decades, but have arisen out of a need to characterize program understanding in the context of much larger and more complex application development. They have also strived to be more concrete about the types of questions that developers ask.

2.1.2. SOCIAL FACTORS

Several previous studies have documented the social nature of development work, much of it finding that despite stereotypes, software developers communicate with each other quite often. Perry, Staudenmayer and Votta reported that over half of developers' time was spent interacting with coworkers [Perry 1994]. Much of this communication is to maintain awareness. de Souza, et al. [2003] found that developers send emails before check-ins to allow their peers to prepare for changes. Collocation is a central factor in determining the quality of awareness information. Seaman and Basili found that the ability to meet face to face facilitates awareness in ways that are unavailable in distributed situations, where communication is remote [Seaman 1998]. Similarly, coordination problems can be exaggerated across sites because of the lack of spontaneous communication channels [Gutwin 2004]. Developers also communicate to obtain knowledge [Hertzum 2002]. LaToza, Venolia and DeLine describe the role of the "team historian," who possesses knowledge about the origins of a project and its architecture [LaToza 2006]. To determine who to ask, developers often gauge expertise by inspecting check-in logs [de Souza 2003], but such information is not always accurate [McDonald 1998]. One consequence of developers' frequent communication is the fragmentation of time. Gonzalez, Mark and Harris found that developers average about 3 minutes on a task and about 12 minutes in an area of work before switching [Gonzalez 2005]. These switches occur due to changing task priorities and getting blocked [Perry 1994]. Perlow related how one software group's frequent interruptions created a sense of a "time famine"—having too much to do and not enough time [Perlow 1999].

Dependencies are also a central factor in software development. Developers use bug reports, content management systems, and version control systems to manage dependencies and notify coworkers of new dependencies [de Souza 2003]. Teams will clone software to avoid dependencies, even though they later have to duplicate fixes to the cloned code [LaToza 2006]. Developers also rush their activities to minimize dependencies between their code and recently committed changes in the repository [de Souza 2003]. Unfortunately, the previous research has failed to explore this notion of dependency in the context of program understanding. For example, we know little about what kinds of dependencies are difficult to find or which ones developers commonly look for.

2.1.3. TECHNICAL FACTORS

Another reason that program understanding is difficult is that modern software is inherently complex: the parts of a system that are related to a developer's particular task are often distributed throughout a system's modules, and can interact in unpredictable ways when a program executes [Eick 2001, LaToza 2005]. This is complicated by the fact that most useful software undergoes a brief period of rapid development, followed by a much longer and more costly period of maintenance and adaptation to new contexts of use [Boehm 1976, Lehman 1985]. The fact that software can change rapidly and in unpredictable ways only exacerbates the cognitive and social challenges mentioned earlier.

Another technical concept, proposed by [Murphy 2005] (and independently in the work described in Chapter 5), is that of a *task context*: the parts and relationships of artifacts relevant to a developer during work on a maintenance task. A number of important contributions have been built around this concept, including ways of representing task contexts [Reiss 1996, Robillard 2003b], tools that enable developers to manually build a task context by selecting program elements [Robillard 2002], and methods of automatically inferring the relevant task context based on a developer's investigations in a development environment [Robillard 2003a][Robillard 2005]. Each of these tools is inspired by the notion that every program is flush with technical dependencies, both internal ones (such as "line 36 calls method M"), but also external ones, linked by natural language (for example, "bug 283 is related to component C").

2.2. CAUSES OF SOFTWARE ERRORS

Since much of program understanding is *debugging*, it is also helpful to understand where errors in source code come from. Many of the causes of these errors lie in the same problems of human cognition that cause difficulty in debugging.

2.2.1. CLASSIFICATIONS OF ERRORS

Prior work on classifying common programming difficulties—summarized chronologically in Table 2.1—has been reasonably successful in motivating novel and effective tools for finding, understanding and repairing software errors. For example, in the early ‘80’s, the Lisp Tutor drew heavily from analyses of novices’ software errors [Anderson 1985] and nearly approached the effectiveness of a human tutor. More recently, the testing and debugging features of the Forms/3 visual spreadsheet language [Burnett 2001] were largely motivated by studies of the type and prevalence of spreadsheet errors [Panko 1998].

Table 2.1. Studies classifying “bugs”, “errors” and “problems” in various languages, expertise, and programming contexts.

Study	Bug/Error/ Problem	Description	Comments
Gould 1975	Assignment bug	Software errors in assigning variables' values	Requires understanding of behavior
Novice Fortran	Iteration bug	Software errors in iteration algorithms	Requires understanding of language
	Array bug	Software errors in array index expressions	Requires understanding of language
Eisenberg 1983	Visual bug	Grouping related parts of expression	
Novice APL	Naive bug	Iteration instead of parallel processing	'...need to think step-by-step'
	Logical bug	Omitting or misusing logical connectives	
	Dummy bug	Experience with other languages interfering	'...seem to be syntax oversights'
	Inventive bug	Inventing syntax	
	Illiteracy bug	Difficulties with order of operations	
	Gestalt bug	Unforeseen side effects of commands	'...failure to see the whole picture'
Johnson et al. 1983	Missing	Omitting required program element	Software errors have <i>context</i> : input/output, declaration, initialization and update of variables, conditionals, and scope delimiters.
Novice	Spurious	Unnecessary program element	
Pascal	Misplaced	Required program element in wrong place	
	Malformed	Incorrect program element in right place	

Study	Bug/Error/ Problem	Description	Comments
Spohrer & Soloway 1986 Novice Basic	Data-type inconsistency	Misunderstanding data types	
	Natural language	Applying natural language semantics to code	
	Human-interpreter	Assuming computer interprets code similarly	
	Negation & whole-part	Difficulties constructing Boolean expressions	
	Duplicate tail-digit	Incorrectly typing constant values	
	Knowledge interference	Domain knowledge interfering w/ constants	
	Coincidental ordering	Malformed statements produce correct output	
	Boundary	Unanticipated problems with extreme values	'All bugs are not created equal. Some occur over and over again in many novice programs, while others are more rare...Most bugs result because novices misunderstand the semantics of some particular programming language construct.'
	Plan dependency	Unexpected dependencies in program	
	Expectation/ interpretation	Misunderstanding problem specification	
Knuth 1989 While writing TeX Data structure in SAIL and debacle Pascal	Algorithm awry	Improperly implemented algorithms	'proved...incorrect or inadequate'
	Blunder or botch	Accidentally writing code not to specifications	'not... enough brainpower'
		Software errors in using data structures	"did not preserve...invariants"
	Forgotten function	Missing implementation	'I did not remember everything'
	Language liability	Misunderstanding language/environment	
	Module mismatch	Imperfectly knowing specification	'I forgot the conventions I had built'
	Robustness	Not handling erroneous input	'tried to make the code bullet-proof"
	Surprise scenario	Unforeseen interactions in program elements	'forced me to change my ideas'
	Trivial typos	Incorrect syntax, reference, etc.	'my original pencil draft was correct'
Eisenstadt 1993 Industry experts COBOL, Pascal, Fortran, C	Clobbered memory	Overwriting memory, subscript out of bounds	
	Vendor problems	Buggy compilers, faulty hardware	
	Design logic	Unanticipated case, wrong algorithm	
	Initialization	Erroneous type or initialization of variables	
	Variable	Wrong variable or operator used	
	Lexical bugs	Bad parse or ambiguous syntax	
	Language	Misunderstandings of language semantics	Also identified why software errors were difficult to find: cause/effect chasm; tools inapplicable; failure did not actually happen; faulty knowledge of specs; "spaghetti" code.
Panko 1998 Novice Excel	Omission error	Facts to be put into code, but are omitted	
	Logic error	Incorrect or incorrectly implemented algorithm	Quantitative errors: "errors that lead to an incorrect, bottom line value"
	Mechanical error	Typing wrong number; pointing to wrong cell	
	Overload error	Working memory unable to finish without error	
	Strong but wrong error	Functional fixedness (a fixed mindset)	Qualitative errors: "design errors and other problems that lead to quantitative errors in the future"
	Translation error	Misreading of specification	

Despite the successful use of these classifications, in hindsight it is clear that the classifications do not actually classify software errors, but rather, the complex relationships between software errors, runtime faults, runtime failures, and cognitive failures. Nevertheless, in analyzing these classifications, four salient aspects of software errors emerge.

The first is a software error's surface qualities: the particular syntactic or notational anomalies that make a code fragment incorrect. Eisenberg's dummy bug is a class of syntax oversights; Knuth's trivial typos and Panko's mechanical errors simply describe unintended text in a program; Gould identifies particular surface qualities of erroneous assignment statements and array references in his study of Fortran. Clearly, the surface qualities of software errors are greatly influenced by the language syntax. While it may seem that these qualities have little to do with the actual cause of software errors, the fact that they are common enough to warrant their own category suggests that syntax can be a cause of software errors on its own.

Other categories allude to several cognitive causes of software errors. For example, Eisenberg's inventive bug, Spohrer and Soloway's data-type inconsistency, and Johnson's misplaced and malformed categories all refer to programmers' lack of knowledge about language syntax, control constructs, data types, and other programming concepts. Knuth's forgotten function category and Eisenstadt's variable bugs suggest attentional issues such as forgetting or a lack of vigilance. Eisenstadt's design logic bugs and Knuth's surprise scenario category indicate strategic issues, referring to problems like unforeseen code interactions or poorly designed algorithms.

A third aspect of software errors is the programming activity in which the cause of the software error occurred. For example, Knuth's module mismatch bugs and Spohrer and Soloway's expectation and interpretation problems all occur in specification activities, in which the programmer's invalid or inadequate comprehension of design specifications later led to software errors. Spohrer and Soloway's plan dependency problem occurs during algorithm design activities, in which unforeseen interactions eventually led to software errors.

A fourth and final aspect of software errors is the type of *action* that led to the error. The classifications suggest six types of programming actions, which are listed in Table 2.2 with examples. For instance, programmers can introduce software errors when *creating* code, but the creation of specifications can also predispose software errors (as in Spohrer and Soloway's expectation/interpretation problems). Programmers also *reuse* code, *modify* specifications and code, *design* software architectures and algorithms and *explore* code and runtime data. The classifications also blame the *understanding* of specifications, data structures, and language constructs for several types of software errors.

Table 2.2. Actions performed during programming activity.

Action	Examples of the action in programming activity
Creating	Writing code, or creating design and requirement specifications
Reusing	Reusing example code, copying and adapting existing code
Modifying	Modifying code or changing specifications
Designing	Considering various software architectures, data types, algorithms, etc.
Exploring	Searching for code, documentation, runtime data
Understanding	Comprehending a specification, an algorithm, a comment, runtime behavior, etc.

While these classifications go a long way in conveying the scope and complexity of several aspects of software errors, they only go so far in relating these aspects causally. For example, what looks like an erroneously coded algorithm on the surface may have been caused by an invalid understanding of the specifications, a lack of experience with a language construct, misleading information from a debugging session, or simply momentary inattention. Each possible cause motivates different interventions.

2.2.2. HUMAN ERROR IN PROGRAMMING ACTIVITY

To fully understand how the interaction between a programmer and a programming system can lead to software errors, a more general discussion of the underlying cognitive mechanisms of human error is necessary. James Reason's Human Error [Reason 1990], grounded in studies of engineering and organizations as well as human cognition, provides a solid foundation for this discussion. This section adapts two aspects of his research to the domain of programming: (1) a systemic view of the causes of failure, and (2) a brief catalog of common failures in human cognition.

Reason distinguishes between *active* errors, whose effects are felt almost immediately, such as syntax errors that prevent successful compilation or invalid algorithms, and *latent* errors, "whose adverse consequences may lie dormant within the system for a long time, only becoming evident when they combine with other factors to breach the system's defenses" [Reason 1990]. The fundamental idea is that complex systems have several functional layers, each with potential latent errors that predispose failure, but also with a set of defenses that prevent latent errors from becoming active. From this perspective, failures are ultimately due to a causal chain of failures both within and between layers of a system.

These ideas are applied to software engineering in Figure 2.1. The figure portrays four layers, each with its own type of latent errors and defenses. On the left, specifications act as high-level defenses against software errors, but if they are ambiguous, incomplete, or incorrect, they may predispose programmers to misunderstandings about a software system's true requirements. By improving software engineering practices, there will be fewer latent errors in design specifications, which will prevent programmers' invalid or incomplete understanding of specifications. Programmers, the next layer in Figure 2.1, have knowledge, attention, and expertise to defend against software errors. However, programmers are also prone to cognitive breakdowns in these defenses, which predispose software errors. The next section discusses these breakdowns in detail. The third layer in Figure 2.1, the programming system, consists of several components (compilers, libraries, languages, environments, etc.). Each has a set of defenses against software errors, but also a set of latent usability issues that predispose the programmer to cognitive breakdowns, and thus software errors. For example, compilers defend against syntax errors, but in displaying confusing error messages, may misguide programmers in correctly repairing the syntax errors. The last layer, the program, has the latent errors known as software errors, which can eventually lead to a program's runtime failure.

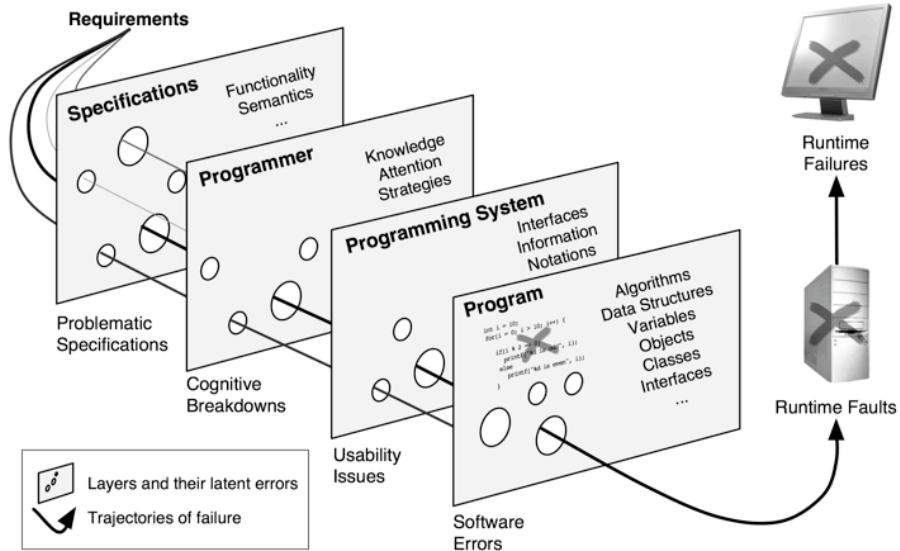


Figure 2.1. Dynamics of software error production, based on Reason's systemic view of failure. Each layer has latent errors (the holes), predisposing certain types of failures. Layers also have defenses against failures (where there are no holes). Several layers of failure must go unchecked before software errors are introduced into code.

It is important to note that latent errors in these layers only become active in particular circumstances. Just as a program may only fail with particular input and in particular states, programming systems, programmers, and specifications may only fail in particular situations.

Within the broad view of software errors portrayed in Figure 2.1, this section focuses on the programmer's latent errors—what I will call *cognitive breakdowns*—and how the programming system might be involved in predisposing these cognitive breakdowns. Reason's central thesis about human behavior is that in any given context, individuals will behave in the same way they have in the past in that context. Under most circumstances, these “default” behaviors are sufficient; however, under exceptional or novel circumstances, they may lead to error. In programming, this means that when solving problems, programmers tend to prefer programming strategies that have been successful in the past. These default strategies are usually successful, but they sometimes break down—hence the term cognitive breakdowns—and lead to software errors.

In order to clarify the sources of these breakdowns, Reason discusses three general types of cognitive activity, each prone to certain types of cognitive breakdowns. The most proceduralized of the three, *skill-based* activity, usually fails because of a lack of attention given to performing routine, skillful patterns of actions. *Rule-based* activity, which is driven by learned expertise, usually fails because the wrong rule is chosen, or the rule is inherently bad. *Knowledge-based* activity, centered on conscious, deliberate problem solving, suffers from cognitive limitations and biases inherent in human cognition. This section discusses all three types of cognitive activity and their accompanying breakdowns in detail.

Skill-based activities are routine and proceduralized, where the focus of attention is on something other than the task at hand. Some skill-based activities in programming include typing a text string, opening a source file with a file open dialog, or compiling a program by pressing a button in the programming environment. These are practiced and routine tasks that can be left in “auto-pilot” while a programmer attends to more problem-oriented matters. An important characteristic of skill-based activities is that because attention is focused internally on problem solving and not externally on performing the routine action, programmers may not notice important changes in the external environment.

Table 2.3 lists Reason's two categories of skill breakdowns. *Inattention* breakdowns are a failure to pay attention to performing routine actions at critical times. For example, imagine a programmer finishing the end of a `for` loop header when the fire alarm goes off in his office. When he returns to the loop after the interruption, he fails to complete the increment statement, introducing a software error. Inattention breakdowns may also occur because of the intrusion of strong habits. For example, consider a programmer who tends to save modifications to a source file after every change so that important modifications are not lost. At one point, he deletes a large block of code he thinks is unnecessary, but immediately after, realizes he needed the code after all. Unfortunately, his *strong habit* of saving every change has already intruded, and he loses the code permanently (a good motivation for sophisticated undo mechanisms in programming environments).

Table 2.3. Types of skill breakdowns, adapted from [Reason 1990]. The → means “causes.”

Inattention	Type	Events resulting in breakdown
Failure to attend to a routine action at a critical time causes forgotten actions, forgotten goals, or inappropriate actions.	Strong habit intrusion	In the middle of a sequence of actions → no attentional check → contextually frequent action is taken instead of intended action
	Interruptions	External event → no attentional check → action skipped or goal forgotten
	Delayed action	Intention to depart from routine activity → no attentional check between intention and action → forgotten goal
	Exceptional stimuli	Unusual or unexpected stimuli → stimuli overlooked → appropriate action not taken
	Interleaving	Concurrent, similar action sequences → no attentional check → actions interleaved
Overattention	Type	Events resulting in breakdown
Attending to routine action causes false assumption about progress of action.	Omission	Attentional check in the middle of routine actions → assumption that actions are already completed → action skipped
	Repetition	Attentional check in the middle of routine actions → assumption that actions are not completed → action repeated

Overattention breakdowns occur when attending to routine actions that would have been better left to “auto-pilot.” For example, imagine a programmer has copied and pasted a block of code and is quickly coercing each reference to a contextually appropriate variable. While planning his next goal in his head, he notices that he has

not been paying attention and interrupts his “auto-pilot,” accidentally looking two lines down from where he actually was. He falsely assumes that the statements above were already coerced, leaving several invalid references in his code.

Rule-based activities involve the use of cognitive rules for acting in certain contexts. These rules consist of some condition, which checks for some pattern of signs in the current context. If the current context matches the condition, then corresponding actions are performed. For example, skilled C programmers frequently employ the rule, “If some operation needs to be performed on the elements of a list, type `for(int i = some_initial_value; i < some_terminating_value; i++)`, choose the initial and terminating values, then perform the operation.” These rules are much like the concept of programming plans [Spoher 1986], which are thought to underlie the development of programming expertise [Davies 1994].

Table 2.4 lists Reason’s two categories of rule breakdowns, the first of which is *wrong rule*. Because rules are influenced by prior experience, they make implicit predictions about the future state of the world. These predictions of when and how the world will change are sometimes incorrect, and thus a rule that is perfectly reasonable in one context may be selected in an inappropriate context. For example, one common breakdown in Visual Basic.NET is that programmers will use the “+” operator to add numeric values, not realizing that the values are represented as strings, and so the strings are concatenated instead. Under normal circumstances, use of the “+” operator to add numbers is a perfectly reasonable rule; however, because there were no distinguishing signs of the variables’ types in the code, it was applied inappropriately.

Empirical studies of programming have reliably demonstrated many other types of wrong rule breakdowns. For example, Davies’ framework of knowledge restructuring in the development of programming expertise suggests that a lack of training in structured programming can lead to the formation of rules appropriate for one level of program complexity, but inappropriate for higher levels of complexity [Davies 1994]. For example, in Visual Basic, the rule “if some data needs to be used by multiple event-handlers, create a global variable on the form” is appropriate for forms with a small number of event-handlers, but quickly becomes unmanageable in programs with hundreds. Similarly, Shackelford studied the use of three types of Pascal while loops, finding that while most students had appropriate

rules for choosing the type of loop for a problem, the same rules failed when applied to similar problems with additional complexities [Shackelford 1993].

Table 2.4. Types of rule breakdowns, adapted from [Reason 1990].

Wrong Rule	Type	Events resulting in breakdown
Use of a rule that is successful in most contexts, but not all.	Ambiguous or hidden signs → conditions evaluated with problematic signs insufficient info → wrong rule chosen → inappropriate action	
	Information overload	Too many signs → important signs missed → wrong rule chosen → inappropriate action
	Favored rules	Previously successful rules are favored → wrong rule chosen → inappropriate action
	Favored signs	Previously useful signs are favored → exceptional signs not given enough weight → wrong rule chosen → inappropriate action
	Rigidity	Familiar, situationally <i>inappropriate</i> rules preferred over unfamiliar, situationally <i>appropriate</i> rules → wrong rule chosen → inappropriate action
Bad Rule	Type	Events resulting in breakdown
Use of a rule with problematic conditions or actions.	Incomplete encoding	Some properties of problem space are not encoded → rule conditions are <i>immature</i> → inappropriate action
	Inaccurate encoding	Properties of problem space encoded inaccurately → rule conditions are <i>inaccurate</i> → inappropriate action
	Exception proves rule	Inexperience → exceptional rule often inappropriate → inappropriate action
	Wrong action	Condition is right but action is wrong → inappropriate action

The second type of rule breakdown is the use of a *bad rule*: one with problematic conditions or actions. These rules come from learning difficulties, inexperience, or a lack of understanding about a particular program's semantics. For example, Perkins and Martin demonstrated that "fragile knowledge"—inadequate knowledge of programming concepts, algorithms, and data structures, or an inability to apply the appropriate knowledge or strategies—was to blame for most novice software errors when learning Pascal [Perkins 1986]. Not knowing the language syntax—in other words, not encoding or inaccurately encoding its properties—can lead to simple syntax errors, malformed Boolean logic, scoping problems, the omission of required constructs, and so on. An inadequate understanding of a sorting algorithm may cause a programmer to unintentionally sort a list in the wrong order. Von Mayrhofer and Vans illustrated that programmers who focused only on comprehending surface level features of a program (variable and method names, for example), and thus had an insufficient model of the program's runtime behavior, did

far worse in a corrective maintenance task than those who focused on the program's runtime behavior [von Mayerhauser 1997].

In knowledge-based activities, Reason's last type of cognitive activity, the focus of attention is on forming plans and making high-level decisions based on one's knowledge of the problem space. In programming, knowledge-based activities include forming a hypothesis about what caused a runtime failure, or comprehending the runtime behavior of an algorithm. Because knowledge-based activities rely heavily on the interpretation and evaluation of models of the world (in programming, models of a program's semantics), they are considerably taxing on the limited resources of working memory. This results in the use of a number of cognitive "shortcuts" or biases, which can lead to cognitive breakdowns.

Table 2.5 describes these biases, and how they cause breakdowns in the strategies and plans that people form. One important bias is *bounded rationality* [Simon 1956]: the idea that the problem spaces of complex problems are often too large to permit an exhaustive exploration, and thus problem solvers "satisfice" or explore "enough" of the problem space. Human cognition uses a number of heuristics to choose which information to consider [Reason 1990]: (1) evaluate information that is easy to evaluate (*selectivity*); (2) only evaluate as much as is necessary to form a plan of action (*biased reviewing*); (3) evaluate information that is easily accessible in the world or in the head (*availability*).

Because of the complexity of programming activity, bounded rationality shows up in many programming tasks. For example, Vessey argues that debugging is difficult because the range of possible software errors causing a runtime failure is highly unconstrained and further complicated by the fact that multiple independent or interacting software errors may be to blame [Vessey 1989]. Gilmore points out that, because of their limited cognitive resources, programmers generally only consider a few hypotheses of what software errors caused the failure, and usually choose an incorrect hypothesis. This not only leads to difficulty in debugging, but often the introduction of further software errors due to incorrect hypotheses [Gilmore 1992]. For example, in response to a program displaying an unsorted list because the sort procedure was not called, a programmer might instead decide the software error was an incorrect swap algorithm, and attempt to modify the already correct swap code.

The second type of knowledge breakdown is the use of a faulty model of the problem space. For example, human cognition tends to see *illusory correlations* between events; it tends to ask questions that confirm beliefs rather than refute them (*confirmation bias*). These biases lead to oversimplified or incorrect models of the problem space. Individuals also display overconfidence, giving undue faith to the correctness and completeness of their knowledge. This results in strategies that are based on incomplete analyses. For example, spreadsheet users exhibit so much overconfidence in their spreadsheets' formulas that a single test case is often enough to convince them of their spreadsheet's correctness [Wilcox 1997]. Corritore and Wiedenbeck have shown that programmers' overconfidence in the correctness of their mental models of a program's semantics was often the cause of software errors in programmers' modifications [Corritore 1999].

Table 2.5. Types of knowledge breakdowns, adapted from Reason [1990].

Bounded Rationality	Type	Events resulting in breakdown
Problem space is too large to explore because working memory is limited and costly.	Selectivity	Psychologically salient, rather than <i>logically important</i> task information is attended to → biased knowledge
	Biased reviewing	Tendency to believe that <i>all</i> possible courses of action have been considered, when in fact very few have been considered → suboptimal strategy
	Availability	Undue weight is given to facts that come readily to mind → facts that are <i>not</i> present are easily ignored → biased knowledge
Faulty Models of Problem Space	Type	Events resulting in breakdown
Formation and evaluation of knowledge leads to incomplete or inaccurate models of problem space.	Simplified causality	Judged by perceived similarity between cause and effect → knowledge of outcome increases perceived likelihood → invalid knowledge of causation
	Illusory correlation	Tendency to assume events are correlated and develop rationalizations to support the belief → invalid model of causality
	Overconfidence	False belief in correctness and completeness of knowledge, especially after completion of elaborate, difficult tasks → invalid, inadequate knowledge
	Confirmation bias	Preliminary hypotheses based on impoverished data interfere with later interpretation of more abundant data → invalid, inadequate hypotheses

2.3. PROGRAM UNDERSTANDING TOOLS

Although the previous sections reveal a great deal of knowledge about the nature of errors and debugging, little of the research informed the design of debugging and program understanding tools. Nevertheless, as long as people have had to understand programs, researchers have devised new technologies to help. Some of these technologies are interactive, playing a supporting role in organizing information about program execution. Others are automated, trying to detect properties of programs to report back to a developer. Others still define query languages for developers to learn and use in order to search for particular facts about a program's design and behavior.

Rather than organize the work in this section by *chronological* appearance in history or by tool categories (e.g., slicing tools, tracing tools), this section discusses tools in terms of their *strategic* goal: what approach does each type of tool take to supporting developers' exploration of a program's execution? The particular work highlighted were chosen to represent a particular type of support and not necessarily research milestones.

2.3.1. MODIFYING EXECUTION

The simplest strategy for searching execution history is to *change* the program's execution until it does what is intended (a naive guess and check strategy). Although this is not likely to be effective at a large scale, it is rational: it ensures a tight feedback loop between a change to the code and a corresponding change in the program output; such feedback loops are a reasonable way of understanding the behavior of a program. Of course, there are obvious problems with this strategy (having to undo each change, forgetting to undo a change, etc.), but the idea itself works quite well if the changes to the program's execution are limited to program *input*. Zeller and Hildebrandt [2002a] describe an approach which intelligently subdivides failure and success inducing inputs until finding a test case for a which a single granular change can determine whether the program fails. A related strategy is to simply "start over," eliminating code that one believes is contributing to a failure and slowly rewriting. In general, these approaches search the space of *possible program executions*, rather than the space of a single execution.

It is interesting to note that this approach to debugging was not initially possible since computer systems were not always interactive. The same holds true today when testing and debugging programs that take considerable time to start and test. Even with today's powerful computers, this approach is only feasible for small test cases and small programs. If the *test cases* are too large, it may take too long to simplify all of the information in the test case by hand. if the *program* is too large, the likelihood that the developer is modifying the right code is low.

2.3.2. WATCHING EXECUTION

Another simple approach to understanding program execution is to simply watch it execute, step by step. One of the first tools to enable watching was the *breakpoint debugger*. Breakpoint debuggers have been available since at least the 1940's [Stockham 1960], but researchers continued to improve their utility and generality [Kessler 1990]. Breakpoint debuggers allow developers to specify the lines of a program on which to pause a program's execution. If a statement with a breakpoint is executed, the program pauses and the developer can inspect variables' values and the execution stack and can step through the program's execution. While this can be helpful in many cases, these tools have several problems. They provide developers access to a vast amount of runtime information, but a very slow means of searching, exploring, and navigating the information. Breakpoint debuggers cannot help a developer determine why a line of code did *not* execute. Furthermore, if a developer steps over a crucial point, breakpoint debuggers do not allow developers to undo the operation and go back. Some researchers have addressed this problem by simulating reverse execution by recording an execution history [Lewis 2003], while others have devised methods of "undoing" a running program's execution [Akgul 2004, Ungar 1997]. One way of using these tools is to find the point in time that the program behaved improperly or produced incorrect output and step backwards from there.

Many of the constraints imposed by breakpoint debuggers are absent in interpreted, interactive programming languages and environments, such as Lisp [McCarthy 1978]. Lisp interpreters allow developers to not only pause execution and step through function evaluations, but also inspect nearly anything in memory and even evaluate full Lisp expressions on the data in memory, potentially altering the program's execution. Skilled Lisp users often speak with fondness of the ability to keep a Lisp interpreter and Lisp program running for many days, reusing all of the

program state accumulated over time in order to test and refine smaller parts of a program without ever having to restart. All of these features are simply ways to watch and inspect program execution.

An interesting form of stepping now out of fashion was called “algorithmic debugging” [Fritzson 1992]. These tools step through the execution of some part of the program and ask the developer to verify the values of variables. There were many variants on this approach that attempted to minimize how much information a developer had to validate. The central limitation of this approach, besides the sheer number of decisions required by the user, is that people are often poor at knowing whether an intermediate value in a program is correct [Phalgune 2005]. It also requires developers to verify many parts of a program’s execution that may not be relevant to the developers’ task, or may already be known to be correct.

Another way to watch program execution is to *visualize* it, utilizing the power of the human visual system to detect patterns. One approach is to show program executing over time, but abstract away irrelevant details and visualize particular information. For example, Incense [Myers 1980][Myers 1983] was one of the first systems to allow custom programmer-specified displays of data structures, to help developers detect problems in the data in memory. Mukherjea and Stasko [1994] describe a variety of algorithm animations and animation authoring tools, which allow developers to see operations on data structures as a program executes. A related approach [Gestwicki 2002] visualized the internal state of specific data structures, explicitly showing the transfer of data between different data structures in the code. Baecker [1997] described a number of algorithm animation techniques for comparing the behavior and performance of various sorting algorithms. While there is evidence that such visualizations can make the complexity of programs and algorithms less intimidating, there is a general consensus that these visualizations are only helpful when directly associated with the source code corresponding to the animated behavior [Kehoe 2001]. Such animations must also be hand-coded for each situation. Rather than showing program execution over time, some tools visualize execution events on a timeline, using space instead of time. Jackson [2001] describes a visualization of high level Java synchronization events to support the identification of indeterminacy and deadlocks. Briggs [1996] describes a timeline visualization that shows different tasks and their dependencies.

One key limitation of all approaches to watching program execution is that they rely on a developer to specify a place or time in a program’s execution to inspect. If setting a breakpoint, a developer must guess what line is relevant. If viewing a visualization, the developer must guess which patterns are related to their problem. If they choose the wrong place or time, the tools will still help them gather information, but only about irrelevant code. Another limitation is that these tools are inherently *control-focused*. They allow a developer to see *where* a program is executing and even what data is used to do so, but they do not reveal information about where *data* comes from or where data goes. If a developer is trying to find where a value was computed, they will either have to manually trace backwards through the code to find its origin, or, guess the origin and step ahead to see if they are correct. There is also the problem of watching things that did *not* happen. If the code with a breakpoint is never executed, the program will never pause.

2.3.3. SAMPLING EXECUTION

One way to explore a program execution from a data-centric perspective is to sample certain events during its execution and analyze the data gathered. The earliest form of sampling support include the dump and the trace, both developed on the EDSAC in the 1940’s [Satterthwaite 1975] (these were some of the earliest program understanding tools in general). A dump contains all of the values in part of a program’s memory space, and is typically used to help a developer find problematic data in memory. One major problem with a dump is that it is only a snapshot, whereas the problem may have occurred earlier or later in the program’s execution. Dumps also contain a lot of information irrelevant to a problem. Likely the most popular form of sampling is the *print statement*, which essentially allows ad hoc instrumentation of code in order to record a customized trace of a program’s execution. This overcomes the some disadvantages of breakpoint debuggers, allowing programmers to instrument any part of the code they think may be responsible and ignore parts that they think are not. Aside from the obvious disadvantages of inserting instrumentation code throughout a code base, print statements also require a programmer to make a commitment about what code might be responsible for a failure.

As computers became more powerful, researchers developed methods of recording execution histories [Boothe 2000][Lewis 2003][Pothier 2007]. Some used this

recordings to support re-execution, others to support reverse execution, and others still to support exploration of the data in a random access fashion. The promise of these tools is to allow developers to get access to any information they need about a program's execution more quickly, for example, by searching the execution history or by only recording particular parts of a program's execution.

The limitation of any sampling approach is how it presents the sampled information to the developer, since each presentation will reveal different characteristics of the sampled data. Dumps will reveal a single state at particular time; print statements will result in a console full of potentially relevant events; recordings result in a wealth of information that the developer must search by some sort of query. In all of these situations, the developer still has to guess the “what” and “when,” which ultimately decides what information they analyze. If they guess incorrectly, they will simply find *irrelevant* information more quickly.

2.3.4. ANALYZING CAUSES OF EXECUTION

Rather than simply watching parts of a program execute, another approach is to follow causal relationships in code and execution. After all, program execution is largely deterministic and governed by fairly simple control and data dependencies.

There are a number of tools that support the manual exploration of program dependencies. Modern IDEs support showing the declaration of a method from a method call, the callers of a method, the uses of a variable. These types of navigations allow the user to track where a program can execute and what data it can use. Robillard and Murphy describe a tool that helps developers navigate static dependencies in code and combine them into a concern graph [Robillard 2002]. Robillard and Murphy also describe techniques for inferring potentially relevant code based on a developer's navigation through a program's source code [Robillard 2003a] and Robillard describes a technique for inferring other relevant code based on a developer's current location in the source code [Robillard 2005]. All of these approaches essentially help a developer navigate *static dependencies* in a program (relationships known at compile-time). These are often less helpful for debugging, however, because of their lack of precision: they help a developer understand a program's behavior for all possible executions, not for a particular one.

Automatic tools for analyzing dependencies began much earlier in history. Weiser [1982] found that developers were navigating these dependencies and decided to define an automated approach to extracting these dependencies called a *program slice*. As originally conceived, Weiser defined a slice to be “expressed as the values of some sets of variables at some set of program statements.” A static program slice, given a particular variable in a program, computes all of the program statements that could affect the value of the variable in all possible executions of the program. A *dynamic program slice* does the same, but for a particular program execution at a particular time. The result of these analyses is typically a set of program statements, highlighted in a source code viewer. This helps developers focus only on the parts of the program that affect the variable of interest. Recent advances have made slicing both time and space efficient [Wang 2004]. There are also many variants [Baowen 2005], including forward slicing, which analyzes the data that a variable affects, and hierarchical slicing [Wang 2007], which divides a slice into syntactic structural phases. Another recent approach called *thin slicing* [Sridharan 2007], only includes “producer statements” in a slice (those statements which transitively produce values for the variable queried), excluding control dependencies, Java heap accesses and pointer dereferencing. Unfortunately, even dynamic slicing, which was designed to produce a smaller, more specific subset of a program’s statements for investigation, can select up to a third of a program’s statements for inspection [Zhang and Gupta 2004]. Furthermore, slicing tools are only helpful if the developer is asking about a relevant variable; to select such a variable, developers must again guess what variable is relevant, and then navigate to it in the source code. Despite these limitations, there is evidence that slicing tools can help developers debug small programs more efficiently than conventional tools [Francel 2001].

A number of approaches outside of software engineering share characteristics with slicing. For example, there are a number of approaches within the areas of knowledge systems and the semantic web that, given some system for drawing conclusions from a knowledge repository, can also generate some explanation about how these conclusions were drawn. For example, McGuinness and da Salva [2004] describe a system for generating explanations of what information sources were used in a semantic web querying system using a variety of proof systems.

2.3.5. ASSERTING EXECUTION

The debugging tools described in previous sections are largely interactive. A more automated approach is to have a developer form declarative assertions about what they *expect* of a program’s execution and let a tool evaluate these assertions as the program executes. At the most primitive level, these are assertion statements inserted into the code halting the program whenever the assertion is violated [Rosenblum 1995]. These assertions are typically written in the target language and allow side-effects, enabling a developer to state, for example, that the result of some computation is within some range or that when reaching a particular function, the program is in a particular mode. These assertions are used primarily for *detecting* failures, but can also be used to check the intermediate states of a program that is already misbehaving. Assertions can also be used to guide a developer to a bug by contrasting user-written assertions with computer generated assertions. Wallace et al. [2002] describe a spreadsheet that extracts computer-generated assertions from data passing through cells. When users write assertions of their own (typically in the form of legal ranges for a cell), the system can highlight conflicts between user and computer assertions. The user can then find the problem by analyzing the conflict.

Some assertions are global in nature and cannot be written within the context of a single function. Recent approaches support the assertion of certain relationships between objects and data structures in memory. For example, Lencevicius [2003] and Potanin [2004] support the expression of constraints on object relationships (“no two nodes may point to the same expression”). The system evaluates these constraints as the program executions and when one violated, the tool notifies the developer so they may take action. *Model checking* is a more general approach to this problem. Tools such as ESC/Java [Detlefs 1998], Fluid [Greenhouse 2005], and PREFix [Bush 2000] aim to detect errors statically by verifying particular properties of programs that are indicative of errors. One advantage of these declarative approaches is that they do not generally require human intervention, except to utilize the results of the analysis. Many analyses, however, require a program to be annotated in particular ways to facilitate analyses; for example, ESC/Java [Bush 2000] requires developers to supply specifications of the program’s intended behavior as code. Such annotations lead to a “garbage in, garbage out” problem, placing the efficacy of the analyses largely in the hands of developers. Furthermore, these tools can generally only confirm a developer’s hypothesis about the cause of a program’s behavior and cannot disconfirm a false hypothesis.

2.3.6. COMPARING EXECUTIONS

Searching a single execution history can be somewhat limited, since an error may only manifest itself under specific conditions. A number of tools have explored the possibility of comparing multiple execution histories in order to detect errors through aggregation or contrast. For example, Sosic and Abramson [1997] discuss a “relative” debugger that determines the difference between two different versions of a program in order to help developers find problems as a program evolves. The approach requires the developer to write a test case, which specifies the expected correspondence between the two programs’ executions. Then, when the test case fails, the tool can compare the executions between the successful and failing executions, isolating the cause of the problem to some change in the program.

A related approach [Cleve 2005], called *delta debugging*, requires a developer to supply a program, input on which the program succeeds, input on which the program fails, and a function that determines whether the program has succeeded. It returns, if possible, a description of the execution events that occurred in the failing execution that did not occur in the successful execution, by comparing the executions in a experimental manner. This is a very powerful approach, often producing explanations of extreme brevity. Although this technique can be very precise about the situations that caused a program to fail, it cannot be used if there is no known input that causes the program to succeed, if the program’s input is difficult to supply (for example, real-time or user input), or if the “success” is difficult to define. Furthermore, even when it can provide an explanation of a program’s failure, developers must still understand the parts of the program that led to the failure in order to implement a solution. There are also many tasks that do not involve a program failure, but still require developers to understand the causes of a program’s behavior in order to modify or enhance the behavior. Perhaps the greatest limitation is that if the two executions are different enough, the code referred to in the explanation may be far from the actual defect.

Another way of using multiple executions is to detect anomalies and invariants in a program’s execution using statistical methods. Miller and Myers used outlier detection to identify potential mismatches in regular expressions [Miller 2001]. Ernst [2000] discusses an approach of finding program invariants by looking for patterns in variable values. Groce and Visser [2003] describe a related approach which compares differences in invariants between multiple unique executions of a

failure. Xie and Notkin [2004] used a similar approach to compare “value spectra” changes across executions of different program versions. Liblit et al. [2003] applied the same ideas, but across programs executing across many machines to extract samples of program executions in a distributed fashion. Brun and Ernst [2004] describe an approach to training machine learning models on characteristics of known errors and using the models to classify and rank properties of code that may lead to errors. This approach also relies on the detection of invariants.

2.3.7. INTERROGATING EXECUTIONS

Some tools in specific domains have directly supported “why” questions like those in the Whyline. One example appeared in the ACT-R cognitive modeling framework (described later in [Bothell 2004]), in which users would write “production rule” systems to model human cognition and performance. In some situations a user might expect one of these rules to “fire” but it would not. ACT-R development environments support a “why not” analysis, which attempts to show the user why the production rule did not fire by tracing back into the reasoning of the model. The Soar cognitive modeling framework has similar support as ACT-R (called “smatches” for “Soar matches”) [Laird 1990].

Knowledge base systems in AI have a similar problem in that queries into a knowledge base often return “unknown” results. The knowledge base may have missing or incorrect information; there may be a problem with the inference engine. Chalupsky and Russ [2002] describe a system that addresses this debugging challenge, generating a set of possible explanations that identify knowledge that may have been missing or inferences that may have been skipped by the inference engine. The Amulet user interface toolkit [Myers 1997] had a debugger with a “why not” command for an object, which would explain why an object was not visible onscreen. Finally, Lieberman [2003] describes an approach to e-commerce debugging, which presents a timeline of relevant web-service events to explain why a package has not shipped, for example. All of these examples of “why” questions, while similar in phrasing to the Whyline, differ in the types of analysis that they used to answer “why” questions. Most of the above used simple heuristics to reason backwards through the behavior of the system.

The Whyline concept described in this dissertation has already inspired question support in other domains. Clark [2007] describes a system for debugging problems

with one-way constraints used in user interface design. Abraham and Erwig's *goal-directed debugging* [2005] take this even further, allowing the user choose a wrong value in a spreadsheet and specify the correct value. The tool then offers several change suggestions that would cause the program to compute the desired value. This is feasible because of the limited domain of spreadsheet functions and the functional aspect of spreadsheet languages. It remains to be seen if such change suggestions are feasible (or even useful) for more complex imperative languages.

2.4. SUMMARY

It is clear from prior work that program understanding is *hypothesis-driven*: people form hypotheses about the causes of a program's behavior and then test them by gathering evidence about its execution. It is a *social-technical* activity, in that people use both tools and their peers in order to gather this evidence. Program understanding also poses many kinds of *temporal* challenges: software changes over time; there can be considerable time between the insertion of an error and its visible consequences; for many failures, there is also a long time between the failure and its cause during a single program execution. Because of all of these factors, *where the search begins* is crucial factor in determining success. If programmers start by analyzing code that has no causal relationship with the failure, they will spend considerable time exploring other unrelated code. This initial choice of what code might be related determines the dependencies they follow, the coworkers they talk to, the analyses performed by automated tools, and the results they use to make their next move. Starting the search by guessing is bound to lead to problems.

Of course, most of these studies were done decades ago and most of these research tools are not in use. Have modern IDEs like Eclipse and Visual Studio remedied these challenges? Which of these findings remain true today? What are the *consequences* of approaching program understanding from a hypothesis formation and testing approach? What kinds of program understanding challenges exist in modern team software development? In the next few chapters, I will describe several studies that attempted to explore these questions observationally and empirically.

3.

BREAKDOWNS IN ALICE⁴

My investigation into program understanding activity began with two exploratory, contextual studies of how program understanding relates to other software development work. The studies involved the use of the Alice 2 programming environment [Cooper 2003], which had numerous users on the Carnegie Mellon campus and was also easy to learn in a short period of time. This meant that it was possible to see whole development cycles for a single program in little time. The goals of the studies were to identify *breakdowns* in programmers' work (a concept from *contextual inquiry* [Holtzblatt 1998]) and consider new types of tools that might prevent these breakdowns.

There were several task design issues to consider. Because the goals were exploratory, it was important to observe tasks of varying complexity. Therefore, the studies observe both experienced Alice programmers working on their own tasks and novice Alice programmers working on a predefined task. This allowed observations of situations where programmers were given specifications, as well as situations where programmers were free to define them as they worked. However, since Alice is a 3D programming system, a large part of writing an Alice program is creating the 3D objects that are manipulated at runtime. Because creating these objects is largely direct manipulation and not programming, I explicitly avoided designing or observing tasks that were primarily 3D object design.

⁴ The results reported in this chapter appeared in part in [Ko 2003a], [Ko 2003b] and [Ko 2005c].

3.1. THE ALICE PROGRAMMING ENVIRONMENT

The Alice 2 programming environment, shown in Figure 3.1, was designed to be the best first exposure to programming. As such, it has a number of unique features that are worth mentioning. First, users create and modify code by dragging and dropping “tiles” around a workspace, rather than using the keyboard to type and edit textual commands. The language itself is strongly typed, enforced by the dragging actions themselves (each 3D object is its own unique type and variables defined in procedures also have a declared type). The language is also object-oriented in the sense that every 3D object in the world has several predefined properties and behaviors. Each object may also be given custom properties and procedures. The execution model is multi-threaded and guided by a global event list, which invokes various commands on the objects in the world. Alice is ultimately intended for creating interactive animations and provides detailed support for many basic kinds of 3D transformations on objects and their parts. In some studies, Alice has been shown to raise grades and reduce attrition in computer science courses [Cooper 2003][Moskal 2004].

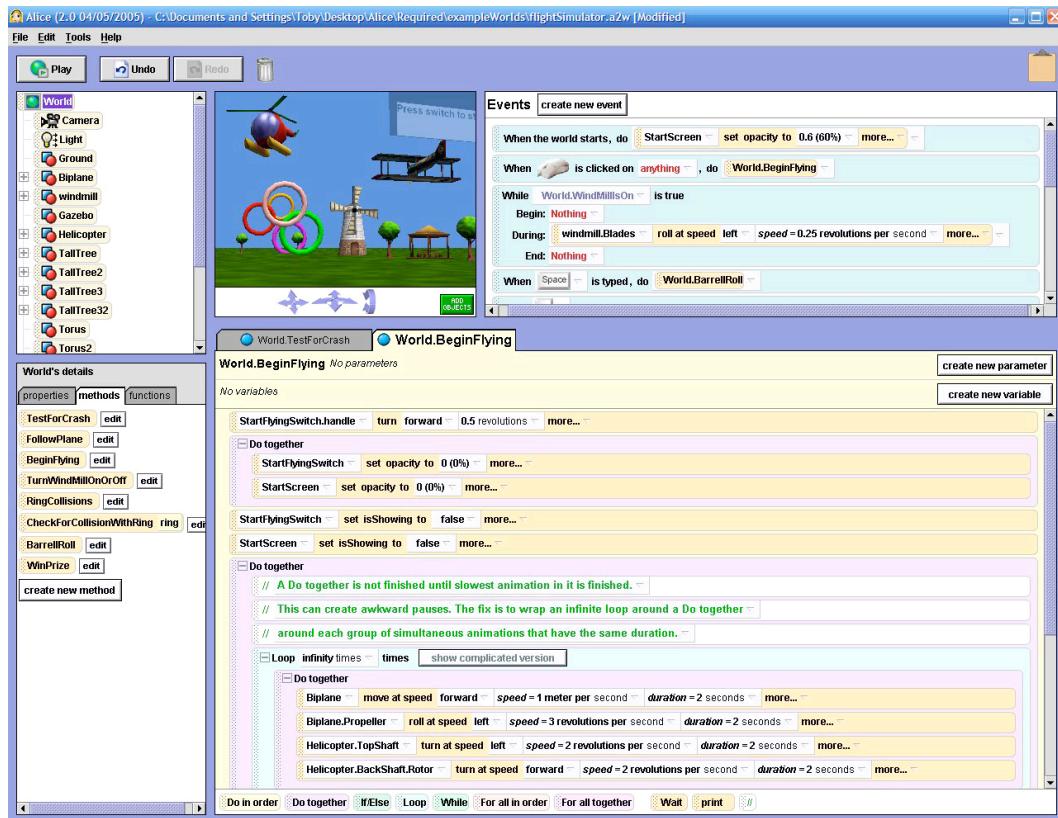


Figure 3.1. The Alice 2 programming environment.

3.2. ALICE IN THE FIELD

The first study explored the use of Alice in the context of the “Building Virtual Worlds” (*BVW*) course offered at Carnegie Mellon University in the Fall of 2002. The course required collaborations among programmers, modelers, sound engineers, painters, and students of other expertise to create a new interactive 3D world. The students were encouraged to work on their projects in a shared workplace, to facilitate face to face communication and collaboration. Each team’s class assignment during the time of observations was to use Alice to prototype a complex, interactive, 3D world over the course of two weeks. Since the projects were collaborative and unspecified, the requirements for each programmer’s Alice program were in constant flux, and the three programmers’ Alice programs were unalike.

3.2.1. METHOD

The Contextual Inquiry methodology [Holtzblatt 1998] was employed to understand the students’ work. The focus of the observations was to look for bottlenecks in the students’ activities. The experimenter recruited programmers by sending e-mail to each *BVW* teams’ class mailing list and soliciting participation. Three teams expressed interest and the observer scheduled separate times to observe each at work. When first meeting each team, the observer explained that the intentions of the observations were to “learn about how the programmer used Alice.” The observer described his role as an “interested learner” and described the programmer’s role as the primary speaker, making clear that the programmer was the domain expert. The observer then requested that the programmer think aloud while working, specifically explaining the rationale behind each decision made. Following this briefing, the observer began videotaping the computer monitor over the programmer’s shoulder using a Sony Digital 8 camcorder while the programmer worked on their own self-initiated tasks using Alice (as seen in Figure 3.2). During observations, the experimenter used the phrases “And now?” and “Please continue” thirty seconds after silence, to remind the programmers to think aloud. If a programmer left the computer to talk to a team member, the experimenter followed the programmer and recorded the discussion. Observations ended when a programmer had to stop working. Programmers were paid \$10 per hour for their participation. Details about the participants and their tasks are given in Table 3.1.

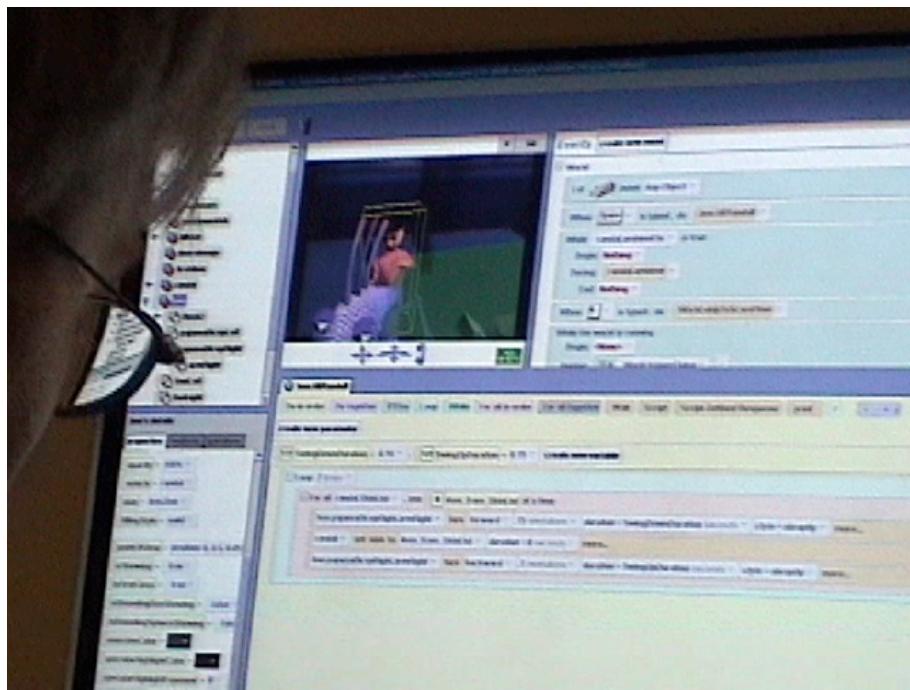


Figure 3.2. A ‘Building Virtual Worlds’ programmer fine-tuning an animation.

ID	Expertise	Observed Work time (min)	Programming Tasks
B1	Average C++, Visual Basic, Java	245	Parameterized a rabbit’s hop animation with speed and height variables Wrote code to make tractor beam catch rabbit when in line of sight Programmatically animated camera moving down stairs Prevented goat from penetrating ground after falling Played sound in parallel with character swinging bat.
B2	Above average C++, Java, Perl	110	Randomly resized and moved 20 handle-bars in a subway train
B3	Above average C, Java	50	Imported, arranged, and programmatically animated objects involved in camera animation.

Table 3.1. For the BVW study, programmers’ self-rated programming language expertise, their total observed work time, and the tasks that they worked on during observations.

3.2.2. RESULTS

Approximately 12 hours of observations were obtained from the three programmers over 12 sessions. Each of the sessions was reviewed for *breakdown scenarios*, in which a programmer's strategy was difficult to perform or unsuccessful. Some breakdowns were related to code. For example, the students reused code to perform similar operations such as animations or calculations, but the code was not properly adapted for its new location (this is the problem that the student in Figure 3.2 was working on). These bugs were particularly difficult to isolate because they propagated through complex animations, which depended on events. Most breakdowns, however, were related to testing and debugging. For example, the programmers used visual cues extensively in order to aid testing tasks. One programmer assigned the color of an object to the triggering of an event handler in order to verify the event occurred at the proper time. This was like writing a print statement or setting a breakpoint to verify that a particular method executed.

Participants also had difficulty testing code in “isolation,” as one participant described it. For example, programmers were forced to wait for long animation sequences to complete in order to test the end of the sequence. To tweak an animation, programmers made a small modification, wrote an event to run the animation when a key was pressed during runtime, ran the world, viewed the animation, and repeated. Also, to test a program’s response to an event in a specific world state, the programmer had to manually recreate the world state, and cause the event to occur. Programmers often had difficulty answering debugging questions of the form “when,” “why,” and “why not.” Students also struggled to conceive of and verify possible explanations. These questions marked the beginning of long and difficult investigations into the execution of a program and often failed. In most cases, the programmer just started over from scratch instead of trying to fix the broken code. This was partly due to the difficulty of modifying Alice code because of its rigid drag and drop editor.

3.3. ALICE IN THE LAB

One of the more interesting findings from the field study of Alice was that each debugging session began with a “why” question. To explore this phenomenon in more detail, the next study attempted to reproduce this finding in the lab and explore its nature in detail. The study involved 4 novice Alice programmers (all masters students in an HCI department) and was performed individually at a desk with a PC and 17” CRT. Programmers were recruited via an e-mail mailing list.

3.3.1. METHOD

In contrast to the BVW study, all 4 programmers were asked to complete the same task, which was to create a simple Pac-Man game with one ghost, four small dots, and one big dot (as in Figure 3.3). After a 15-min tutorial on how to create code, methods and events, programmers were given the same briefing as in BVW study, and were then given these requirements for the Pac-Man game:

- Pac must always move. His direction should change in response to arrow keys.
- Ghost must move in randomly half of the time and towards Pac the other half.
- If Ghost is chasing and touches Pac, Pac must flatten and stop moving forever.
- If Pac eats big dot, ghost must run away for 5 seconds, then return to chasing.
- If Pac Man touches the running Ghost, the Ghost must flatten and stop for 5 seconds, then chase again.

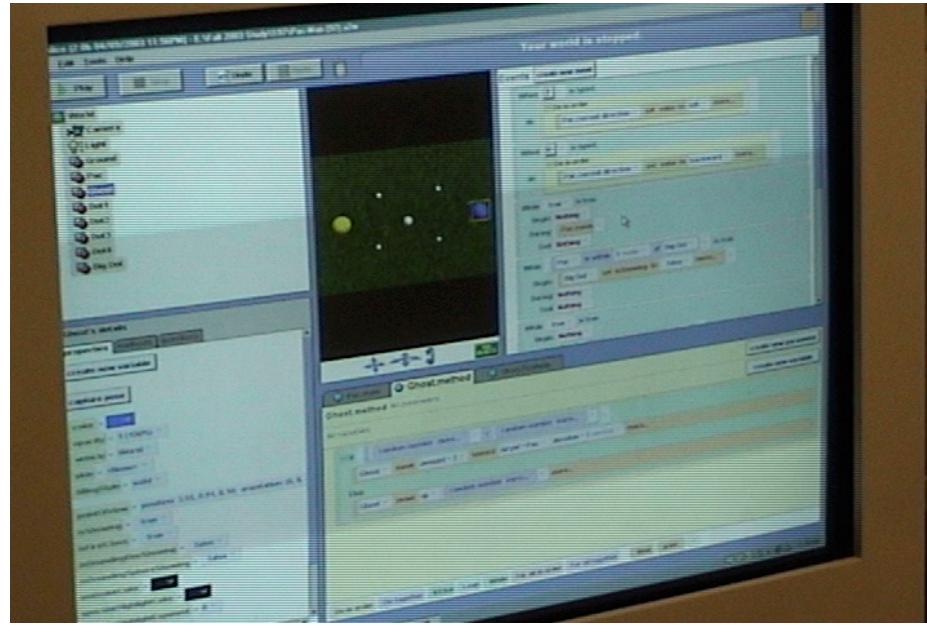


Figure 3.3. An image from the recording of a study participants' work on the Pac-Man task.

ID	Expertise	Work time (min)
P1	Above average Java, C	95
P2	Below average C++, Java	90
P3	Above average Java, C++	215
P4	Above average Visual Basic	90

Table 3.2. Details about the four participants of the Pac Man study.

Programmers remained at the desk throughout observations, and were videotaped over the shoulder with a Sony Digital 8 camcorder. As with the BVW study, the experimenter used the phrases “And now?” and “Please continue” as reminders to think aloud. Programmers worked for 90 minutes or longer if they wished to work more on the game. Programmers were paid \$15 for their participation as long as they completed at least 90 minutes of work. Table 3.2 lists programmers’ self-rated programming language expertise, their total work time. All participants were male except for P2. The questionnaire, tasks, and tutorial for this study appear in the Appendix.

3.3.2. RESULTS

There were a number of interesting general findings from the study, summarized here. First, the data was broken down into several debugging sessions. This process involved tracing backwards through each session and inferring the goal structure of the participants based on their actions and their verbal statements. These sessions were counted from when a “why” question was asked and ended when the developer was satisfied with the solution to the problem or gave up, moving on to another task. In analyzing these sessions in more detail, the first thing that was apparent was that when participants noticed failures while testing their program, they verbalized “why did” and “why didn’t” questions about their program’s output and behavior (about 68% of all of the participants’ questions were “why didn’t” questions). They only asked “why didn’t” questions about behaviors that they expected to happen because of code they had written (or thought they had written). In looking at these debugging sessions in more detail, an average of 46% of participants’ time was spent debugging. All of the time that each developer spent debugging was the result of an average of two or three false hypotheses about the cause of the program’s behavior. Because of these false hypotheses, about half of *all* of the developers’ errors were inserted while debugging some other error. No developer formed an accurate explanation of a program’s behavior on the first attempt.

3.4. AN ANALYSIS OF BREAKDOWNS

Although the results of each study in this chapter stand alone, there is much to learn by analyzing them together. To perform this analysis more systematically, this next section formalizes the notion of a “breakdown” along with a process to identify causes of breakdowns from verbal utterances from the participants. The end of this section then uses this formalization to analyze the results from the two studies discussed in the prior sections.

3.4.1. A FRAMEWORK FOR STUDYING THE CAUSES OF SOFTWARE ERRORS

The causes of software errors are rarely due to a programmer’s cognitive failures alone: a myriad of environmental factors, such as hidden or ambiguous cues in a programming environment, unfortunately timed interruptions, or poorly conceived

language constructs may also be involved. Thus, to truly support design, the programmer *and* the programming system should be considered together. To this end, there are a number of concepts to define:

1. Programmers perform three types of *programming activities*: specification activities (involving design and requirements specification), implementation activities (involving the manipulation of code), and runtime activities (involving testing and debugging).
2. Programmers perform six types of *actions* while interacting with a programming system's interfaces: design, creation, reuse, modification, understanding, and exploration.
3. *Skill breakdowns*, *rule breakdowns*, and *knowledge breakdowns* occur as a result of the interaction between programmer's cognitive limitations and properties of the programming system and external environment (as discussed in Chapter 2.2.2).

These aspects are combined into two central ideas:

1. A *cognitive breakdown* consists of four components: the *type* of breakdown, the *action* being performed when the breakdown occurs, the *interface* on which the action is performed, and the *information* that is being acted upon.
2. *Chains of cognitive breakdowns* are formed over the course of programming activity, often leading to the introduction of software errors (as defined in Section 2.2.2).

These ideas map directly to elements in the framework portrayed in Figure 3.4. The three grey regions, stacked vertically, denote specification, implementation, and runtime activities. The four columns contain various types of the four components of a breakdown. Breakdowns are read from left to right in the figure, with '[]' meaning "choose one within the brackets." For example, in specification activities, a single breakdown consists of one of three categories of breakdowns, one of three types of

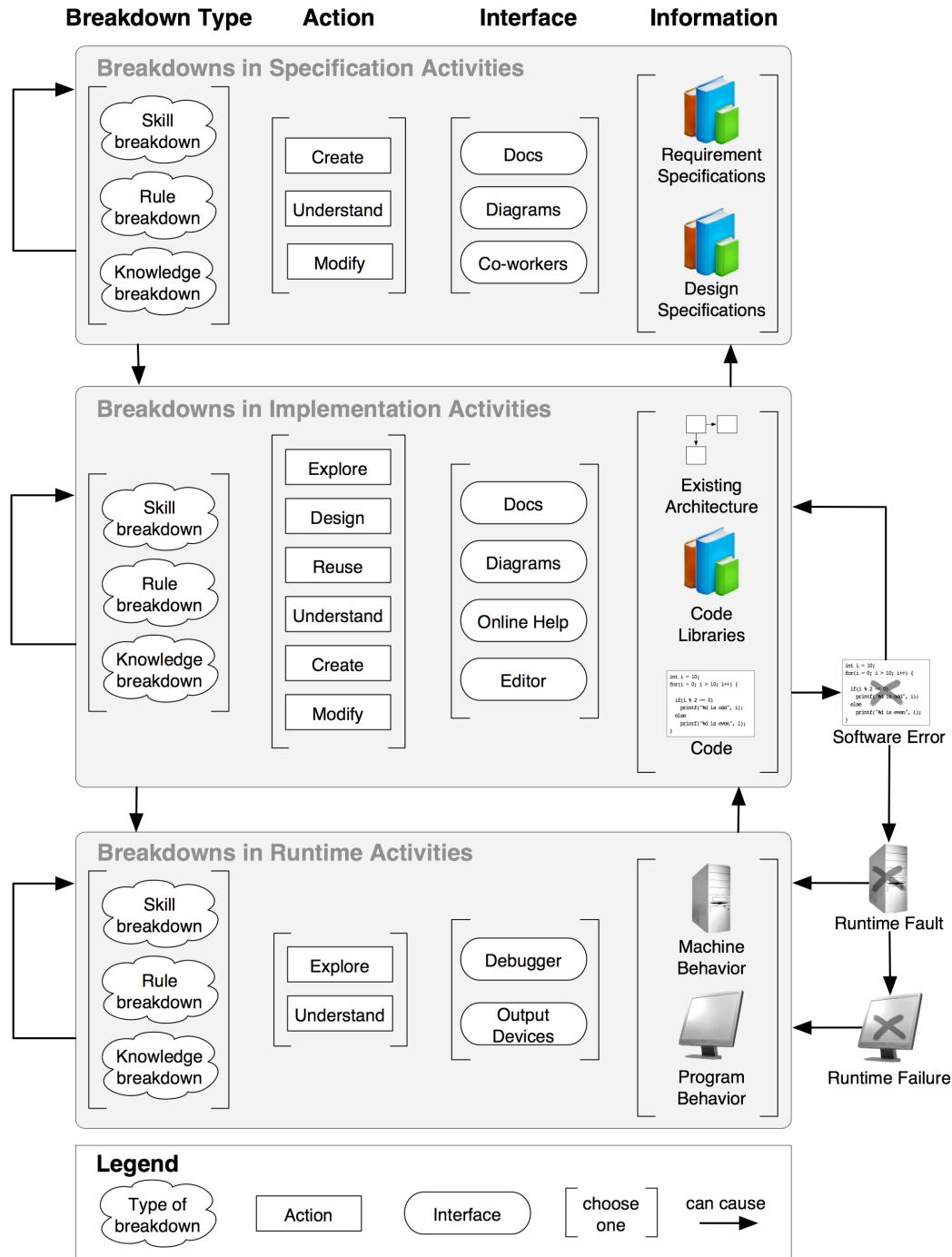


Figure 3.4. A framework for modeling the causes of software errors based on chains of cognitive breakdowns. Breakdowns occur in specification, implementation, and runtime activities. A single breakdown is read from left to right and consists of one component from each column within an activity. The cause of a single software error can be thought of as a path through these various types of breakdowns, by following the “can cause” arrows between and within the activities.

actions, one of three types of interfaces, and one of two types of information (therefore, the framework can describe $3 \times 3 \times 3 \times 2 = 54$ types of breakdowns in specification activities; in fact, because there are specific types skill, rule, and knowledge breakdowns, there are even more). Thus, one possible breakdown described by the framework would be “a knowledge breakdown in understanding a diagram of a design specification.” The actions, interfaces, and information for a particular activity are determined by the nature of the activity. For example, in runtime activity, programmers explore and understand machine and program behavior, but they do not create or design it.

Chains of breakdowns are represented by following the arrows in Figure 3.4, which denote “can cause” relationships. For example, by following the arrow from specification activities to implementation activities, we can say, “a knowledge breakdown in understanding a diagram of a design specification can cause a knowledge breakdown in modifying code.” The framework allows all “can cause” relationships *within* each activity; for example, during specification, “a software architect’s breakdowns in creating design specification diagrams can cause programmers to have knowledge breakdowns in understanding them.” The framework also supports relationships *between* activities, as in “breakdowns in modifying design specification documents can cause breakdowns in modifying code,” or, “breakdowns in understanding code in an editor can cause breakdowns in understanding design specification documents.”

In addition to describing “can cause” relationships within and between activities, the framework also describes relationships between software errors, runtime faults, runtime failures, and other breakdowns. For example, software errors can cause breakdowns in modifying code before ever causing a runtime fault: when a programmer makes a variable of Boolean instead of integer type, any further code that assumes the variable is of integer type is erroneous. A runtime fault or failure can cause various types of debugging breakdowns if noticed.

While the framework suggests many links between breakdowns, it makes no assumptions about their ordering. High-level software engineering processes, such as the waterfall or extreme programming models, assume a particular sequence of specification, implementation, and debugging activities; models of programming, program comprehension, testing, and debugging assume a particular sequence of

programming actions. The framework describes the causes of software errors in any of these models and processes.

To illustrate how these chains of breakdowns occur, consider the scenario illustrated in Figure 3.5. A programmer had little sleep the night before, which causes an *repetition* breakdown (a type of skill breakdown in Table 2.3) in implementing the swap algorithm for a recursive sorting algorithm; this causes a repeated variable reference. At the same time, a *faulty model* knowledge breakdown in understanding the algorithm's specifications causes an *overconfidence* breakdown in implementing a statement in the recursive call; this causes another erroneous variable reference. When he tests his algorithm, the two software errors cause two runtime faults, causing the sort to fail. When observing the failure, the programmer has a *problematic signs* breakdown in observing the program's output because it is displayed amongst other irrelevant debugging output, and he perceives a "10" instead of the "100" that is on-screen. This causes the programmer to have a *biased reviewing* breakdown in understanding the runtime failure: he forms an incorrect hypothesis about the cause of the failure, and neglects to consider other hypotheses. This invalid hypothesis causes a *selectivity* breakdown in modifying the recursive call, ultimately causing infinite recursion.

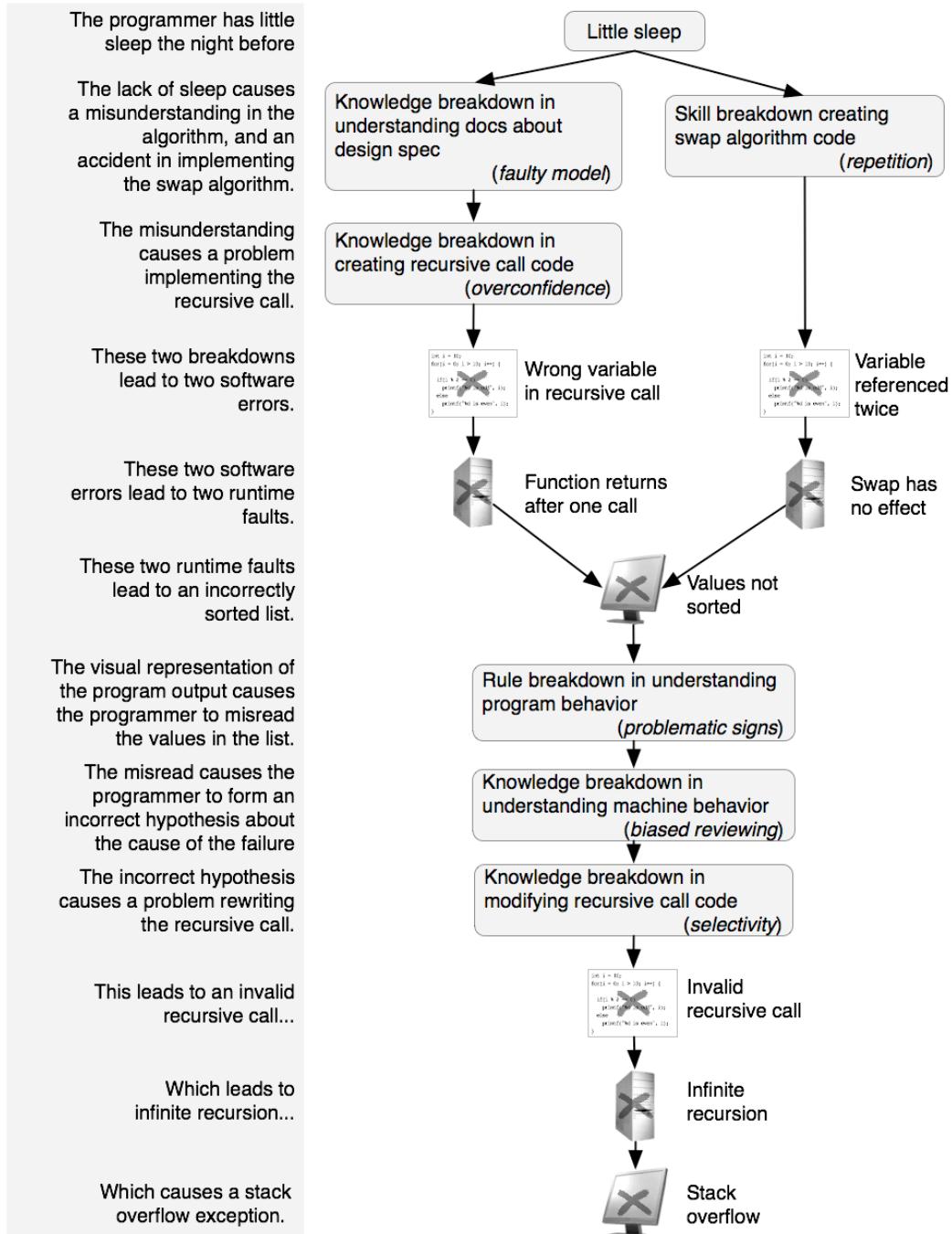


Figure 3.5. An example of a cognitive breakdown chain, where a programmer has several breakdowns while implementing a recursive sorting algorithm.

3.4.2. A METHODOLOGY FOR STUDYING THE CAUSES OF SOFTWARE ERRORS

One use of our framework is as a vocabulary for the causes of software errors in a programming system. It enables statements such as “This window in the code editor might make programmers prone to *problematic sign* breakdowns in cutting and pasting code, since it obscures part of the pasted text.” In this sense, it can complement other “broad-brush” frameworks such as the Cognitive Dimensions of Notations [Blackwell 2003]. However, the real strength of the framework is in using it to guide the empirical analysis of programming systems, with the goal of collecting design insights and inspirations that would have otherwise not been obvious.

This section describes a methodology for performing such analyses. It has four major steps:

1. Design an appropriate programming task.
2. Observe and record suitably experienced programmers working on the task, using *think-aloud* methodology [Ericsson 1984] to capture their decisions and reasoning.
3. Use the recordings to reconstruct chains of cognitive breakdowns by working backwards from programmers’ software errors to their causes.
4. Analyze the resulting set of chains of breakdowns for patterns and relationships.

How should programmers be observed and what should be recorded? Because the underlying assumption of the methodology is that a programming system is prone to a subset of all possible chains of breakdowns described by the framework, recordings should capture all four aspects of a cognitive breakdown. As just defined, a breakdown consists of four components: the *type* of breakdown, the *action* performed by the programmer, the *interface* used to perform the action, and the *information* acted upon. The latter three components are directly observable. For example, by watching a programmer use a UNIX environment to code a C program, one can observe the programming *interfaces* she uses (emacs, vi, man pages, etc.), the *actions* she performs using these interfaces (editing, shell commands, searching, etc.), and the *information* that she is acting upon (code, makefiles, text console output, etc.). The only unobservable component of a breakdown is its *type*—one of

the many types discussed in Section 2.2.2. Merely analyzing a programmer’s actions will not reliably suggest a programmer’s goals and decisions, since a single action may have many possible motives. Instead, *think-aloud* methodology [Ericsson 1984] should be used to elicit the causes of programmers’ actions. In think-aloud studies, the experimenter asks participants to provide self-reports of the decision-making and rationale behind their actions⁵.

To actually record breakdowns, videotaping programmers at work or recording the contents of their screen using video capture software with an accompanying audio recording works well. While this may seem like an unnecessarily large amount of data to gather and analyze, anything less than a full recording of a programmer’s interaction with a programming system can severely hinder the validity of assessments of the causes of a software error. In our experience, observations such as the pauses between clicks, the code scrolled to, what code is being focused on and even the *speed* of scrolling can all be reliable indicators of a programmer’s goals and decisions when combined with verbal utterances. For example, many environments show tool tips when the mouse cursor is hovered over particular code fragments; by only instrumenting a programming system to record high-level actions such as “tool tip shown,” “button pressed,” and “text deleted,” there would be no indication of whether the programmer actually meant to inspect the tool tip, or whether he just happened to leave the mouse cursor at that position while he consulted some printouts on his desk.

To reconstruct chains of cognitive breakdowns from a recording, a deductive approach in which one asks questions about a software error, runtime fault, runtime failure, or cognitive breakdown in order to determine its cause works well. This backwards reasoning proceeds until no further causes can be determined from the evidence. This process is illustrated in Figure 3.6, which reconstructs the chain presented in Figure 3.5.

⁵ The study followed Boren and Ramey’s guidelines for think aloud usability testing [Boren 2002]

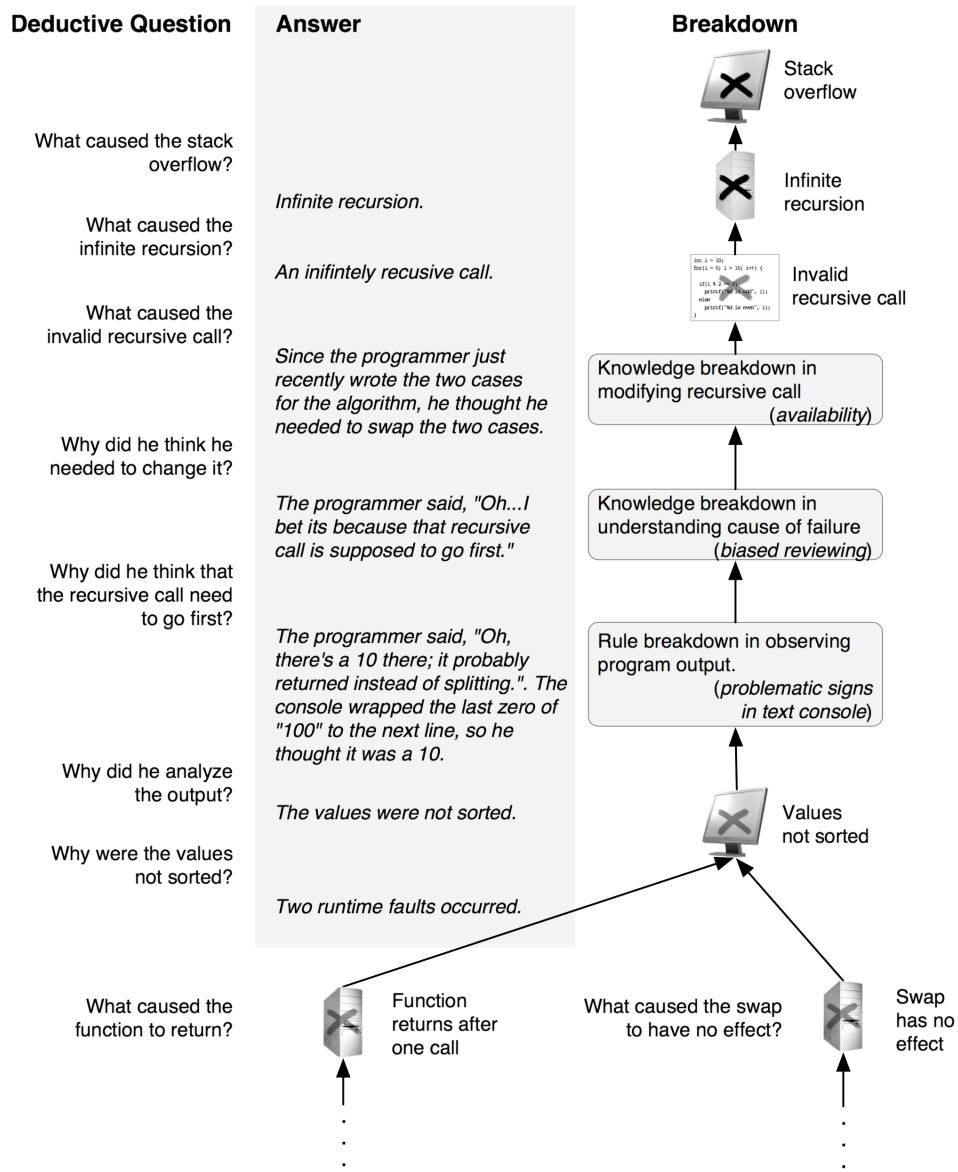


Figure 3.6. Deductively reconstructing the causal chain of breakdowns represented in Figure 3.5, using a programmer’s actions and speech.

The reconstruction begins from the program’s runtime failure, a stack overflow exception, by asking the deductive question, “What caused the stack overflow?” Deducing the chain of causality from this failure, to the runtime fault, from the fault to the software error is essentially debugging—to analyze the situation, one must understand the programmer’s code well enough to be able to determine all of the software errors that contributed to the program’s failure. This deduction can be done objectively, given enough knowledge of the program’s code and runtime

behavior. Performing this analysis from the videotape requires repeated rewinding and fast-forwarding, and thus having the video in digital format is quite helpful.

Once the software errors leading to the runtime failure have been deduced, one must determine what types of cognitive breakdowns led to the software error. For example, Figure 3.6 asks, “What caused the invalid recursive call?” Had the programmer said nothing about his actions, there would have been several explanations, but none with supporting evidence. However, because the programmer said, “Oh, I bet it’s because that recursive call was supposed to go first” and then proceeded to move the recursive call in his code, we can be relatively confident that it was an *availability* breakdown: the programmer assumed that his most recent changes to the code were responsible, rather than the swap code. We then proceed to ask deductive questions about each breakdown, until no further causes can be deduced from the evidence.

In some circumstances, there can be multiple events responsible for a single breakdown, at which point the chain is split in two. For example, in Figure 3.6, there are multiple reasons why the sort failed (two runtime faults, two corresponding software errors, and thus at least two cognitive breakdowns). In general, chains can branch at runtime failures (due to multiple runtime faults), at software errors (due to multiple cognitive breakdowns), and at breakdowns (due to multiple external events, such as interface problems or interruptions).

How should a breakdown’s type be determined? The analyses have used programmers’ verbal utterances and other contextual information to answer deductive questions about some action. For example, if a programmer types the wrong variable name in a method call, our deductive question would be, “Why did the programmer use variable X instead of variable Y?” This question is answered by considering the programmer’s past actions and verbal utterances. For example, if the programmer said, “What do we have to send to this method? Um, I think X.” we might deduce that he had a *biased reviewing* knowledge breakdown because he was in knowledge-based cognitive activity and only considered one course of action.

Table 3.3. A summary of skill, rule, and knowledge breakdowns, which can be used to answer deductive questions from observations.

Detecting Skill Breakdowns	
<i>Skill-based</i>	<i>The programmer...</i>
<i>activity is when...</i>	<ul style="list-style-type: none"> • Is actively executing routine, practiced actions in a familiar context • Is focused internally on problem solving, rather than executing the routine actions
<i>Skill breakdowns happen when...</i>	<ul style="list-style-type: none"> • Is interrupted by an external event (interruption) • Has a delay between an intention and a corresponding routine action (delayed action) • Is performing routine actions in exceptional circumstances (strong habit intrusion) • Is performing multiple, similar plans of routine action (interleaving) • Misses an important change in the environment while performing routine actions (exceptional stimuli) • Attends to routine actions and makes a false assumption about their progress (omission, repetition)
Detecting Rule Breakdowns	
<i>Rule-based</i>	<i>The programmer...</i>
<i>activity is when...</i>	<ul style="list-style-type: none"> • Detects a deviation from the planned-for conditions • Is seeking signs in the environment to determine what to do next
<i>Rule breakdowns happen when...</i>	<ul style="list-style-type: none"> • Takes the wrong action • Misses an important sign (favored signs) • Is inundated with signs (information overload) • Is acting in an exceptional circumstance (favored rules, rigidity) • Misses ambiguous or hidden signs in the environment (problematic signs) • Acts on incomplete knowledge (incomplete knowledge) • Acts on inaccurate knowledge (inaccurate knowledge) • Uses an exceptional, albeit successful rule from past experience as the rule (exception proves rule)
Detecting Knowledge Breakdowns	
<i>Knowledge-based</i>	<i>The programmer...</i>
<i>activity is when...</i>	<ul style="list-style-type: none"> • Is executing unpracticed or novel actions • Is comprehending, hypothesizing or otherwise reasoning about a problem using knowledge of the problem space
<i>Knowledge breakdowns happen when...</i>	<ul style="list-style-type: none"> • Makes a decision without considering all courses of action or all hypotheses (biased reviewing) • Has a false hypothesis about something (confirmation bias) • Sees a non-existent relationship between events (simplified causality) • Notices illusory correlation, or does not notice real correlation between events (illusory correlation) • Does not attend to logically important information when making decision (selectivity) • Does not consider logically important information that is unavailable, or difficult to recall (availability) • Is overconfident about the correctness and completeness of their knowledge (overconfidence)

To help make these judgments about a breakdown's type, Table 3.3 summarizes the various types of skill, rule, and knowledge breakdowns from Section 2.2.2.

Everything in this table specific to programming was adapted from the more general research on human error in [Reason 1990] and informed by experiences analyzing the data in the studies in this chapter. This table can be used to find an appropriate answer for each deductive question. This table has been an indispensable aid in considering the possible explanations for a programmer's behaviors. In our experience, when considering the context of some action, either a *single* type of breakdown stands out, or none do. If it is unclear which type of breakdown was to blame, the observations are probably insufficient for objectively deducing the cause of the cognitive breakdown. However, even in this case it is useful to record all of the possible interpretations, since programmers' actions or decisions that have yet to be analyzed may disambiguate a breakdown's type.

Although programmers' verbal utterances can be a valuable and reliable indicator of a breakdown's type, this is only true if the verbal data is analyzed in a reliable way. It is useful to test the reliability of interpretations by having multiple individuals reconstruct a subset of the software errors independently, and then checking for agreement in the types of breakdowns and structure of chains of breakdowns. This comparison is non-trivial, since it involves comparing not only the categories chosen from Table 3.3 (whether at the level of *skill*, *rule*, or *knowledge* breakdowns, or a finer grained comparison on subtypes of these breakdowns), but also for agreement on the causal links between breakdowns. Formalizing such a comparison technique, and how such comparisons might be converted into reliability statistics, is outside the scope of this dissertation.

Once a set of reliable chains of cognitive breakdowns has been reconstructed from observations, there are a wide variety of questions that can be asked:

- What activities are most prone to cognitive breakdowns?
- What aspects of the language and environment are involved in breakdowns?
- What types of actions are most prone to breakdowns?
- How do novice and expert programmers' types of breakdowns compare?
- What breakdowns tend to cause further breakdowns?

3.4.3. BREAKDOWNS IN ALICE

Because the programmers in each of the studies were responsible for their own design specifications (the actual implementation of their requirements), the analysis only involved chains based on software errors that caused failures. Errors that did *not* cause failures could not be identified: when specifications exist only in a programmer's head, the programmer is the only person who can deem that program behavior violates a specification. The observations across both studies resulted in 895 minutes of recordings, all of which was analyzed. In total, it took about 40 hours to analyze the 15 hours of recordings. The first phase of analysis was to search each recording for runtime failures by finding incidents where a programmer explicitly labeled some program behavior as incorrect (as in, "What? Pac's not supposed to be bouncing!"). Once these failures were found and time stamps were recorded for each, the next phase was to informally scan the programmers' actions before and after the failure in order to get a sense for what software errors were responsible for the runtime failure; in many cases, the software errors were obvious because the programmer later found the errors after debugging. Once the software errors were determined, deductive questions were asked about each, and programmers' verbal utterances in close temporal proximity were used to determine the answers. Time stamps were recorded for each of the breakdowns in the chain, along with other contextual details such as the interface and information involved in the breakdown.

To illustrate the nature of these analyses, consider the example of Figure 3.7 from one of P2's resulting chains. In the figure, the instigating breakdown in creating the specifications for the Boolean logic led to a breakdown in implementing the logic, which led to incorrect logic in the code. At the same time, P2 had another breakdown, assuming that a reference to "BigDot" was already included, but off-screen, when in fact it was not. This led to a missing reference error. Both of these software errors caused the conditional to become true after a single dot was eaten, causing Pac-Man to bounce before he had eaten all of the dots. When P2 observed the failure, she had a breakdown, in only forming one incorrect hypothesis about its cause. This false hypothesis led to a breakdown in modifying the expression, which caused Pac-Man to bounce immediately, before eating *any* dots. However, because P2 had moved the camera position to look down on Pac-Man, the failure was no longer visible. This caused P2 to have another breakdown, where she believed the earlier failure had been repaired because Pac-Man did not *seem* to be bouncing. Twenty minutes later, after repositioning the camera, she noticed that Pac-Man was actually

still bouncing at runtime, but assumed that her recently modified code was to blame, rather than the still incorrect Boolean expression. The interesting thing about this example is that it illustrates how easily a single mistake can propagate throughout a task. Each little mistake along the way compounds earlier mistakes. This means that the cost of a mistake early on is likely to be underestimated.

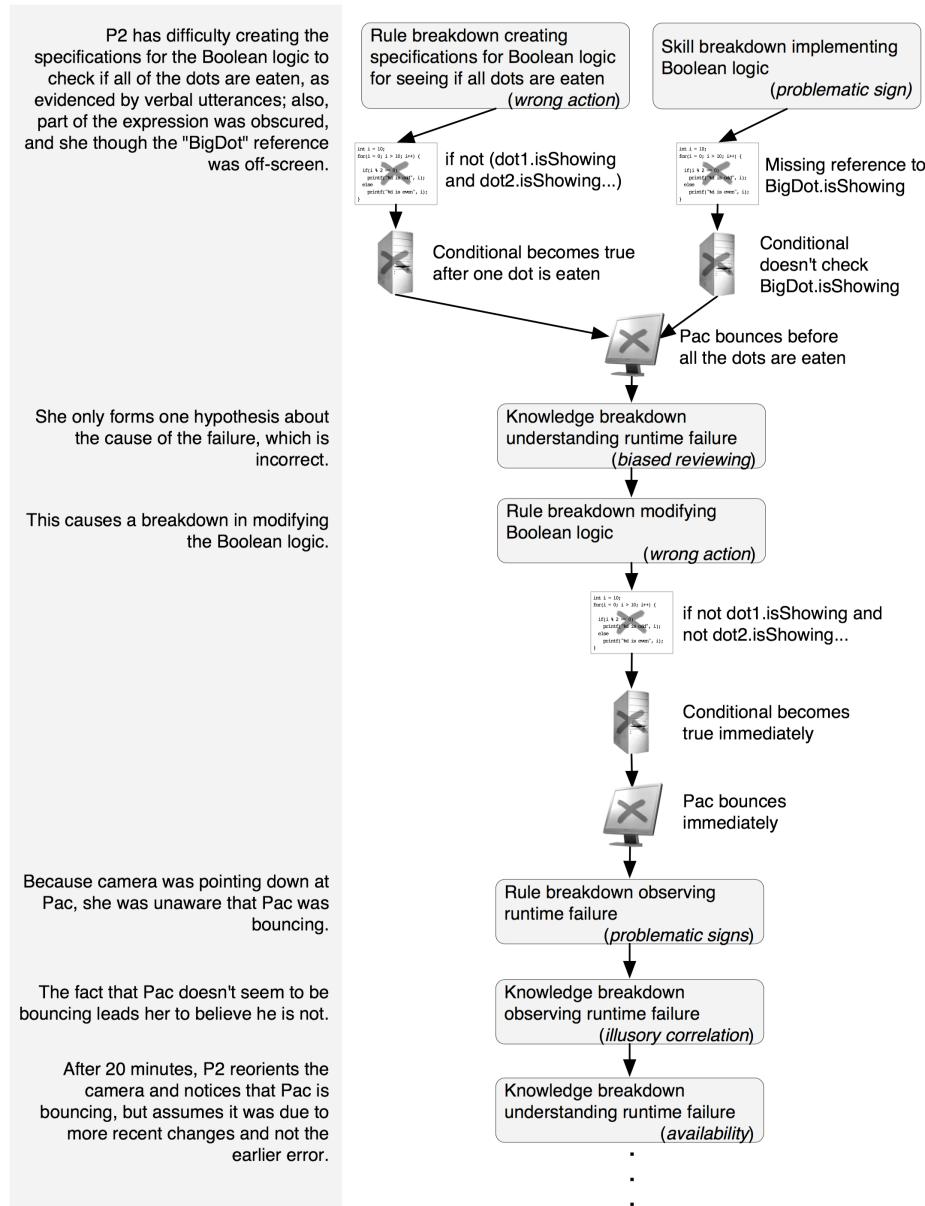


Figure 3.7. A segment of one of P2's cognitive breakdown chains. The last breakdown shown here did not cause further breakdowns until 20 minutes later, after the camera position made it apparent that Pac was still jumping.

One analyzer reconstructed breakdown chains like the one in Figure 3.7 from all of the 7 programmers' runtime failures. A second analyzer reconstructed chains for a random 10% of the runtime failures, to test for inter-rater reliability. Upon subjective comparison, the causal links between breakdowns in the reconstructed chains were largely the same, although each of the analyzers noticed some breakdowns that the other had not (as noted in Section 3.4.2, there is not yet a formal method for performing such comparisons). Although the time to construct the chains was not recorded, the analysis time was largely dependent on how much time the chain of breakdowns covered in the recording: for example, a chain spanned 10 minutes of video took twice as long to reconstruct than one that spanned 5 minutes of video. Overall, the analyses spanned approximately one 40-hour week.

Over 895 minutes of observations, there were 69 root breakdowns (breakdowns with no identifiable cause) and 159 total breakdowns. These caused 102 software errors, 33 of which led to one or more new software errors. The average chain had 2.3 breakdowns (standard deviation 2.3) and caused 1.5 software errors (standard deviation 1.1). Table 3.4 shows the proportions of time programmers spent programming and debugging. On average, 46% of programmers' time was spent debugging (and thus a little more than half was spent implementing code and understanding the problem). The BWW programmers, whose code was more complex, had longer chains of breakdowns than the Pac-Man programmers', suggesting that the causes of their software errors were more complex.

Table 3.4. Programming and debugging time, and the number of software errors, breakdowns, and chains, as well as chain length, by programmer.

ID	Programming Time		Debugging Time		# of Software Errors	# of Breakdowns	# of Chains	Average Chain Length Mean (SD)
	minutes	minutes	% of time					
B1	245	142	58.0%		23	41	10	4.1 (3.5)
B2	110	35	32.8%		16	32	7	4.6 (3.3)
B3	50	11	22.0%		3	5	4	1.2 (0.5)
P1	95	23	36.8%		14	23	11	2.1 (1.7)
P2	90	30	33.3%		7	7	7	1.0 (0.0)
P3	215	165	76.7%		34	44	25	1.8 (1.2)
P4	90	27	30.0%		5	7	5	1.4 (0.5)
Total	895	554	46.4%		102	159	69	2.3 (2.2)

As seen in Table 3.5, about 77% of all breakdowns occurred during implementation activity; these tended to be skill and rule breakdowns in implementing and

modifying artifacts and knowledge breakdowns in understanding and implementing artifacts. About 18% of all breakdowns occurred in runtime activity; these tended to be knowledge or skill problems in understanding runtime failures and faults. The proportion of skill, rule, and knowledge breakdowns were about equal. The root breakdowns of most chains were knowledge breakdowns understanding runtime failures and runtime faults and skill and rule breakdowns implementing code.

Table 3.5. Breakdowns split by activity and type.

Activity	Type of Breakdown	% of all Breakdowns
Specification	Skill	0.0%
	Rule	3.1%
	Knowledge	1.2%
	Total	4.4%
Implementation	Skill	22.0%
	Rule	28.3%
	Knowledge	27.0%
	Total	77.4%
Runtime	Skill	8.1%
	Rule	0.0%
	Knowledge	10.1%
	Total	18.2%

Table 3.6 shows which aspects of Alice were most often involved in breakdowns. Most breakdowns involved the construction of algorithms and the use of language constructs and animations. This is to be expected, since the majority of observations were of programmers completely new to the Alice programming system.

Table 3.6. Frequency and percent of breakdowns and software errors by type of information and the average debugging time for software errors in each type of information.

Type of Information	Breakdowns		Software Errors		Debugging Time Mean (SD) in minutes
	Frequency	% of all breakdowns	Frequency	% of all errors	
Algorithms	37	23.3%	34	33.3%	4.8 (6.2)
Language constructs	35	22.0%	31	30.4%	4.6 (5.5)
Animations	21	13.2%	19	18.6%	7.1 (6.9)
Runtime Failures	20	12.6%	-	-	-
Events	18	11.3%	10	9.8%	3.6 (4.2)
Runtime Faults	9	5.7%	-	-	-
Data Structures	8	5.0%	7	6.9%	3.3 (4.1)
Run-Time Specification Environment	5	3.1%	-	-	-
Requirements	4	2.5%	1	1.0%	1.0 (-)
Software Failures	0	0%	-	-	-

Table 3.7 shows the number of software errors and time spent debugging by problem and action. Most software errors were caused by rule breakdowns in implementing, modifying, and reusing program elements (rather than understanding or observing program elements). The variance in debugging times was high, and the longest debugging times were on rule breakdowns in reusing code and knowledge breakdowns understanding code.

Table 3.7. Software errors and debugging time by cognitive breakdown type and action. Only actions causing software errors are shown.

Breakdown	Action	Software Errors		Debugging Time Mean (SD) in minutes
		Frequency	% of errors	
Skill	Implementing	15	14.7%	5.2 (4.3)
	Modifying	14	13.7%	4.6 (7.1)
	Reusing	4	3.9%	1.2 (1.2)
	Total	23	22.5%	4.0 (5.1)
Knowledge	Implementing	15	14.7%	4.2 (4.8)
	Modifying	5	4.9%	5.4 (4.0)
	Reusing	1	1.0%	5.0 (-)
	Understand	6	5.9%	6.8 (5.7)
	Total	27	26.5%	5.3 (4.2)
Rule	Implementing	23	22.5%	4.2 (3.4)
	Modifying	16	15.7%	4.7 (5.1)
	Reusing	3	2.9%	6.6 (9.3)
	Total	52	51%	5.1 (5.4)

This analysis revealed four major causes of software errors in the studies. In each case, the Alice design shared a considerable portion of the blame. The most common breakdowns that led to software errors were breakdowns in implementing Alice numerical and Boolean expressions (33% of all errors, from the “% of all errors” column and “Algorithms” row in Table 3.6). Most were breakdowns in implementing complex Boolean expressions. For example, when programmers in the Pac-Man study wanted to test if all five dots were eaten, their expressions were *“if not (dot1.isEaten and dot2.isEaten...)”* which evaluates to true if *any* dots are eaten. In other cases, whether or not they had created a correct expression, programmers suffered from breakdowns in modifying the expressions: for example, after placing operators for *and* and *or* into the code, it was not always obvious which part of the expression they had affected because the change was off-screen or obscured.

With so many software errors introduced because of the implementation breakdowns, the breakdowns in debugging (18% of the total, from Table 3.5’s “Runtime” total) only complicated matters. These debugging breakdowns were due

to breakdowns in understanding runtime faults and failures. In particular, programmers often generated only a single, incorrect hypothesis about the cause of a failure they observed, and then because of their limited knowledge of causality in the Alice runtime system, generated an incorrect hypothesis about the code that caused the runtime fault. Because Alice provides virtually no access to runtime data, there were few ways for programmers to test their hypotheses, except through further modification of their code.

The 18% of knowledge breakdowns in “runtime” (in Table 3.5), in turn, were ultimately responsible for nearly all of the 24% of rule and skill breakdowns in modifying code, leading directly to software errors. This was because their hypotheses about the cause of the runtime failure had led them to the wrong code, or led them to make the wrong modification. However, these modification breakdowns were also due to interactive difficulties in modifying expressions. When programmers tried to remove intermediate Boolean operators, they often removed other code unintentionally, and because the structure of the code was not clearly visualized, did not realize they had introduced new errors during modification.

A final source of software errors, largely independent of the cycles of breakdowns described above, were the reuse breakdowns (7% of the total). These were breakdowns in reusing code via copy and paste, caused by problems in the copied code. In particular, after pasting copied code into a similar context, programmers began the task of coercing references from the old context to the new context. Oftentimes, several uncoerced properties were off-screen, causing the programmer to overlook the references. These software errors were very difficult to debug, because of breakdowns in understanding their copied code’s correctness. Thus, when programmers attempted to determine the cause of their program’s failure, their hypotheses were instead focused on other recent changes. Furthermore, because these software errors caused complex, unpredictable runtime interactions, programmers rarely found them.

Figure 3.8 summarizes these trends, portraying the most common causal links between breakdowns in the set of chains. The percentages on each line represent the proportion of each particular type of causal link between breakdowns among all links in the data set; together, they account for approximately 74% of all of the links in the chains (the remaining 26% were each below 2% of the data set, and thus are not shown in the figure).

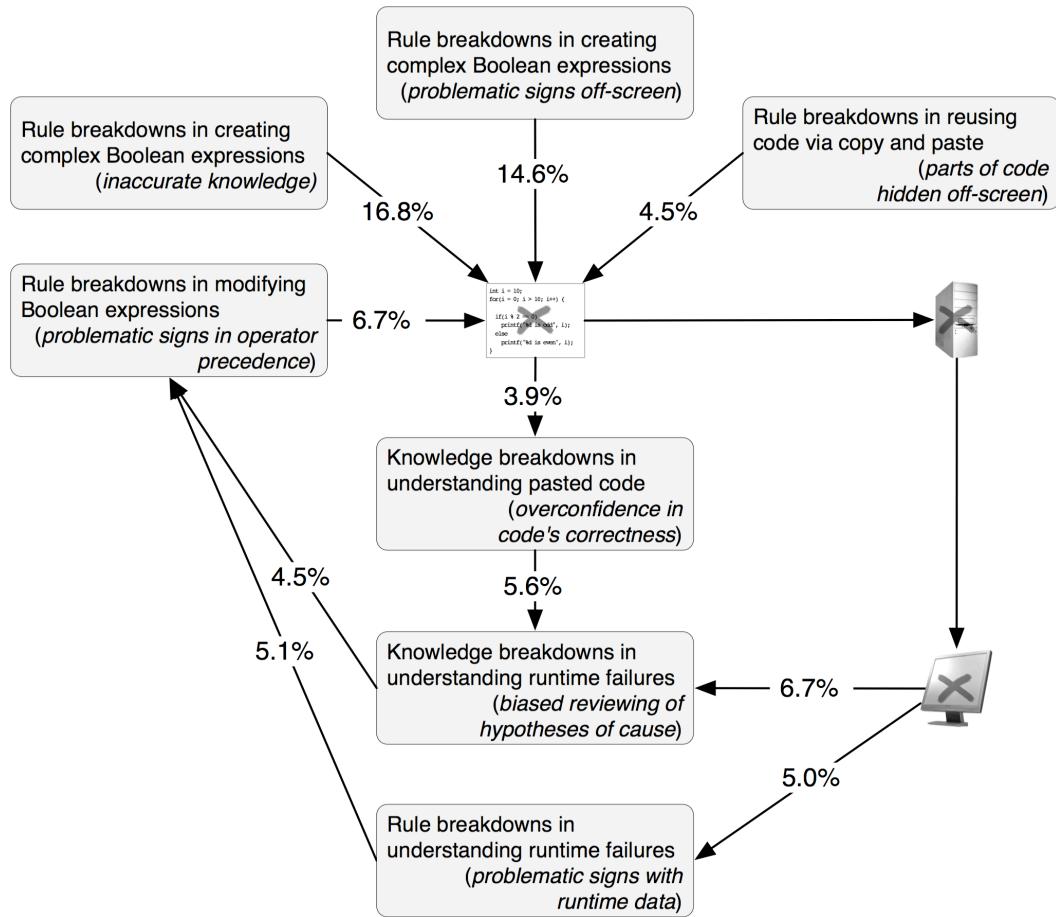


Figure 3.8. A model of the major causes of software errors during programmers' use of Alice. Each line represents a causal link between one type of breakdown and another; the number on the line represents the proportion of the type of link out of all links in all chains. (This does not include numbers for the links between software errors, runtime faults, and runtime failures, since it was only possible to identify errors that led to failures; also, the numbers do not add to 100% because not all types of breakdowns are shown).

Most of the root causes of software errors were from inexperience in creating Boolean expressions and forgetting to fully adapt copied code to a new context, but the impact of these early software errors was compounded by difficulties with debugging and modifying the erroneous code. In particular, only 18% of the breakdowns that occurred while forming hypotheses about the causes of runtime faults and failures (Table 3.5's total for "Runtime") were the cause of nearly all of the software errors introduced. Therefore, even in the simple tasks in this study, there

were complex relationships between the programming system's interfaces, the programmers' cognition, and the resulting software errors.

3.5. LIMITATIONS

Across all of the methodologies used in the studies in this chapter, the greatest limitation is the reliance on subjective interpretations of the participants' intents and beliefs. This limitation comes directly from the reliance on verbal data. The degree to which such data can be trusted, as discussed in section 3.4.2, depends on the type of task in which participants' thought aloud. For example, the participants in the studies in this chapter were working on largely high-level problem solving tasks, rather than proceduralized low-level activities such as typing. This increases the likelihood that what the participants said reflects at some level what they were thinking. Nevertheless, the conclusions made from such verbal data must be made cautiously and hopefully with other converging evidence from more reliable methodologies.

Another limitation comes from the vigilance of the person watching the videotapes. An important point regarding the methodology proposed in this chapter is that the causes of software errors can be quite small and seemingly insignificant, making it difficult for someone watching a video to know what to look for and when. It is possible that many interesting and explanatory events occurred in the video data that the video observer did not notice. This type of coding of behaviors and actions requires a degree of consistency and thoroughness unnatural to human attention. The recommendations made in this chapter may be one way to help focus an observer's attention on relevant events.

3.6. SUMMARY

This chapter contributed a number of findings and techniques:

- A framework for modeling the cognitive causes of software errors.
- A detailed a methodology for reconstructing the causes of software errors from video and verbal data.
- Characterizations of identified common breakdowns in Alice.

- Characterizations of the relationship between different types of breakdowns in Alice programming.
- Evidence that developers verbalize “why did” and “why didn’t” questions in response to program failures.
- Evidence that developers tended to form false hypotheses about the causes of program failures.
- Evidence that developers tend to insert new errors while debugging.

In addition to providing a high-level view of the common breakdowns in using Alice, the actual process of reconstructing the chains of breakdowns directly inspired several design ideas for error-preventing programming tools. For example, the dozens of breakdowns in understanding runtime failures provided a rich source of inspiration for the design of the Whyline. In the observations, immediately after programmers saw their program fail, they asked a question of one of two forms: “why did” questions, which assumed the occurrence of an *unexpected* runtime action, and “why didn’t” questions, which assumed the absence of an *expected* runtime action. It was immediately obvious from looking at these chains that the programmers’ implicit assumptions about what did or did not happen at runtime had gone unchecked, which led to a lengthy and error-prone debugging session to determine if their false hypothesis was correct. Not only did the data from the studies provide the key inspiration for the Whyline, but it also provided a valuable source of information for specific design decisions. For example, since programmers could easily verbalize their why questions during the think-aloud, why not directly support an interface for asking these questions? Furthermore, if users could choose from a set of “why did” and “why didn’t” questions rather than generate the question themselves, they might not form incorrect hypotheses about their program’s runtime behavior. These ideas are explored in later chapters.

4.

LEARNING BARRIERS IN VB.NET⁶

Although there were many interesting results regarding program understanding in the studies in the previous chapter, the greatest weakness is their generality. It may be that many of the findings were due simply to aspects of the Alice environment itself and not fundamental to software development in general. To address this limitation, the study in this chapter considers another environment intended for beginners: Visual Basic.NET.

The approach of the study was to look for “learning barriers” in not only the programming language, but the tools provided with the language, such as the debugger, the help system, the searching mechanisms, libraries, APIs, and so on. We know much about the learning barriers in programming *languages* [Pane 1996], but little about the rest. To clarify the phrase “learning barrier,” consider a beginning programmer, like the one portrayed in Figure 4.1. Let us suppose Jill is a user interface designer who just began learning Visual Basic. Shortly after starting, she realizes that she must learn about event handlers to proceed. This poses a potential learning barrier. From an attention-investment perspective [Blackwell 2000], she will weigh the cost, risk, and reward of overcoming the barrier, and if the risk of failure outweighs the reward, she may abandon VB for other tools.

⁶ The results in this chapter appear in part in [Ko 2004b].

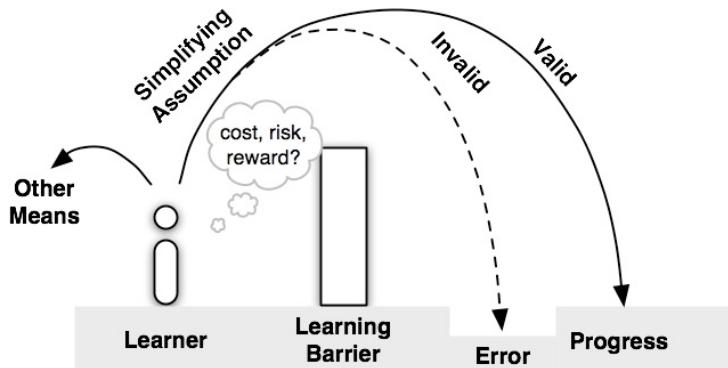


Figure 4.1. In overcoming barriers, learners risk making invalid assumptions that often lead to error.

Jill may also decide that progress is worth the risk of failure, but in doing so, will make several simplifying assumptions about VB's language, environment, and libraries in trying to acquire the necessary knowledge (as discussed in Section 2.2.2). If her assumptions are valid with respect to the programming system, she will make progress. If her assumptions are invalid, she is likely to make an error. Within this framework, *learning barriers* are aspects of a programming system or problem that are prone to such invalid assumptions.

4.1. METHOD

To discover learning barriers in end-user programming systems beyond just those in languages, the study involved observations of 40 non-programmers learning to use Visual Basic.NET 2003 in a course called Programming Usable Interfaces (<http://www.bam.hcii.cmu.edu/pui> offered in Fall 2004). VB was a suitable candidate because it shares many features with other end-user programming systems, such as an imperative syntax and limited support for designing visual features of a program and then augmenting them with code. It also has the added complexities of a compiled language, the Visual Studio environment and the object-oriented .NET framework.

The study methodology sampled incidents of learners reaching *insurmountable* barriers: properties of VB or a programming problem that the learner could not understand despite considerable effort. Learners were told that if they were stuck they could consult an oracle (the experimenters) for guidance. When learners sought advice via e-mail or in a public lab, they were asked to report (1) what they

were stuck on, (2) how they became stuck, and (3) how they tried to get “unstuck.” The oracles used the form shown in the Appendix to capture the barriers. The oracles then helped learners overcome their barrier. Learners worked on the tasks in Table 4.1 over 5 weeks. None of the learners who sought advice had taken more than one programming course.

- (1) Create a form that computes the average of 3 numbers in text fields.
- (2) Fix a form so that it sorts the names in the list reverse-alphabetically.
- (3) Write a program that is impressive in its utility or entertainment value.
- (4) Create a form with a chain of interaction using all of the VB widgets.
- (5) Design an alarm clock that can be set to ring at a certain time.
- (6) Make a simulation that shows 3 elevators’ directions and floors.
- (7) Design a copy machine interface that supports collating and stapling.

Table 4.1. The seven Visual Basic.NET tasks.

The sample included 74 insurmountable barriers (74 instances of students asking for help), but many of these were reached as a result of invalid assumptions that learners had made to overcome earlier barriers (as shown in Figure 4.2). By including these intermediate barriers, the final set included 130 barriers, all different situations in which a learner could not make progress without help.

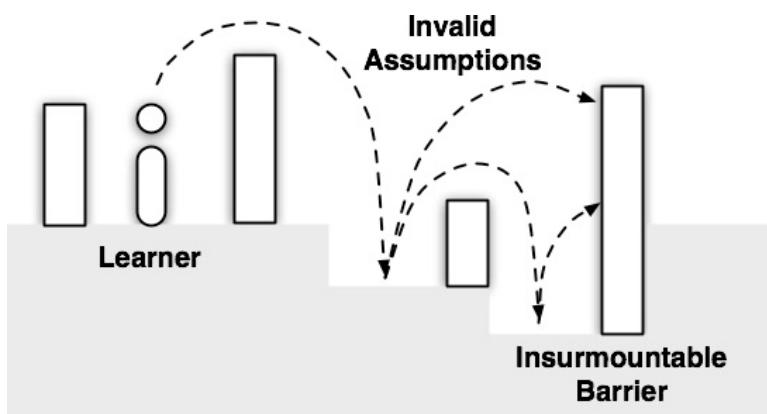


Figure 4.2. Learning barriers overcome with invalid assumptions often led to insurmountable barriers of a different type.

The goal of the analyses was to extract from these 130 situations a small set of distinct categories that capture the strategic goals of the learners. The process for identifying these categories involved printing out each scenario onto a small piece of paper and manually sorting and grouping scenarios that seemed similar in terms of the kind of information that the learner was seeking and the artifacts and content involved. This process, which involved two people, resulted in six distinct categories. Both people involved in the analysis then independently classified all 130 barriers using the six categories, and 94% of the data points were assigned the same category by both people. The disagreements were discussed until 100% agreement was attained.

4.2. SIX LEARNING BARRIERS

The six barriers are called *design*, *selection*, *coordination*, *use*, *understanding*, and *information* barriers. The definition of each relies on the concept of a *programming interface*, any element of a programming system's language or accompanying libraries that can be used to achieve some software behavior. These include language constructs such as loops and operators, and library calls such as animations and math routines, as well as any of the APIs included in the programming environment. Programming interfaces should not be confused with the *user interfaces* in an environment (compilers, editors, menus, etc), although they are conceptually equivalent.

4.2.1. DESIGN BARRIERS

Design barriers (4 of 130) are inherent cognitive difficulties of a programming problem, separate from the notation used to represent a solution (i.e., words, diagrams, code). Several problems posed design barriers, including sorting, communication between forms, conditional logic, and event concurrency.

Half of the design barriers were insurmountable (2 of 4) because solutions to a problem were difficult to visualize. For example, a learner working on task 2 (see Table 4.1) was unable to conceive of a systematic way to sort names. Her best solution was "Just keep moving the names until it looks right!" Learners who were able to conceive of a sorting algorithm made invalid assumptions about their solution. For example, one learner successfully tested one cycle of her algorithm on a

single data set on paper, and believed it to be correct. When her algorithm failed, she faced the insurmountable understanding barrier of determining what her algorithm did and did not do at runtime.

4.2.2. SELECTION BARRIERS

Selection barriers (13 of 130) are properties of an environment's facilities for finding what programming interfaces are available and which can be used to achieve a particular behavior. These emerged when learners could not determine which programming interfaces were capable of a particular behavior.

Half of the selection barriers were insurmountable (6 of 13). Many learners faced selection barriers in task 5 in trying to get their program to keep time. Some tried using the help system, but could not guess which keywords to use. If they happened to find a relevant article, they were unable to understand the description of VB's timing abilities. Many learners overcame selection barriers by using their peers' timing code as examples, but faced insurmountable *use* and *coordination barriers* in adapting them.

4.2.3. COORDINATION BARRIERS

Coordination barriers (25 of 130) are a programming system's limits on how programming interfaces in its language and libraries can be combined to achieve complex behaviors—what one learner called “the invisible rules.” Learners encountered these when they knew what set of interfaces could achieve a behavior, but did not know how to coordinate them.

Most coordination barriers were overcome with invalid assumptions (20 of 25). For example, learners correctly assumed that inter-form communication involved creating a new form programmatically and accessing its data (in VB a “form” is a window). However, most made invalid assumptions about how to access data and tried to “pull” values from the new form instead of “pushing” values to the old form. Because form controls are inaccessible if their form is not visible, “pulling” led to runtime exceptions. Learners also overcame coordination barriers by finding examples that revealed VB’s invisible rules. However, as with selection barriers, they faced *use* barriers and further *coordination barriers* adapting when trying to adapt these examples to their needs.

4.2.4. USE BARRIERS

Use barriers (36 of 130) are properties of a programming interface that obscure (1) in what ways it can be used, (2) how to use it, and (3) what effect such uses will have. These arose when learners knew what interface to use, but were misled by these obscurities.

About half of the use barriers were insurmountable (17 of 36), often because a programming interface did not indicate in what ways it could be used. For example, task 4 required learners to make a *Label* interactive, but many did not know that a *Label* could respond to mouse events. Some overcame these use barriers by using VB's facilities for obtaining a list of an object's methods. However, learners made invalid assumptions about how to use the methods or what effects they would have, passing syntactically correct but semantically *incorrect* parameters (also use barriers). Use barriers were also insurmountable when they involved syntax. For example, learners could not determine how to declare or initialize arrays; when they guessed, they made invalid assumptions, and encountered insurmountable *understanding barriers* in determining the meaning of the resulting syntax errors.

4.2.5. UNDERSTANDING BARRIERS

Understanding barriers (38 of 130) are properties of a program's *visible* behavior (including compile- and run-time errors) that obscure what a program did or did not do at compile or runtime. These emerged when learners could not evaluate their program's behavior relative to their expectations.

Most understanding barriers were insurmountable (34 of 38). Compile-time errors were insurmountable when learners could not determine what parts of their code were deemed right or wrong by the compiler, based on its error message. For example, when learners wrote a function call without a '=', they received the error message "*expected: =*". Learners faced an understanding barrier of determining if and where the '=' should be placed, and why it was "*expected*".

Runtime-errors and other unexpected behavior were insurmountable when they obscured what did or did not happen at runtime. For example, some learners wanted to pass data between forms, but did not know how to create references to forms to do so. To overcome this use barrier, they assumed that they could

instantiate a form of the appropriate type in the *Form_Load* event of each form, not knowing this would cause infinite recursion and a stack overflow exception. Most learners did not associate the exception with their earlier assumption, because it did not suggest a relationship to their code.

In other cases, learners expected a behavior that did not occur. For example, many learners created a `Timer` object, assuming that it would start counting at runtime, when it was in fact disabled by default. When their label's text did not update as expected, they overlooked their assumption, and as a result, could not imagine what prevented the label from updating. Most assumed that their update code was incorrect, and rewrote it. Of course, this led directly to the same understanding barrier.

A common strategy for overcoming these barriers was to seek out potential explanations for the problems from other more experienced classmates. For example, if a student was expecting some output that did not occur, they might ask a classmate, "why do you think this isn't happening?" and the classmate would offer some actionable explanation such as, "Have you tried inserting a print statement on this line, to see if the program reaches this point?" When the teaching assistants showed students how to use print statements to print out information while the program executed, many remarked that they did not know what to print out that would help them solve their problem.

4.2.6. INFORMATION BARRIERS

Information barriers (14 of 130) are properties of an environment that make it difficult to acquire information about a program's *internal* behavior (i.e., a variable's value, what calls what). These arose when learners had a hypothesis about their program's internal behavior, but were unable to find or use the environment's facilities to test their hypothesis.

Many information barriers were insurmountable (10 of 14) because the places to search for appropriate tools were numerous, or it was unclear how to use a tool. For example, many learners accidentally closed VB's property panel and could not determine how to redisplay it. Some learners caused null pointer exceptions, but did not notice that the exception dialog contained a link to the code responsible.

Some learners overcame information barriers by assuming something about their program's behavior. For example, when learners could not find the code that caused a null pointer exception, they deleted all of their recently modified code, confident that part of it must be guilty. When learners encountered barriers in using VB's debugger, rather than overcome them, they abandoned the debugger and simply guessed which statement was to blame.

4.3. DISCUSSION

The six barriers are related to other concepts. For example, they share characteristics of Norman's *gulf of execution* (the difference between users' intentions and the available actions) and *gulf of evaluation* (the effort of deciding if expectations have been met) [Norman 1988]. Three barriers pose gulfs of execution exclusively:

- *Design*: mapping a desired program behavior to an abstract description of a solution.
- *Coordination*: mapping a desired behavior to a computational pattern that obeys "invisible rules."
- *Use*: mapping a desired behavior to a programming interface's available parameters.

Two pose gulfs of execution *and* evaluation:

- *Selection*: mapping a behavior to appropriate search terms for use in help or web search engines, and interpreting the relevance of the results.
- *Information*: mapping a hypothesis about a program to the environment's available tools, and interpreting the tool's feedback.

Understanding barriers pose gulfs of evaluation exclusively, in interpreting the external behavior of a program to determine what it accomplished at runtime.

Norman's recommendations on bridging gulfs of execution and evaluation are easily adaptable to programming system design. For example, Norman recommends bridging gulfs of execution by establishing visible constraints on what actions are possible. For coordination barriers, this might involve a more explicit representation of a system's "invisible rules." To overcome gulfs of evaluation, Norman recommends

that a system's state be accessible and understandable relative to users' expectations.

The barriers are also related to each other. Figure 4.3 reveals common paths of failure in learning VB. The edges show the percent of each type of barrier that was overcome with invalid assumptions and the type of barrier to which the assumptions led. (Since Figure 4.3 only show edges greater than 10% and it excludes *insurmountable* barriers, the outgoing edges' of each node do not add up to 100%).

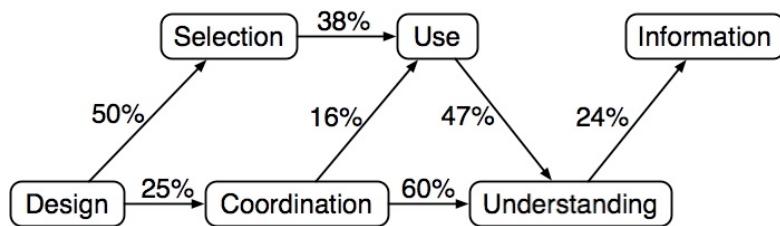


Figure 4.3. For surmountable barriers, the percent of each type overcome with invalid assumptions, and the type of barrier to which the assumptions led.

Selection barriers tended to lead to use barriers, suggesting that preventing invalid assumptions about programming interfaces' *capabilities* might avoid assumptions about their *use*. Furthermore, selection, coordination and use barriers—half of all observed—often led to understanding and information barriers. This implies that while debugging is a problem, a bigger concern is how prone VB's programming interfaces are to invalid assumptions *prior* to their use.

The six barriers were useful for classifying observations from studies of several end-user programming systems. For example, one common information barrier in the Alice programming system [Dann 2003] is that variables' values are inaccessible at runtime because the output window is modal. This is in contrast to VB's information barriers, where the most common information barrier is determining where to insert print statements. This also in contrast to informal observations of Macromedia Flash, where a common information barrier is finding a particular line of code among hundreds of frames. Thus, the barriers are general enough to capture differences between the barriers of at least three diverse programming systems. However, because the barriers have not been applied extensively, it is unlikely that they describe *all* possible barriers. With more data and more experience with the barriers, other categories may be apparent.

4.4. LIMITATIONS

There are a number of limitations of the methodology employed in this study. First, there are many kinds of learning contexts and the classroom is only one. The motivations of the students in the class observed may differ significantly from those of a learner on their own at home, learning to program as a hobby. These other contexts may have resulted in a different notion of “insurmountability,” since the concept of insurmountability depends directly on the concept of motivation. There may also have been some variation in the students’ willingness to ask for help; some students may have been intimidated by the teaching assistants or less willing to ask for help. These particular students may have experienced different learning barriers than those students who *were* willing to ask for help.

There is also the possibility that in the process of collecting data about the context of the students’ learning barriers, that the teaching assistants may have misunderstood the students’ problems. All of the assistants were familiar with the programming environment and language, but there may have been details about the students’ programs that the assistants overlooked, causing them to misinterpret the problem that the student was having. Other kinds of misinterpretations may have occurred when the barriers were being categorized, because the only information available about each barrier was the information recorded on paper. This verbal detail, especially since it lacked any visual context, could have been misinterpreted.

4.5. SUMMARY

With respect to program understanding, the approach of identifying learning barriers emphasized a number of students’ difficulties:

- There are at least six major types of barriers in learning to use programming systems that span across implementation, APIs, testing, debugging, and design.
- About 20% of the reported problems were *coordination barriers*, involved multiple objects not working together appropriately, for example, information from one window not being sent to another. (Section 4.2.3).
- In the majority of reported problems, students were stuck because particular behaviors did not occur, even though the students had

implemented code for the behavior. This is just like the “why didn’t” questions observed in the Alice studies (Section 4.2.5).

- Students struggled to even form a hypothesis about the cause of a problem, and so many recruited help from their more experienced peers in the form of hypotheses such as “have you tried to do...?” (Section 4.2.5).
- About 11% of the reported problems were *information barriers*, in which students could not find a tool in the environment that would help answer their question, or could not understand how to use a tool that they had found. (Section 4.2.6).

All of these results were consistent with the findings of the Alice studies.

5.

EXPLORING JAVA CODE IN ECLIPSE⁷

The weakness of the studies described in previous chapters is their focus on novice programmers. To address this limitation, this chapter describes a study of several experienced Java developers using Eclipse to work on debugging tasks. The goal of the study was to gain a more detailed understanding of how developers form their task contexts and how software development environments (SDEs) are related to this formation. *Task contexts*, discussed in [Murphy 2005] and (identified independently in [Ko 2005b] as *working sets*), consist of all of the information relevant to a developers' software development task, potentially including code, specifications, documentation, bug reports, and other artifacts. This study investigated the following:

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

The next section describes the design and methodology of the study and the following section investigates Eclipse's relationship to developers' work both qualitatively and quantitatively. The chapter concludes with a discussion of the limitations of the method and the implications of the study results on program understanding.

⁷ The results in this chapter appear in part in [Ko 2005b] and [Ko 2006b].

5.1. METHOD

Developers in the study were asked to correctly complete as many of five maintenance tasks over a 70-minute period as possible, while responding to intermittent, automated interruptions. Three of the tasks were debugging tasks, requiring developers to test the program and diagnose a particular failure. The other two tasks were enhancement tasks, which required developers to understand some portion of the system and modify it in order to provide a new feature. Interruptions were included because of recent evidence that interruptions are frequent in software engineering workplaces [Gonzalez 2004, Perlow 1999]. The decision to study developers in the lab instead of in the context of developers' actual work was driven by an interest in comparing multiple developers' strategies on identical tasks. Had each developer worked on different code, as would have been the case in a more realistic context, differences in developers' work, if any, could have been due to variations in their strategies, their code, or more likely, some combination of factors.

5.1.1. PARTICIPANTS

A total of 31 experienced Java developers from the local community participated, including both undergraduate and graduate students (some who were staff programmers). Analyses of various subsets of these developers' data have appeared in other publications [Fogarty 2005][Ko 2005a][Ko 2005b]; the analyses here focus on the 10 developers most experienced with Java, based on a pre-test, self-report survey: seven described themselves as "Java experts" and the remaining three described themselves as having "above-average" Java expertise (the other 21 developers described themselves as "average" or below). All reported using either Eclipse or Visual Studio "regularly," and reported programming a median of 17.5 hours a week (the distribution was bimodal, with developers programming either less than 20 hours or more than 35). Although all claimed some degree of Java expertise, having a variety of approaches to completing the tasks was important. Novice Java programmers' work was not of interest, however, because of the high variability in their knowledge and decision making [Curtis 1981]. The ten developers studied were all male, had ages ranging from 19 to 28, and included six senior computer science students, two doctoral students in CS, and two MS students in computer engineering and information systems.

5.1.2. THE PAINT APPLICATION

All of the tasks involved a program called *Paint* (shown in Figure 5.1). This was a Java Swing application, implemented with nine Java classes across nine source files and 503 non-comment, non-whitespace lines (available at <http://www.cs.cmu.edu/~natprog/data/paint.zip> and also in the Appendix). The application allowed users to draw, erase, clear and undo colored strokes on a white canvas. Its implementation was based on the `PaintObjectConstructor` class, which created a single `PaintObject` for each list of mouse locations accumulated between mouse down and up events. The canvas consisted of an ordered list of `PaintObject` instances, which was rendered from least to most recent. The application declared two subclasses of `PaintObject`: `PencilPaint` and `EraserPaint`. The `PencilPaint` class painted itself by iterating through the list of mouse coordinates and drawing beveled line segments between consecutive pairs. The `EraserPaint` class subclassed `PencilPaint`, overriding its `getColor()` method to return the color of the canvas, simulating the effect of an eraser. Developers were given no documentation about the implementation and the code was not commented.

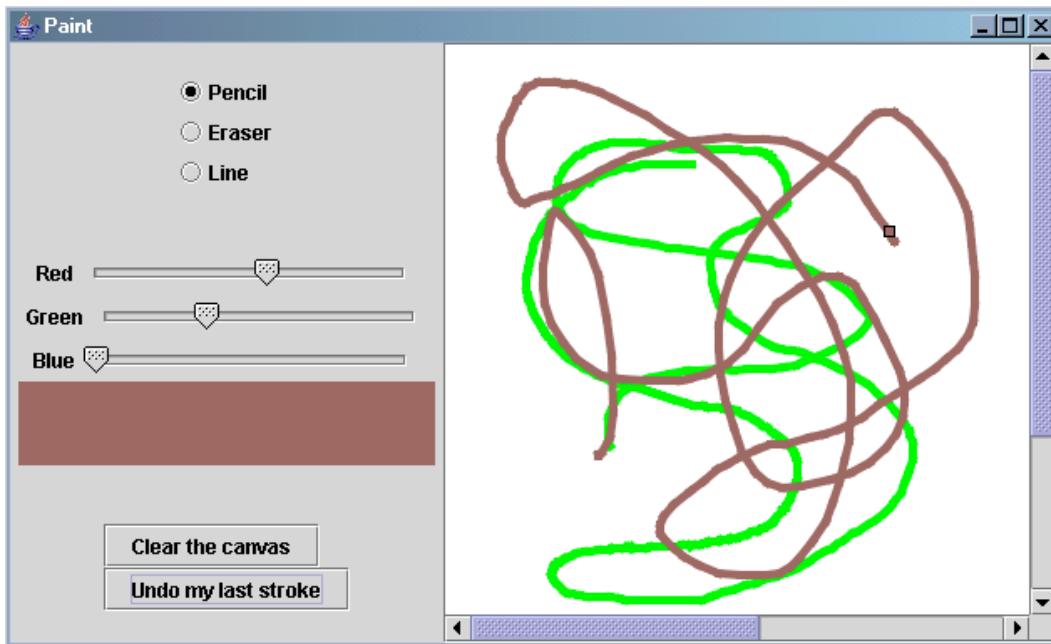


Figure 5.1. The Paint application.

Although the program was reasonably complex given its small size and the lack of documentation about its design, it was not as complex as programs that have been used in other studies [Robillard 2004], which were on the order of tens of thousands

of lines long. The primary reason for studying a smaller program was that it allowed an investigation of developers' work on several different tasks, and allowed detection of variations in developers' strategies on these different tasks; most prior studies have focused on a single task on a larger system.

5.1.3. TASKS

Developers were given a sheet of paper with the text in the middle column of Table 5.1, which describes five invented user complaints and requests (the sheet given to participants appears in the Appendix). The task names in Table 5.1 are used throughout this chapter, but were not given to developers. The descriptions explained the requirements for each task, so that each developer would have a similar understanding of the functional requirements. The last column of Table 5.1 describes a solution to each problem, including the minimum number of lines that had to be added, removed, or modified, and in how many files. These solutions were deemed by the author of *Paint* to be most consistent with the program's design. Because there were many valid solutions for each task, any solution that led to appropriate behavior (developers' actual code was verified later before the analyses) was considered correct. The errors for the debugging tasks were not artificial, but emerged during the actual creation of *Paint*.

5.1.4. TOOLS AND INSTRUMENTATION

Developers were given the Eclipse 2.1.2 IDE (released in March of 2004) and a project with the 9 source files. They were allowed to use debuggers, text editors, paper for notes, and the Internet. The only resource they were *not* allowed to use was the experimenter, who was only permitted to answer clarifying questions about the functional requirements described in the task descriptions. The browser's default page was the Java 1.4 API documentation. Developers used a PC with Windows XP, a keyboard, a mouse with a scroll wheel, and a 17" LCD. Because the analyses would involve a careful inspection of developers' actions, even at the level of mouse cursor movements, every detail of developers' work was captured with full screen-captured videos at 12 frames per second in 24-bit color, as well as audio. The display was limited to a resolution of 1024x768 to prevent any impact of the recording on the PC's performance.

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

Table 5.1. The five maintenance tasks.

5.1.5. INTERRUPTIONS

Interruptions came from a server on the experimenter's machine and appeared on the developer's machine as a flashing task bar item with an audible alert, as shown on the top of Figure 5.2. The interruptions were designed to require developers' full

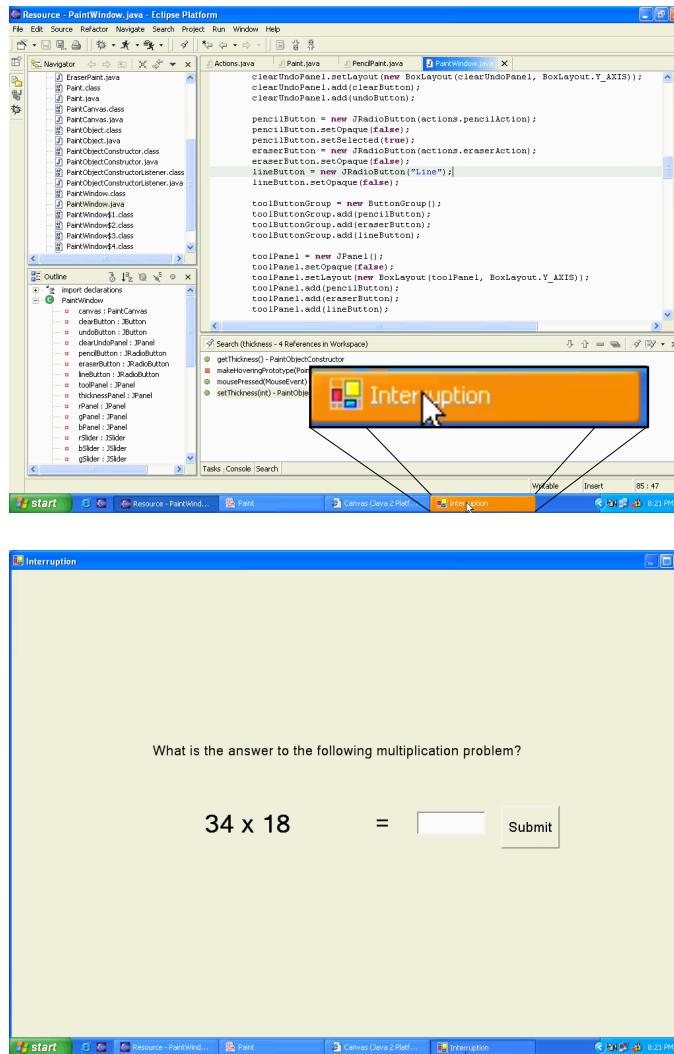


Figure 5.2. The flashing taskbar notification (top) and one of the arithmetic interruption tasks (bottom).

attention, mimicking real interruptions such as requests from coworkers for help on unrelated projects [Perlow 1999]. Thus, when clicked, a full-screen dialog appeared with a 2-digit multiplication problem and a text box for the answer, as shown on the bottom of Figure 5.2. Although the developers were told that they were not allowed to use any external resources to solve these problems, most found them so difficult

that they used the text field to store intermediate results while the experimenter was not looking. The server sent interruptions every two and a half to three and a half minutes. The order of the multiplication questions was fixed and identical for all developers. Each question was unique and did not contain 0 digits.

5.1.6. PROCEDURE

The experiment was run in the fall and spring months of 2004. Developers worked alone in a lab and began by completing a survey on their programming expertise. They were then told that they would be given five user complaints and requests for an application and would have 70 minutes to complete as many as possible (the 70 minute limit was made to keep the full session under two hours). Developers were told they would be paid \$10 for each request correctly completed. They were then told that a flashing taskbar item would occasionally interrupt them and that they should click it "when they were ready" and answer the arithmetic problem presented. The experimenter then explained that they would lose \$2 for each interruption ignored or answered incorrectly (this was used to give skipping the interruptions some cost during the study, but was not actually enforced when developers were paid). Developers were then told that their work would be recorded with screen capturing software and were then given the user complaints and requests and asked to begin. Afterwards, the experimenter tested the developers' solutions for correct behavior on the five tasks, paid the developer accordingly, and then answered any questions about the study.

5.2. RESULTS

This section discusses both qualitative and quantitative evidence for a number of patterns, based on about 12 hours of screen-captured video across 10 developers' work. The method for analyzing the videos involved two phases. The first phase involved looking ahead in each developer's video to find what code they inspected and modified and what behaviors they tested. Because there were only five tasks, this was enough information to determine the task they were working on. Once the task was determined, the video was scanned in reverse to find the moment when the developer began the task. This was obvious from pauses in activity after the developer tested the behavior they were modifying or implementing in their previous task. Once the sequence of tasks that a developer worked on was

determined, each task was then observed in detail, studying developers' individual actions, navigations, and choices, attempting to infer their high-level goals, and noting any interesting patterns regarding information seeking and management. This process also produced a list of developer actions that were important to understanding their behavior. These are listed in Table 5.2. Two people performed all of these observations together over about 40 hours.

Developer Action

Reading code, identified by text caret movement, mouse cursor hovering, text selection, and hovering of scroll bars in a fixed region.

Editing code, by typing, copy and pasting, refactoring, or quick fixes.

Navigating a static dependency, including any navigation from or to method declarations, method calls, class declarations, class references, and declarations, assignments, and uses of variables.

Navigating an indirect dependency, between code fragments that were indirectly related by two or more static dependencies.

Searching for text strings, within a file or the whole project.

Testing Paint by executing it from Eclipse.

Switching to documentation, either in the browser or inside of Eclipse.

Switching to Eclipse from Paint, web browser, interruption, etc.

Switching to a source file, and the Eclipse user interface used to do so.

Reading the task description, indicated by experimenter notes.

Starting a new task, identified by the task structure inferred.

Handling an interruption by clicking on the taskbar item.

Introducing an error that later caused Paint to behave inappropriately.

Table 5.2. Developer actions transcribed from the screen-captured videos.

In the second phase, the actions in Table 5.2 were logged for each developer's work. The same two people, on separate computers, stepped through the video, cooperatively creating a single log of each action, its start and stop time, and if relevant, a description of the code that was operated on and the user interface that was used to perform the action (for example, static dependencies could be followed using the Eclipse *Open Declaration* command, using one of the commands in the *Java Search* dialog, or manually). In addition to the actions in Table 5.2, a number of inferences about developers' questions and hypotheses were also recorded, based on the information they investigated. To help detect navigations of dependencies in the program, the *Paint* application's static dependencies were enumerated prior to

transcription. Each 70 minute video took about 3 to 4 hours to transcribe, resulting in 2,870 actions (shown by task and developer in Table 5.3). During this process, there were never disagreements about whether a developer action had actually occurred, but there were many cases where one evaluator missed an action that the other found. This synchronized logging caught many actions that would have otherwise been omitted.

Once the transcripts for each developer were created, the next step was to analyze the patterns that observed in the first phase of the analyses. Throughout this section, per-developer averages for reasonably normal distributions, as *average (\pm standard deviation)* and medians for other distributions. All time proportions *exclude* any time spent on handling the interruptions, which accounted for an average of 22% (± 6) of the developers' time.

5.2.1. DIVISION OF LABOR

Table 5.3 lists the number of developers attempting and completing each task and the average time spent on each. Developers finished an average of 3.4 (± 0.8) tasks in 70 minutes. Almost everyone finished the **YELLOW**, **UNDO**, and **THICKNESS** tasks, but spent most their time on the more difficult tasks, **SCROLL** and **LINE**. One developer completed all five correctly.

The bar chart shown in Figure 5.3 portrays developers' average division of labor in terms of the actions in Table 5.2 (some of the actions are grouped together, for example, the three types of application switching are grouped as "Switching Applications"). The error bars indicate the variation between participants. Developers spent about a fifth of their non-interrupted time reading code, a fifth of their time editing code, a quarter of their time performing textual searches and navigating dependencies, and a tenth of their time testing the *Paint* application. An average of 5% (± 2) of each developer's time was spent switching and reorienting between Eclipse, the web browser, interruptions and Paint. Of the 6% (± 4) of time that was spent reading the Java APIs, nearly all of it was read in the context of the JavaDoc documentation within the web browser, despite evidence that each developer knew that documentation was accessible within Eclipse. In a few cases, developers used Google to search documentation and examples. Of course, each

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

Table 5.3. Task completion statistics for the ten developers, including the average time spent on each task and the number of actions per task per developer.

developer had a unique distribution of labor, as noted by the error bars. For example, some developers spent more time editing than others and correspondingly less time on other activities.

5.2.2. TASK STRUCTURE

The actions in Figure 5.3 were not independent: before editing code, developers had to determine what code to edit, and before determining this, they had to find it. Although all of these low-level actions were interleaved to some degree, the observations of developers' work indicated a higher-level sequence of choosing a task to work on, searching for task-relevant information, understanding the relationships between information, and editing, duplicating, and otherwise referencing the necessary code. Because developers' searches often failed and developers often inserted errors that had to be fixed, portions of this sequence were interleaved.

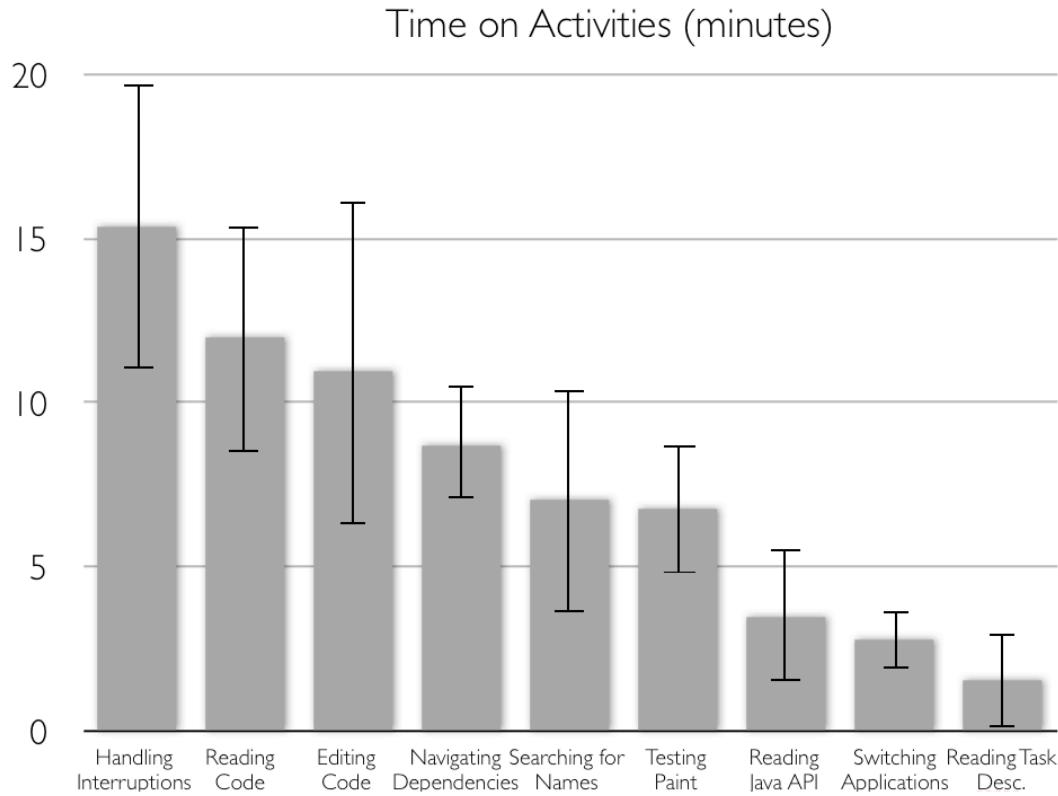


Figure 5.3. Developers' division of labor in terms of time on activities. The vertical bars represent one standard deviation above and below the mean.

To attempt to illustrate this sequence with data, the actions in Table 5.2 were grouped into four categories: *searching*, *navigating*, *editing*, and *other*. In the first category were textual searches and reading task descriptions; in the second were static dependency navigations, switching between files, and reading API documentation to understand API dependencies; in the third were copying and editing code and indirect dependency navigations (as defined in Table 5.2), which occurred later in the task, once the developer had comprehended the necessary code. The remaining actions, such as testing and switching files, were categorized as *other*, since they were activities that seemed to occur throughout the developers' work. Each action from the **THICKNESS** and **YELLOW** tasks was then categorized (allowing a comparison of one enhancement task and one debugging task that all developers completed).

Using these categorizations, each developer's action sequence was plotted, resulting in Figure 5.4. The vertical axis is the category of action (excluding the *other* category for clarity), and the horizontal axis is time, normalized between the start and end of work on the task. The categorization was only an approximation: a textual search

did not *always* indicate that the developer was looking for task relevant code, because in many instances, developers used textual search as a navigational tool to get from one place in a file to another. Within the plots, there were also several activities interleaved. For example, while determining where to add a declaration for a thickness slider, several developers also inspected the nearby slider event handler. Once developers had implemented a solution in both of these tasks, they often returned to correct errors, which involved further navigations and edits.

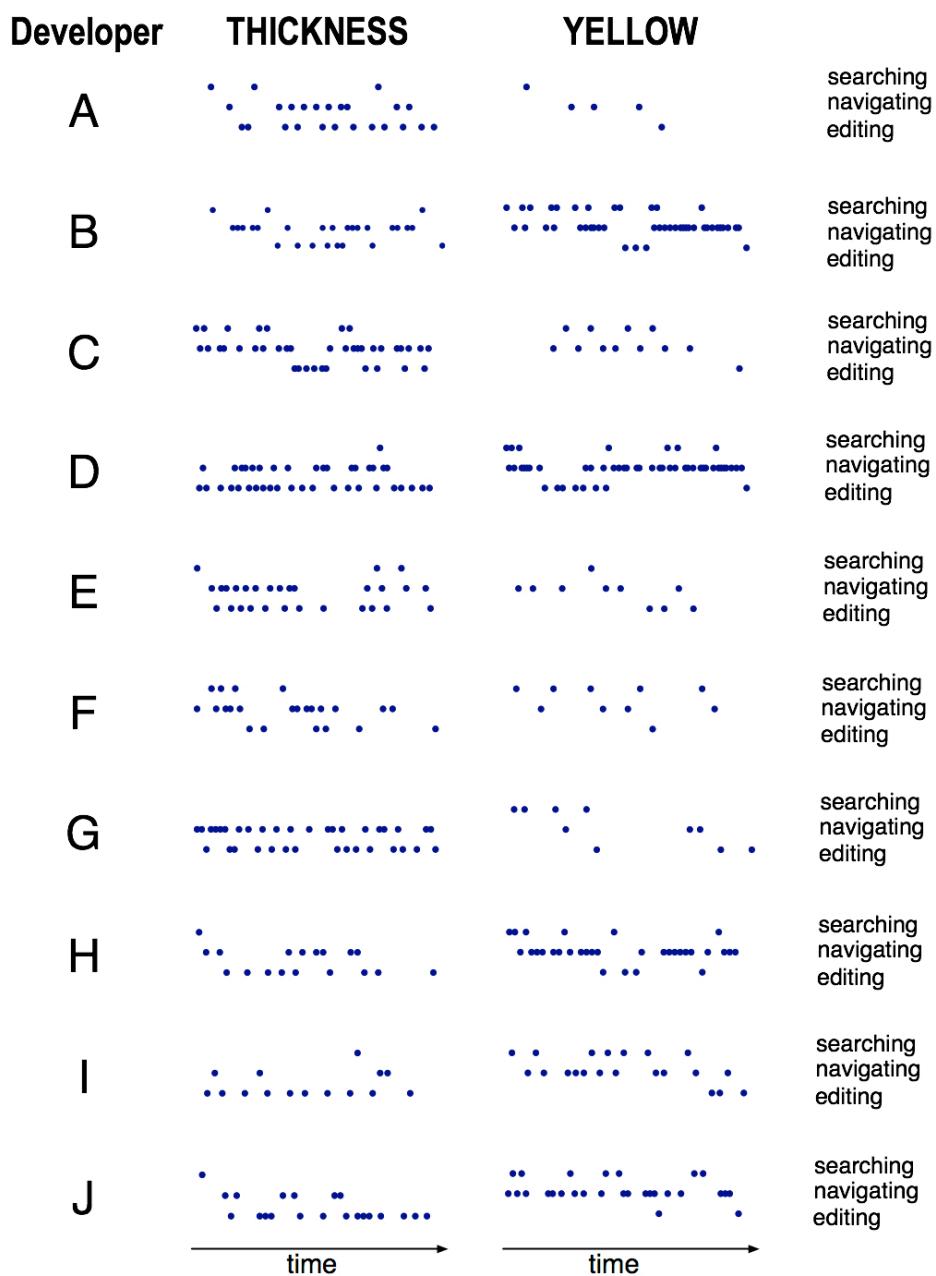


Figure 5.4. The developers' actions for the THICKNESS and YELLOW tasks.

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

5.2.3. SEARCHING FOR TASK RELEVANT INFORMATION

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

For the debugging tasks (**SCROLL**, **YELLOW**, and **UNDO**), developers used *symptoms* and *surface features* of the program's failure to guide their searches. For example, 8 of the 9 developers who attempted the **SCROLL** task first resized the `Paint` window and noticed that the canvas was only partially painted; thus, searched for a method with the name "paint" in it, which always resulted in the `paintComponent()` method of the canvas, which was not responsible for the bug. An average of 88% (± 11) of developers' searches led to nothing of later use in the task. These failed searches were at least partially responsible for the average of 25 (± 9) minutes of their time (about 36%) spent inspecting irrelevant code. That no one identifier in the code could fully represent the code's purpose is related to the *vocabulary problem* [Furnas 1987]. The cost of these incorrect guesses in the debugging tasks demonstrates how much the developers' early perceptions of relevance impacted their work.

When developers began the enhancement tasks (**LINE** and **THICKNESS**), their investigations of the source code were driven by a search for *extension points* in the code (places where additions to the code would be consistent with the program's existing architectural design). For example, 5 of the developers began the **THICKNESS** by searching for how the other sliders were implemented, and duplicating the code,

3 learned how to create an action object for the thickness slider, and 2 began by searching for how the stroke thickness might be set, investigating the `PaintObject` and `PaintCanvas` classes. Of the 8 developers who attempted the `LINE` task, 3 began by inspecting how the pencil and eraser tools were implemented, eventually copying one of them, and 2 began by investigating how the application created paint objects from the mouse events, 2 began by investigating the `Action` objects defined by for the pencil and eraser tools, and 1 began by investigating how to render lines.

5.2.4. FORMING PERCEPTIONS OF RELEVANCE

The process that developers used to determine the relevance of code or information involved several levels of engagement with information and several types of cues to which developers attended in order to decide whether to continue comprehending some information. For example, a common progression in the observations was as follows: a developer would look at the name of a file in the package explorer in order to judge its relevance. If it looked interesting, he would hover over the file icon with the mouse, and possibly select (but not open) the icon⁸. At this point, if he thought the name seemed relevant, he double-clicked on the icon to open the file, or expanded the node in the package explorer in order to inspect its contents. Developers who expanded the explorer node hovered over the names of methods and fields, looking for relevant identifiers, whereas developers who opened the file tended to scroll through the file, skimming the code for identifiers that looked relevant, or comments that might explain the intent of the file (though there were no comments). If developers found a method or algorithm of interest, they would inspect it more closely, sometimes even selecting the text of the code repeatedly.

The user interfaces that Eclipse provided for summarizing code, such as the package explorer and search tools, determined the structure of these investigations. For example, Eclipse's package explorer allowed developers to consider file names and identifiers before looking at more detailed information. Had these interfaces not been available, developers would have had to open many more files and look at

⁸ Although the developers could see the names without hovering with the mouse, for some developers, hovering seemed to be a common way of expressing additional interest in some name. Perhaps this was because hovering is the precursor to clicking.

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

Table 5.4. Types of dependencies navigated, the average percent of each type for a developer, and the tools that developers used to perform each.

more information before finding what they believed to be relevant code. One problem with these summaries was that they were often misrepresentative of the content. The most glaring examples of this in the data involved misleading names. For example, when developers worked on the **YELLOW** task, half of them first inspected the `PencilPaint` class, but the file that was actually relevant was the generically named `PaintWindow`.

5.2.5. TYPES OF RELEVANCE

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

and added user interface components in order to guide their own implementation. Of course, there may be other types of relevance that were not observed.

5.2.6. NAVIGATING DEPENDENCIES OF RELEVANT INFORMATION

After reading a segment of code, developers explored the code's incoming and outgoing dependencies. During this exploration, developers generally followed the static relationships listed in Table 5.4. Overall, each developer navigated an average of 65 (± 18) dependencies over their 70 minutes session; these were inferred during the transcription process, since few of them used Eclipse's navigation commands. There were two types of dependency navigations transcribed. Some were *direct* dependencies that could be determined by static analyses, such as going from a variable's use to its declaration, or from a method's header to an invocation of the method. The other type of navigation was of *indirect* dependencies, such as going from a variable's use to the method that computed its most recent value. These were program elements that were indirectly related by two or more static dependencies. Developers tended to make these indirect navigations later in each task, as seen in the "editing" phases in Figure 5.4. Developers' proportions of each type of dependency navigation are given in Table 5.4.

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

developer spent six minutes searching for a name elsewhere before finding that there were several uses in the original file.

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

An average of 42% (± 20) of developers' navigations were of *indirect* dependencies (this proportion may be even higher, given the difficulty of detecting them in the videos). Because Eclipse's support for navigating direct dependencies was unhelpful for these, developers used the scroll bars, page up and down keys, the package explorer and the file tabs instead. When navigating *within* a file using the scroll bars, scroll wheel, or page up and down keys, developers had to perform visual searches for their targets, costing each developer, on average, a total of 10 (± 4) minutes. Three developers avoided this overhead by using Eclipse's bookmarks to mark task-relevant code but then always ended up having more than two bookmarks to choose from and could not recall what code each one represented. This required clicking on each bookmark, which was no faster than their average scrolling time. Bookmarks also incurred the "cleanup" costs of their later removal when starting a new task. To navigate indirect relationships that were *between* files, developers had to use the *package explorer* and the *file tabs*. When several tabs were open (as in Figure 5.5), developers could not read the file names because they were displayed in abbreviated form and many shared the common prefix of "Paint" in their name. If the package explorer had several expanded nodes (as in Figure 5.5), developers had to scroll to find their targets. Overall, all of this overhead cost each developer 5 (± 1) minutes.

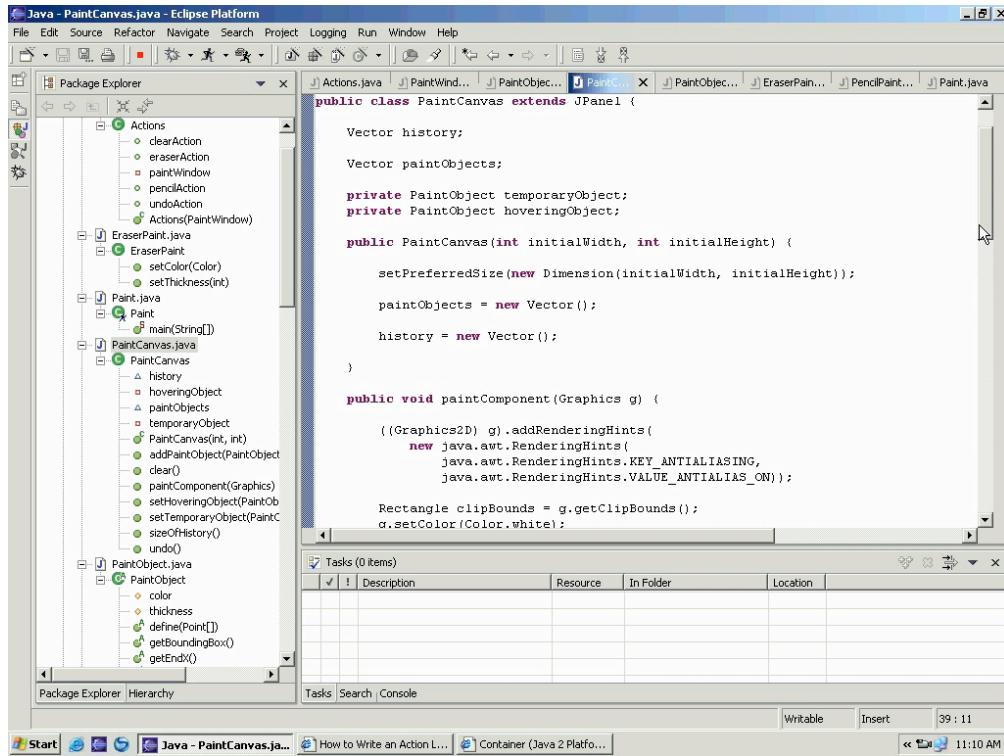


Figure 5.5. The package explorer, file tabs, and scroll bars of Eclipse 2.0.

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

5.2.7. REPRESENTING TASK CONTEXTS

*Task contexts*⁹ [Murphy 2005], defined at the beginning of this chapter, include all of the information relevant to a developer's software development task, potentially including code, documentation, specifications, bug reports and other information.

⁹ I identified this concept in [Ko 2005b] as a developer's *working set*, but later settled on Murphy's phrase because it was more descriptive of the concept.

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

Although these interfaces were helpful, they were far from perfect. The scroll bars only helped developers remember the most recent relevant section of code in a file, and as soon as they moved it, the mark was lost. Five developers temporarily left the **LINE** and **SCROLL** tasks to work on easier tasks, but because part of their task context was represented by the open file tabs and the state of the package explorer (see Figure 5.5), they often lost their task context when closing tabs or package explorer nodes to make space for information relevant to the new task. When developers returned to their earlier task, they spent an average of 60 seconds (± 28) recovering their task contexts. Furthermore, tabs opened during previous tasks made it more difficult to find relevant tabs, because the tab names were truncated (as seen in Figure 5.5). One problem with the package explorer was that developers often found a relevant method or field in a file, but to use the explorer to navigate to it, they had to keep the whole *file* expanded. For example, developers used the explorer to navigate to `pencilAction` for reference during the **LINE** task (shown in Figure 5.5), but in doing so, they also had to show all of the *irrelevant* code in `Action.java`.

5.2.8. VARIATIONS IN DEVELOPERS' TASK CONTEXTS

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

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

Answering these questions requires knowledge of what code developers thought was relevant, and what code they did not. Because this information was unavailable—and developers may not have even explicitly formed this knowledge—it was approximated by looking for developer actions that might indicate a developer's decision that some code or information was relevant. Because of evidence that there were several stages involved in forming perceptions of relevance, the more preliminary decisions of relevance were ignored, such as opening a file or reading a method, and instead the analysis focuses on more final indicators: (1) editing a code fragment, (2) navigating a dependency from a particular line of code, and (3) dwelling on API documentation found as a result of reading a particular identifier in a particular line of code. Although these indicators are not without uncertainty, they allowed an approximation of the set of code fragments that each developer may have thought relevant. Unfortunately, what developers would have actually chosen is unknown, and so the error in the approximation cannot be assessed.

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

ID	Relevant Information for THICKNESS	Time & Success	Relevant Information for LINE	Time and Success
A	36 lines PaintWindow EraserPaint PencilPaint JSlider documentation 14 returns across 3 files.	17 min ✓	34 lines PencilPaint, PaintObjectConstructor PaintWindow PaintObject PaintCanvas LinePaint class created JRadioButton documentation 6 returns across 3 files.	26 min ✗
B	50 lines PaintWindow PencilPaint PaintObject EraserPaint JSlider documentation SliderDemo.java example code ChangeListener documentation 39 returns across 3 files	29 min ✓	(didn't attempt)	■
C	36 lines PaintWindow PencilPaint Actions PaintObjectConstructor 15 returns across 2 files	16 min ✓	13 lines Actions PaintWindow PaintObject PaintObjectConstructor LinePaint class created 15 returns across 3 files	25 min ✓
D	41 lines PaintWindow PaintObjectConstructor LineThickness class created 23 returns across 2 files.	11 min ✓	12 lines Actions PaintWindow, LinePaint class created 11 returns across 3 files.	16 min ✓
E	18 lines PaintWindow EraserPaint, JSlider documentation 10 returns across 1 file	8 min ✓	16 lines PaintWindow PaintObjectConstructor PaintCanvas Actions LinePaint class created 6 returns across 2 files	30 min ✓
F	27 lines PaintWindow PaintCanvas PaintObjectConstructor Returned 19 times across 1 file	21 min ✓	(didn't attempt)	■
G	29 lines PaintWindow PaintObjectConstructor EraserPaint JSlider documentation 13 returns across 3 files	10 min ✓	19 lines Actions PaintCanvas PaintObjectController PaintWindow PaintObjectConstructor PaintObject LinePaint class created Rectangle2D.Double class Rectangle class 32 returns across 4 files.	47 min ✗
H	31 lines PaintWindow EraserPaint PaintObjectConstructor LinePaint class created 16 returns across 3 files.	35 min ✓	11 lines Actions PaintWindow PaintObject, LinePaint class created 3 returns across 1 files.	12 min ✓
I	26 lines PaintWindow EraserPaint 7 returns across 1 file	11 min ✓	13 lines Actions PaintWindow, PaintObjectConstructor LinePaint class created 15 returns across 3 files	20 min ✓
J	33 lines PaintWindow PaintObjectConstructor 22 returns across 1 files	11 min ✓	18 lines Actions PaintWindow PaintObjectConstructor PaintObject PencilPaint LinePaint class created 9 returns across 3 files	13 min ✓

Table 5.5. An approximation of developers' task contexts for THICKNESS and LINE, derived from edits, dependency navigations, and searches.

The resulting sets of relevant information are shown for all 10 developers in Table 5.5. The first column is the developer ID, and the second and fourth columns contain the number of relevant lines for the **THICKNESS** and **LINE** tasks respectively, the files they were in (in order of decreasing number of relevant lines), and also the number of times relevant code was returned to. The third and fifth columns list the amount of time each developer spent on the tasks and whether they succeeded or not. The minimum number of lines to successfully complete each task was 12 for **THICKNESS** and 15 for **LINE**.

What information did all developers find relevant, and how did it relate to the code that was actually relevant to a task? In general, successful developers' relevant information included the parts of the program that were part of the solutions described in Table 5.1. For example, everyone found similar segments of the `PaintWindow` class's constructor method relevant, because that was the place where the user interface was constructed, and thus where the thickness slider would be added. Everyone who was successful at creating a line tool found the `Actions` class relevant, because that class had to be modified to include an action for the line tool radio button.

How did developers' task contexts differ? One way was in how much information they deemed relevant. For the **THICKNESS** task, developers deemed a median of 32 lines relevant, and for **LINE**, a median of 17 lines, not including the `LinePaint` class that each developer wrote, which was generally about 20 lines (the median is reported for the latter task because the distribution was non-normal. For both of these tasks, this was about 7% of the 508 lines in the whole program. Note that this is less than the standard 40 lines visible in an Eclipse editor or other standard code editor, but in none of these editors is it possible to show these exact lines together in a single view. Developers also differed in the kind of information they found relevant. For example, many developers consulted documentation on the `JSlider` class, and many looked for examples on how to use the class. Other differences seemed due to strategy. For example, developers differed on which lines of `PencilPaint` were relevant to the **THICKNESS** task because some noticed the `setThickness()` method, but those that did not changed the rendering algorithm instead. Other differences were due to prior understanding of the program; for example, some developers on the **THICKNESS** task only looked at the few files necessary to modify, possibly because they learned about other information in earlier tasks. Others indicated part of almost every file as

relevant, possibly because they needed or wanted to understand more about how the strokes were being generated and rendered.

How often did developers return to code they perceived as relevant? For the **THICKNESS** task, developers returned an average of 18 (± 9) times to code that was transcribed as relevant, and for the **LINE** task, an average of 12 (± 9) times. Not only does the magnitude of these numbers give some validity to the measurements of perceived relevance, but it also reinforces the earlier findings of navigational bottlenecks in the Eclipse user interface (Section 5.2.6). There was no linear relationship between the time that a developer spent on a task and the number of relevant lines of code deemed relevant. There was a linear relationship between time on task and the number of returns on the **THICKNESS** task ($R^2=.65$, $F=17.8$, $p < .005$), but not for **LINE**. This difference may be explained by that fact that most of the new code required for the **LINE** task was contained within a single method, unlike the **THICKNESS** task.

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

5.2.9. IMPACT OF INTERRUPTIONS

In order to understand the impact of interruptions, developers' actions before and after the interruption were compared subjectively. Interruptions only had an impact on developers' work when two conditions were true: (1) important task state was not externalized into the environment at the time of acknowledging the interruption and (2) developers could not recall the state after returning from the interruption. Developers were very careful to externalize important task state before acknowledging the interruption. For example, in *every* case where a developer was interrupted while he was performing an edit, whether large or small, the developer

always completed the edit before acknowledging the interruption. This was even the case when one developer had just copied the entire `PencilPaint` class in an effort to convert it into a new `LinePaint` class: before acknowledging the interruption, he modified the class name, constructor name, commented out the old rendering algorithm, and wrote a comment about an implementation strategy for the new algorithm. In other cases where the task state was stored implicitly in Eclipse, developers forgot to externalize the state. For example, seven developers were interrupted just after repairing a syntax error, but just before saving the changes. If they had saved, it would have caused Eclipse to incrementally compile and remove the syntax error feedback. Because they did not save, when they returned from interruptions, they often saw the underlined syntax error, and tried to repair the already valid code. In these seven cases, developers spent an average of 38 seconds before they realized that Eclipse's feedback about the syntax errors was out of date because they had not invoked the incremental compiler (more recent versions of Eclipse have rectified this problem).

5.3. LIMITATIONS

5.3.1. MEASUREMENT ERROR

Many of the findings were based on subjective interpretations of developers' behaviors, so it is important to characterize the sources of error in the measurements and their impact on the findings. All of the data reported in this chapter was based on the transcription of developers' actions. To assess the error rate in this transcription, three task sequences from different developers were randomly sampled and compared to the videos to look for events that may have been missed in the original transcription. This revealed 3 omitted dependency navigation actions and 3 transcription errors out of 108 actions. Therefore, one estimate of the error rate in the data would be about 5%. In general, these errors were not due to disagreements about whether an action had occurred or what type of action it was, but rather to the high level of difficulty of coding particular actions. For example, it was easy to identify application switching and code editing, but more difficult to identify dependency navigations.

This error rate affects a number of results. The statistics in Figure 5.3 would likely be impacted. For example, if 5% of dependency navigations were missed, there

would probably be an average of 68 navigations per participant instead of 65. This would then impact the estimate of the time spent reading code, which was generally the default category when no other actions were observed. The transcription error rate also impacts the proportion estimates of kinds of navigations (Table 5.4), and likely increases the number of navigations that were coded as returns, which would increase the amount of time estimated that developers spent doing visual searches and scrolling. Because the transcription errors were omissions, most of the raw numbers reported would simply increase, so the interpretations remain the same.

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

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

5.3.2. EXTERNAL VALIDITY

Because this was a lab study, there are obvious limitations on this study's external validity. First, the size and complexity of *Paint* program is not representative of most heavily maintained software systems. This may have led to observations of work strategies that are dramatically different from those when developers face hundreds of thousands of lines of code, rather than hundreds. It is likely that the general strategies that developers used in this study would still be present in these situations, but particular activities, such as searching for relevant code and navigating dependencies, might require different tools and occur at different time scales (for example, developers would probably not use the package explorer to navigate amongst thousands of items). This is partially confirmed by some of the findings of Robillard, Coelho and Murphy's study of the medium-sized *JEdit*

application [Robillard 2004], in which successful developers used Eclipse tools in much more systematic, goal-driven ways, rather than browsing.

In addition to the program itself, the tasks in this chapter's study, which can be thought of as "quick fix" maintenance tasks, may differ from other types of tasks, such as impact analysis and reverse engineering. The tasks in this study were also GUI-related tasks and it is unknown whether developers strategies' differ for less visual programs. The lack of comments in the source code in this study is another limitation, especially given the importance of information cues suggested by the results in section 5.2.4. Had these cues existed, developers may have performed more successful searches and less navigation.

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

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

The time constraints imposed were also somewhat artificial. For example, there may have been little incentive for developers to deeply investigate any problem on the web or with a debugger because they may have felt pressured to complete *all* of the tasks in the 70 minutes. Furthermore, they may have felt unable to leave their work and focus on some other non-development task, such as learning about the Swing API for future use; this may have impacted their problem solving efficacy, given evidence that time away from difficult problems can help people change their mindsets and conceive of new solutions [Anderson 2000]. The interruptions in the study were also artificial in several ways. No interruption was more or less *useful* to acknowledge than another; in reality, some interruptions are beneficial [Gonzalez 2004]. Furthermore, no interruption was more or less valuable socially; interruptions by friends and family may have caused developers to acknowledge interruptions without first externalizing important task state, possibly leading to errors. Finally, all of the interruptions took a similar amount of time; in the real world, some interruptions can be hours or days long.

The limitation of screen resolution to 1024 x 768 could have been the source of many of the interactive bottlenecks that were observed in the data. It is possible that with a larger screen resolution, many of these effects would disappear, or be lessened (certain kinds of cognitive limitations are known to disappear at larger screen sizes [Tan 2006]). However, while more space would leave more room for file tabs and result in fewer off-screen code fragments, this could have easily introduced issues with screen real estate *management*, replacing one interactive bottleneck with another, because of the single-windowed nature of Eclipse 2.0 and 3.0.

5.4. IMPLICATIONS FOR THEORY

The findings about developers' search strategies, the importance of the user interface in perceptions of relevance, the frequency with which developers returned to certain fragments of code, and the variation in developers' task contexts call for a model of program understanding based on information seeking. The one proposed here describes program understanding as a process of *searching*, *relating*, and *collecting* relevant information, by perceiving relevance cues in the programming environment.

To help explain this model, consider a program and its metadata such as comments and documentation as a graph consisting of individual fragments of information as nodes, and all possible relationships between information (*calls*, *uses*, *declares*, *defines*, etc.) as edges. In this representation, the code relevant to a particular task will be one of many possible subgraphs of this graph, with the particular subgraph for a developer depending on the implementation strategy, the developer's experience and expertise, and other factors. Using this representation, it is possible to think of a developer's program understanding process as described in Figure 5.6.

A developer begins a task by looking for a node in the graph that seems relevant (*searching*). To do so, they use cues throughout the development environment, such as identifier names, comments, and documentation, to form perceptions about the relevance of information. Once a developer has found what is perceived to be a relevant node, the developer attempts to understand the node by relating it to dependent nodes (*relating*). Because each node in the graph could have a vast number of related nodes, the developer uses cues in the programming environment to determine which relationship seems most relevant. After choosing and navigating a relationship, the developer may investigate nodes related to the new node, and so on, or return to a previous node. If at any point in this cycle of relating, the developer believes there are no more relevant cues, the developer drops out of the relating cycle and goes back to searching for a new node to comprehend. As this searching and relating continues, the developer gathers any nodes that seem necessary for completing the task, whether for editing, reference, or other purposes (*collecting*). If at any point the developer believes that the nodes that have been collected are sufficient to implement a solution for the task, the developer drops out of this understanding process altogether, and focuses on the information collected to implement a solution. Problems during this implementation process then may lead to further search, relate, and collect activities.

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

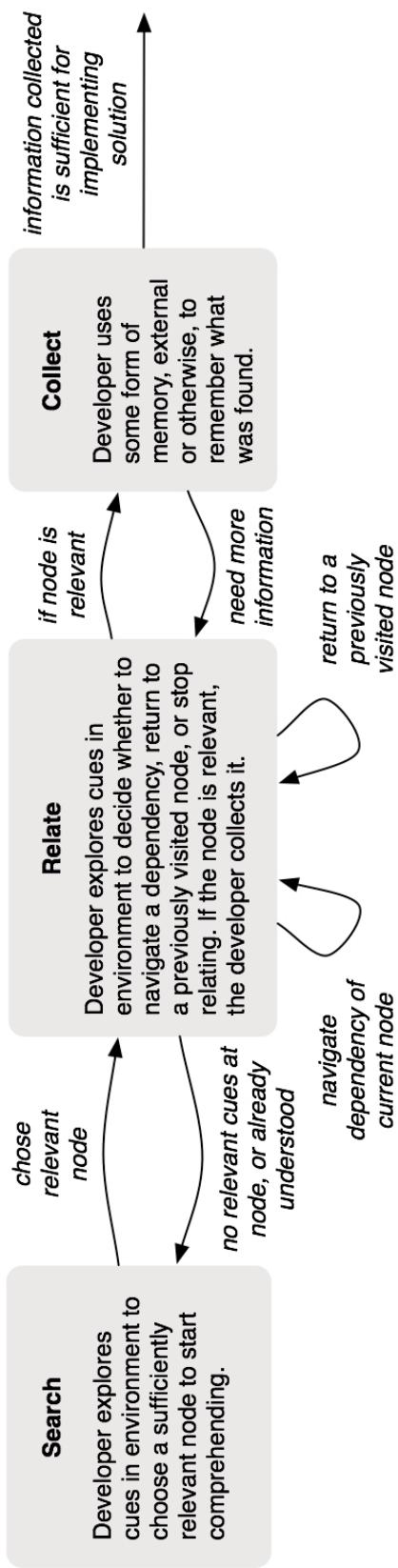


Figure 5.6. A model of program understanding in which developers search for relevant information and relate it to other relevant information while collecting information necessary for eventually implementing a solution.

This model of program understanding is directly informed by *information foraging* theory [Pirolli 1999], which posits that people adapt their strategies and environment to maximize gains of valuable information per unit cost. It proposes that a central mechanism of this adaptation is *information scent*: the imperfect "perception of the value, cost, or access path of information sources obtained from proximal cues." In general, these cues include artifacts such as hyperlinks on a web page or graphical icons in a toolbar. In software development environments, they include the names of program elements, comments, the source file names, and so on. Information foraging theory may suggest more rigorous explanations of how developers might form their perceptions of relevance, so future work should further investigate its relationship to this model.

With regard to existing models of program understanding, the model proposed here is largely consistent with their predictions; the difference is that the model suggests a lower-level explanation of developers' actions than prior work. For example, many models have argued that developers begin with questions and form hypotheses [Brooks 1972, Vans 1999, LaToza 2007]; this corresponds to the *searching* part of this model, in which developers ask "What is relevant?" and use cues to both form and test hypotheses about what is relevant. Other models have focused on high-level strategic differences, such as whether developers understand programs from the top down or bottom up [Corritore 2001, Littmann 1986, Vans 1999], and whether they use systematic or as-needed strategies [Koenemann 1991]; recent work on these issues tend to suggest that developers do all of these [Robillard 2004]. Under the model here, a top-down strategy involves choosing a high-level node and following more specific dependencies; a bottom-up strategy is just the reverse. An as-needed strategy might involve many short paths through this graph, whereas a systematic strategy would likely involve longer and more consistent paths. The model presented here allows for all of these possibilities, and predicts that the particular strategy chosen depends on the cues provided in the environment. Models of knowledge formation during program understanding [Navarro-Prieto 2001, Wiedenbeck 1993], which have suggested that a developer's mental model consists of relationships between code elements, and the purpose and intent of these elements, is consistent with the description of knowledge as the combination of paths that a developer has traversed in a program over time and their existing knowledge. Finally, because the model describes a pattern of activity that is fundamentally driven by cues offered by the environment and developers'

perceptions of their relevance, it is also consistent with research on the influence of the visual representation of code on program understanding [Baecker 1990, Green 1996, Miara 1983, Teasley 1994].

5.5. IMPLICATIONS FOR TOOLS

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

5.5.1. HELPING DEVELOPERS SEARCH MORE EFFECTIVELY

Much of the navigational overhead in our study was caused by developers' use of inadequate or misrepresentative cues in the development environment to guide searches (Section 5.2.4). One approach to alleviating this is to provide better relevance cues. Some studies suggest that the most important information in understanding code is its purpose and intent [LaToza 2006]. For example, a method named `paintComponent` likely does something close to what it describes. Although our data suggests that names are an important way to indicate purpose, names can also be misleading, causing developers to form invalid perceptions of relevance. Comments are a common means of conveying intent, and perhaps if written well and kept up to date, would be more indicative of purpose and intent. Imagine, for example, if the Eclipse package explorer annotated each file icon with a brief description of its purpose, extracted from `Javadoc` documentation. Another idea would be to annotate the icons with the number of other files using the code in the file to help a developer know how "important" the code is to a project, much like ranking is computed in search engines. Future work should investigate other types of information that correlate with the relevance of information. For example,

TeamTracks helps developers explore code that other developers found relevant in their tasks [DeLine 2005].

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

Tools such as Hipikat have tried to automatically find potentially relevant code for developers to inspect [Cubranic 2000]. However, because the recommendations are based on a developers' investigation activities, or their current location in the code, when the developer is investigating irrelevant code, these tools may recommend more *irrelevant* code. Furthermore, the relevance cues in these recommendations can be misleading, since these systems present the names of program elements. Recommender systems such as these should be further studied in order to understand their impact on developers' perceptions of relevance.

5.5.2. HELPING DEVELOPERS RELATE INFORMATION MORE EFFICIENTLY

Once developers found relevant code, our observations suggest that they began navigating its dependencies, using relevance cues in the programming environment to decide whether to continue investigating, and if so, what dependency to navigate. One reason that developers did not use Eclipse navigation commands to perform these navigations is the overhead that they incurred by opening new tabs and requiring a return navigation. One way to reduce this overhead is to make dependency navigations more provisional. For example, nearly one-fifth of developers' time was spent reading code within a fixed view in the Eclipse editor (Section 5.2.1), so it could be helpful to highlight dependencies in the code automatically based on the current text caret position or text selection (Eclipse does have basic support for this, but it must be invoked). Developers also spent a lot of time going back and forth between related code (Section 5.2.6), so interaction techniques that allow developers to *glance* at related code could be helpful. Tools such as FEAT [Robillard 2002] might be a good place to start, by replacing the context menus used to inspect dependencies with something requiring fewer steps.

5.5.3. HELPING DEVELOPERS COLLECT INFORMATION MORE RELIABLY

Collecting information was central to developers' success, but developers currently have a limited number of ways to do it. Each has its own flaws: memorizing information tends to be unreliable [Altmann 2001]; writing it down is archial in some sense, but slow, imprecise, and requires developers to re-navigate to relevant code; and finally, using the interactive state in Eclipse was precise, but unreliable (Section 5.2.7). None of these approaches help developers compare information side-by-side, which our study suggests was quite important (Section 5.2.6).

The mockup in Figure 5.7 illustrates one way that these fragments could be collected and viewed. In this hypothetical situation, the developer has already found all of the code thought relevant, and has just copied the `rSlider` setup code in order to create code to add a thickness slider. The developer is in the middle of changing `rSlider` to `tSlider` in the duplicated code. The basic concept of the workspace is to provide a single place for developers to view all relevant information for a single maintenance task side-by-side, in order to eliminate much of the navigational overhead observed in our study. The rest of this section describes the features portrayed in our conceptual workspace.

One issue with collection tools is how the workspace refers to code and metadata internally. For example, some tools have used a virtual file, which allows developers to combine a set of line ranges [Reiss 1996] or methods [Chu-Carroll 2002], and edit it as if it were a single file. Robillard proposed the concept of a concern graph [Robillard 2003b], which represents information as a set of relationships between program elements (such as declares and subclasses relationships). This approach is more robust to changes to a program, but the tradeoff is that arbitrary subparts of the smallest granularity element (such as parts of a method) cannot be referenced

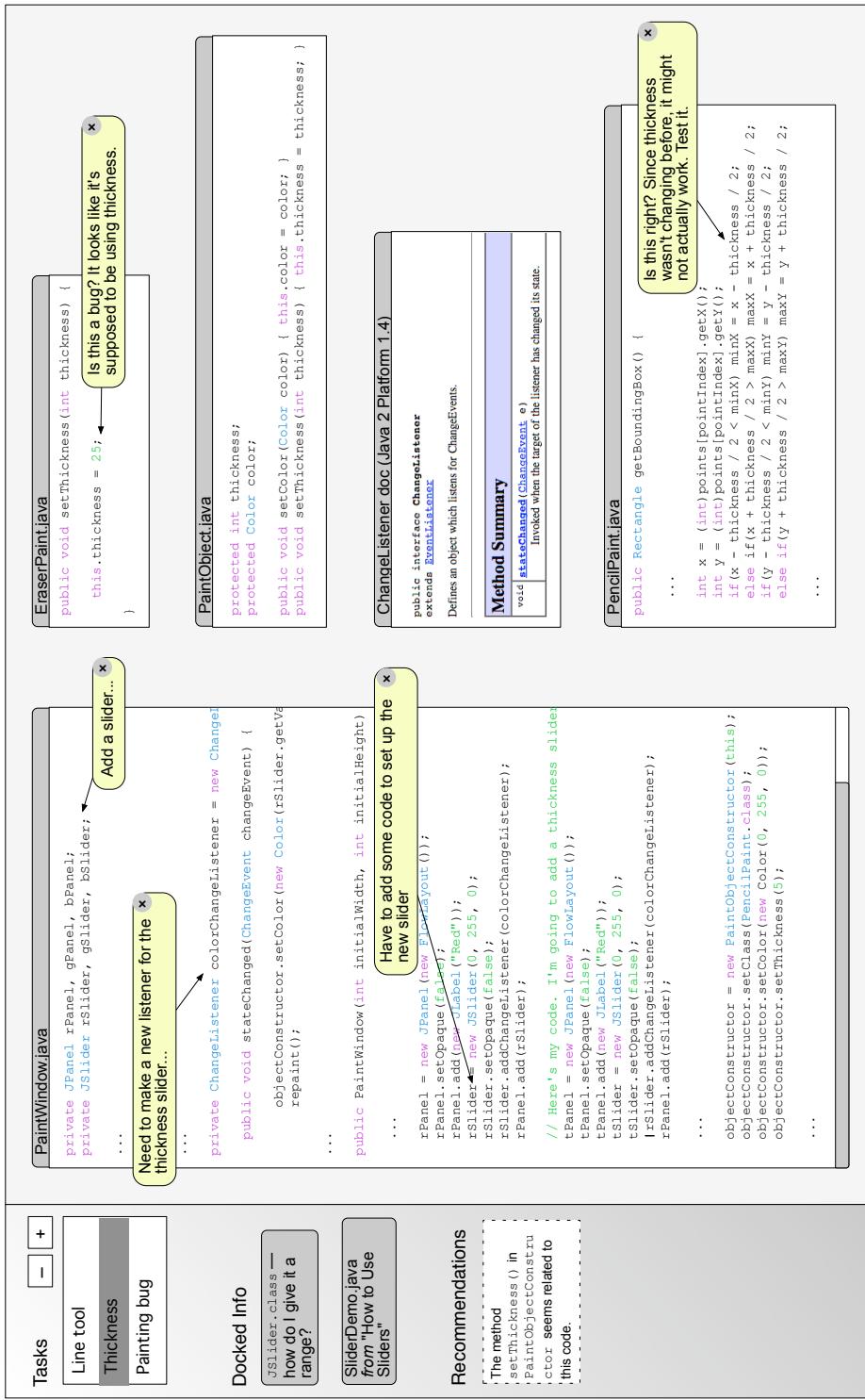


Figure 5.7. The 50 lines of code and other information that developer B indicated as relevant, portrayed in a mockup of a workspace that help developers collect relevant information for a task in one place, independent of the structure of a program.

(FEAT can reference the *calls* and *uses* of a method, but not arbitrary lines of code). The representation envisioned in the workspace in Figure 5.7 would involve regions of code that one could imagine a developer circling on a code printout; this approach might better match the granularity in which developers think about code.

The interaction technique that developers might use to add information to this workspace, once found, largely depends on the representation used to refer to the information (if information is collected as lines of code, for example, the interaction technique must allow developers to select lines). However, it must also be simple enough that developers can quickly specify the relevant information and continue with their task, uninhibited. Tools proposed in the past have tended to be heavyweight. For example, Desert [Reiss 1996] requires users to know the line numbers beforehand and enter them manually. FEAT [Robillard 2002] requires users to navigate a program by its relationships, and add "relations" to a concern graph through a contextual menu, which is a somewhat oblique way for a developer to say "this code is important to me." To add information to our workspace, developers could use a keyboard shortcut to add a single line, and possibly a gesturing technique with the mouse to circle the relevant code and its surrounding context. This would be quite similar to the snapshots of code that developers claim they see when trying to recall the shape and location of familiar code [Petre 1997].

Given that nearly all of the navigational overhead in the study was due to the way code and information was organized on-screen, the visual representation of information in collection tools is also an important issue. Desert [Reiss 1996] presents relevant lines of code as a single integrated file; FEAT [Robillard 2002] represents concern graphs as a hierarchical tree and requires developers to select an element in this hierarchy, and use a pop up menu to request the source file; Concern Highlight [Nistor 2006] highlights relevance code in a conventional editor. There are several problems with these approaches: (1) they treat all information as if it had the same role in the task; (2) they do not support side-by-side comparison of information; and (3) they incur much of the same interactive overhead observed in relying on file tabs and scroll bars. Our proposal in Figure 5.7 represents the code and information that a developer deems relevant concretely, rather than using abstract icons or names. Not only does this avoid the navigational overhead of navigating to the information, but it affords other advantages: code can be placed side-by-side in order to aid comparison and editing; views could be collapsed to the bar on the left of Figure 5.7, in order to allow developers to focus on the subset of

information that is necessary for the current task; and code and other information could be directly annotated, as seen throughout Figure 5.7.

Of course, there are some limitations to representing code concretely, rather than summarizing it and allowing users to navigate to it. One obvious concern is whether such a workspace would be able to fit all of a developer's relevant information on a single screen. To consider a lower bound, developers in our study found about 30 lines of code relevant on average. To consider an upper bound, a study of the CVS repository of GNOME, which is over a million lines of code, found that the average check-in was about 28 lines of code, with a standard deviation of 38 and a maximum of 237 [Koch 2000]. This suggests that an average task, even with a few lines of surrounding context for contiguous fragment, would be likely to fit inside a full-screen window.

I worked with student Michael Coblenz to explore the concept in Figure 5.7 in more detail, designing the Jasper environment, which allows Eclipse developers to gather fragments of Java code relevant to their current task [Coblenz 2006]. An example of this is shown in Figure 5.8. The environment allows developers to select ranges of Java code from the Eclipse Java code editor, as well as documentation from a web page, and combine these selections into a single view. This allows a developer to save the task context for later, in case the task is interrupted, and also share the task context with other developers who may be collaborating on the task. The environment automatically sizes and lays out the selections, so that developers can focus on the information, rather than its presentation.

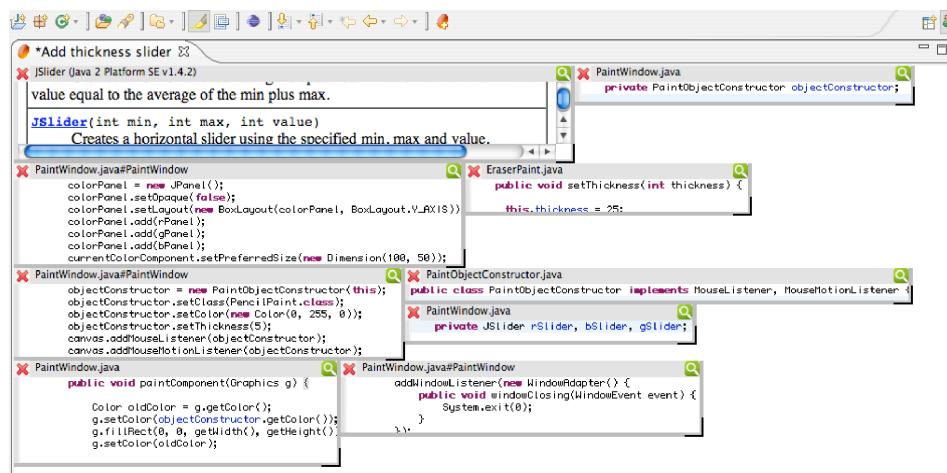


Figure 5.8. Jasper, an Eclipse plug-in that allows developers to gather arbitrary fragments of Java code in a single view.

One challenge of implementing the workspace was how to reference code fragments. As code changes, Jasper must detect these changes and decide how to re-anchor the selected code given the new changes, or whether to simply notify the developer about unanchored code. Jasper stores code references as a project name, path within the project to a file, and a line number range within the file. Jasper also stores the last known text of the item. If a file changes, Jasper searches for the code and scores each new line in the file if it matches the last known text. If more than half of the text was found based on this scoring mechanism, the code is reanchored; otherwise, the user is alerted about the problem, giving feedback about the change. More details on the exact algorithms used are available in [Coblenz 2006].

5.6. SUMMARY

The goal of this study was to investigate the program understanding strategies of developers with more experience than in the studies in Chapters 3 and 4. There were a number of findings:

- Developers generally form hypothetical explanations of program execution and then use a variety of tools to verify or reject their explanations.
- Developers based their guesses about the cause of program execution on surface features of its output, such as labels found in UIs (Section 5.2.3).
- 88% of developers' hypotheses about the causes of a program behavior of were false (Section 5.2.3).
- The consequences of guessing incorrectly caused developers to spend an average of 36% of their time investigating irrelevant code (Section 5.2.3).
- Developers tend to form *task contexts* of relevant code in order to capture the information necessary to find a bug or add a new feature (Section 5.2.7).
- The information in developers task contexts can vary considerably on the same task, probably due to differences in experience and in the actual process of gathering relevant code (Section 5.2.8).
- Information foraging theory can help model the information cues that developers use to guide their search for relevant information (Section 5.4).

All of these findings and ideas contributed to the Whyline concept and design, described in later chapters, as well as inspiring tools like Jasper [Coblenz 2006].

6.

INFORMATION NEEDS AT MICROSOFT¹⁰

The central limitation of the prior studies was essentially that they all involved students. Therefore, the next study explored the daily lives of developers at Microsoft working on various Microsoft products. This study was an attempt to extend the ecological validity of my findings by exploring the same issues in the software development industry.

Rather than approach these observations with a narrow focus on program understanding, the focus was broader, exploring software development work from an *information* perspective. Some studies have investigated information *sources*, such as asking other people [LaToza 2006], looking in code repositories [de Souza 2003], and investigating bug reports [Sandusky 2005]. Others have studied means of *acquiring* information, such as email, instant messages (IM), and informal conversations [Sandusky 2005]. Studies have even characterized developers' strategies [Hertzum 2002], for example, how they decide whom to ask for help.

While these studies provide several concrete insights about aspects of software development work, we still know little about what information developers look for and why they look for it. For example, what information do developers use to triage bugs? What knowledge do developers seek from their coworkers? What are developers looking for when they search source code or use a debugger? By

¹⁰ The results in this chapter appear in part in [Ko 2007].

identifying the *types* of information that developers seek, we might better understand what tools, processes and practices could help them more easily find such information.

To understand these information needs in more detail, a two-month field study of software developers at Microsoft was performed in the summer of 2006, involving 17 groups across the corporation, focusing on three specific questions:

- What information do software developers' seek?
- Where do developers seek this information?
- What prevents them from finding information?

The observations identified several information needs. The most difficult to satisfy were design questions: for example, developers needed to know the *intent* behind existing code and code yet to be written. Other information seeking was deferred because the coworkers who had the knowledge were unavailable. Some information was nearly impossible to find, like bug reproduction steps and the root causes of failures.

The next sections discuss the study's methodology and then detail the information needs that were identified in both qualitative and quantitative terms.

6.1. METHOD

The method was to record notes while observing developers' normal work. To recruit developers, 250 developers from the corporate address book were surveyed. Of these, 55 responded and 49 volunteered for observation.

Each observation session was about 90 minutes and involved a single observer taking handwritten notes. To encourage the participants to narrate their work, they were asked to think of the observer as a newcomer to the team (in Microsoft terms, a "new hire"), doing a "job shadow." The observer focused on recording goal-oriented events like "finding the method that computed the wrong value" rather than low-level events like keystrokes or menu selections. Since the observers shared the participants' programming background, they understood much of the work and where and how information was obtained, without inquiry. In some cases, the observers prompted with questions like "what are you looking for?" to learn their

information needs, but most developers thought aloud without prompting. The observers time-stamped the recorded events and conversations each minute. After 90 minutes, the observers looked for a good stopping point and wrapped up. Immediately after each observation, the observers transcribed the handwritten notes, as in the excerpt shown in Figure 6.1.

9:41 am So this copies the files onto the server, then allocates a machine to do the setup. In the meantime, I'm going to get this local fix [of this other bug] over [checked in].
9:41 am [opens diff tool to see changes he's made to code]
9:43 am Oh damn, I have some mixed changes. Some are part of this DCR [design change request] I'm working on and some are part of a bug fix, so I have to mix it out.

Figure 6.1. An excerpt from J's observation log.

During the allotted time for the study the observers were able to observe 17 developers, which was enough to see common patterns in their information needs. (Section 6.2 touches on the potential value of observing more developers.) Figure 6.2 describes these developers' experience levels, types of work, and phases of development and introduces the initials used to refer to them in this chapter. In Microsoft's terminology, *dev leads* manage software *development engineers (SDEs or devs)* while also performing a development role. In general, the teams developed a variety of applications, some internal components to the company, other end user products, and all developers used widely varying software development processes.

6.2. TASK STRUCTURE

The observations spanned 25 hours of work and resulted in 357 pages of handwritten notes, which were then transcribed into 4,231 time-stamped rows in a spreadsheet. The logged activities were partitioned into work categories common across the participants: *writing code; submitting code (check-ins); triaging bugs; reproducing failures; understanding program behavior; reasoning about design; maintaining awareness; and non-work activities* (e.g. personal phone calls). Causes of task switching were also identified: face-to-face dialogue; phone calls; instant messages (IM); email alerts; meetings; task avoidance; getting blocked; and task completion. The logs were annotated with these switches, based on remarks like "I want to get back to my repro[duction]..." All of these causes of task switches are

forms of interruption, except getting blocked and task completion. When participants voluntarily switched activities, the switch is labeled as *blocked* if they could no longer make progress on the activity (typically due to an information need) and *task avoidance* if they could make progress but chose to switch anyway.

Figure 6.2 visualizes these task switches, which occurred an average of every 5 minutes (± 1.7), mirroring the rate reported in [Gonzales 2005]. Time fragmentation varied considerably per participant. For example, M reproduced failures without interruption, whereas R was frequently blocked. Many interruptions were due to face-to-face, IM, or phone conversations, which occurred from 0 to 6 times per session (median of 1), each lasting for much of the session. Developers were also interrupted by notifications, such as email and alerts about changes to the bug database. Developers experienced 0 to 9 notifications per session (median of 1).

Blocking, shown in Figure 6.2 as dark vertical bars, occurred when information was unavailable. Some blocks were about waiting for the results of compilations and tests suites. Developers also waited for email replies and for other teams to submit changes or bugs. Other blocks were due to missing knowledge, like when a developer stops coding to learn about an API. Developers were blocked a median of 4 times per session and between 0 and 11 times overall.

6.3. INFORMATION NEEDS

Of the 4,231 rows in the spreadsheet, there were 334 instances of information seeking events, which were then abstracted from the particulars of the work context into 21 general information needs. This section presents these information needs clustered by the work category in which they arose. Throughout, the initials of the developers for whom we observed a trend are listed within braces.

6.3.1. WRITING CODE

Developers had several questions situated in the code they were writing:

- (c1) What data structures or functions can be used to implement this behavior?
- (c2) How do I use this data structure or function?
- (c3) How can I coordinate this code with this other data structure or function?

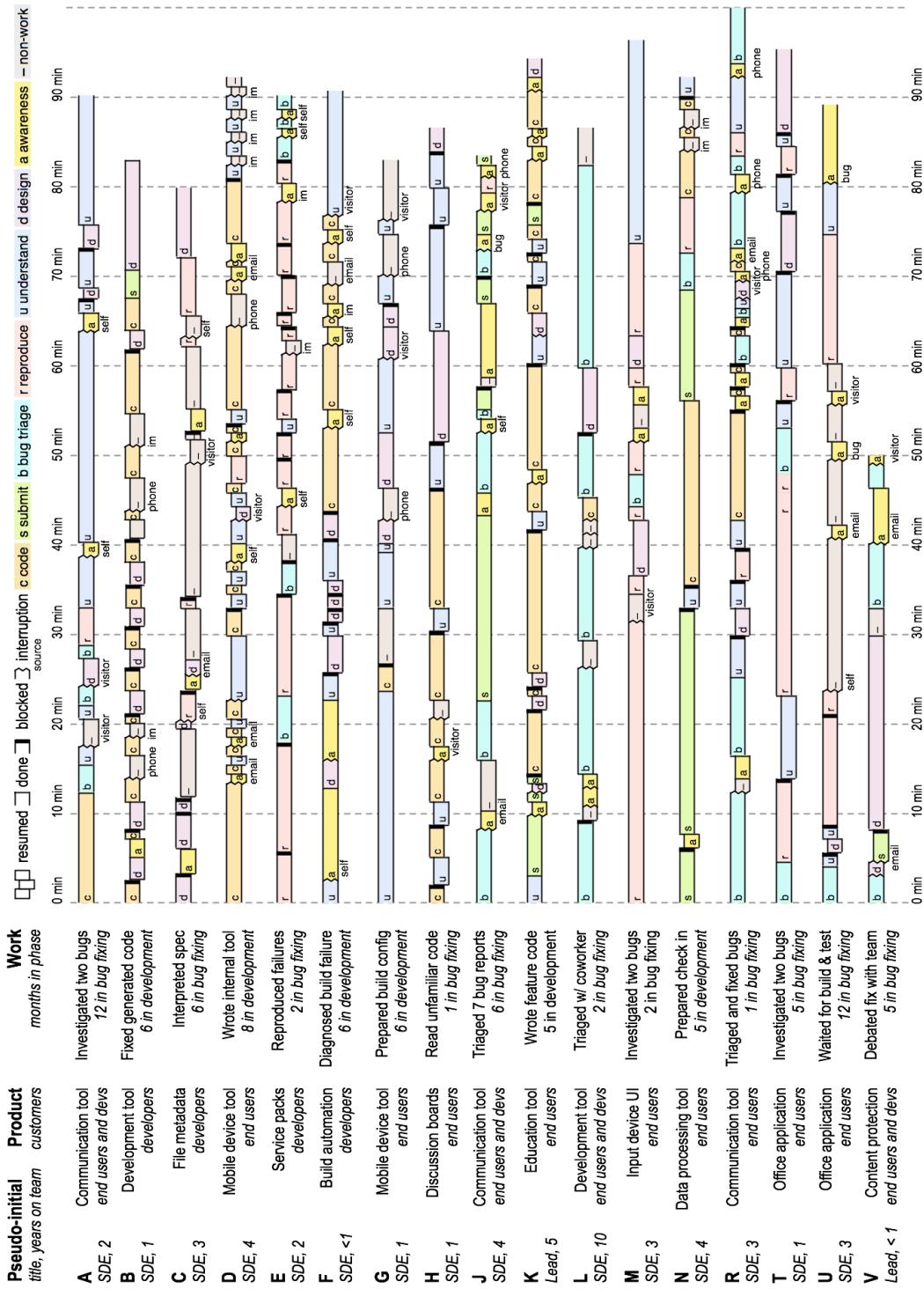


Figure 6.2. The backgrounds and task structures of the 17 observed developers. The right edge of each task block indicates the reason for the task switch (Tasks are labeled time blocks with coded ends (thin line for done, thick line for blocked, jagged line for interrupted). When a task is interrupted or blocked, its fragments are drawn at the same vertical level to show resumption.

Although the first of these questions (c1) was uncommon, when it occurred, developers searched API documentation {K} and inspected other code for examples {H}. These searches can be thought of as a search through the space of existing reusable code; for example, K looked for an appropriate serialization API by searching a large database of public documentation.

Once a developer had a candidate in mind, they sought its *syntactic* usage rules (c2). For example, which method is appropriate to call? What data structures does this require? What constructors does this class have? Developers used documentation when it was available {DFHJKN}, but sometimes needed to use code that was only fully understood by its author {AFL}. Others found example code from which to infer rules {GH}.

Because developers had to coordinate APIs with their own code, they also sought *behavioral* usage rules (c3), implicit in the API design. For example, is it legal to call this method after calling this other method? What state do I have to be in before this call? Such information was rarely explicit. Developers used their colleagues {A}, documentation {K}, and example code {HN} to infer these rules.

6.3.2. SUBMITTING A CHANGE

Developers had three primary questions when exposing their code to their teammates:

- (s1) Did I make any mistakes in my new code?
- (s2) Did I follow my team's conventions?
- (s3) Which changes are part of this submission?

Besides building their code to assess its *syntactic* correctness, developers answered questions of correctness (s1) by considering the scenarios and range of input that they intended to cover. They used debuggers {DKR}, diff tools {R} and unit tests {BN}, but primarily relied on their own reasoning {ADHJ}. Another common filter for mistakes was code reviews. Before a review, developers first looked for mistakes:

K: I think I'm ready to check in, so I'm just making sure I didn't do anything stupid. Like, I forgot to write those tests! Yeah, stupid like that.

One developer wrote assertions {N}, but these interfered with other developers' work {ATU}, even when the assertions were correct:

A: I never want to see [that product's] asserts but they always pop up. They have nothing to do with my work!

Three developers used static analysis tools to check for fault-prone design patterns {JKU}, but expressed disdain for such tools' false positives or could not understand the tools' recommendations.

Developers also considered their team's conventions (s2). Some teams required tags or other documentation to be embedded in method headers, which developers were careful to remember, often with the help of tools {BELN}. Sometimes two submissions intersected (like in the transcript in Figure 6.1) or developers had to merge their code with another's and developers had to determine which differences were part of the current submission (s3) {JN}.

6.3.3. TRIAGING BUGS

Most developers were swamped with bug reports from tests, customers, and internal employees. Triage occurred in isolation as a developer partitioned their time {AEJMNTUV}, but also in triage meetings {LR}. For each report, the goal was to determine:

- (b1) Is this a legitimate problem?
- (b2) How difficult will this problem be to fix?
- (b3) Is the problem worth fixing?

Assessing legitimacy (b1) involved deciding whether a failure was due to a problem with the code or an unrealistic configuration of a test {BL}:

B: It might but not really be a failure. It might just be a setup problem. This particular component doesn't depend on anything. Probably locked a file, so it's returning an exit code. Not a real failure.

Legitimacy also depended on whether a report was a duplicate. People reported similar failure symptoms {L}, but also different failure symptoms that developers believed had a common cause {LT}:

A: These subjects are just busted! I have a feeling I'm seeing the same bug. I'm going to do a quick search to see if there are busted subjects [in the bug database]—this one kinda sounds like it, blah blah, category name is corrupted? Ooh, screenshots are the same!

Bug triage is a cost/benefit analysis. To assess the cost of repair (b2), developers considered whether a redesign would be necessary {CJTV}, whether other teams might be affected {V}, and whether a fix could be written and tested by a deadline {V}. Teams also close bugs “by design,” treating them as work items for later releases:

V's teammate: I think the best thing is a new overlay to indicate something's going on.

V: We can't do that by the release. Looks like a work item.

Another factor affecting the repair cost was a reports' clarity {JLR}. Does it have detailed reproduction steps? Is the failure clearly described? Does it have hints about possible causes or an error message? Reports with inadequate clarity were rejected {R}.

On the benefit side of the analysis (b3), developers considered the number of users affected {CJLV} and the user experience {LRV}. For example, V discussed a fix:

V's teammate: If we want to push it back, we can, but I think an overlay is easiest.

V: But it's a totally broken experience for the user.

If there was a known workaround, developers might focus on more severe bugs {LV}.

6.3.4. REPRODUCING A FAILURE

To reproduce a failure, developers asked:

- (r1) What does the failure look like?
- (r2) In what situations does this failure occur?

The primary source for both of these types of information was bug reports. Reports would often include screen shots {MT}, but more often developers relied on the descriptions of the failure to help them imagine its appearance {AELNRTU}.

Developers relied heavily on the bug report's reproduction steps to understand the situations in which a failure occurred (r2). Given the complex configurations that were necessary to reproduce some problems, even detailed steps omitted crucial state {ERT}. In other cases, the state was known, but difficult to reproduce {AT}:

A: Originally, the repro steps said I need a blog count [as a test case] but I couldn't set one up, so I went back and forth.

To overcome this, some developers set up a remote desktop connection with the report's author, so that the full configuration was available for debugging {EL}. Developers would also guess what state was wrong and begin modifying their environment and test cases until reproducing the failure:

A: I'm looking at [the report] to see if I have this configured the same way, but I'm not getting the problem. Maybe we've changed it in the past half year this has been open.

In one situation a failure could not be reproduced and the bug had to be deferred {A}. The developer documented his attempts in the report for the sake of other testers and developers.

6.3.5. UNDERSTANDING EXECUTION BEHAVIOR

Developers had to understand unfamiliar code in several circumstances: using vendor code {GM}; joining a new team {V}; obtaining ownership of code {H}; during workload balancing {T}; or when debugging, with unfamiliar code on the call stack {ENT}. Each time, they addressed three basic questions:

- (u1) What code could have caused this behavior?
- (u2) What is statically related to this code?
- (u3) What code caused this program state?

Developers began these tasks with a “why” question and a hypothesis about the cause of the failure:

A: Why did I get gibberish? Storing field, given PPack, what is an MPField? I have no idea what this data structure contains. SPSField? I suspect SPS is just busted.

Developers acquired their hypotheses (u1) by using their intuition {ALM}, asking coworkers for opinions {AFM}, looking at execution logs {F}, scouring bug reports for hints {ER}, and using the debugger {GTU}. Although developers used many sources to obtain hypotheses, only a few gathered and considered more than one at a time {FM} (it is an open question as to whether this leads to more success). The accuracy of developers' hypotheses was only obvious to them in hindsight (the methodology of this study is not suited to knowing just how accurate they were).

To test and refine hypotheses, developers asked a broad array of questions with a variety of tools. Many of these questions were about the structure of the code (u2), like *what is the definition of this?* and *what calls this method?* Such questions were easy to answer with tools. Other, more broadly scoped questions, like *what code does a similar operation?* has no tool support, but developers were good at answering them with search tools.

Developers answered questions about causality (u3) such as *where did this value come from?* {ATU} and *how did the program arrive at this method?* {AM} by a series of lower level questions, such as *what thread is the program in right now?* {AT} *what is the value of this variable or data structure now?* {AEMTU}. Sillito, Murphy and De Volder report similar indirect questioning [Sillito 2006]. These questions were primarily answered with breakpoint debuggers, which required developers to translate their questions into an awkward series of actions:

A: Here we're formatting WSTValue...I can't do highlighting, so I go to Source Insight. Find where I am in devns—this is the guy [the code] that screwed up. Shift F8, highlight all occurrences, where it gets its value from. Here's where we set it. So I want a breakpoint here.

As developers refined their hypotheses, they changed their concern from the behavior of the *existing* system to the *hypothetical* behavior after some change:

T: There's no file there, so something forgot it and I have a suspicion of what it is. Might mean that the free code has to get moved later.

Intuition was essential in answering all of these questions. The cost of testing hypotheses and the risk of a false hypothesis often prevented developers from

finding a root cause. Instead, developers frequently assessed the value in continuing their investigation, stopping when they were satisfied {ATU}.

6.3.6. REASONING ABOUT DESIGN

Developers sought four kinds of design knowledge:

- (d1) What is the purpose of this code?
- (d2) What is the program supposed to do?
- (d3) Why was this code implemented this way?
- (d4) What are the implications of this change?

The purpose of code (d1) was often unclear when developers found an API to use: Is it a public artifact or intended only for a particular component? Is it regularly maintained or no longer used? Some developers inferred purpose by finding example API uses {GHK}; sometimes they directly asked the code's author {T}.

Developers needed to know what the program was *supposed* to do (d2), for example, to evaluate the correctness of a variable's value {ABCDFMR}:

D, yelling across the hall: Is 'B' not a legal license key letter?

Sometimes this assessment was obvious. For example, a crash in a basic use case must be unintended. In other cases, what a program was supposed to do was an explicit, documented decision:

M: I just want to double check and make sure the convert key only shows up in languages that it's supposed to, based on the spec.

It was rarely sufficient to understand the cause of a program behavior. Developers also needed to know the historical reason for its current implementation (d3) {AEHRTV}. For example, when assessing whether a variable's value was "wrong," developers had to consider whether the value was anticipated by the designer and explicitly ignored or whether it was overlooked. They would do this by investigating the code's change history {AT} or by looking for bug reports that contained hints about its current design {ET}. Developers would seek this design rationale from the author of the code through face-to-face conversation or some other means {TV}, but in one case the author was unavailable {T}. Even when developers found a person to

ask, identifying the information that they sought was hard to express, as developers struggled to translate detailed and complex runtime scenarios into words and diagrams.

The *consequences* of decisions were also important (d4). For example, when triaging, developers often discussed hypothetical scenarios {FHKLRV}:

V's teammate: Let's go ahead and block and make it into a single operation.

V: But the upgrade script needs to look for individualization.

Design knowledge of all types was scattered among design documents {M}, bug reports {AHV}, and personal notebooks {AHMT}. Email threads sometimes contained design rationale {CFJ}, but were not shared globally. Code comments sometimes contained design rationale {H}, but developers hesitated to write them because of the cost of submitting code changes. Developers rarely searched these sources, because such sources were thought to be inaccurate and out of date:

H: Given that I'll be the one fixing the bugs, I need to make sure I know, not what we are doing, but why we are doing it. We have these big long design meetings, and everybody states their ideas, and we come to a consensus, but what never gets written in the spec is why we decided on that. Keeping track of that is really hard.

These problems led all but two developers to defer decisions because of missing design knowledge.

6.3.7. MAINTAINING AWARENESS

Developers worked to keep track of hardware, people and information needed for their tasks:

- (a1) How have resources I depend on changed?
- (a2) What have my coworkers been doing?
- (a3) What information was relevant to my task?

Some awareness information was “pushed” to developers through IM clients and alert tools {BDEFLMN}, and through check-in emails {CFJ}. Developers obtained other types of awareness by actively seeking it. One group had brief meetings throughout the day, to keep aware of problems that teammates were working on and

issues on which they were blocked {M}; other groups had weekly meetings to keep awareness about triage and design choices. Developers would stop by coworkers' offices to update them on problems or to see what problems they were facing {AFGHJKLMRT}:

F: I talked to [coworker] a bit about the execution, and `gatherObjects()` is on track, but I still need to make the base class.

F's boss: Yeah, [coworker] talked to me about it. We need to make sure files are not delay assigned. He's in this big whoop-de-doo about it.

Developers tracked their time and others', checking their calendars, glancing at schedules and asking their managers about priorities {BCK}. Managers communicated to their developers about upcoming changes in informal meetings, email announcements, or planning meetings {FL}. Because developers were often interrupted, they also sought awareness about their own work (a3):

G: Sometimes I have like 20 windows, 5 or 6 build windows, each one is a state that I'm working on and I lose it! If I could just save it...I would be really happy! I hate those midnight reboots.

6.4. QUANTIFYING INFORMATION NEEDS

The information needs discussed in section 6.3 are summarized in Figure 6.3. The *time spent searching*, *search frequencies*, *search outcomes*, and *source frequencies* are based on the observational data gathered. The outcomes include when developers *acquired* information, *deferred* a search with the intent of resuming it, or *gave up* with no intent of resuming it; a few searches continued beyond the observations. Also, in two cases, a need was initially deferred, then satisfied afterward by a coworker's email response; these were coded as *acquired*.

The most frequently sought and acquired information includes whether any mistakes (syntax or otherwise) were made in code and what a developers' coworkers have been doing. The most often deferred information was the cause of a particular program state and the situations in which a failure occurs. Developers rarely gave up searching. There was no relationship between deferring a search and whether the source involved people (bug reports, face-to-face, IM, email) versus data sources (code, version control, etc.) ($\chi^2(1)=.6$, $p > .05$).

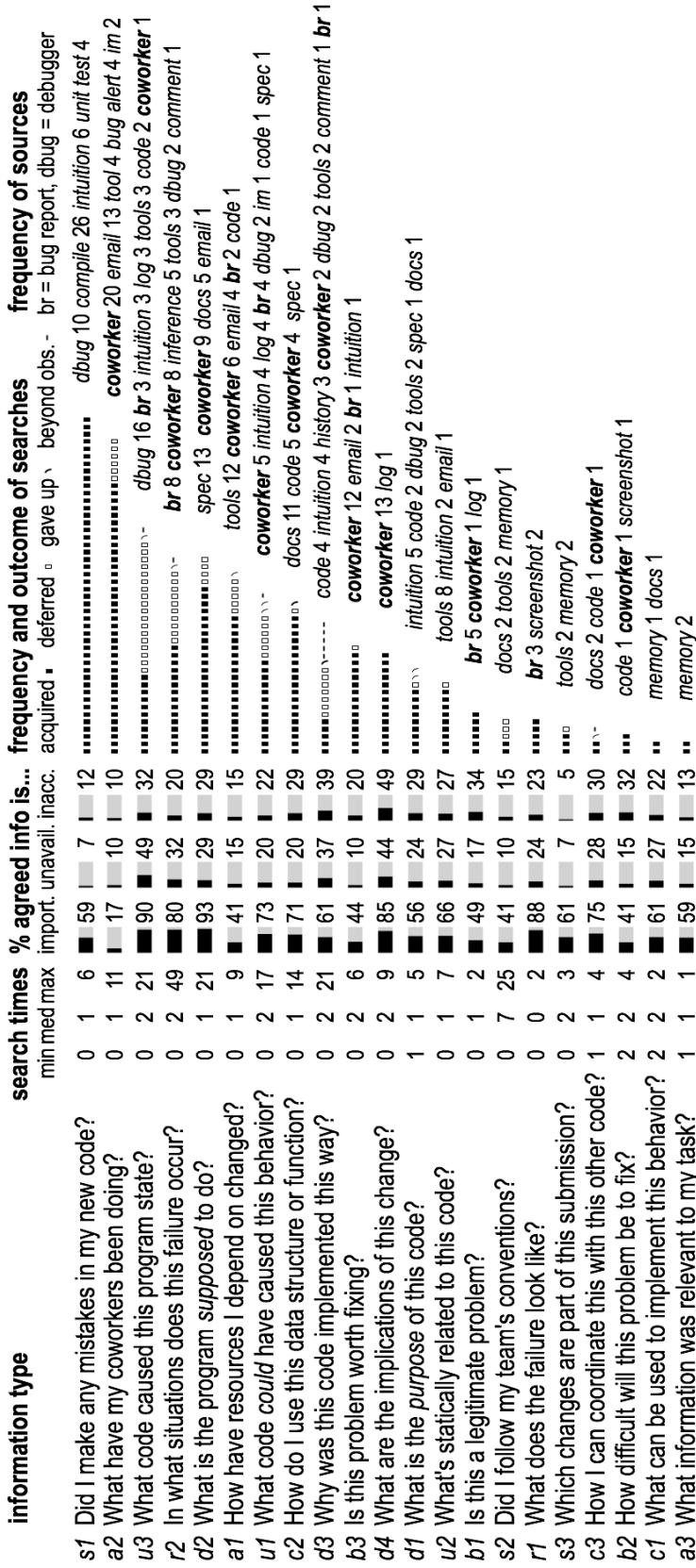


Figure 6.3. Types of information developers sought, with search times in minutes; perceptions of the information's importance, availability, and inaccuracy; frequencies and outcomes of searches; and sources, with the most common in boldface.

Based on the medians in Figure 6.3, the information that took the longest to acquire was whether conventions were followed (s2); based on maximums, the longest to acquire was knowledge about design (d2, d3) and behavior (u1, u3). No one source of information consistently took longer to acquire than another ($F(17, 321)=.53, p>.05$), nor was there a consistent difference in search times between sources involving people and sources that did not ($F(1, 339)=.07, p>.05$). These times are misleading, however, as many of the maximums were on deferred searches, so they were likely longer than shown here. Further, developers gave up or deferred searches because they depended on a person known to be unavailable. They were also expert at assessing the likelihood of the search succeeding and would abandon a search if the information was not important enough.

6.5. MOST COMMON INFORMATION NEEDS

Figure 6.3 sorts the needs by frequency across all of the observations. Figure 6.4 shows a different sorting, by the number of participants for whom a need was observed. The most common need across participants was coworker awareness. Most of the information needs occurred among several developers from different teams in different business divisions, which suggests that these are representative of development work in general. A few of the information needs occurred for only a

a2 What have my coworkers been doing?	15	ABCDEFGHIJKLMNRTU
u3 What code caused this program state?	11	ABDEFGMNRTU
a1 How have resources I depend on changed?	10	ACEFGJKLRT
u1 What code could have caused this behavior?	9	AEGFLMRTU
c3 How do I use this data structure or function?	9	ADFGHJKLN
s1 Did I make any mistakes in my new code?	9	ABDFHJKNR
d2 What is the program supposed to do?	7	ABCDFMR
r2 In what situations does the failure occur?	7	ELMRTUV
b3 Is this problem worth fixing?	7	BCJLRTV
u2 What's statically related to this code?	6	AEHKNT
d3 Why was this code implemented this way?	6	AEHRTV
d4 What are the implications for this change?	6	FHKLRV
r1 What does the failure look like?	5	AMNRT
c3 How can I coordinate this with the other code?	4	AHKN
s2 Did I follow my team's conventions?	4	BELN
d1 What is the purpose of this code?	4	HKT
b2 Is this a legitimate problem?	3	BLR
s3 What changes are part of this submission?	2	JN
b2 How difficult will this problem be to fix?	2	LR
a3 What information was relevant to my task?	1	V

Figure 6.4. Information needs per participant.

few participants, which suggests that this list is not complete. Observing more developers over longer periods of time could reveal other, less frequent, needs. (Had V not been observed, for instance, we would not have seen need a3.)

Many of the frequent information needs are problematic, in that searches for the information were often unsatisfied (deferred or abandoned) and had long search times. The most frequently unsatisfied information needs were the following, with their percentage of unsatisfied queries and maximum observed search times:

u3	What code caused this program state?	61%	21 min
d3	Why was the code implemented this way?	44%	21 min
r2	In what situations does this failure occur?	41%	49 min
u1	What code could have caused this behavior?	36%	17 min
a1	How have the resources I depend on changed?	24%	9 min
d2	What is the program supposed to do?	15%	21 min
a2	What have my coworkers been doing?	14%	11 min

This ranking may reflect that 11 of the 17 participants' teams were in a bug fixing phase. In particular, the information needs ranked 1, 3 and 4 are largely about bug reproduction and the ones ranked 2 and 6 are largely about evaluating possible fixes for bugs. Nonetheless, the fact that these information needs are so often unsatisfied and take such a long time clearly hindered developer productivity.

6.6. RATING INFORMATION NEEDS

A survey was deployed after the observations to investigate how well the results generalized to a larger population of developers. The percentages in the middle of Figure 6.3 come from a survey of 42 different developers (of 550 contacted), asking them to rate their agreement with statements about each of these information types, based on a 7-point scale from strongly disagree to strongly agree. The bars represent the percent of developers who *agreed* or *strongly agreed* that the information was (from left to right) *important to making progress*, *unavailable or difficult to obtain*, and *had questionable accuracy*.

The survey results reveal interesting trends. The majority of developers rated the most frequently sought information in the observations as more important, and they also rated frequently deferred information as more unavailable. One discrepancy is

that developers rated coworker awareness (a2) as relatively unimportant, which conflicts with its frequency in the observations. It may be that coworker awareness is so frequent sought and successfully obtained that developers do not think about it. In the observations, developers successfully obtain knowledge about the implications of a change (d4), whereas in the survey, developers rated it relatively difficult to acquire. The survey also begins to reveal which information types have more questionable accuracy, namely knowledge about design (d2, d4), behavior (u1), and triage (b1, b2).

6.7. DISCUSSION

The motivation for this study was to identify and characterize software developers' information needs. The list, while likely not comprehensive, has several implications.

6.7.1. COWORKERS AS INFORMATION SOURCES

Coworkers were the most frequent source of information, accessed at least once for 13 of the 21 information needs and in 83 of the 334 instances of information seeking. The importance of coworkers as information sources probably explains why coworker awareness is the second most frequent information need. (Developers checked on coworker availability almost as many times as they looked at output from the compiler or debugger.) This is consistent with prior work on awareness in software development, with regard to sources, strategies, and frequency of information seeking [de Souza 2003][Hertzum 2002][Perry 1994].

Why should developers turn so often to coworkers? One possibility is the topics being discussed. Outside of awareness, the information needs where coworkers were most often consulted were either about design, i.e.

- *d4: What are the implications of this change?* (13 times)
- *d2: What is the program supposed to do?* (9)
- *d3: Why was the code implemented this way?* (2)

or about execution behavior, i.e.

- *b3: Is this problem worth fixing?* (12)
- *r2: In what situations does this failure occur?* (8)

- *u1: What code could have caused this behavior? (5)*

In several instances coworkers were unavailable for these questions, and the developers' tasks were blocked once they sent their questions via email {ABCEFJR}.

Developers consulted coworkers about design because in most cases, design knowledge was only in coworkers' minds. The lack of design documentation may be due to inadequate notations, particularly for design intent and rationale. Two of the observed developers did have design documentation—a prototype for a user interface, a syntax grammar for a parser—which answered some of their questions {MG}. However, they still turned to coworkers when they questioned the accuracy of the documents.

Questions about program *behavior* were difficult to acquire because of the number of possible explanations. Developers had to use primitive tools to search this explanation space, and so searches were driven by intuition or expert opinion. Developers also went to great lengths to learn behavior information. As one example, to fix a bug recently assigned to him, E had a tester nine times zones away reproduce the bug (at 2 am) since no one else had the right machine configuration. Because behavior information was hard to acquire, developers made triage decisions quickly based on implementation concerns and resource availability, rather than the organization's overall goals {BCJLRT}. That is, developers would favor those tasks with the fewest information needs.

6.7.2. AUTOMATING INFORMATION SOURCES

One approach to reducing this communication burden is to *automate* the acquisition of information. Given the frequent desire for awareness information, it is no surprise that researchers are already creating awareness displays for development teams, like FASTDash [Biehl 2007] and Palantír [Sarma 2003].

For example, many of developers' questions about static relationships depended on metadata such as build numbers and version histories, but developers manually incorporated such data in their searches. Similarly, tools for analyzing programs' dynamic behavior only partially helped with determining the cause of a program state; the rest had to be determined by hand using a breakpoint debugger and through guesswork. Task-specific applications of program slicing would be a way to

automate some of this searching [Sridharan 2007]. Implementation questions (c1, c2, c3) also lacked adequate tools (it is worth noting that these needs have also been discussed relative to end-user programming in Chapter 4). Tools were often appropriated for unanticipated uses, so it is within tool designers' interests to create new tools that are *amenable* to appropriation. This might entail using standards, so that information may be passed between tools and transformed as needed.

Some information seeking cannot be automated because the information is currently unavailable. For example, when developers could not reproduce a failure, there was little they could do to find it. Tracing tools that can *record* the failure context, like those discussed in Chapters 9, 10, and 11, would be a major advance. Failures could be debugged separately from their original context and the trace could be analyzed by multiple people. Design intent was also difficult to find. Information about rationale and intent existed sometimes in unsearchable places like whiteboards and personal notebooks or in unexpected places like bug reports. Some awareness information is difficult to acquire, for example, developers often wondered who is *reading* their code. Tools could make this available.

Aside from tools, one could address these information needs through process change, for example Agile methods. The frequent need to consult coworkers for information is an important motivation for Scrum meetings and radical collocation [Fowler 2001]. Chong and Siino recently compared interruptions among radically collocated pair programmers versus cubicle-base solo programmers and found that the Agile team's interruptions were shorter, more on-topic, and less disruptive [Chong 2006]. The data about the importance of design knowledge provide evidence about the value of prototyping in software design, as well as the value of prototypes during implementation. The observations about the importance of error checking, coupled with the distributed nature of design knowledge, also support the claims of pair programming: with two developers, with slightly different design knowledge, errors seem more likely to be caught or even prevented.

Last is the issue of *notations* for software design. While there is already considerable research on architecture description languages, UML, and various forms of model checking, the observations raise several pertinent questions. What can be written down cost-effectively? How can it be written to be searchable and so that its accuracy and trustworthiness are assessable by developers who consult it? It is worth noting that several participants perceived that face-to-face meetings are a

pleasant and efficient way to transfer design knowledge. The frequent conversations promote camaraderie and no effort is wasted recording design information that might never be read or might go stale before being read. Hence, a demand-driven approach to recording design knowledge might succeed over an eager “record everything” approach.

6.7.3. LIMITATIONS

Because the study was performed in the context of developers’ real work, the external validity of these results is high. The diversity of the sample provides support for generalizing across different products, team structures and development phases within the organization that was studied. The study did not control for corporate culture, although the communication patterns and development processes that were observed are consistent with studies of other corporations [Perlow 1999] [de Souza 2003][Chong 2006]. Other variations, such as testing practices, the talent and expertise of a company’s developers, and more or less formal development processes, may have biased these findings.

Studies that rely on observations are subject to observers’ biases and a single observer may have misunderstood what developers were looking for in an individual observation session. This study had precisely this limitation, since to minimize intrusion, there was a single observer *per session*, who took only written notes. (In several cases, the participants worked in shared or noisy offices.) Even with a single observer, there were several instances of missed interruptions, where a visitor peeked inside the office, saw the observer, and chose to leave rather than interrupt. There were also many information seeking tasks that could not be observed because they were either too subtle, like glancing at a coworker’s IM status, or invisible, like the use of memory to recall facts about the code. The data was also biased by those issues that developers chose to mention during think aloud. To reduce the risk of misunderstandings, the study did involve two observers (this author and Rob DeLine) over the whole set of observation sessions, who carefully compared notes and discussed interpretations of quotes and other actions.

The time stamps in the handwritten notes taken during observations are accurate within the minute, but are subject to typical clerical errors during transcription and copying. Also, some time was spent talking to the observers, but this bias was likely distributed throughout the observations. Only a single coder categorized the logs,

which affects the quantitative data; however, the orders of magnitude in the data are likely to be accurate.

6.8. SUMMARY

The goals of this study were to identify software developers' information needs and characterize the role of these needs in developers' decision making. The results were numerous:

- Developers' work is highly fragmented in time, with developers experiencing interruptions an average of every three minutes.
- Software development is a highly social activity involve many forms of communication and collaboration.
- Developers had at least 21 observable types of information needs, spanning implementation, design, testing, and collaboration .
- Some needs were easy to satisfy accurately (awareness) but others had only questionable accuracy (the value of a fix and the implications of a change).
- Some needs were deferred often (knowledge about behavior and design), and some were impossible to satisfy in certain cases (reproduction steps).
- Information needs regarding debugging and program understanding, especially those regarding the causes of program behavior and conceiving of *potential* causes of program behavior, were particularly difficult to satisfy.
- Even after finding the cause of a bug, there is often the more daunting task of deciding what to do about it, which requires collaborating with coworkers and uncovering the design rationale underlying particular code fragments.

Not only do these needs call for innovations in tools, processes, and notations, but they also reveal how the collective responsibility for design knowledge can lead to intense awareness and communication needs observed in this study and others. These findings, particularly the information needs regarding understanding program execution, reinforce a number of findings from the studies in previous chapters. Industry developers, despite their relative expertise compared to the participants in the other studies, also have difficulty conceiving of explanations for their program's behavior and with verifying these explanations; supporting this process is precisely the goal of the Whyline, which is discussed in the next chapter.

7.

THE WHYLINE CONCEPT¹¹

While the studies in the past chapters revealed complex and detailed accounts of program understanding, they have a number of facts in common:

- To understand program execution, people ask *causal questions* about *program output* (when debugging, one might call this a *symptom of failure*).
- People try to answer these questions by conceiving of one or more potential explanations (hypotheses) of the causes of the program's behavior. However, because there is so little data about the program's *actual* execution at the time of forming these hypotheses, these guesses are based on surface features of the program output, prior experience and other personal and situational biases, few of which tend to correlate with a program's actual execution.
- Because these initial hypotheses are usually wrong and because current tools are so poor at helping people confirm or reject their potential explanations, people spend considerable time trying to find a correct explanation. People also form inaccurate notions about a program's execution, inserting new errors in the process and further complicating later explorations of code.

In essence, people must *guess* where to start their debugging search before learning anything about what the program did internally. This characterization of the problem motivates a simple solution: what if people could *directly ask* about the

¹¹ Patent pending, application serial # 11/246,331

causes of program output and have the system identify the parts of program execution responsible? Such a tool could diminish guesswork and perhaps prevent the unfortunate consequences documented by my studies and others'. There are several basic ideas that make up this concept:

- Asking “why” and “why not” questions by explicitly selecting program output.
- Deriving questions by identifying source code and execution events relevant to the selected output.
- Answering questions with complete causal explanations that are easy to explore.
- Correcting misperceptions of program output by allowing questions about output that did or did not occur, despite appearances.

This chapter will explore this idea conceptually, discussing definitions and ideas behind the Whyline concept. Later chapters will detail concrete implementations of these ideas for three specific environments.

7.1. ASKING ABOUT OUTPUT

What does it mean for a person to ask a question about a program? First, almost all questions are plausibly useful—what, when, where, why, how—even who when thinking of team software development. All of these types of questions are seeking some form of declarative fact about a program, its execution, its design. “What” and “where” questions usually seek code. “When” questions are used for code executed at a particular time. “How” questions usually refer to algorithmic details. The Whyline concept focuses specifically on “why” questions, which ask about causality. “Why” questions tend to precede other questions about what, when, and where, since the answer to a “why” question usually consists of the answers to several “what,” “when,” and “where” questions.

In the Whyline concept, the subject of a “why” question is always *program output*. Output is the visible part of a program’s execution, such as the signals transformed into light on a display, the printed dots on a piece of paper, the sound emitted from speakers. It is also any information that leaves the physical boundaries of the program in memory, such as data written to disk or sent across a network.

The reason why the Whyline concept requires users to reason about *output* and not code is simple. Program output is the first sign that something is right or wrong, the final effect of a series of causes propagated through potentially millions of decisions internal to a program. If a person has knowledge that a program is misbehaving, it is because that person first *sees* so in the program's output. This is a subtle, but crucial point: asking a question about a *precursor* to output assumes that nothing between the *precursor* and the *output* was faulty (see Figure 7.1). As the results of the studies in this dissertation have demonstrated, these assumptions are frequently wrong. For example, consider some failure in which a person sees an incorrect value printed in

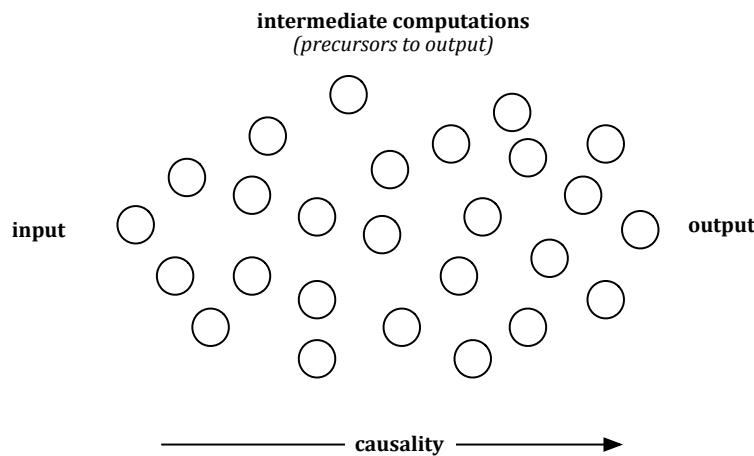


Figure 7.1. Asking about precursors to output presumes that the fault did occurred *before* the precursor, which is not always the case.

a console. Their first inclination might be to inquire about the code that produced that value. This is a reasonable guess; however, there are several other possibilities. Perhaps the output was computed by an entirely different module. Perhaps there is an error in the print statement that printed the value. Perhaps the output was printed by a different program entirely! Because there are so many possibilities and human cognition is generally so poor at conceiving of all the alternatives, people are likely to choose incorrectly, as we have seen in previous chapters. Therefore, it is crucial that any analysis of the causes of program output begin with the output itself.

7.2. QUESTIONS FROM CODE

A central part of the Whyline concept is that the user asks questions by *choosing* them, and not by verbally expressing them with speech or text. To attempt to understand developers' speech or text would reveal all kinds of challenges in

interpreting the meaning of a user's natural language. Instead, a Whyline derives questions from the program and execution history. The program, in this context, is treated as a *specification* of all of the output that a program can produce. It is also used as a repository of *names* for the different conceptual types of output in a program. For example, a painting program will have many kinds of painting concepts and, especially in an object-oriented program, these concepts will be given human-readable names. A Whyline utilizes the organization of these concepts to create menus of relevant human-readable questions.

In addition to questions about *visible* program output, there are several types of questions about output that a person expected but did not see. For example, if a user thinks that a window did not appear as they expected, a Whyline should support a question about the window not appearing. It is in answering such negatively phrased questions that things become interesting. For example, what if the window *did* appear, but on a different screen or behind another window, or in a different virtual desktop? A Whyline should still support the question, but then respond by explaining the user's misperception of the output. These misperceptions happened throughout all of the studies reported in earlier chapters.

7.3. EXPLAINING CAUSALITY

When we ask why questions in conversation, we often expect a single answer. For example, if a child asks a parent, "why do we have to move," a parent will typically reply with "because Daddy has a new job." But as any child knows, no single answer is sufficient. Even for trivial questions about causality, there are usually multiple causes and each of those causes has causes of its own. The same is true for program output. If a person is simply understanding an algorithm or a pattern of calls, there will be many software artifacts involved. If a person is debugging, there may in fact be a single place that contains the error, but there will a long chain of causality between that error and the eventual visible symptoms of the failure, potentially spanning many parts of the software system.

Although it might be possible to intelligently isolate parts of these causal chains that seem relevant or important, the current Whyline approach is to determine *all* places relevant to the queried output and then help the user explore, understand, and relate these places. This way, the tool can guarantee that the cause of the problem

appears somewhere in the answer, as long as the user is willing to search for it. In general, this approach has the goal of clarifying the relationship between a program's *input*, *execution*, and *output*. Whatever the task, it is making these causal associations that will ensure the developer has an accurate understanding of the program's execution. Therefore, a fundamental part of the Whyline concept is to provide a visual workspace that shows developers their source code, its execution over time, and the program output that emerged as a result.

7.4. SUMMARY

The concept of a Whyline includes:

- Selecting program output to provide entity context
- Deriving “why” and “why not” questions by identifying source code and execution events relevant to the selected output.
- Answering such questions with complete causal explanations that are easy to explore.
- Correcting misperceptions of program output by allowing questions about output that did or did not occur, despite appearances.

The coming chapters will detail implementations of this concept for three specific programming languages and applications.

8.

A WHYLINE FOR ALICE¹²

The first effort to explore an implementation of the Whyline concept began as a feasibility study. The question was whether such a concept was possible to implement, and if so, whether an implementation would actually help people be more successful at debugging and program understanding tasks. We considered a number of platforms, including Flash, Visual Basic, and Java, among others. For this first implementation, we chose Alice (described in Chapter 3). We had access to the source code and could modify any part of the system; the language and execution model were simpler than these other languages, meaning the focus could be on the question and answer interaction, rather than supporting every subtle detail of the other languages. Most important, the notion of output in Alice is clearly defined, consisting of 3D graphical objects and their properties. This was not true for the other platforms, where there were many types of output with different characteristics. Choosing Alice was a way to limit the scope of the design exploration to just questions and answers, saving other issues for later. This exploration and the eventual implementation of the Alice Whyline took place during the summer and fall of 2003.

This chapter begins with an example of the Whyline in use and then explains the implementation of the prototype. It ends with a discussion of a user study that compared the Alice Whyline to regular Alice, assessing the Whyline's influence on typical Alice debugging tasks.

¹² The results in this chapter appear in part in [Ko 2004a].

8.1. AN EXAMPLE

Alice is shown in Figure 8.1. Before describing the Whyline's implementation, its design is described through a debugging scenario that comes directly from the user study discussed later in this chapter:

Ellen is creating a Pac-Man game, and trying to make Pac shrink when the ghost is chasing and touches Pac. She plays the world and makes Pac collide with the ghost, but to her surprise, Pac does not shrink...

Pac did not shrink because Ellen (a pseudonym) has code that prevents Pac from resizing after the big dot is eaten. Either Ellen did not notice that Pac ate the big dot, or she forgot about the dependency.



Figure 8.1. The Alice programming environment, before the world has been played: (1) the object list, (2) The 3D world view, (3) the event list, (4) the currently selected object's properties, methods, and questions tabs, and (5) the code area.

When Ellen played the world, Alice hid the code and expanded the worldview and property panel, as seen in Figure 8.2. This relates property values to program output. Ellen presses the “why” button after noticing that Pac did not shrink, the program is paused, and a menu appears with the items “why did” and “why didn’t,” as in Figure 8.3. The submenus contain the objects in the world that were or could

have been affected. The menu supports exploration and diagnosis by increasing questions' visibility and decreasing the viscosity of considering them.

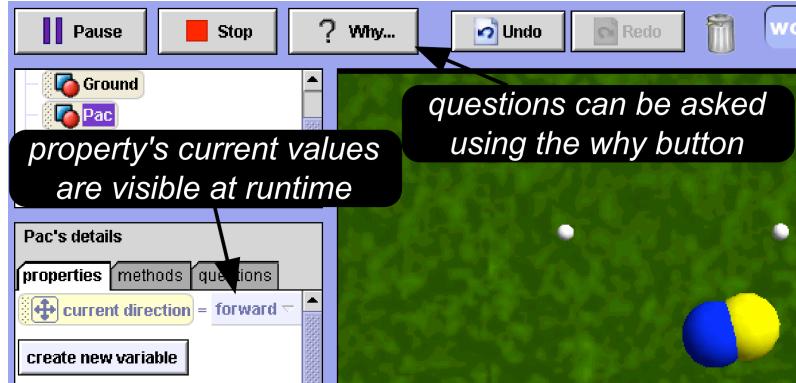


Figure 8.2. Ellen expected Pac to resize, but he did not.

Because Ellen expected Pac to resize after touching the ghost, she selects “why didn’t Pac...” and scans the property changes and animations that could have happened. When she hovers the mouse over a menu item, the code that causes the output in question is highlighted and centered in the code area (see Figure 8.3). This supports diagnosis by exposing hidden dependencies between the failure and the code that might be responsible for it.

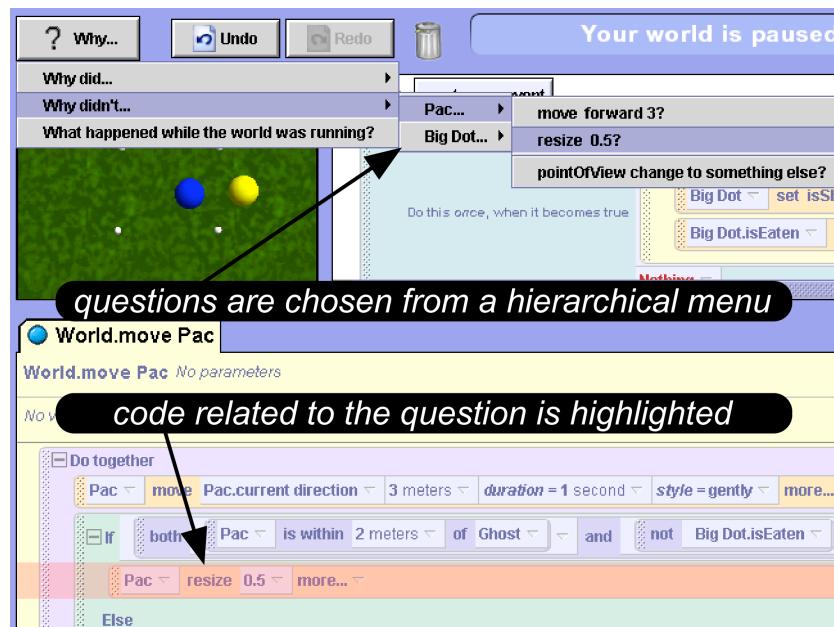


Figure 8.3. Ellen explores the questions and decides to ask “Why didn’t Pac resize .5?” which highlights the code.

Ellen asks “why didn’t Pac resize .5?” and the camera focuses on Pac to increase its visibility. The Whyline answers the question by analyzing the execution events that *did* and *did not* happen, and provides the answer shown in Figure 8.4. The execution events included are only those that prevented Pac from resizing: the predicate whose expression was false and the actions that defined the properties used by the expression. By excluding unrelated execution events, the tool supports observation and hypothesizing by increasing the visibility of the actions that likely contain the fault. To support diagnosis, the events’ names and colors are the same as the code that caused them. This improves consistency and closeness of mapping with code.

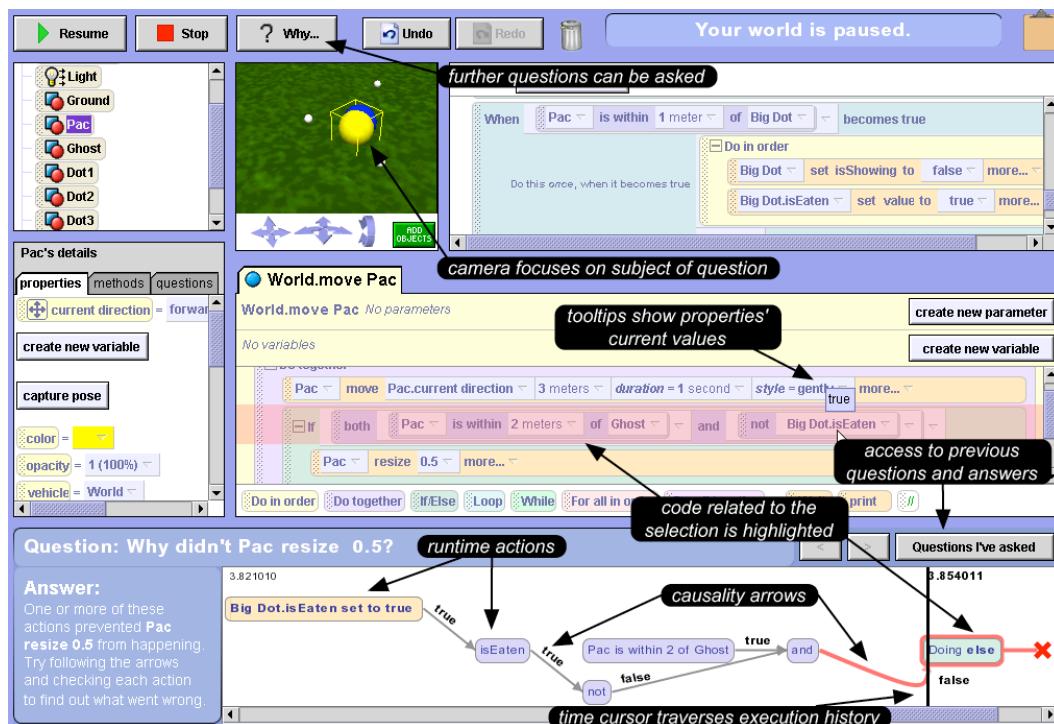


Figure 8.4. The Whyline’s answer shows a visualization of the runtime actions preventing Pac from resizing. Ellen uses the time cursor to “scrub” the execution history, and realizes that Pac did not resize because isEaten was true.

The arrows represent data and control flow causality. Predicate arrows are labeled true or false and data flow arrows are labeled with the data used by the action they point to. The arrows support progressive evaluation, and thus hypothesizing, by helping Ellen follow the runtime system’s computation and control flow.

Along the x-axis is event-relative time, improving the closeness of mapping to the time-based Alice runtime system. Along the y-axis are event threads: this allows co-occurring events to be shown, supporting juxtaposability.

Ellen interacts with the timeline by dragging the time cursor (the vertical black line in Figure 8.4). Doing so changes all properties to their values at the time represented by the time cursor's location, supporting exploration of the runtime data. When Ellen moves the cursor over an execution event, the event and the code that caused it become selected, supporting diagnosis and repair. These features allow Ellen to rewind, fast-forward, and even "scrub" the execution history, receiving immediate feedback about the state of the world. This exposes hidden dependencies between execution events and data that might not be shown directly on the Whyline, and between properties' current values and program output.

To reduce the viscosity of exploration, Ellen can double-click on an action to implicitly ask what caused this to happen and actions causing the runtime action are revealed. Ellen can also hover her mouse cursor over expressions in the code to see properties' current values and to evaluate expressions based on the current time. This improves the visibility of runtime data. The Whyline also makes previous answers available through the "Questions I've Asked" button.

By analyzing the events in the Whyline's answer, Ellen discovers her misperception:

"So this says Pac didn't resize because BigDot.isEaten is true...Oh! The ghost wasn't chasing because Pac ate the big dot. Let's try again without getting the big dot."

Without the Whyline, the misperception could have led to an unnecessary search for non-existent errors.

8.2. USER INTERFACE

The user interface for the Alice Whyline is fairly simple. There is a single global question menu at the top, which can also be shown by right-clicking anywhere in the timeline. The timeline itself supports a single selection model, allowing the user to express interest in any one of the events shown in the timeline. The selection is based on the horizontal position of the time cursor (the black vertical line in Figure 8.4). When the user clicks anywhere in the timeline, the selection is updated, and the

program state (all variables and object state) are set to their most recently assigned value at that point in the execution history. This automatically updates the program output view, showing the user the state of the program at the selected time. When the user drags the time cursor at the edges of the timeline window, the window automatically scrolls left or right (like selecting a large block of text in a word processor and hovering at the edge of a window).

The visual design of the timeline events deserves some discussion. All arrows in the timeline are straight, except for the curved arrows showing the use of values used for predicates (for example, the true or false value used by an “if” statement). This allows users to segment an answer into expression parts and control statement parts. Animation events such as moves and resizes are also given a proportional amount of horizontal space in the timeline to represent their duration (a two second animation was twice the horizontal length of a one second animation). The “x” at the right edge of the timeline in Figure 8.4 represents a predicate that evaluated in the wrong direction. The colors also carry meaning: peach represents imperative comments, purple represents data flow computations, and turquoise represents conditionals.

8.3. IMPLEMENTATION

The Alice Whyline used a modified version of Alice 2.0¹³. There are several issues to discuss regarding the Alice Whyline’s implementation: recording an execution history, deriving questions, and answering questions.

8.3.1. RECORDING EXECUTION

Alice programs are represented as an abstract syntax tree. When users drag tiles around an program, they are directly modifying this abstract syntax tree. For the Alice Whyline, this representation was augmented such that every operation on these trees was instrumented to maintain control flow and data flow graph representations of the program. In particular, data flow expressions were

¹³ At the time of the implementation, Randy Pausch and his students and staff were preparing an Alice textbook and needed to freeze the feature list, meaning that the implementation was never reintegrated with the later released Alice 2.0 versions.

represented as data flow graphs attached to control flow graph nodes. These graphs were constructed incrementally as programmers create and modify code.

The original Alice implementation executes Alice programs by traversing their abstract syntax trees. The modified version executes an Alice program by traversing the control and data flow graphs at runtime. As individual nodes in this graph execute, the Whyline maintains histories for individual nodes in the program graph, remembering assignments and uses of property values. This value history is maintained in memory and discarded at the beginning of each program execution. When the user drags the time cursor in the interface, these property value histories are used to “rewind” the program output to the selected time by identifying the value of each property in the program output at the current time. Because Alice programs are small, there were no serious issues with maintaining this information in memory, but in general, this approach does not scale.

8.3.2. DERIVING QUESTIONS

“Program output” in the Alice Whyline is defined as all of the 3D objects in an Alice program, all of those objects’ properties (including properties such as color, opacity, shape, etc.) and all primitive animations supported by default (such as “resize”, “move”, and other basic transformations). Output does *not* include user-defined procedures, variables declared inside those procedures, or other custom properties added to objects.

Given this definition of output, the Alice Whyline has a single “Why” menu at the top of the screen. Inside the menu, there are “why did” and “why didn’t” menus, each containing a menu of all of the objects in the program (typically a very small number). Each “why did” object menus contains two types of “why did” questions. First, there are questions about animations executed on an object (such as “why did Pac resize 0.5”). There is a question for each animation call that appears in the program. Therefore, if there are three locations in the code that Pac was resized by 0.5, there would be three questions. In addition to questions about these animations, there are also “why did” questions about changes to the selected object’s properties’ current values. For example, if Pac’s color was assigned to red, there would be a question about why Pac’s color was red.

In the “why didn’t” object menus, like the one in Figure 8.3, there are similar questions, but for *all* of the animation statements in the program regarding the selected object and *all* of the assignments to that object’s properties. Consequently, this menu can be much larger. Each question about an animation or assignment refers to a particular instruction in the program. In general, the prototype only distinguishes between commands and assignments with constant value expressions (resize 0.5, set color to red). For expressions or continuous valued expressions (such as position or opacity), the Alice Whyline supports the generic question “why didn’t this property change?”

In the original design, the structure of the “why” menu had four top level menus, including “why did,” “why didn’t”, and “why is”, “why isn’t.” These “why is” questions were particular to questions about object properties. Usability testing revealed that users did not understand the difference between these two, because animations often changed state and state often affected behavior. For example, changing the `isShowing` flag of an object made it “disappear,” which seemed like a behavior.

Another important aspect of the “why” menu was that the “why didn’t” menu contained questions about program behavior that *did* occur. These questions were allowed so that the tool would have an opportunity to correct developers’ misperceptions about the program’s behavior. For example, if the developer thought an object was not moving, but in fact it was, only not in a perceptible way because of the camera angle, the system would be able to explain that movement was occurring. The “why” menu did *not* contain questions about program behaviors that *did* occur, because in general, it was obvious when something had occurred. (As discussed in later chapters, such questions about misperceptions require slightly different support, because the type of misperceptions depend on the type of program output being misperceived).

8.3.3. ANSWERING QUESTIONS

Positively phrased questions (“why did Pac resize 0.5”) refer to a specific animation call or variable assignment. The challenge in interpreting these questions is to decide *which* call to analyze, since the particular call or assignment may have executed multiple times in the recorded history. Because observations of people asking “why” questions showed that developers asked questions immediately after

failures, the tool chooses the most recent execution of the call or animation. This was not always expressive enough to support users' questions (especially when developers delayed pausing), but worked in most cases.

To answer the question about the selected event, the Alice Whyline uses backward dynamic slicing [Zhang 2003] to discover the control and data dependencies responsible for the call or assignment executing. Alice programs generally work by invoking procedures in response to global event handlers, therefore most of these dynamic slices resulted in very short chains of causality. Nevertheless, the tool limits answers to two or fewer conditionals in the slice, to keep answers simple. If the problem occurred upstream of these conditionals, when the user selects the conditional furthest upstream, the system automatically computes other upstream conditionals and adds them to the answer. In addition to these calls and conditionals, the answers include all of the data dependencies in the causal chain, such as assignment statements and properties used in evaluating expressions.

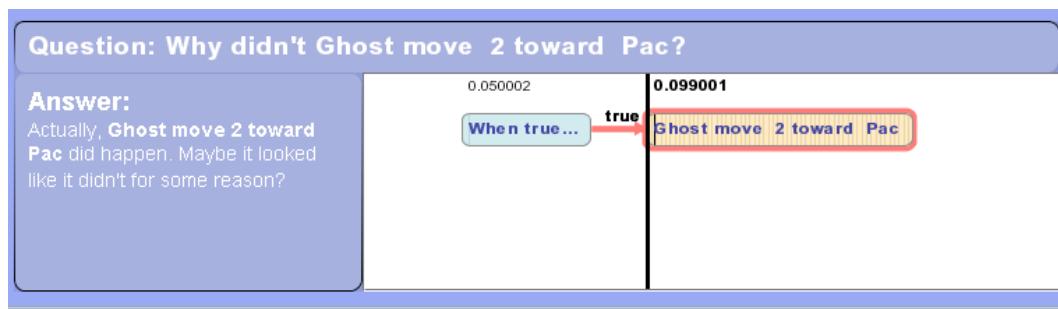


Figure 8.5. A *false proposition* answer, which explains to the developer that the code they thought did not execute, actually *did* execute.

For “why didn’t” answers, there are several cases to check in order to answer the question. First, the system checks if the command queried did in fact execute, despite the user’s beliefs. If it did, the system generates a *false proposition* answer, named so because of the implicit assumption in the user’s question (as shown in Figure 8.5). If it did *not* execute, the system then checks if the procedure that contains the queried command is *reachable* in the program’s control flow graph. If there are no edges to the procedure in the graph, then the system replies with an *invariant* answer (“this code can never be executed”, as in Figure 8.6). Finally, if the procedure is reachable but did not execute, the tool finds the enclosing conditional or procedure and recursively determines why the conditional or procedure was not executed (as in Figure 8.4).

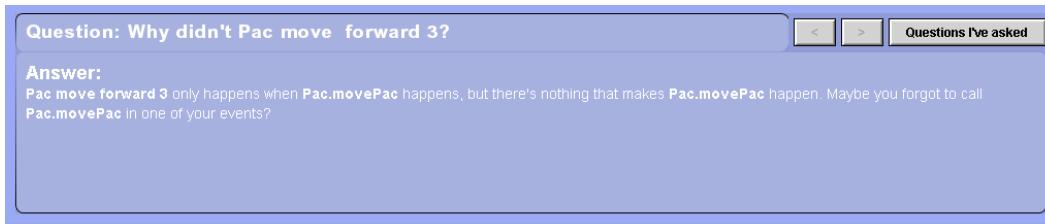


Figure 8.6. An invariant answer, which explains to the developer that the code can never be reached.

For determining why a procedure was *not* executed, the system recursively analyzes why none of the calls to the procedure were executed. If in the process, the system finds a conditional that did execute, then it must have executed in the wrong direction, skipping over the desired instruction. In this case, a dynamic slice is performed on the conditional's expression and included with the answer. An example of such a “why didn’t” answer is shown in Figure 8.4.

8.4. EVALUATION

The Alice Whyline was the first Whyline prototype, so there were several basic questions to answer:

- Would the Whyline be considered usable and useful?
- Would the Whyline reduce debugging time?
- Would the Whyline help complete more tasks?

To investigate these questions, the observational lab study discussed in Chapter 3 was replicated (using the same materials shown in the Appendix), but including the Whyline in the Alice environment. The first study without the Whyline will be called the *Without* study, and the present study will be called the *With* study. The *With* study used an iterative design method: observations from user session were used to fix usability problems and inform the design of features for successive sessions.

In both studies, participants were recruited from the Carnegie Mellon HCI Masters program. Programming experience ranged from beginning Visual Basic to extensive C++ and Java. The four participants in the Without study will be referred to as C1-C4, and the five in the With study as E1-E5.

Sessions began with a 15-minute tutorial on creating Alice code. Participants were given the layout in Figure 8.1 and 90 minutes to make a Pac-Man game with these specifications:

- Pac must always move. The arrow keys should control his direction.
- Ghost must move in random directions half of the time and directly towards Pac the other half.
- If Ghost is chasing and touches Pac, Pac must flatten and stop moving.
- If Pac eats the big dot, the ghost must run away for 5 seconds, then resume chasing.
- If Pac touches a running ghost, Ghost must flatten and stop for 5 seconds, then chase again.
- If Pac eats all of the dots, Ghost must stop and Pac must hop indefinitely.

For the purposes of analysis, these six specifications are treated as six distinct tasks, since the code necessary to accomplish them were only related by the character's state of behavior. In both studies, participants were asked to think-aloud to track goals, strategies and intents. Participants were also videotaped while they worked, for later analysis.

	Question Type	Answer Type	Frequency Question/Answer Pair was Asked	# of times Whyline found useful in the With study
			<i>Without (4 programmers)</i>	<i>With (5 programmers)</i>
<i>Why Did</i>	<i>Invariant</i>		0	0
	<i>False Proposition</i>		1	0
	<i>Control/Data Flow</i>		7	5
Total # of Why Did Questions		8	5	3
<i>Why Didn't</i>	<i>Invariant</i>		5	5
	<i>False Proposition</i>		5	7
	<i>Control/Data Flow</i>		7	7
Total # of Why Didn't Questions		17	19	16

Table 8.1. Frequency of question/answer types in each study and times the Whyline was found useful for each.

Table 8.1 shows the distributions of question/answer types in each study. “Why didn’t” questions were more common than “why did” questions, and programmers rarely asked invariant or false proposition “why did” questions. Participants used the Whyline for 19 of 24 of their questions (the sum of totals in the last column of Table 8.1) and the proportion increased in successive user tests: the Whyline was useful for 0 of E1’s 3 questions, but all 5 of E5’s.

Programming Error and the Failure it Caused	Strategy, Outcome, and Time in Seconds from Failure → Error Diagnosis			
	Without		With	
Code resized Pac to 0, which Alice ignores → Pac doesn't resize after touching ghost.	Read events, moved method call to event, moved camera, toggled state variables. “So it's the resize function that's not working.”	330 sec	Asked, why didn't pac resize 0? and got invariant answer. “So resize to 0 must not work. I'll try .5 instead.”	38 sec
Forgot event to call method that moves ghost → ghost doesn't move after playing.	Stared at screen and held head in hands. “Oh! I need an event to start it.”	75 sec	Browsed why didn't ghost questions and saw highlighted ghost movement code: “...oh, I didn't call it anywhere!”	8 sec
Thought dot2 was dot1 and referenced wrong dot → dot1 not eaten after collision.	Looked at code; searched for dot in worldview; removed then recreated collision code, this time without error.	91 sec	Inspected why didn't menu and realized her misunderstanding: “Oh, no questions about the other dots. That must be dot2”.	9 sec
Maps right key to right direction → Pac moves down instead of right.	“I think this is wrong; it doesn't go to where I want it to be...this is terrible!” Tried all possible mappings until she realized direction was relative to Pac.	182 sec	Asked why did Pac move right? Inspected control and data flow answer; noticed direction set to right because down was pressed: “So direction is relative to Pac.”	28 sec
Dot collision threshold too small for test to happen while Pac is over dot → dot not eaten after Pac touches.	“I made some methods that I thought would help me rid of the dots...I'm pretty sure I got close enough.” Rewrote collision events and slowed down Pac so test had time to happen.	207 sec	Asked why didn't dot1.isShowing change to false? Scrubbed predicate tests: “that's really intuitive...so when it actually did this test, this was the state of the world. I should increase the threshold.”	27 sec
Forgot event to call eatBigDot method → Dot not eaten after touching Pac	“There is definitely nothing happening.” Browsed and inspected code and event list. “Oh, of course not!”	49 sec	Asked why didn't big dot isShowing change to false? and read invariant answer which noted that nothing called the method. “Oh, you're right!”	10 sec

Table 8.2. Identical debugging scenarios in the With and Without studies. Scenarios are described by the programming error, the failure caused, each programmer’s strategy and outcomes, and the time from failure to error diagnosis.

Because the participants in the study created and debugged different errors, comparing debugging times was somewhat challenging. However, because the tasks were relatively constrained, many of the participants wrote very similar programs and introduced and debugged the same errors over the course of their work. Table 8.2 describes six of these identical debugging scenarios and the strategies and times of participants in the *With* and *Without* studies (there were more than six *similar* scenarios, but they were not similar enough to warrant comparison). In the *Without* study, participants tended to hypothesize and diagnose by inspecting and rewriting code. In the *With* study, they tended to hypothesize and diagnose by asking questions and analyzing the Whyline's answer. A repeated measures ANOVA shows that across the six scenarios, the Whyline participants were significantly faster at debugging similar errors ($F(1,5) = 12.64$, $p < .02$). This decrease was by an average factor of 7.8 across the six scenarios.

Overall, in the 90 minutes allotted, programmers with the Whyline completed significantly more tasks ($M = 3.20$, $SD = .457$) than those without ($M = 2.25$, $SD = .500$), $t(7) = 3.0$, $p < .02$. This was a 40% increase in tasks completed.

8.5. DISCUSSION

The Whyline appears to have great potential as a usable and effective debugging tool for Alice users. In generalizing these results, there are many issues to consider. For example, in the user testing, there were a few significant usability issues with the initial designs, some of which were predicted and others that were not. These have implications for the design of other Whyline designs. In session 1, for example, the prototype did not support "why didn't" questions. When E1 first used the Whyline, he wanted to ask a "why didn't" question, but could not, and immediately decided "this thing is useless." This suggests that support for "why didn't" questions may be crucial to programmers' perceptions of the utility of the tool. In session 2, the prototype distinguished between questions about output statements ("why did") and questions about property changes (phrased as "why is"). E2 observed a failure based on Pac's direction property and searched the "why did" menu, ignoring the "why is" menu. The experimenter asked her later if she had noticed the menu: "I wanted to ask about something that already happened." This is consistent with the observations that programmers phrased questions in terms of failures instead of runtime data: she said "why did Pac's direction change to forward?" and not "why is

Pac's direction forward right now?" In session 3, the prototype answered questions relative to the time cursor's placement. When E3 asked his first question, he moved the time cursor, and upon asking his second question, noticed that the contents of the question menu changed considerably: "Where did all my questions go?" This was the rationale for using a statement's latest execution, regardless of the time cursor's placement. (This interaction was changed in the Java version described in Chapter 10, where time is important only when asking "why didn't" questions). Usability issues were also found in sessions 4 and 5, but not of the same magnitude as in the first three.

The most helpful feature of the Whyline seemed to be the question menu. Observations confirmed the hypothesis that asking questions in terms of program output, rather than code or execution events, would make it easier for programmers to map their question to related code. By restricting the programmer's ability to make assumptions about what did and did not happen, the tool enabled them to observe and explore the execution events that most likely caused failures. Similarly, relating code to execution events interactively with the time cursor and visual highlighting helped with diagnosis and repair activities, as predicted. Had this relationship not been explicitly visualized, more text would have been needed to denote what caused the execution events, decreasing visibility, and programmers would have had to manually search for the code responsible. Finally, the data and control flow arrows directly supported hypothesizing about which execution events caused failure, as predicted. This seemed to be because the visualization acted as an external memory aid to help programmers simulate runtime execution. In the *Without* study, participants were forced to calculate expressions manually, allowing for attentional breakdowns during calculation. When the time cursor, reversibility, and other features were used, the observations suggest that they played supporting roles in the Whyline's overall effectiveness.

There are many issues to consider in generalizing from the Alice Whyline to other languages. For example, for a given language and programming task, what output will programmers want to ask about? In a modern code base, output might be numerical, message-based, or simply the execution of a stub of code. The Java Whyline (Chapter 10) starts from a notion of primitive level output and finds data and code that indirectly affects this primitive output, identifying code structures that are likely to be viewed as output relevant.

Because the implementation requires the complete execution history, another issue is memory and performance. Researchers have developed time- and space-efficient approaches to recording data definitions and uses, building control flow and data flow graphs, and generating dynamic slices [Tip 1995][Zhang 2003]. Another challenge is, for a given task and language, what heuristics generate the most understandable, concise answers? The prototype only included a small portion of a dynamic slice because of the simplicity of most Alice worlds. For more complex software, there would be a host of visualization and interactive issues in presenting a dynamic slice. These issues are explored in more detail in Chapter 10.

8.6. LIMITATIONS

The Alice Whyline has a number of limitations. Questions about complex Boolean and numerical expressions give equally complex answers. This is because the level of detail in the questions is not enough to know which particular part of the data flow path is problematic. Reichwein et al. describe one solution that allows spreadsheet users to mark intermediate values in data flow paths as correct or incorrect, which feeds into a visualization of which computations may be faulty [Reichwein 2000].

Programmers often needed to inspect the internals of Alice primitives. For example, choosing the distance for “is object a within distance of object b” was difficult, because programmers could not see the measured distance value used by the command at runtime; commands like these were black boxes like any other API, only returning true or false and nothing about the actual internal behavior. One solution would be to instrument the surface-level internal logic of primitives, so that such expressions could be shown on the Whyline. The Java Whyline, described in Chapter 10, allows developers to inspect the internal behavior of API calls.

The Alice Whyline does not support object-relative questions (such as “why did Pac resize after Ghost moved”), which were fairly common in early observations of Alice programmers. The Java version in Chapter 10 gives the developer more control over specifying this context and thus more control over implicitly specifying the problematic behavior relative to some time. Some object relative questions are still unsupported, however, such as “why is this object so close to that object?” and other

questions that are spatial in nature. Future prototypes may involve constructing object-relative questions using direct manipulation of the objects on the screen.

Finally, in the user studies, using the latest execution of the queried statement was sufficient. In more complicated Alice worlds, this may not hold true. One possible interaction would allow programmers to further specify their questions with a time, which would allow them to find a particular execution in the recent history. This is exactly what the Java Whyline (Chapter 10) supports.

8.7. SUMMARY

The Whyline for Alice was a first attempt at supporting why questions about program output and it clearly demonstrated the feasibility of such an interaction on a number of dimensions:

- Questions can be derived from source code to represent program output and behavior.
- Concise answers can be given using precise backward dynamic slicing.
- Why didn't questions can be answered effectively by using reachability algorithms.
- The Alice Whyline increased developers' productivity at debugging tasks by an average factor of 8.
- Most of the questions that participants wanted to ask were supported by the Whyline.

Of course, the Alice Whyline also revealed a number of challenges to address in generalizing the Whyline concept to other languages and usage contexts. Issues of scale, the concept of output, the support of more complicated languages and other issues will be explored in depth in the coming chapters.

9.

A WHYLINE FOR APPLICATIONS¹⁴

Modern applications often have several complex commands, automated features and hidden dependencies. For example, Microsoft Word is full of detailed settings about automatically correcting capitalization errors, replacing misspelled words, and defined intricate dependencies between paragraph and character styles. E-mail clients and web browsers often support a myriad of application settings and rules that can affect a user's data or browsing experience in subtle but frustrating ways.

Unfortunately, when users need to understand how these features work, applications provide little support. Online help systems are quite generic, having little to do with the specific application state or the user's specific document. Instead, users may spend significant time searching help systems and exploring application settings to figure out how to turn off an automatic feature or find out why a paragraph is always indented. It is even more difficult when something expected before does *not* happen. For example, a user might see a misspelled word in their document that the application thinks is spelled correctly. How can a user find out why the application did *not* mark it as misspelled? There have been a number of research approaches to address these problems, helping to teach users how to use commands to perform various tasks. Some involved animated help [White 2002] and others that involved sophisticated instructions and query languages [Ramachandran 2005] [Lin 2003]. Others have proposed user interface enhancements such as "stencils" for focusing users' attention [Kelleher 2005] and mechanisms such as a special help

¹⁴ The prototype described in this chapter was implemented by David Weitzman. Details in this chapter appear in part in [Myers 2006].

modes [National Instruments 2005]. The problem with all of these approaches is that they are generic and do not help users with problems specific to their application and document state.

We can think of these types of usability problems as a special form of program understanding. Just like when a programmer is trying to understand the execution of source code, users constantly must understand dependencies between application and document state. The key difference is that the user of an application has no access to the source code and would not want to see code anyway. What they *do* want to know is why the output they see is *not* what they expect, and more specifically, how they can change the application settings and document properties to have what they want happen. The Whyline concept is perfectly suited for these types of questions. The central difference from the users perspective being the kind of answer to give: rather than showing the user source code and explaining the program's execution, instead it ought to explain the application behavior in terms of the *user modifiable state in the application and document*.

This chapter will describe such an interaction technique through an example and then details of the implementation and evaluation of the technique.

9.1. AN EXAMPLE

The example is based on a simple text editor meant to mimic Microsoft Word and many of its more complex features, such as auto correction and paragraph styling. The prototype, called Crystal, works by dynamically building question menus based on the current application and document state. One common scenario in Word is typing some word and having it automatically corrected, even if it was not desired. For example, if one types “teh” and then space, the editor automatically replaces the text with “the.” This is not always the desired behavior.

In this editor, the user can hover over the word “the” and type F1 to reveal a menu of questions about the word, one of which is “Why was the text changed from ‘teh’ to ‘the?’” Upon choosing this question, the editor shows the answer in Figure 9.1. At (a), a question mark is left which shows where the question was asked. The highlighting shown at (b) shows that the text was corrected because the “Replace text as you type” checkbox, which was in the “AutoCorrect” menu, was checked by default. This answer is also explained verbally at (c).

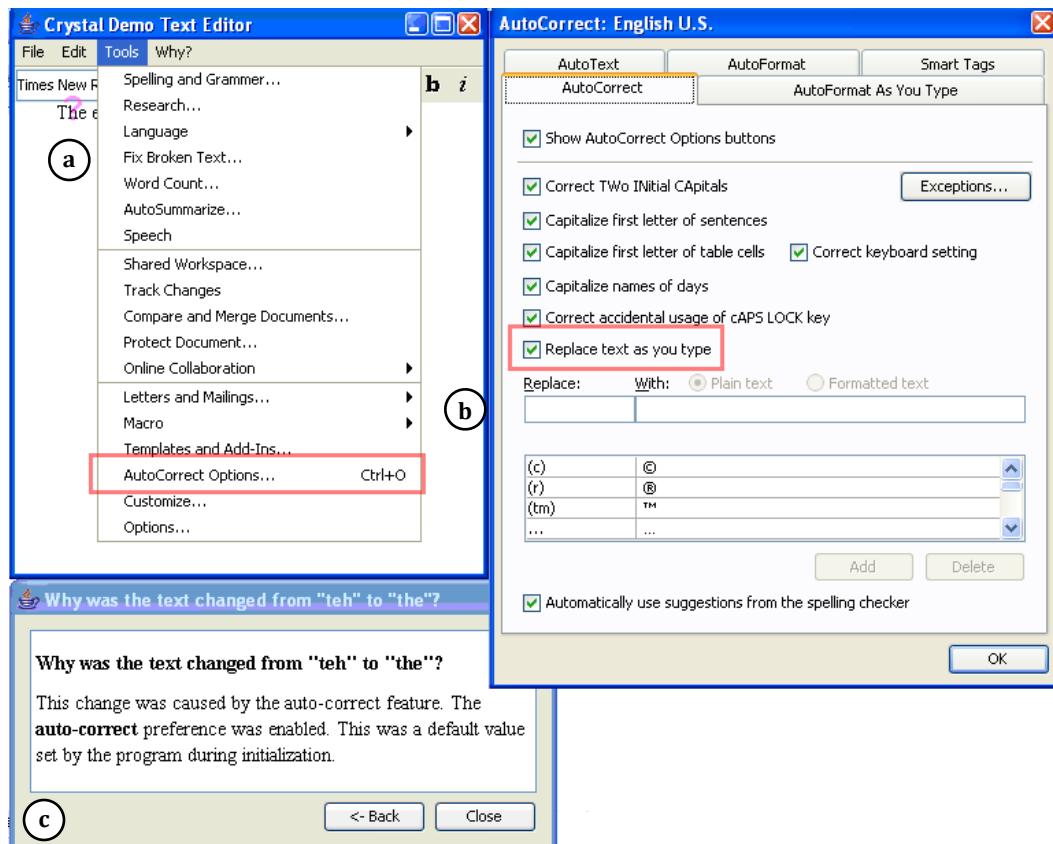


Figure 9.1. The answer for why “Teh” was changed to “The” The “?” in the upper left shows where the F1 key was pressed.

Unlike a Whyline for code, Crystal replies by providing answers in terms of the user-modifiable input and output of the program. In this example, this includes the menu “AutoCorrect Options...” and the checkbox “Replace text as you type.” This answer shows users not only the cause of the unexpected change to their text, but also how they can prevent such a change in the future.

9.2. USER INTERFACE

There are a number of unique aspects to the design of Crystal. For example, there are two ways to ask questions: by moving the mouse over some output of interest and pressing the F1 key, and also, by clicking on a global “why” menu. This latter approach allows the user to ask about application behaviors that have no suitable output to inquire about or behaviors that did not happen. In both of these menus, the question menu is derived from commands that are executed as the application

runs. A command is anything a user might find in the undo history of an application. For example, in the case of a word processor, these might be things like “set selection to bold” or “indent bullet.” Typically, applications are built with hundreds of such commands internally and they are exposed through various user interface elements such as buttons, menus and keyboard shortcuts.

As with any Whyline tool, a key consideration is the definition of program output and the lowest granularity at which questions may be asked. For this prototype, the various kinds of output include common word processing data such as characters, paragraphs, sections and the properties of these word processing data types such as indentation, font, and so on. Input includes user actions but also other application preferences that affect documents, such as the “Replace text as you type” checkbox in the earlier example. The primitive level of output was explicitly chosen to be the *character*, since it is the smallest indivisible element in a textual document. It was also important to consider the definition of “input.” At the lowest level, the prototype could have supported questions relative to mouse and keyboard events, but this is not how users typically think of using applications. Instead, the prototype defines input events as user and application invoked commands. The tool does not include questions about “typing” events, since they occur too frequently to be useful and are already well understood by users. In other application domains, the application designer would make similar decisions, for example, whether to support questions about the back and forward buttons in a web browser or scrolling through an e-mail in a mail client.

Several examples of these questions are seen in Figure 9.2. There are questions about the letter “h” selected and its properties, including why it has a particular font, whether it is bold or italic, and so on. The paragraph menu in Figure 9.2 includes properties of the paragraph such as indentation, style, alignment and other common paragraph properties. Below the paragraph menu, there are also questions about recent commands applied to the selected output. In Figure 9.2, there is also a question about why the text was changed from “teh” to “the.” This represents the command that was automatically executed by the application due to an application setting. Some output is invisible to the user, for example, the whitespace in the margin of a document that may be part of a paragraph. Figure 9.3 shows support for clicking on this whitespace to inquire about its properties and the related paragraph.

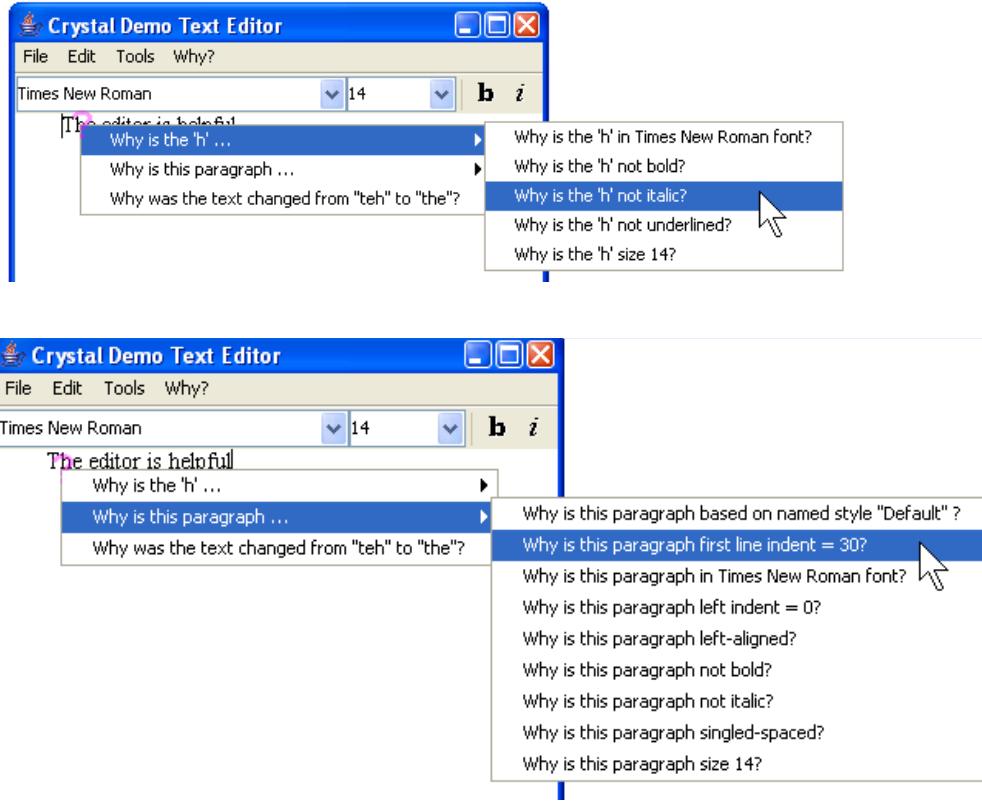


Figure 9.2. Menus resulting from typing F1, showing sub menus for character and paragraph properties.

In addition to questions about the document, the prototype also supports questions about the application UI. For example, the user can hover over a disabled button or menu item and find questions about why the control is disabled.

In addition to “why did” questions about application and document statement, the prototype also supports “why didn’t questions” about commands or behavior that did not happen (as seen in Figure 9.4). Of course, any number of things did not happen, but the scope of these questions is limited to the application features and forms of input. For example, if a user types an unsupported keyboard shortcut and nothing happens, Crystal includes a question about why the input was ignored. Or, if the application could have auto-corrected some text but chose not to because the setting was disabled, Crystal includes a question about why the command was not invoked. All of these appear in the global why menu in Figure 9.4.

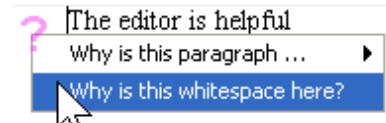


Figure 9.3. A question menu about whitespace.

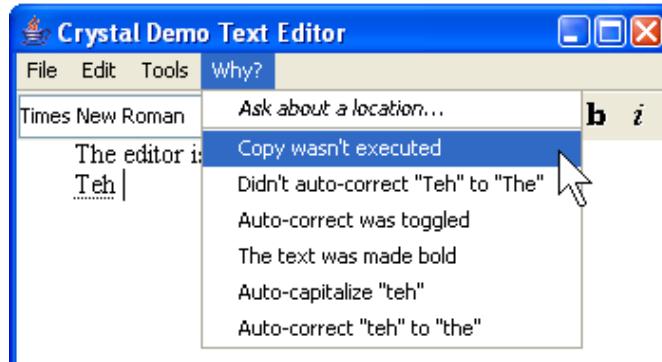


Figure 9.4. The “global” why menu, showing recent commands that did and did not execute.

The user can also ask questions about deleted data by clicking where the data used to appear (as shown in Figure 9.5). When the user deletes an object or word, Crystal leaves invisible objects to track their prior location and existence, linking the commands that deleted them. These markers flow with the text so that at any point, the user can ask about data that used to exist in the past.

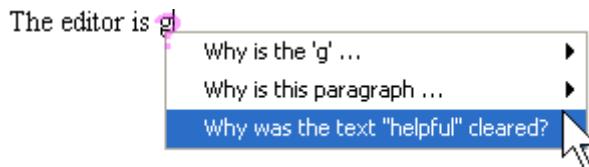


Figure 9.5. The user typed “g” in while “helpful” was selected, so “helpful” was deleted. Crystal inserts an invisible marker in the text so a question will appear about the deleted object.

Crystal answers questions by showing a textual explanation and highlighting user interface elements relevant to the user’s question. For example, Figure 9.6 shows the answer to “why is the letter p bold?”, including both a textual description of the command that caused p to be bold and a highlight of the user interface that was used. The key aspect of this design is that rather than explaining the decisions that the application made *internally*, the answer only includes user-modifiable events and decisions, such as actions that the user took explicitly themselves or commands applied by the application automatically because of user modifiable settings. An interesting side effect of this form of answer is that users can use Crystal to navigate to a settings dialog *faster* by asking a question, even if the user already knows the answer. When the user presses the close button, the highlighting is dismissed.

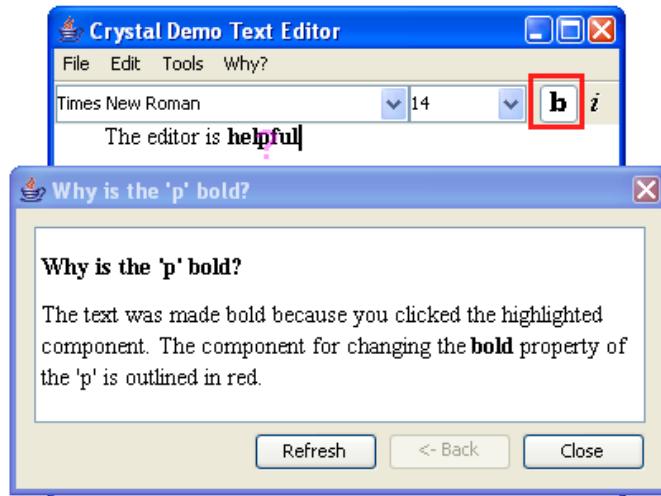


Figure 9.6. The answer to “Why is the ‘p’ bold?”, explaining that the user set the property using the toolbar button.

The textual part of Crystal’s answers are sometimes crucial. For example, there is often a chain of events that are responsible for a particular application state or behavior and such chains are difficult to portray through highlighting. For example, Figure 9.7 shows an answer explaining why a particular word’s font size was 20, which was inherited from its style. Crystal adds a link for each of the successive causes in the answer and provides a back button to navigate between these causes.

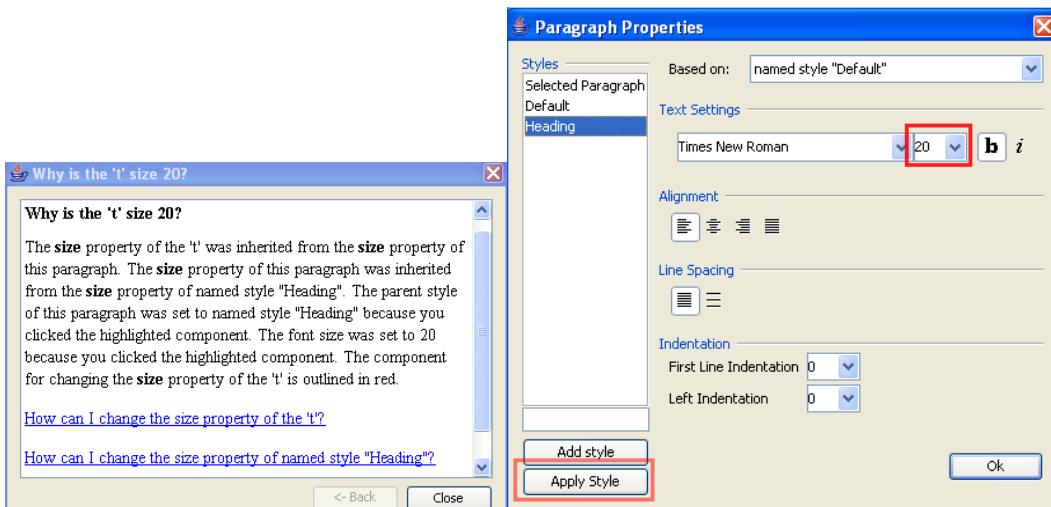


Figure 9.7. The answer shown for when a property’s value, in this case the font size, is inherited from a style.

9.3. IMPLEMENTATION

For supporting questions about program execution, a Whyline tool will typically record everything about a program's execution. For Crystal, such full recording is unnecessary, because the answers only need to explain causality at a specific level of abstraction. Instead, the approach was to augment a common application design pattern of an undo history with an explicit notion of commands and command history. In particular, Crystal uses a notion of *hierarchical command objects* [Myers 1996]. The top level command objects represent user executed commands, such as when a user explicitly sets a paragraph style. The lower level children of these top level command objects are the individual actions performed by the top level command; for setting a style, this might involve changing the paragraph's font, font size and other properties.

Each command object has a number of specific properties, shown in Table 9.1. Many are typical to command object based systems [Myers 1996]; Crystal adds the ones in bold.

Table 9.1. Fields and methods of the command objects in Crystal. Properties in bold are novel.

Name	Function
Do-Method	Performs the action, e.g. changes the font to bold
Undo-Method	Undoes the action
Redo-Method	Redoes the action
Object-Modified	Object affected by this action, so the command can be undone.
Enabled	Boolean to determine if action can be invoked now
Label	String that describes this command
Dependencies	Which properties of which objects are used by this command
Invoking-Control	Which control was used to invoke this command
Questions-Method	Supports application-specific questions
Undoable/Undone	Field that notes whether this command was undone yet
Show-In-Why-Menus	Whether this command should appear in Why menus

A key part of answering questions is tracking dependencies between these commands (the `Dependencies` part of the command object). For example, the auto-correct command in Figure 9.1 depends on the current state of the “Replace text as you type” checkbox. For every command executed, Crystal remembers the current values of these application settings, so that even if they change in the future, the command will remember the original state at the time it was executed. This allows

Crystal to generate a message like “the auto-correct preference was disabled” even if the property is now enabled. When values are inherited for properties, such as when the font size for a character comes from a named style, the `Dependencies` parameter is used to record where the value came from.

Each command also records the input event that was used to invoke it (the `Invoking-control` part of the command object). This could be a particular user interface control, a keyboard shortcut, or perhaps some automated feature that the application invokes after particular events (such as auto correction and spelling features). This is used to highlight the control in the user interface. This is especially important since each user may be used to using a different control to invoke the same command.

The `Questions-Method` part of the command object allows the application designer to create custom phrasings of the answers. The method returns an object that contains a method to generate the corresponding answer. This is used in the sample text editor for example, by the background auto-correction process. For standard property setting (e.g., “make bold”) and actions like creation and deletion, Crystal automatically creates the questions and answers, and the designer does not need to supply a method here.

The `Show-In-Why-Menus` flag (in Table 9.1) allows the application designer to determine that some commands should not be shown to the user as part of “why” menus even though they are undoable. For example, the Crystal text editor allows regular typing to be undone, but does not add this to the “why” menus. Conversely, normally sub-commands are not shown to users in the “why” menus, and instead just the top-level command would be included. However, if the programmer wants to allow the user to ask about a sub-command, then its `Show-In-Why-Menus` can be set to true. An example is that when a new character is typed, the top-level typing command is not displayed in the “why” menus, but if the new character inherits its formatting from a named style, the programmer might want the sub-command that sets the character’s properties from the style to appear on the “why” menus, since that may be mysterious to some users. When a command’s `Enabled` property specifies that it is disabled, but the user tries to execute it anyway (e.g., typing `Control-C` with nothing selected), then a command object is put on the command list with its `Enabled` property set to false to show that it was not actually executed. These unexecuted

commands allow Crystal to support asking of “why not” questions. Of course, these commands are not undoable, since they were never executed.

There are several useful side effects to this application design. For example, because the application knows all of the prior states of various document and application properties, implementing undo is extremely simple. The only practical difference between current application design and the design that Crystal imposes is that the command history in Crystal is never thrown away. In fact, when a user invokes an undo command, that command itself is added to the command history.

Generating question menus is straightforward. The “why” menu contains the last few user visible items in the command history. This is typically different than what would be visible in an undo history, since unexecuted commands and undo commands themselves will also appear in this menu. To generate questions for a specific output entity, such as a character or paragraph, Crystal finds all of the output entities and user interface components under the mouse. For each of these entities in the UI that supports questions, there are a number of types of questions included in the menu. Crystal includes questions about recent commands in the command history that reference the selected output. Crystal also includes questions about properties of the selected output in a sub menu. Finally, Crystal also maintains a history of objects that used to be part of a document but are no longer, such as paragraphs or images that were deleted. Objects that are deleted by the user leave invisible objects where they used to be, linked to the commands that deleted them. In a regular graphical editor, this would make it easy to ask about the object that used to be at a location. In the sample text editor, the objects are invisible markers that flow with the text. In the text editor, a custom method for whitespace was added that adds an extra question that asks about the whitespace itself. Alternatively, the programmer can provide special invisible objects in all the blank areas, and let them generate questions about why the area is empty.

The question answering algorithm is fairly generic. Questions about properties simply involve checking the command responsible for the property’s current value, recursively. Similarly, questions about commands recursively identify the nested commands responsible for the command in question, traversing the command hierarchy. When the property’s value is inherited, for example when a font size property comes from a named style, then the answer must include a discussion of the inheritance, as well as the final place in which the value was set, as in Figure 9.7.

This required a custom answer method in the sample text editor, to generate understandable messages. However, facilities in the Crystal framework automatically traverse the command's `Dependencies` to determine the properties that contributed to the current value. If any of those properties themselves were inherited, then Crystal recursively goes to those properties' commands, and then to their `Dependencies`, and so on. At each step, Crystal checks to see if the property is marked as `Show-In-Why-Menus`. If so, another sentence is added to the answer window. (Internal properties are often involved in dependencies, but should not be shown because users cannot change them.) When there are multiple steps, then a "How can I..." question is added to the end of the answer, so the user can ask about each step individually.

To highlight the controls, Crystal needs the ability to bring up widgets programmatically, set them to specific values, find their location, and highlight them, while still having them be operational for the user. Furthermore, the dialog boxes need to keep track of what causes them to be displayed, so Crystal can highlight the appropriate menu item. We were able to implement all of these using the Swing toolkit. Such support is also available in other commercial toolkits such as Mac OS X's Cocoa, where it has been used to implement several types of universal access features.

9.4. EVALUATION

A study was performed to determine whether the questions supported by Crystal were useable and useful. The study was a between-participants design to avoid learning effects. One group used the Crystal word processor portrayed throughout this chapter and the other used an identical application but without the question support. Each group had 10 participants, between the ages of 18 and 53 with an average age of 24. Twelve participants were male and eight female. Participants who reported "little or no" experience with Microsoft Word were recruited, although they all had extensive general computer experience, and all but two had experience with other text editors. Those two happened to both be in the group with the "why" menus. Participants were randomly assigned to one of the two groups and were paid to participate. The experiment was conducted on a laptop and was recorded. All of the materials used to run the experiment are shown in the Appendix.

Both groups received the identical six tasks. These were derived from real observations of Microsoft Word users, published articles about difficulties with Word, and an inspection of Microsoft's support pages. The tasks represent common issues that real Word users encounter. In summary, the tasks were:

1. turn off automatic capitalization;
2. turn off automatic spelling correction;
3. change paragraph formatting;
4. explain why the "Paste" menu item is grayed out;
5. use the Styles mechanism to change italics of some headings; and
6. use the inheritance property of the Styles mechanism to adjust the font size of all headings.

However, the tasks were not presented this way. The experimenter demonstrated a problem or a surprising behavior (or let the user do it), and then asked them to fix it. For example, the experimenter read the following script as the stimulus for the first task:

- Type in the following sentence "The abbreviation fl. oz. stands for fluid ounce."
- You notice that the word processor has capitalized some characters for you, but you don't want this to happen.
- Your task is to make the automatic capitalization not happen again.
- When you think you're done, type "fl. oz. stands" again to make sure it works.

In order to make the experiment somewhat realistic, Microsoft Word 2003's "Tools" menu and the "Options" and "Auto Correct Options" dialogs that are invoked using the Tools menu were all copied (see Figure 9.1). All of the submenus and the various tabs on each of these were live, so the users would have to search through more places. Both tasks 1 and 2 required using the "Auto Correct Options" dialog (Figure 9.1), and no task required using the Options dialog. Tasks 3, 5 and 6 required using the paragraph styles dialog (Figure 9.5). The participants were given brief training on the 'why' features with an example problem, but none that involved the details of any of the tasks. The participants were *not* trained on the any aspects of the

application menus, because the point of the study was to assess Crystal's ability to help learn these details.

Dependent measures included time on task and task success. A few users got stuck and required hints, and they were counted as unsuccessful. Because not all participants completed all tasks successfully, the data could not be analyzed using a standard repeated-measures ANOVA. Instead, both the number of tasks completed and the mean time per completed task were analyzed using between-participants ANOVA. Participants in the "why" menus condition completed an average of 5.60 (93%) of the tasks whereas those without "why" menus completed an average of 4.20 (70%) of the tasks ($F [1, 20] = 12.60, p < .005$). As shown in Figure 9.8, participants with "why" menus had an advantage in each of the six tasks.

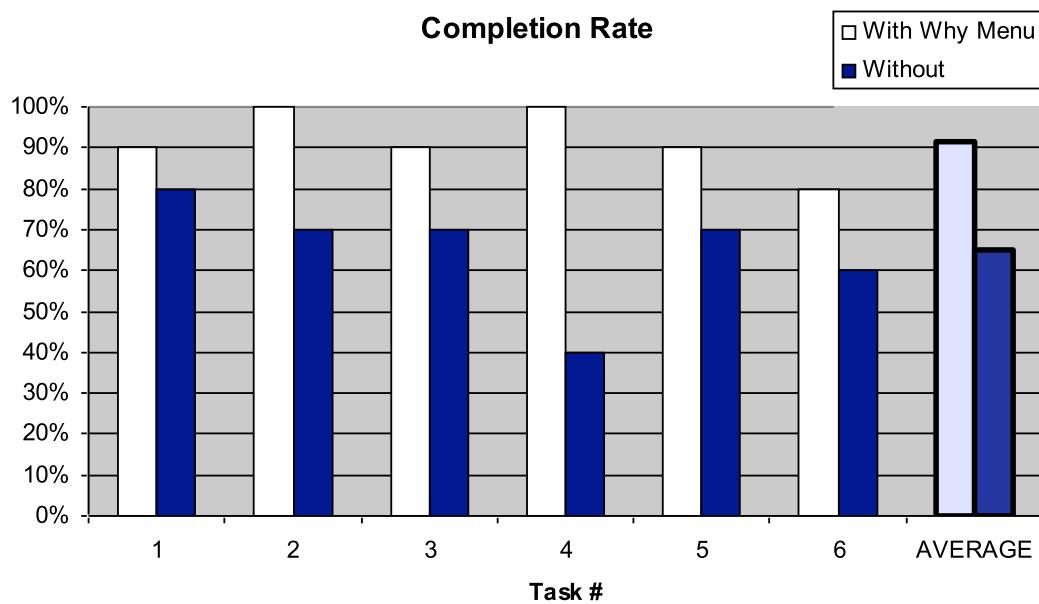


Figure 9.8. Percent of people in each group that completed the tasks and the overall average. Taller bars are better.

Figure 9.9 shows the average time per task for those participants who could finish it. Participants with "why" menus completed each task in an average of 91.38 ($SD = 51.66$) seconds, whereas those without "why" menus required an average of 137.74 seconds ($SD = 49.62$). This difference approached significance ($F [1, 20] = 4.19, p = .06$).

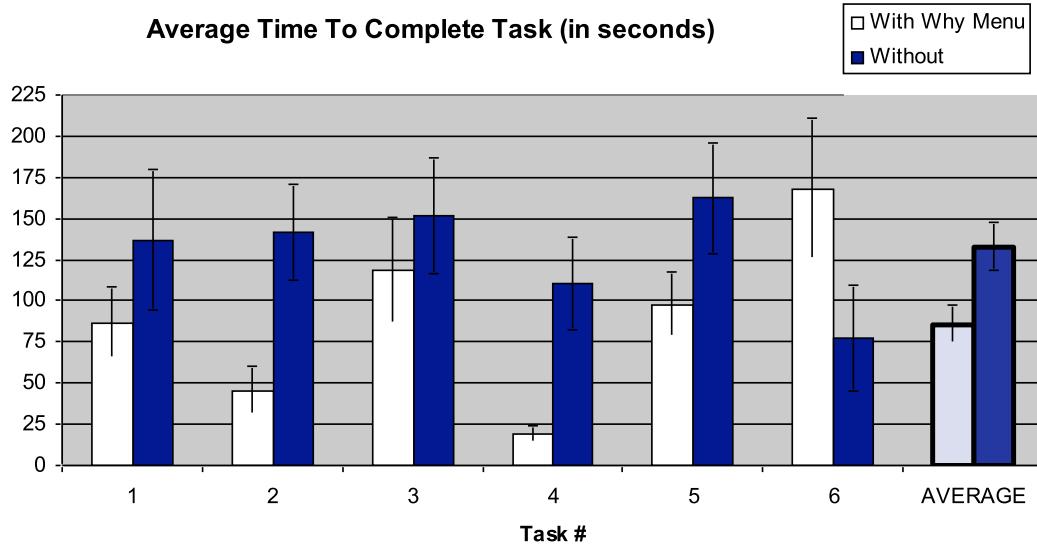


Figure 9.9. For the participants who could complete the task, the average time they took, with bars showing the standard error of the mean. Shorter bars are better.

The anomalous value for task 6 seems to be due to a few participants in the “without” group accidentally figuring out a workable strategy during task 5, compared to the “why” menu group who almost all used the “why” menus to try to learn how inheritance works.

The participants who saw the “why” features liked them. Each of the statements got an average agreement value of greater than 6.2 out of 7: “I understand how to use the Why feature in Crystal”, “I found the Why feature easy to use”, “The Why feature improved my word-processing experience”, “The answers provided by the Why feature were easy to understand”, “The answers provided by the Why feature were what I wanted to know”, “I was comfortable using the Why feature”, and “I would really like a Why feature like this in the programs I use.”

Clearly, the “why” menus were helpful to users. It is not surprising that the later tasks fared worse, since these tasks were quite difficult, even for some experts. For some people, the “why” features played the crucial role of explaining the concept to some of the participants, which directly led to successful task completion. However, Crystal is not necessarily designed to serve as a tutorial, and it probably did not teach participants about the concept of inheritance if they did not know it already.

There also were several usability observations about the system. Most of the participants preferred using the F1 key to have more control over the questions they

could ask. It seemed that the most efficient people used the F1 key first. Some participants were reticent to use the F1 key—this apparently was not a natural interaction for them. They used the “Ask about a location...” item in the “why” menu when the desired question was not in the “why” menu directly.

Participants using the “why” features generally knew which objects they should ask questions about, and the questions that showed up matched their expectation. Participants without the “why” features used a lot of trial-and-error clicking of menus, while those *with* the “why” features did not, seeming more purposeful and effective in their actions.

9.5. DISCUSSION

By attempting to adapt the Whyline concept to regular application use, a number of interesting issues were raised. For example, it became obvious that what to show in a Whyline answer depends a lot on what the user asking the question will understand and be willing or able to change. In the Alice Whyline, any line of code that the user could change could potentially be part of a Whyline answer. For Crystal, it was important only to include program execution events that the user would recognize either because they were responsible for them or the application had some setting that they could modify. One way to characterize this class of execution events is that they included all events and state at the level of *event handlers*, in the word processors implementation. Any data structure the event handlers used was subject to questioning, whereas data internal to the implementation and manipulation of such data was not.

The word processing domain also revealed that in designing a Whyline tool, one must have a deep knowledge of the *nature* of a domain’s output. This issue arose in making tradeoffs in question support, as was done by not including questions about typing events, in order to make the menus of reasonable size. These same kinds of tradeoffs were made in the Java Whyline (Chapter 10), again using knowledge about the types of output supported. In many cases, it was only clear what types of questions would be useful because we had an intuition about common problems in the word processing domain. Were one to design a Whyline tool for Photoshop one would probably need considerable Photoshop experience to choose the right granularity of primitives for questions.

9.6. SUMMARY

The Crystal approach is an example of using the Whyline concept to help people understand application behavior. Adapting the Whyline concept to application use resulted in several contributions:

- Answering algorithms that explain causality in terms of user modifiable document and application state.
- Interaction techniques for asking questions about document and application entities, whitespace, and global events.
- A software architecture for augmenting a conventional undo stack with information about command histories and the data they depended on for execution.
- Evidence that Crystal users are significantly more effective at resolving common issues with complex and automated features of a word processor than users of regular online help.

10.

A WHYLINE FOR JAVA¹⁵

While the prototypes discussed in previous chapters were quite successful, they left unexplored the challenges of generalizing the Whyline concept to programming languages typically used by professional programmers. These challenges are numerous: first, the increased complexity of the languages and the programs written in these languages mean that the answers to Whyline questions are likely to be more complex. Supporting the variety of programs that can be written in general purpose languages also means that the Whyline must support questions about a wider variety of output and in a generic way, since the output of a particular program will not be known in advance.

To explore these challenges, the final Whyline prototype described in this dissertation supports desktop Java programs. The prototype supports graphical output drawn as pixels on a display, textual output printed to a console, and Java exceptions. In designing the prototype, a number of unique algorithms and data structures needed to be designed and there were several HCI challenges that constrained its design. This chapter discusses these contributions in detail and then explores the effectiveness of the prototype compared to conventional Java debugging tools.

¹⁵ Details in this chapter appear in part in [Ko 2008a] and [Ko 2008b].

10.1. AN EXAMPLE

To motivate the implementation, let us begin with an example of the Java Whyline in use. The study in Chapter 5 involved a simple painting application, which supported drawing colored strokes (see Figure 10.1a). Among the 500 lines of code, there were a few bugs in the program unintentionally inserted, which were left in for the study. One problem was that the RGB color sliders did not create the right colors. Participants took a median of 10 minutes (from 3 to 38) to find the problem; the high variation in times was largely due to their strategies: most used text searches for “color” to find relevant code, revealing 62 matches over 9 files; others manually followed data dependencies, sometimes using breakpoints.

With the Whyline, the process would be greatly simplified (see Figure 10.1). The user simply demonstrates the behavior they want to inquire about (a), in this case by drawing a stroke that exhibits the wrong color. The user then quits the program and the trace is automatically loaded by the Whyline. The user then finds the point in time they want to ask about by moving the time controller, the black vertical bar in (b). Then, they click on something related to the behavior to pop up questions about it (c). In this case, they could click on the stroke with the wrong color, resulting in the question, “why did this line’s color = ■?”

After clicking, the Whyline shows a visualization explaining the sequence of executions that caused the stroke to have its color (d)(e). This visualization includes assignments, method invocations, branches, and other events that cause the behavior. When the user selects an event, the corresponding source file is shown (f), along with the call stack and locals at the time of the selected execution event (g). In this case, the Whyline selects the most recent event in the answer, which was the color object used to paint the stroke (d). To find out where the color came from, the user could find the source of the value selecting the label “(1) why did color = `rgb(0,0,0)`” (d). This causes the selection to go to the instantiation event (e) and the corresponding instantiation code (f). Here, the user would likely notice that the green slider was used for the blue component of the color; it should have used the blue slider.

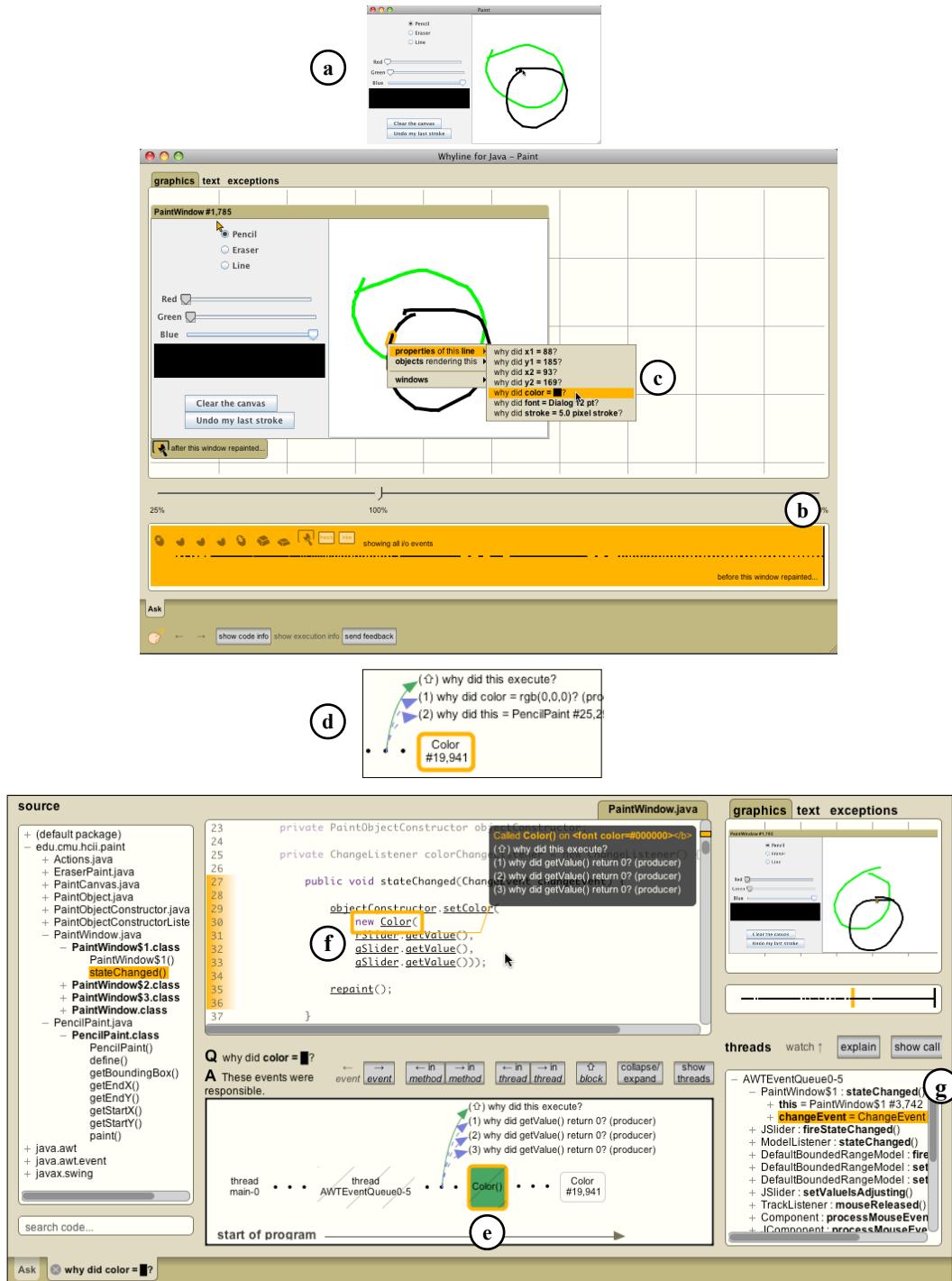


Figure 10.1. Using the Whyline: (a) The developer demonstrates the behavior; (b) after the trace loads, the developer finds the output of interest by scrubbing the I/O history; (c) the developer clicks on the output and chooses a question; (d) the Whyline provides an answer, which the developer navigates (e) in order to understand the cause of the behavior (f). (g) shows the call stack.

In a user study of this task, reported at the end of this chapter, people using the Whyline took half the time that participants with traditional tools took to debug the problem. This was because participants did not have to guess a search term or speculate about the relevance of various matches of their search term, nor did they have to set any breakpoints. Instead, they simply pointed to something that they knew was relevant and wrong, and let the Whyline determine the related evidence.

10.2. USER INTERFACE

Before discussing the implementation in detail, this section documents the many interactive details of the Whyline user interface.

10.2.1. RECORDING A PROGRAM EXECUTION

When starting the Whyline, the user first sees the screen shown in Figure 10.2. This window has two sides: the left lists the user's *launch configurations* and the right lists *saved recordings*. (Recordings can be saved and named after the recording has loaded; each recording has a unique name, since the same launch configuration can be run and recorded multiple times, demonstrating different problems). Saved recordings can be loaded by just selecting the saved recording from the list and pressing the appropriate loading button at the bottom of the main Whyline window.

To begin using the Whyline for a new program, a user must first create a launch configuration, so that the Whyline knows how to execute the program (this is the same process used by most modern software development environments). This launch configuration window is shown in Figure 10.3. Each configuration takes the root folder of the program and the relative paths to the program's compiled class files and original source files. The user also must specify the class with the `main()` method to execute and the arguments to pass to it.



Figure 10.2. The main Whyline window, showing the user's launch configurations on the left and saved recordings on the right.

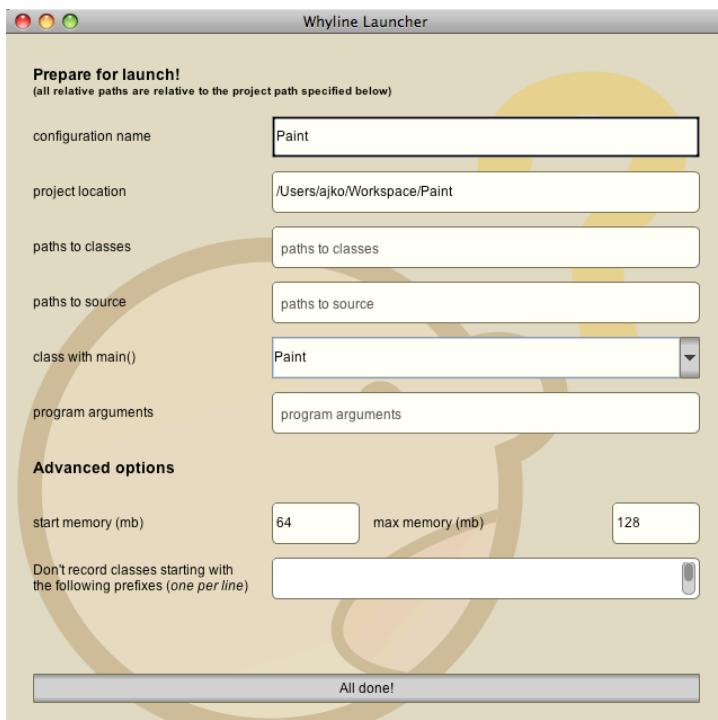


Figure 10.3. The launch configuration window.

In addition to these basic options, the user can also specify how much memory to launch the program with, as well as a list of class names and package prefixes to skip during the instrumentation process. This allows the user to control which code is recorded and which is not. By default, everything in the user's program is recorded, as well as all libraries loaded by the program at runtime. The defaults in Figure 10.3 are fine, since the paths to the source files and classes are contained within the Paint directory specified for "project location"; if they were not at the root folder, these fields would contain the relative paths to the root of the source files and class files.

Once the configuration is complete, the user simply clicks the "Record" button in Figure 10.2 and the Whyline launches the program in a separate Java virtual machine. The user then demonstrates the behavior that they want to understand, and when finished, quits the program. The Whyline knows

when the program has halted (using platform-independent Java mechanisms to watch platform-specific processes) and then reads the recording from the disk, also opening a Whyline window that represents the recording. Progress is shown in a simple progress bar, as in Figure 10.4. Loading time depends on the complexity of the program execution that was recorded. User interfaces, which are often idle, are fairly lightweight; computationally dense programs, such as compilers, can take much longer, since there is more data to load and process (see [Zhang 2004] for a more detailed performance analysis of such programs).



Figure 10.4. The loading progress bar in the Whyline window.

10.2.2. ASKING QUESTIONS

A Whyline window represents all of the data in a Whyline recording. There are two basic modes for this window: *question asking mode*, which shows the program output and a time slider and nothing else, and *answer mode*, which shows a visualization, source code and many other types of runtime information. It was a conscious decision to only show program output in question asking mode, as it is important for developers to begin their search by working backwards from output. As explained in Chapter 7, if the search begins at the level of code, developers are prone to overlooking problems by making assumptions about what information they see in program output.

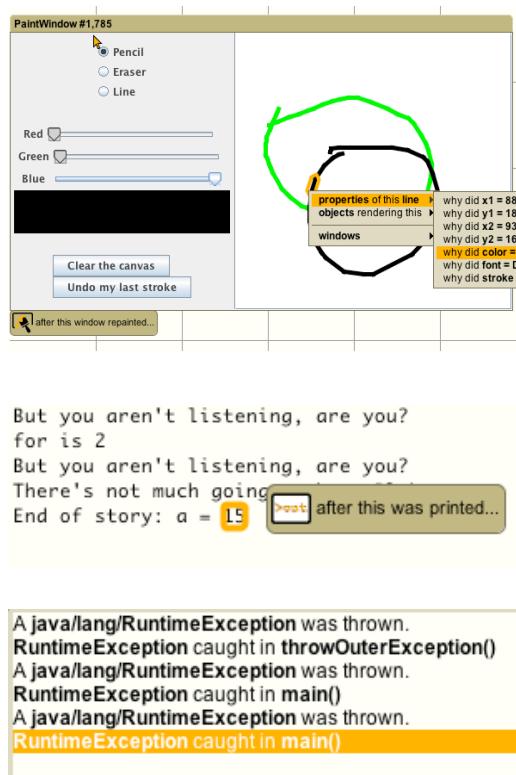


Figure 10.5. Hovering over graphical, textual, and exception output. The graphical and textual output both include pop-ups indicating the temporal context of the output (such as “after this was printed” and “after this window repainted”).

10.6), about fields that affect output and their current value or why they were not assigned after a certain time (Figure 10.7). Users can also ask questions about why particular methods were not executed after a certain time. Also notice in Figure 10.7 that the questions largely consist of names extracted from the source code (such as `PaintCanvas "canvas"`); the idea of including class and variable names in the menus came from the information foraging results in Chapter 5. Including such names should help developers find the questions they want to ask more easily by giving them more cues about the relevance of the contents of each sub-menu.

For textual and graphical output, users can ask questions about why text was printed and why an exception was thrown. There is also limited support for asking why some text was *not* printed, by finding the desired text in a global menu of all print statements in a program. The implementation section later in this chapter explains about how these questions are derived.

In the Java Whyline question asking mode, there are three output tabs: *graphics*, *console*, and *exception*. The Whyline initially shows whichever tab contains the most number of output events. When hovering over output with the mouse in any of the three output modalities, the Whyline highlights individual pieces of primitive level output. For example, Figure 10.5 shows an individual line segment highlighted as part of a larger line stroke, an individual value printed from a variable in a console, and also an exception thrown. Clicking on any of these types of output shows a menu of questions about the content underneath the mouse.

The types of questions supported depend on the type of output. For graphical output, users can ask questions about the properties of primitive output (Figure

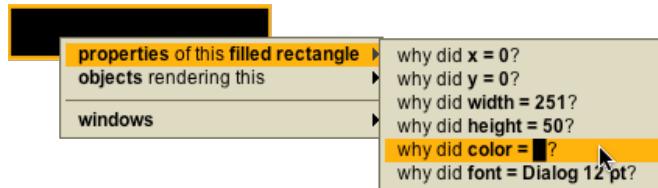


Figure 10.6. Questions about properties of a rectangle.

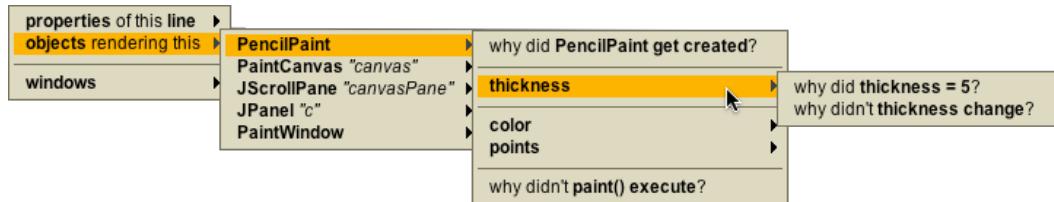


Figure 10.7. Questions about fields and methods that indirectly affect output.

Another factor in asking questions other than the *subject* of the question is the *temporal context* of the question. The Java Whyline provides a time slider, as in Figure 10.8, to allow the user to explore the output history of the program by simply dragging the cursor with the mouse or using the keyboard arrows to step between individual I/O events. Each black dots in the time slider represents a single I/O event, such as a mouse click, a keyboard press, or a window repaint. There are also several icons at the top of the time slider, each representing a particular kind of input or output. Figure 10.8 shows the mouse move event filter selected, so that only drag events are shown in the slider and only mouse move events are selectable. This allows the user to easily find particular kinds of events, ignoring those of a type irrelevant to the problem being investigated.

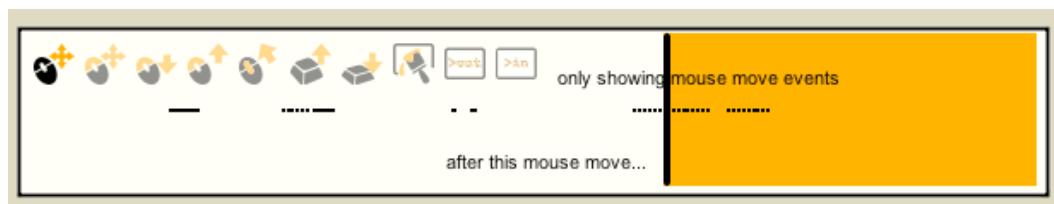


Figure 10.8. The time slider, showing only mouse events (the left-most icon is selected, indicating this filter).

When asking a question, time is treated differently depending on the intent of the question. For positively phrased “why did” questions, the user needs to select the time at which the output in question is visible in order to select the output. These

“why did” questions then reason backwards about the cause of the output prior to the time the output was rendered. The top of Figure 10.9 shows the Whyline’s highlighting to indicate this fact. Conversely, “why didn’t” ask about why something did not occur or change *after* a particular point in time. As in the bottom of Figure 10.9, these questions reason in a *forward* direction about code that should have executed after the selected time. These distinctions differ from the behavior of the Alice Whyline, which required the user to pause the execution of the program at the desired time. “Why did” questions in the Alice Whyline always reasoned backwards from the paused time, whereas “why didn’t questions” always reasoned globally to the whole execution history, rather than being scoped after the paused time. This was because Alice programs’ executions are often more simple, in that they involve far fewer instruction executions.

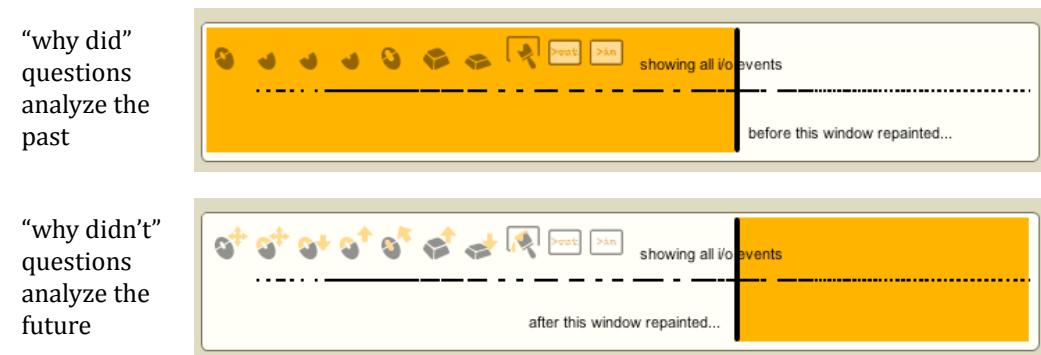


Figure 10.9. Time relatively for positively and negatively phrased questions.

The Alice Whyline (Chapter 8) and Crystal (Chapter 9) both contained global why menus. The Java Whyline does not, since there are too many events that occur within a Java program’s execution globally. Having the user first choose a particular output at a particular time is central to providing a reasonably sized question menu.

10.2.3. VIEWING CODE

When presenting answers like the one in Figure 10.1, part of the answer is the source code shown at the top of the Whyline window. The view of source is a standard syntax colored, fixed width view, but is not a source code *editor* (all of the data in a Whyline recording is immutable). Instead, it offers a number of interactive features to make it easy to read, understand, and highlight.

First and foremost, source files are broken down into *lines* and *tokens* for rendering onto the screen. This allows the Whyline to have precise control over highlighting and transparency so that specific information can be highlighted with ease. For example, the Whyline can highlight the boundaries of even the complicated method header shown in Figure 10.10, because it knows about the individual visual boundaries of each token in the source file. Figure 10.10 also shows the Whyline's support for indicating *unfamiliar files* by crosshatching the window (unfamiliarity is described in Section 10.3.6).

The screenshot shows a Java code editor with the following code:

```
6      *      class could not be found
7      * @exception InstantiationException if a new instance of the class
8      *      couldn't be created
9      * @exception IllegalAccessException if the class or initializer isn't ac-
0      * @exception UnsupportedLookAndFeelException if
1      *      <code>laf.isSupportedLookAndFeel()</code> is false
2      */
3  public static void setLookAndFeel(String className)
4      throws ClassNotFoundException,
5          InstantiationException,
6          IllegalAccessException,
7          UnsupportedLookAndFeelException
8  {
9      if ("javax.swing.plaf.metal.MetalLookAndFeel".equals(className)) {
0          // Avoid reflection for the common case of metal.
1          setLookAndFeel(new javax.swing.plaf.metal.MetalLookAndFeel());
2      }
3      else {
4          Class lnfClass = SwingUtilities.loadSystemClass(className);
5          setLookAndFeel((LookAndFeel)(lnfClass.newInstance()));
6      }
}
```

The code is annotated with several yellow highlights:

- A rectangular highlight surrounds the entire method header: `public static void setLookAndFeel(String className)`.
- A horizontal highlight spans from the opening brace of the method to the first parameter: `throws ClassNotFoundException,`.
- A horizontal highlight spans from the second parameter to the third: `InstantiationException,`.
- A horizontal highlight spans from the third parameter to the fourth: `IllegalAccessException,`.
- A horizontal highlight spans from the fourth parameter to the end of the line: `UnsupportedLookAndFeelException`.
- A vertical highlight surrounds the entire conditional block: `if ("javax.swing.plaf.metal.MetalLookAndFeel".equals(className)) {` and its corresponding closing brace: `}`.
- A horizontal highlight spans from the opening brace of the conditional block to the first line of the block: `if ("javax.swing.plaf.metal.MetalLookAndFeel".equals(className)) {`.
- A horizontal highlight spans from the first line of the block to the second line: `// Avoid reflection for the common case of metal.`.
- A horizontal highlight spans from the second line of the block to the third line: `setLookAndFeel(new javax.swing.plaf.metal.MetalLookAndFeel());`.
- A vertical highlight surrounds the entire `else` block: `else {` and its corresponding closing brace: `}`.
- A horizontal highlight spans from the opening brace of the `else` block to the first line of the block: `else {`.
- A horizontal highlight spans from the first line of the block to the second line: `Class lnfClass = SwingUtilities.loadSystemClass(className);`.
- A horizontal highlight spans from the second line of the block to the third line: `setLookAndFeel((LookAndFeel)(lnfClass.newInstance()));`.

The background of the code editor has a diagonal cross-hatch pattern where the code is located.

Figure 10.10. Token level highlighting in the Java Whyline source viewer and crosshatching over an unfamiliar source file.

When a user is mousing over a source file, there are three types of selectable content: *tokens*, *lines*, and *methods*. All identifier tokens are selectable; hovering over a non-selectable token selects the line. Hovering over any of the whitespace in the file selects the enclosing method, if there is one. These three types of selections are shown in Figure 10.11.

```
50         for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex-  
51  
52             Point one = points[pointIndex];  
53             Point two = points[pointIndex - 1];  
54             g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(), (int)  
55  
56     }  
57  
58     g.set  
59     method paint()  
60  
61     ]
```

```
50     for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex-  
51  
52         Point one = points[pointIndex];  
53         Point two = points[pointIndex - 1];  
54         g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(), (i  
55             line 54  bookmark  
56         } method paint() copy to clipboard  
57  
58         g.setStroke(oldStroke);  
59  
60     ]
```

```
49
50     for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex-
51
52         Point one = points[pointIndex];
53         Point two = points[pointIndex - 1];
54         g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(), (i
55
56     }                                method paint() ▾ show callers
57
58     g.setStroke(oldStroke);
59 }
```

Figure 10.11. Three types of selections (tokens, lines, and methods) and the menus for each shown upon clicking.

Each type of selection supports a different set of context menu commands. Clicking on an identifier shows questions about the variables current value (based on the time slider position) as well as static questions about its uses in the program (whether a field, local, or global). Clicking on a line allows bookmarking or copying the line to the clipboard. Clicking on the whitespace of a method supports questions about the methods callers and overridden method, among other commands. These commands are typically found in modern software development environments. Figure 10.11 also shows the source file view's support for revealing the method that one is in if the header is offscreen (the method `getBoundingBox` is partially visible on the top of the windows, and the Whyline shows its header at the top of the first source file view).

For each selected execution event in a Whyline visualization (discussed in the next section but also briefly in the earlier example), the Whyline automatically arranges relevant source files in the source file viewing space. For example, Figure 10.12 shows two files arranged by the Whyline automatically. Rather than have the user manually find relevant files and manually arrange them onscreen, the Whyline organizes the files to optimize the readability and highlighting of relevant information the user has selected.

```
public Rectangle getBoundingBox() {
```

PencilPaint.java

```
41    }
42
43    public void paint(Graphics2D g)
44    {
45        Stroke oldStroke = g.getStroke();
46        g.setStroke(new BasicStroke(thickness));
47        g.setColor color;
48
49        for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex-
50
51            Point one = points[pointIndex];
52            Point two = points[pointIndex - 1];
53            g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(), (i
54
55
```

PaintObject.java

```
protected int thickness;
protected Color color;

public void setColor(Color color) { this.color = color }
public void setThickness(int thickness) { this.thickness = thickness; }

public abstract double getStartX();
```

Figure 10.12. Some event selections will show multiple files, if multiple files are relevant to the selection. The example above shows both the use of the field color and the assignment to the field color, because the user has selected a question about why the field color had its current value.

10.2.4. PRESENTING ANSWERS

For every given Whyline answer, the companion to the source files is the timeline visualization. The value of the visualization can be seen from a few different perspectives. From one perspective, it organizes relevant events temporally and by thread, providing a concise view of the important execution events related to the subject of the user's question. This is the way that some people use it. Other users view the visualization merely as a temporally organized bookmarking tool, where

they can leave a trail of all of the places that they have explored in the code and easily return to those places by selecting an event in the visualization. In fact, the visualization was explicitly designed as a navigational aid, to help users understand the relationship between events that occurred at runtime and the code that caused these events.

Each visualization is structured as a sequence of events, along with a set of unexecuted instructions attached on the right. The events do not overlap horizontally (since code does not technically execute in parallel, though some processors do this at the hardware level) and are separated by thread in vertical rows. Figure 10.13 shows events occurring in three different threads, the name of each thread appearing at the left of the events that occurred in the thread.

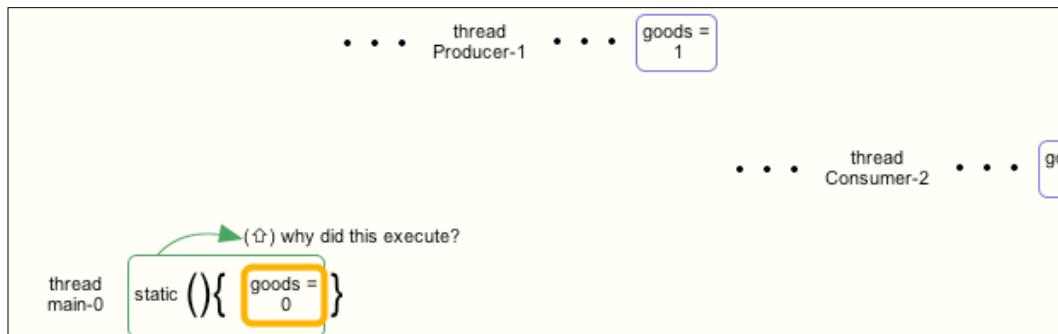


Figure 10.13. Threads separated along the y-axis.

There are several types of events that appear in a Whyline visualization, each of which has a particular color (as shown in Figure 10.14). Orange always highlights information about the current selected event in the visualization (orange also highlighting the corresponding code in the source files above the visualization). Events with green borders refer to *control dependency* events, such as method calls and branches (such as `if` statements and loops). Events with blue borders refer to *data dependency* events, such as assignments to different types of Java variables. Grey events are code that was not recorded by the Whyline and events with cross hatching (slanted vertical lines) refer to API calls (code for which editable source is unavailable). In addition to colored

||
||
||
||
||

////////// API calls ///////////

Figure 10.14. The meaning of various colors in the Java Whyline.

borders, there are certain notations used in the visualization to mimic Java syntax. For example, in Figure 10.13, parentheses are used to group arguments passed to method calls (“static” refers to a class initialization method, which has no arguments). Curly braces are used to group events that occurred within a particular method call, as in Figure 10.13; these can be nested if the visualization contains nested method calls (a call stack depth greater than one). Also, the “• • •” in the visualization indicates that other events occurred between the ones currently visible in the application. Clicking on the ellipses reveals the most recent hidden event.

There are many ways to interact with a Whyline visualization. From any given selection, clicking on any other event changes the selection and using the left and right arrow keys navigates to previous and next events shown in the visualization, if there are any. The study in Chapter 6 found that “peeking” at a control or data dependency, and then returning back to a line of code was quite common. Therefore, for every action affecting the selection in the prototype, `backspace` always goes back to the prior selection, giving users confidence that they will be able to return to their previous location after a navigation. These types of interactions will not change the visualization in any way and can be used to navigate between code and execution events that one has already explored.

For every event selection, the Whyline also shows a number of *followup questions* about the event. Figure 10.15 shows an event that refers to a reference to a `color` field, and three followup questions are shown. The first, which is in green, asks why the reference to the `color` field was executed; this is the *control dependency* of the event. Choosing this shows the conditional or method call that led to this reference (in the figure, it was the call to `paint()`, as indicated by the green arrow). The other two questions in Figure 10.15 refer to *data* that was used to execute the reference to `color`, namely the object of the field that was referred to and the value of the field itself. These are the two data dependencies of the field reference. By default, asking either of these two questions will show the *origin* of the value referred to in the question. This adds the originating event to the visualization, selects the event, and thus shows that event’s corresponding source files. The origin is essentially where the value is computed, skipping over all of the other places that the unmodified value was passed through the source code. If the user desires to follow these individual steps, they can hold `shift` and choose the question to see the *direct* data dependency, rather than the *originating* data dependency.

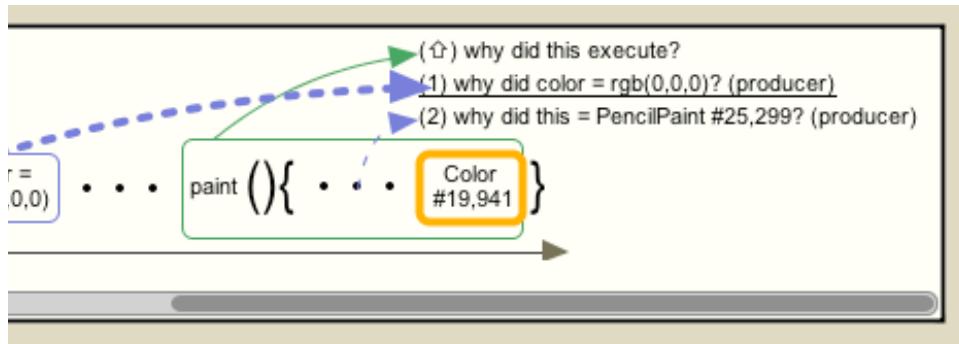


Figure 10.15. Followup questions about the selected execution event.

Aside from asking followup questions, there are several other ways to navigate from a selected event in the visualization. These are listed in Table 10.1. Many of these are supported to mimic some of the stepping commands found in a conventional breakpoint debugger. For example, `<` and `>` navigate to the previous and next event in a method, much like the “step over” command in a breakpoint debugger. The `meta-left` and `meta-right` shortcuts navigate to the previous and next event in the thread, like the “step into” command in a debugger. All of these commands will automatically add the new event to the visualization and select it. Currently there are no interaction techniques for removing an event from the visualization once its added, but this is an obvious and simple addition.

Table 10.1. Keyboard commands supported in the Whyline visualization.

key	action
<code>left</code>	go to previous event
<code>right</code>	go to next event
<code><</code>	go to previous event in method call
<code>></code>	go to next event in method call
<code>meta-left</code>	go to previous event in thread
<code>meta-right</code>	go to next event in thread
<code>up arrow</code>	go to event that caused selection to execute
<code>down arrow</code>	go to event that selection caused to execute
<code>[or backspace</code>	go to previous selection (back)
<code>]</code>	go to next selection (forward)
<code>any digit</code>	go to source of value
<code>meta + any digit</code>	go to direct producer of value
<code>escape</code>	collapse/expand sequence of events
<code>b</code>	bookmark the line of the current selection

By default, the Whyline also hides information that the user would find unfamiliar or irrelevant. For example, when first showing a causal chain, the Whyline shows causes on demand, rather than everything at once. It also collapses events that occurred in unfamiliar methods (using the same definition as in question derivation), effectively black boxing API calls and other code for which the developer has no source (Figure 10.16.1). In addition, if events from familiar code occur in methods that were called by unfamiliar methods (for example, a user-defined call back method called by an API), those events are shown, but the surrounding calling context is not (Figure 10.16.2). Both of these filtering mechanisms dramatically reduce the number of events presented in a dynamic slice (a major criticism of slicing in the past [Baowen 2005]). One could argue that it gives just the right amount of information, assuming the familiarity metric is right; after all, if everything in a slice is familiar, anything in that slice might be a candidate for a bug fix. (The results in the study discussed later in this chapter are consistent with this claim). Another form of filtering is to only include certain types of events in the causal chains. These include invocations, branches, returns, argument values, and assignments, but not uses of variables or the results of computation. These latter two are visible statically from the code, and thus redundant.

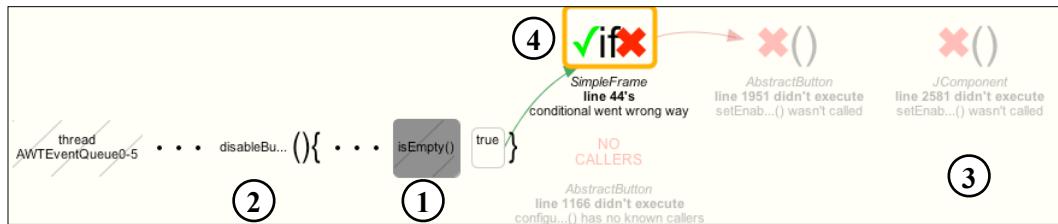


Figure 10.16. An answer showing (1) a collapsed invocation, (2) a hidden call context, (3) several instructions not executed and (4) a conditional that evaluated in the wrong direction, preventing the desired instruction from executing.

For “why didn’t” answers, the Whyline also includes instructions that were *not* executed (Figure 10.16.3). This represents a subgraph of the call graph that needed to execute for the output in question to occur. When selected, the Whyline shows the code for the unexecuted instruction, and draws arrows from the instructions that could have caused the selection to execute. The Whyline includes events when the answer includes a conditional or call that branched in the wrong direction. For example, in Figure 10.16.4, the Whyline shows that an instruction was not executed because the conditional evaluated to true.

10.2.5. OTHER WINDOWS

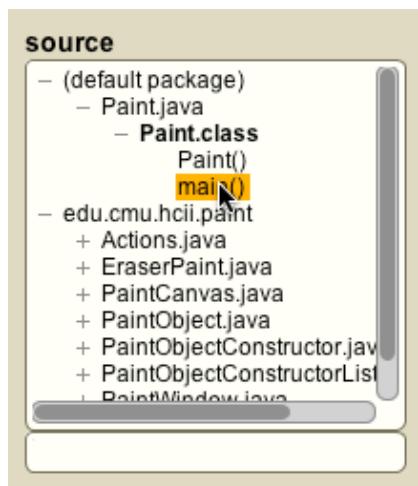


Figure 10.17. The source file outline and search field.

The Whyline window also contains a simplified view of the program output and time cursor, (Figure 10.18) allowing the user to explore the output history for reference. The orange vertical line in the timeline shows the position of the selected event in the program's overall execution history.

When showing an answer, the Whyline also provides an integrated view of each active thread, each thread's call stack, each call's local variables, and the field state for each local variable pointing to an object (Figure 10.19). The state of this view is based on the current selection in the visualization. This way, the user can easily navigate between different states in the program and see the call stack state and changes to variables and object state. There are a few buttons above this view. When a method call in the call stack is selected, the "show call" button will add the call to the visualization and select it. When a local variable is selected, the "explain" button will show the event that assigned the local variable its most recent value.

There are a number of supportive tools included in the Whyline window once an answer is shown. The source file outline, shown in Figure 10.17, shows all of the source files in the program, allowing the user to expand and collapse different Java packages and select files in order to display them in the source file area. Expanding a file shows the methods contained in the file.

The Whyline window also contains a simplified view of the program output and time cursor, (Figure 10.18) allowing the user to explore the output history for reference. The orange vertical line in the timeline shows the position of the selected event in the program's overall execution history.

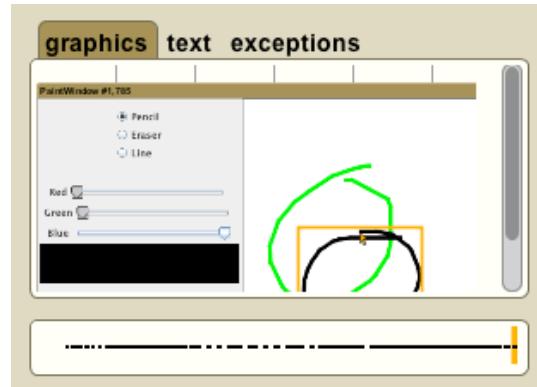


Figure 10.18. The simplified output history and time controller, which shows the position of the currently selected event in the visualization.

In addition to the call stack window, the Whyline also provides an object watch window. Users can select a local variable containing an object and press the “watch” button to add it to this list. The Whyline then updates the object state as the visualization selection changes. When an object is expanded, all of its fields current values are shown. The “previous =” and “next =” buttons show the previous and next assignments to the selected field based on the current selection in the visualization.

10.3. IMPLEMENTATION

The Whyline is intended to support interactive debugging (unlike automated debuggers, which take a specification of correctness to find potential causes of a problem [Cleve 2005]). Therefore, the Whyline needs new incremental and cache-reliant algorithms to ensure near immediate feedback for most user actions. The Whyline also takes a postmortem approach to debugging, capturing a trace [Wang 2004, Zhang 2005] and then analyzing it after the program has stopped, like modern profilers. This choice was based on evidence that bug fixing is generally a collaborative process (as noted Chapter 6), which could benefit from the ability to share executions of failures. The post-mortem approach was chosen explicitly for this purpose; an alternative design of “live” debugging could have been implemented, but would have required a different approach.

10.3.1. ARCHITECTURAL AND HISTORICAL NOTES

The Java Whyline has three major parts, all implemented in Java: the instrumentation framework, the trace representation and the user interface. As a historical note, I had the opportunity to use some of my earlier work unrelated to debugging to implement these various components, namely Citrus [Ko 2005d], a programming language designed for implementing interactive tools, and Barista [Ko

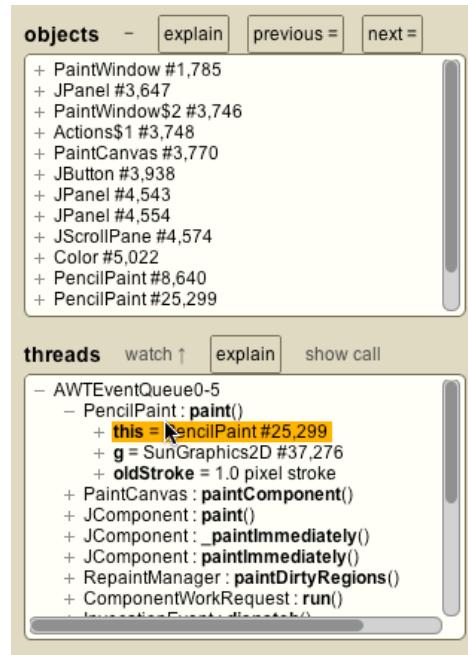


Figure 10.19. The object watch window (top) and the threads, call stacks, locals and objects (bottom).

2006c], a richly visual structured code editor. There would have been a number of benefits to implementing the Whyline the user interface with Citrus and reusing the code in Barista for visualizing Java code. In fact, I used many of the same architectural ideas in these two projects to design the Whyline, including much of the animation framework for transitioning between user interface states. Ultimately, however, the instability of these two projects, given their nature as research prototypes, precluded their use and I instead built the Java Whyline as a pure Java application from scratch.

The three major components of the Java Whyline's architecture are discussed in detail in subsequent sections, but it is worth noting their basic functions and their relationships with each other. The instrumentation and recording framework consists of an API for representing Java classfiles and a few source files that use this API to read in classfiles in their byte representation, analyze them, instrument them, and translate them back into byte arrays. I chose not to use other common bytecode APIs, such as BCEL (<http://jakarta.apache.org/bcel/>), because they had a number of error-prone properties (such as not enforcing a valid classfile structure). Instead, I wrote my own to be less error-prone. For example, upon insertion of new instructions into a bytecode method, my API automatically manages constant pool indices and checks for illegal properties specified in the Java bytecode specification, such as method length limits and rules about references to uninitialized objects. I found that these design decisions prevented a number of instrumentation mishaps by modularizing changes to fragile classfile data structures such as the constant pool.

The trace representation is designed to read in the multiple files that represent a Whyline trace from disk and represented it compactly in memory, shuttling information to and from disk on demand. The data itself is intended to be immutable, except for log information and annotations that the user might place on the immutable data. The implementation of this data structure has numerous query and predicate functions to gain access to specific static and dynamic facts about a recording, such as all of the executions of a particular method or all of the potential callers of a method. The results of these queries are typically cached to improve performance and in many cases, these caches are serialized to disk so that subsequent loads of the trace can benefit from these prior analyses. The trace representation encapsulates all of a program's source, class files and execution history into a single data structure. The data structure has detailed knowledge of

precisely how the instrumentation library records events to disk, as it needs to know how to read them from disk, but they are separate components.

The user interface part of the Java Whyline essentially provides a front end to the trace data structure, facilitating users' questions and queries by making particular queries to the trace data structure and providing usable views of the information in the trace data structure. Views of source files, execution events, call stacks, and so on, are generated on demand and discarded for optimal memory management. Many specific kinds of user interface state are serialized along with a trace, including window size and the visibility of various views, much like in any modern software development environment.

10.3.2. RECORDING PROGRAM EXECUTION

A Whyline trace of an execution consists of a number of types of information: sequences of events that occurred in each thread (many of which regard program output), all class files executed and the source files that represent them, and other types of metadata recorded to interpret the data in the trace. This section describes this information in detail.

Recording source files. Before launching the program, the Whyline scans the user specified folders for user-defined source code, copying all of the current versions of source. The directory structure of the source is maintained, whether in a platform-specific directory or a JAR file, so that the qualified name of the class defined by the source file can be recovered.

Analyzing a method for instructions to instrument. There are two major ways to capture an execution history of a Java program. One is to instrument a Java Virtual Machine to record a history of the program as it is executing. This has the advantage of having lower performance overhead, but the disadvantage of being platform and VM dependent. Instead, the Java Whyline uses bytecode instrumentation. As each Java class is loaded, the tool intercepts its byte array (using the `java.lang.instrument`¹⁶ package, standard across most JVM implementations since version 1.5), instrument

¹⁶ An alternative to instrumentation would have been to use the Java Platform Debugger Architecture (JPDA), which was not implemented for most platforms at the time of the implementation of the Java Whyline. This would allow more control over the subject program's execution, while still providing access to the same kinds of data.

each of the methods in the class, and return the modified code as a byte array to the JVM. This approach allows the prototype to work in a largely platform independent manner (although there may be inconsistent support for this particular instrumentation mechanism). The disadvantage is the complexity of inserting bytecode instructions into a Java program to capture information about its execution as well as the additional overhead of executing the instrumentation code. (A third option would have been to instrument the source, which is the most platform-independent manner of capturing a trace, but the approach with the most overhead).

The instrumentation process involves an analysis step and then an instrumentation step. The analysis step has the goal of identifying control and data instructions to instrument. Control instructions include invocations of methods, thrown exceptions, exception catches, and branch instructions (all of these instructions are part of the Java bytecode instruction set). These are straightforward to identify by simply parsing the bytecode and looking for particular opcodes. The data instructions in Java bytecode are more various, but generally involve instructions that affect the JVM operand stack (essentially arithmetic) and instructions that affect the JVM heap or local variable space (assignments to local variables, fields, globals, etc.). The Java Whyline instruments all of these latter category instructions. For the former category that only have operand stack effects, the prototype instruments only those instructions that compute values for control instructions or data instructions with heap or frame side effects. For example, in the Java assignment statement “`x = a + b + c;`” the prototype would instrument the value produced by the final addition and the assignment instruction, but not the values produced by “`a`” and “`b`.” This omission is purely for performance purposes. The value of “`a`” and “`b`” will be known from prior instrumented assignments, so recording their value at the time of use would be redundant. The one exception to this case is uses of global variables or public fields. These variables may be changed by uninstrumented code.

Of course, to know which instructions produce a value consumed by a control or assignment instruction, the prototype must first analyze the operand stack dependencies within a method. To do this, the prototype uses an algorithm that explores all execution paths through a method, and for each path, pairs instructions that push values onto the operand stack with instructions that later pop them off. (this is an algorithm similar to the verification steps performed by JVMs for security purposes.) While exploring paths, the algorithm maintains a simulated operand

stack, with each producing instruction on the stack representing the value produced. (The rules for whether an instruction produces and/or consumes a value are based on the Java bytecode specifications for each instruction.) This process determines a set of stack dependencies for each instruction in a method, allowing the system to perform a variety of analyses on the data dependencies within a method. For performance, the system caches these stack dependencies as a method attribute (method attributes are defined in Section 4.7 of the JVM specification, second edition) to make class loading and analysis more efficient when a trace is loaded for the first time.

Instrumenting a method. Once the instructions that need to be instrumented have been determined, the prototype then steps through each instruction, inserting a call to a global instrumentation method either before or after the instrumented instruction. Stack duplication instructions are also inserted if the instrumentation needs a copy of a value from the operand stack. For example, to record the result of an integer addition, `dup` instruction would be inserted to push a copy of the result onto the stack. An `invokestatic` instruction would be inserted afterwards to call the `record_int()` method, which would pop this copied result and record it to the trace file. In other cases, the instrumentation call is inserted *before* the instruction; for example, to record a thrown exception, the event must be recorded before the `throw` instruction executes.

Each instrumentation call records specific information as a prefix to any other arguments included in the type of event. Each event has a header containing the following information:

- 1 bit `switch` flag to represent whether the event is the first occurring after a thread switch. If it is set, a 32-bit serial event ID is recorded. The IDs for all subsequent events follow this ID in sequence, until the next switch. Switches are identified when reading the trace by checking whether the next event ID follows the last in a thread.
- A 1 bit `io_callstack` flag; set to true if the code represents I/O or is necessary for maintaining a call stack, which helps the trace loader know which events to process immediately.

- 6 bits to represent the event type (there are currently 55 types, such as “value produced” events for computed values, four kinds of invocations to parallel the four types of invocation instructions in Java bytecode, and so on).
- 32 bits to represent an instruction ID, consisting of two parts: a 14 bit class ID (maintained for all instrumented classes, across all programs), and an 18 bit integer represent the index of the instruction as it appears in the class file. (The largest JDK class file I have seen contains fewer than 200,000 instructions and is an outlier).

Event types include assignments, invocations and returns, thread synchronization events, exception throws and catches, instantiations of objects and arrays, and some special events to represent I/O events that are generated natively (such as mouse and keyboard events). All 55 are shown in full in Table 10.2 (with some events grouped, namely those that cover the 8 primitive Java types but with the same semantics). The studies described in Chapters 3-7 also suggest that developers find concrete values essential for interpreting program state. Therefore, unlike prior work [Baowen 2005][Wang 2005], many of these events also include a value after their header. For example, for an invocation event, the tool would record values passed as arguments to the invocation, or for an assignment, the value assigned.

Table 10.2. The 55 different kinds of events recorded by the Java Whyline. The constant, value, and argument categories contain 8 events each, to cover each of the 8 primitive types in Java. The group of six events at the bottom are custom instrumentation to capture certain I/O events.

event	purpose	event	purpose
putfield	object field assignments	throw	captured just before throw
putstatic	global variable assignments	catch	captured at beginning of catch block
setarray	array index assignments	monitor	before and after synchronized blocks
setlocal/ iinc	local variable assignments	constant	for 8 primitive types; placeholder for constant used in expression
comrefs/ comnull	reference comparison branch	value	for 8 primitive types; records value of expression or call
comints/ compzero	integer comparison branches	this	records occurrence of event, but not value of reference (to save space)
tablebranch	switch statement branches	newobject	captured after object constructor completes
invoke	the four JVM invocation instructions	newarray	captured after array instantiated
start	marker for the beginning of a method's execution, in case its call was not instrumented	argument	for 8 primitive types; captures value argument passed to method, in case call not instrumented
return	marker for just before method return		
repaint	used to mark a graphical repaint cycle	context	used to track duplications of graphics context used for rendering
mouseevent	tracks mouse input arguments	keyevent	tracks keyboard input arguments
window	tracks window size and state changes	imagesize	tracks the size of images drawn to screen for use in placeholders

In every case where the value recorded is an object, the tool obtains a unique 64-bit ID for it, creating a new one if the object has not yet been encountered. These are stored in a thread safe weak reference hash table, so that objects can be garbage collected. For each new object encountered, the tool also writes the type of the object (as a class ID) with its object ID to a separate file. Thread IDs are managed in the same way at runtime.

Special instrumentation for I/O events. Most I/O events are extracted from the regular event sequence. For example, calls to `java.awt.Graphics` are captured as `invoke` events in the trace just like any other call, and these are used to identify graphical output events and their arguments. Some I/O events benefited from or required

special support, namely the last six event types in Table 10.2. For example, the prototype replaces all calls to `java.awt.Window.getGraphics()` with a custom call, which gathers information about the size and location of `Window` instances before returning the value originally requested. The prototype also inserts custom instrumentation into the constructors of the `java.io.KeyEvent` and `java.io.MouseEvent` constructors to capture information about low-level I/O events and their parameters. These latter two were added for performance reasons; extracting them from the normal event sequence at load time would have been possible, but slower than capturing it as a custom event.

Instrumenting programs. As the prototype intercepts loading classes, it does a number of things to reduce the overhead of instrumentation on future recording sessions and to support analysis of the full executing program:

- It copies the uninstrumented version of each class in a trace folder, in case the class was loaded off a network.
- In order to keep track of code not executed (for answering “why didn’t” questions), the tool also keeps track of each class referenced by the dynamically loaded class, and just before the program halts, the tool writes each of these unexecuted class files to the same trace folder. Ideally, this would be done recursively, in order to get the complete call graph of all of the code that the program could have executed, but this would take considerable time and likely include all known classes. Tracking classes that are never loaded is important for “why didn’t” questions, since code that does not execute is often not dynamically loaded by the JVM.
- As classes load, the Whyline skips those that the user has marked to skip, as well as a number of classes that are used in the instrumentation code itself. The tool also skips methods that, once instrumented, exceed the 65,536 byte length limit imposed by the JVM.
- The Whyline caches instrumented versions of the class files and their modification date so that later executions of the target program or other programs that use the same classes can load faster by avoiding redundant instrumentation. This is particularly useful for API classes that are used by many programs.

Overall, there are four major files recorded to disk in a Java Whyline trace:

- A file declaring a list of fully qualified class names that were loaded or referenced by an program execution, along with global class identifiers for each. The names are listed in loaded order.
- An “immutables” file, which stores constant values used by the program execution. This includes all of the strings referenced, the names of threads, and custom support for common immutables, such as `java.awt.Color`.
- A source file hierarchy, as described above.
- A set of event sequences, one for each thread. Each event in the thread histories is formatted as stated above in a separate file.

When a program halts, the Whyline writes a few bits of metadata to disk as well, to note how many events were written, how many objects were instantiated, and so on, to help the loader later create reasonably sized data structures to store the information. If the program halts in such a manner that prevents this information from being captured, the trace can still be loaded, but less efficiently. A summary of the information in a Whyline recording is shown in Table 10.3.

file	purpose
metadata	object field assignments
static...	global variable assignments
call graph	constructed on the first load to speed up subsequent loads
class identifiers	unique identifiers for each loaded class, used to identify instructions in the event sequence
class names	used to find classes stored on disk that were loaded at runtime
source...	a source file hierarchy for the executed program
dynamic...	
immutables	a table of strings, colors, fonts, gradients, strokes, rectangles, and transforms, stored by object ID, used to recreate output history efficiently
objects	a history of object instantiations, stored by object ID
thread histories	a set of files, each containing a thread history

Table 10.3. The file hierarchy of a recorded Whyline trace.

10.3.3. LOADING A RECORDING

When a Whyline trace is loaded, the loader performs a number of duties to prepare for question asking. First, the source files and class files are loaded, since these are used for nearly every aspect of question asking and answering. As classes are loaded, the loader also processes several types of static information extracted from the class files:

- Associating invocation instructions with methods potentially called.
- Associating field references with field declarations.
- Associating class references with class declarations.
- Gathering all known “primitive output” instructions, which in this prototype include all calls on `java.awt.Graphics`, `java.io.PrintStream`, thrown exceptions, and exception catches. These are later analyzed to generate questions.

After loading this static information, the Whyline generates a precise call graph, using all of the invocations found in class files. *Precise* in this situation means that rather than using the type declared in the invocation instructions, the tool uses the static analysis algorithm in Figure 10.20, which scans definition-use edges to find transitively reachable new statements from the code's receiver. The result is a set of “new” instructions, which represent potential sources of new instances for the given instruction. This set is used to conservatively finds all of the *potential* types of the actual instance used in the call, and resolves the method on these types. This omits many types of infeasible calls, increasing the precision of “why didn't” answers. This algorithm is called on demand whenever the Whyline needs to identify a set of potential callers to a method. (Whenever an algorithm in this chapter refers to a “caller,” the set of potential callers is identified using this algorithm). It should be noted that many call graph construction algorithms have been proposed, which may help make the algorithm in Figure 10.20 more efficient [Grove 2001].

```

getSources(Instruction inst)

    if values for inst are cached in global table, return them
    value_producers = {}
    add value_producers to global table, with key inst      // mark inst as visited
    if inst pushes a constant, the result of an expression,
        or a new object or array
        add inst to value_producers

    else if inst manipulates operand stack           // find what inst manipulates
        add getSources(inst's value producer) to value_producers

    else if inst gets a variable                  // find variable's potential value
        if inst gets an method argument
            for each call to inst's method
                for each producer of the arg that inst gets from call
                    add getSources(producer) to value_producers
        else
            for each assignment to the variable used by inst
                add getSources(definition) to value_producers

    else if inst gets an array element          // find array's potential values
        arrayAllocations = getSources(inst's array argument)
        for each allocation of arrayAllocations
            for each assignment of getElementSources(allocation)
                for each producer of assignment's element value
                    add getSources(producer) to value_producers

    else if inst is an invocation             // find potential values returned
        for each method that the inst could invoke
            if method is native, add inst to value_producers
            else for each return_instruction of method
                for each producer of return_instruction's return value
                    add getSources(producer) to value_producers

    if instruction after inst is a type cast      // filter out illegal values
        exclude instructions from value_producers that produce types
            that do not conform to the type cast.

    return value_producers

```

```

getArraySources(Instruction newarray)

    if visited with newarray, return, otherwise, mark visited
    consumer = instruction that uses the array instance produced by newarray
    if consumer sets an array element
        include consumer as source of array element value
    else if consumer sets a local, field, or global
        for each use of the variable defined, getArraySources(use)
    else if consumer invokes a method
        for each method consumer could call
            for each use of newarray in method, getArraySources(use)
    else if consumer returns a value
        for each caller of consumer's method
            getArraySources(caller)

```

Figure 10.20. Algorithm **getSources**, which gathers instructions that could produce a value for a given instruction's argument, and **getArraySources**, which gathers instructions that could produce values for an array.

Next, the loader reads the thread traces, loading events in the order of their event IDs, switching between thread trace files as necessary using the `switch` flag in each event. This allows the Whyline to have a complete ordering of the events in the execution. As events are read, events whose `io_callstack` flag are set are processed immediately (essentially output and events needed to maintain a call stack); others are processed on demand. As call stacks are maintained, they are cached at equal intervals to provide constant time access to the call stack state at any event.

To improve the performance of question derivation and answering, the Whyline constructs tables of invocations, assignments to fields and other types of variables, and the values produced by expressions, all by event ID. All of these histories are extracted from the serial thread histories the first time a Whyline recording is loaded. Most histories are stored as integer sequences of event IDs. For example, if there were thirty calls to some method `foo()`, the event IDs for each call would be stored in a sorted list. These lists can then be efficiently searched using a binary search.

10.3.4. RECREATING AN I/O HISTORY

After loading the static and dynamic information, the final duty of the loader is to create a primitive I/O history. This step is fundamental to the Whyline's question support, since every question derived depends on the Whyline's ability to relate the pixels on the screen to the program logic responsible for them. The prototype assumes that a program uses standard Java I/O interfaces and their subclasses to produce output: `java.awt.Graphics2D` for graphical output, `java.awt.Window` to represent windows and `KeyEvent` and `MouseEvent` for input events in these windows, `java.io.Writer`, `OutputStream`, `PrintStream`, `Reader` and `InputStream` for console and file I/O, and `java.lang.Throwable` for exception output. (The Java Whyline does not record the native I/O, such as those used in some Java look and feels or in native UI toolkits such as the Simple Windowing Toolkit (SWT) used in Eclipse. This would involve instrumenting and recording code compiled for platforms other than the JVM, which was out of the scope of this implementation.)

Recreating graphical output. The Whyline derives graphical output events from the standard event sequence in a whyline recording (described in previous sections). For example, a call to `Graphics2D.drawRect()` and the events producing its

arguments are combined into an I/O event representing the rectangle drawn. This eventually produces a sequence of I/O events, which the Whyline then uses to construct a user interface for navigating the I/O history, like the one seen in Figure 10.1. To recreate this history, the Whyline iterates through each I/O event, segmenting the event sequence into repaint cycles using the `repaint` event (mentioned in Table 10.2). Within each repaint, the Whyline tracks the creation and duplication of `Graphics2D` instances, determining when and where each render event occurred on screen. A `Graphics2D` instance stores information about the current color, stroke, font, and origin, among other information, all used to determine the appearance of the next render call. Java programs duplicate these `Graphics2D` instances when painting in order to modify the render context and draw some output, without having to explicitly revert the rendering context to its previous state. Each render event is related to the `Graphics2D` instance that was responsible for rendering it. The Whyline then uses the history of modifications to these `Graphics2D` instances to determine the font, color, stroke, and so on, of each of the graphical primitives rendered.

During this process of iterating through graphical output events, the Whyline tracks the *visual occlusion* of render events. For example, if a hundred small rectangles were drawn into a buffer and then a large rectangle drawn over all of them, the hundred small rectangles would be marked “occluded” after the time of the larger rectangle’s rendering. Once this process is complete, there is enough information to render the program output for any given time in the program’s execution history. For a given time t , the Whyline finds all repaint cycles that began before t and renders all of the render events in each repaint cycle until reaching a render event that occurred after t . The Whyline uses the visual occlusion information to skip render events that are not visible at t .

In order to allow users to point and click on render events, the Whyline uses a basic picking algorithm. Given a point p , the Whyline first finds which window contains p . Then, it searches through all of the visible render events in the window, gathering a bottom-to-top list of render events that contain p . This list of events is used to generate a list of questions about the top most render event (i.e., a text label and a background rectangle), as well as the objects that the list of render events represent (i.e., the button represented by the text and rectangle). This process is described in the next section. However, for each item in the list, the Whyline also has to

determine the *original renderer* of the output. This is because render events can be drawn into arbitrary image buffers and these buffers can then be rendered into other buffers (this includes double-buffering, which is used to ensure that whole screens are drawn at once to a physical display rather than pieces at a time, avoiding a flickered appearance). For example, to paint a gradient for a button in a UI, which can be an expensive operation, some systems will render part of the button's gradient into a buffer, then quickly paint the buffer at multiple locations to "tile" the output. Such techniques are now ubiquitous in modern operating systems, meaning that any Whyline that supports graphical output needs mature support for tracking which buffer a graphical primitive is drawn to. What makes such support challenging is that a single event can be rendered into a buffer, but the buffer can be drawn multiple times and multiple locations into other buffers. This means that a single event can have effects on multiple places in the screen. Therefore, when mapping a user's mouse cursor to the graphical primitives underneath the cursor, the system must distinguish between the *render time* of the event and one or more *appearance times* in which the rendered output appeared in the physical display. Therefore, a single user click in the graphical output refers to a list of render event pairs, which is then processed to create a list of questions.

Recreating console output. Console output (seen in Figure 10.21) and exception output are relatively straightforward to create, as it is just a list of strings and exceptions to display as a vertical list. Textual output events are extracted from the standard event sequence by watching for calls on `java.io.PrintStream`. Each textual output event, rather than referring just to the string printed by the print stream (as in the string in `System.out.println("message =" + message)`), refers to each individual argument used to concatenate the final argument. This way, the Whyline can support questions about both the string "`message =`" and the string variable `message`, independently. Once these are extracted, the sequence of textual output event is processed in order of execution, and text printed to the console is laid out onto a single line, advancing a single line on the occurrence of each '`\n`' character. Clicking on a string in the console output generally results in a single question "`why did this get printed?`".

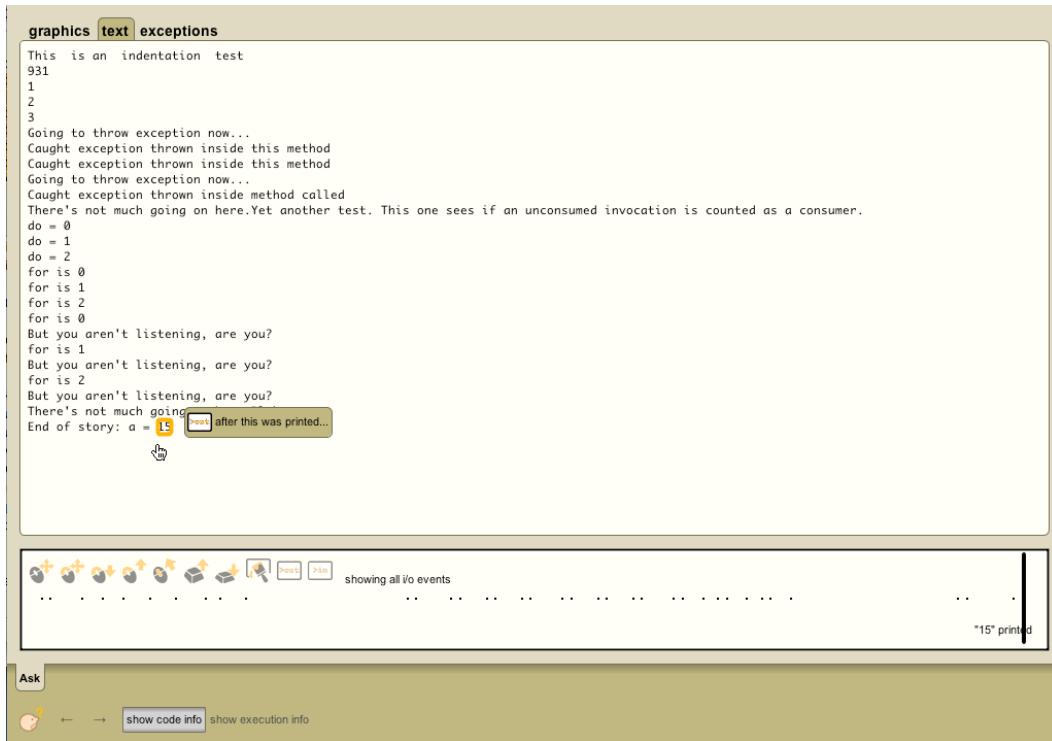


Figure 10.21. The textual output question user interface.

10.3.5. DERIVING QUESTIONS

In any program execution, many things happen, and many things do not. The Whyline uses both static and dynamic analyses to derive questions about these behaviors that the developer may or may not have expected.

“Why did” questions refer to a specific event from a trace; the questions available for asking depend on the input time selected by the user (Figure 10.1b), since this time also determines what events are visible on screen (this differs from the Alice version, which required the user to pause the program at the time desired). When the user clicks on an output event, the Whyline shows questions related to the output event selected. For example, in Figure 10.6, “why did” questions relate to the properties of the rectangle the user has selected.

In addition to questions about output primitives, it is also helpful to have questions about higher level concepts that these primitives *represent* (for example, in addition

to questions about rectangles, also supporting questions about the button that is drawn using the rectangle). The Java Whyline supports questions about two types of higher level entities. The first kind of entities are fields of Java objects that indirectly influence an output primitive's arguments—these are *data* that affect primitive output¹⁷. For example, imagine a drop down menu with a list of items; it is important to not only be able to ask about the individual items, but also about the list itself. The prototype follows all upstream dynamic control and data dependencies of the primitive output's arguments to identify fields that affected the primitive output's arguments, stopping when finding no more upstream dependencies. This amounts to a backwards dynamic slice, tuned to gather field references. The size of this list can be relatively large, since there are typically many upstream data dependencies, but the entities are organized by Java class, making the list easier to navigate.

The second kind of higher level question regards entities responsible for *indirectly rendering* low-level output—these are *callers* that affect primitive output. For example, when clicking on the label of the button, the user may also want to ask about the button itself and its properties, such as its visibility, enabled state, and so on. These objects are found on the call stack of the invocation that rendered the primitive output, and all such objects are included in the list (with the exception of those filtered out as described in the next section).

10.3.6. FILTERING BY FAMILIARITY

A fundamental problem for both kinds of higher level question is *filtering* the menus by *familiarity*. It is important to include only objects that are relevant to output and that the user is likely to have created or used, since questions about unfamiliar classes or data structures will not likely seem relevant to the user. For example, in the Swing UI toolkit, the `Button` class does not know how to draw itself. This is delegated to a `ButtonUI` class, which renders the button. A developer may write code to instantiate a `Button`, but have no idea about the existence of its look and feel delegate. The same is true of Swing's `ButtonModel`, which is a helper class for storing a button's pressed state. To avoid presenting questions about these types of delegate and helper classes, the Whyline defines a notion of *familiarity*. A class is *familiar* if user-owned code either *defines* or *references* the specific class. In the prototype,

¹⁷ These types of questions were not originally included in the Java Whyline design; the need for such questions became apparent after piloting the design of the study in Section 10.4.4.

user-owned code consists of those classes that were derived from source on the last compile (thus excluding APIs and libraries for which the developer has no source). One could imagine more sophisticated definitions for familiarity and ownership based on authorship, checkins, or other measures.

This notion of familiarity is used to filter the two types higher-level questions about data and callers. For callers, the Whyline inspects the call stack of the invocation that produced the selected output primitive and for each call stack entry that represents a call on an object and only includes questions about that object if the object is of a familiar type. This causes the tool to include questions about `Buttons`, but not `ButtonUIs`, unless the user had directly referenced `ButtonUI` in their code. To filter questions about data, the Whyline only includes questions about *familiar* data structure classes, thus excluding helper classes such as `ButtonModel`.

10.3.7. FILTERING BY OUTPUT AFFECTING

Another way to filter question menus is to exclude code structures that do not affect the modality of output in question. For example, if a user is asking about *graphical* output, it can exclude questions about fields, methods, and classes that only indirectly affect *textual* output. To accomplish this, the Whyline finds fields and invocations that could have *affected* each known output instruction, using the first algorithm in Figure 10.22. For example, the color used to draw a rectangle might be affected by some field in an object, or by the return value of a call to some method. To find these fields and invocations, the algorithm follows upstream static data dependencies, marking fields and methods as “output affecting” along the way, keeping track of the *modality* of the output. This way, methods and fields are marked as affecting graphical output, textual output, or other types.

Next, if the output instruction directly *invokes* output (such as *drawing* a rectangle, unlike setting the color, which merely *affects* appearance), all potential indirect callers to the output instructions method are marked as *output invoking*. This is done by following potential callers of a method, starting with the output instruction’s method (bottom of Figure 10.22). Each algorithm is run on each primitive output instruction, and halts either when reaching an instruction already visited for a particular modality or code with no dependencies.

One detail not mentioned in the algorithms in Figure 10.22 is in how the algorithm traverses potential callers of methods. Aside from the precise call graph mentioned earlier in this chapter, the algorithm also tracks the class of the method that the propagation begins in, and remembers this class during the traversal of potential callers. For example, if the propagation started in a method of the `javax.swing.JButton` class, then later arrived at some call on a `java.awt.Component` (a superclass of `JButton`) and finally ended up in a method of `javax.swing.JComboBox` (which is *not* a subclass of `JButton`), the algorithm would know not to follow any calls on the `JComboBox`, because the original source of output was a `JButton`. This allows the algorithm to exclude infeasible calls as part of the call graph traversal, by propagating the class that originated the output.

```
markAffectors(Instruction inst)

  if instruction been visited, return, otherwise, mark inst as visited

  if instruction acquires a field value      // mark assignments to fields
    mark field as affecting instruction
    for each definition of field, markAffectors(definition)

  else if instruction is an invocation      // mark data used by return statements
    for each method potentially called by instruction
      mark method as affecting instruction
      for each return instruction in method, markAffectors(return)

  for each control dependency of instruction // mark code causing instruction to execute
  markAffectors(control)

  for each instruction data that instruction is data dependent on
  markAffectors(data)



---


markInvokers(Instruction inst)

  if instruction has not been visited      // mark callers to method of instruction
    mark instruction as visited
    mark instruction's method as invoking inst
    for each caller of instruction's method, markInvokers(caller)
```

Figure 10.22. Algorithms `markAffectors` and `markInvokers`, which mark methods and fields that affect or invoke output (the two algorithms do not invoke each other).

Intuitively, it would seem these algorithms mark everything; after all, what code is not responsible for affecting or invoking some output? The insight is that particular code is responsible for particular output: because the algorithms in Figure 10.22 are run on each output instruction, the Whyline knows for what output a particular field or method is output-affecting. The Whyline can then use this knowledge to generate and filter questions based on what output the user expresses interest in.

10.3.8. CREATING QUESTIONS

Once the Whyline identifies each entity represented by the selected output primitive, the Whyline generates questions for each entity. These include “why did” questions about each of the familiar, output affecting fields’ current values, such as “why did this Button’s visible = true?” (Figure 10.23.2) and also “why didn’t” questions about why these fields were not assigned after the selected time (Figure 10.23.3). Each of these questions points to the most recent assignment to the field on that instance. The Whyline also generates questions about objects that indirectly invoked the selected output primitive, including questions about the creation of the object (Figure 10.23.4), about the objects output-affecting fields, and about output-invoking methods that the user believes did not execute (Figure 10.23.5).

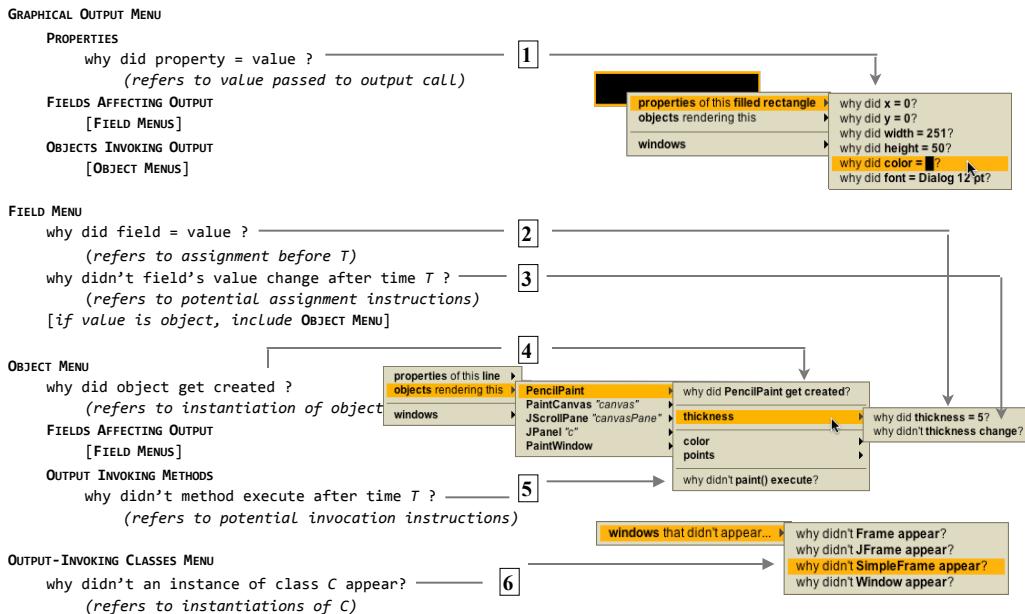


Figure 10.23. All supported questions for a graphical output event in the Java Whyline prototype, showing six types of questions currently supported by the prototype (numbered 1-6) and three types of menus. For each, the content on the left lists the meaning of the question (items in []'s represent nested menus of the specified type) and the content on the right gives an example screen shot.

The actual phrasing and presentation of questions depends on the type of output. Exceptions thrown by the program, caught or uncaught, are phrased as “why did” questions, and map to a throw event. Output in the console history supports questions about why a particular string was printed (mapping to the event that produced it). The questions supported for graphical output are somewhat more

diverse, because the output itself is more complex in nature. For primitive level output, such as a line, circle, or rectangle, users may ask “why did” questions about any of the properties used to render the output. These correspond to arguments passed to the render method, such as position and size, as well as state in the `Graphics2D` object such as color and font.

“Why didn’t” questions refer to one or more instructions in the code. The prototype currently supports three types of “why didn’t” questions:

- Questions about *output-affecting fields*, such as “why didn’t this `Button`’s `hidden` field change?”,
- Questions about *output-affecting methods*, such as “why didn’t this `Button` `repaint()`?”, and
- Questions about *output-affecting classes*, such as “why didn’t a `Button` appear?”

For “why didn’t” questions about fields, there are two types. For discrete-valued variables such as booleans or enumerated types as well as constant-valued objects, the system can identify specific values for “why didn’t” questions. For example, one might ask “Why didn’t the filled rectangle’s `color = red`? ” if the program referred to the constant `Color.red`; these values are found by following upstream data dependencies until reaching constant values. For continuous-valued variables (integers, floats, etc.), this is usually is not feasible; for these variables, the system instead supports questions of the form “why didn’t the variable get assigned?” For both kinds of questions, there may be numerous places that could have caused a variable to be assigned, these questions refer to the set of potential assignment statements. These instructions are grouped into a single question to avoid user speculation about which particular source should have executed; instead, all of them are considered together. For “why didn’t” questions about methods, the system analyzes all of the potential callers of the subject method.

The last type of why didn’t question supports questions about output that has no representative output to click on. For example, a user might have expected a dialog box to appear after a certain input event, or a console string after a certain action; there are no primitives to choose that would enable questions about such output. To support these, the Whyline includes a question for each familiar class that has output invoking methods, inherited or declared. An example of the resulting menu is

seen in Figure 10.23.6, showing several different types of windows that could have appeared (the prototype does not yet add questions about non-window classes, but this is a straightforward addition).

There are a few types of questions that the Whyline does not support. First, it does not support questions about concrete values of continuous variables (“why didn’t $x = 6.08$?”). This is partly because answering such questions can be computationally expensive and that such answers can pose too many possible reasons to be useful. Moreover, developers do not often know precisely what value to expect. Rather, they might guess that $x > 0$ and around 5; such specifications would require new algorithms to support. As an alternative to these kinds of questions, the Whyline allows users to ask “why didn’t x change?” questions or to simply ask the positively phrased version of the same question to find the source x ’s value.

One other type of unsupported question is about the effects of input, such as “why didn’t this drag event do anything?” The problem with such “forward looking” questions is that even in very simple situations, input events have many effects on programs and with no expectation provided to the system, there is no way to filter all of these things that do occur. Instead, users can ask the “why didn’t” questions discussed throughout this chapter, which inquire about some expected output *after* an input event. This is an unfortunate but necessary restriction. Future work might investigate techniques for adding forward reasoning questions (one place to start might be the Garnet toolkit’s limited support for forward reasoning “why didn’t” questions about input events [Myers 1990]).

The Alice Whyline (Chapter 8) and Crystal (Chapter 9) both contained global why menus. The Java Whyline does not, largely because there are far too many events that do and do not occur in a Java program globally. If there were to be a menu, there would be too few constraints on what appear in the menu. Having the user choose a particular output at a particular time is central to providing a reasonably sized question menu.

10.3.9. ANSWERING “WHY DID” QUESTIONS

Although there are a variety of types of why did questions, each maps to an event from the program’s execution history. The approach to answering these is to work backwards from the event to find its chain of causality. Dynamic slicing techniques

[Baowen 2005], which use a concept of control [Cooper 2001] and data dependencies, are a natural approach to constructing these chains. However, the typical approach of dynamic slicing algorithms is to generate a set of instructions and present them to the user as a set of highlighted lines of code. Given the data from the studies discussed in prior chapters, this seems less helpful, therefore the Whyline produces a visualization of events, temporally sequenced. This visualization is the tree of events that are traversed in a typical backwards dynamic slicing algorithm. Although the algorithm is essentially the same [Baowen 2005], the difference in data structures affect how the information is presented to the user: a chain of events shows what happened at runtime temporally, whereas a set of instructions simply states dependencies, many of which a user might already know. Furthermore, each event's control and data dependencies are computed *on demand* when a user selects an event. For example, rather than have the algorithm automatically traverse all of the data dependencies in a slice, the user explicitly chooses data dependencies to navigate in the form of “followup questions” (Figure 10.15), much like the interaction in the recently proposed *thin slicing* [Sridharan 2007]. This means that answers are produced almost immediately, making slicing time largely moot, unlike previous slicing systems, which process answers in full as a batch process [Baowen 2005].

Some “why did” questions use some pre- and post-processing to increase the utility of the answers. For example, when answering a question about an argument value passed to a method, the system first finds the “source” of the value, by default. The source essentially follows data dependencies backwards until reaching a data dependency with multiple incoming dependencies (not counting control dependencies). To be concrete, imagine a color is instantiated in a call and then the color is passed, unmodified, through a dozen other calls until it is finally used. When asking about this color (the “followup questions” about data dependencies mentioned in Section 10.3), the system follows these data passing dependencies backwards until reaching the source, which might be an instantiation, an expression, or the return value of an unrecorded method. The assumption that this analysis makes is that the *calls* made to pass such data are not buggy, but that the data itself is buggy. If this assumption is not true, the analysis will skip over the buggy code. For example, perhaps the color was obtained from the wrong call. To account for this, the system also allows users to follow direct data dependencies and avoid skipping these potentially erroneous intermediate steps.

10.3.10. ANSWERING “WHY DIDN’T” QUESTIONS

“Why did” questions analyze an event by searching *backwards* in the history at a certain time. “Why didn’t” questions analyze one or more *potentially unexecuted* instructions *forward* from the I/O event the user has selected using the time cursor. A “why didn’t” query thus consists of one or more instructions, a time, and in addition, one or more constraints on the expected conditions of the given instructions’ execution (which will be discussed shortly). For example, imagine a question about a button’s `enabled` field; there may be three places this `enabled` field could be assigned. Each potential assignment is analyzed individually, generating individual answers. These answers are then combined into a final single answer.

To explain each individual instruction, the Whyline uses two analyses: (1) determining why an instruction was not executed, and (2) determining why a particular dynamic data dependency did not occur. Each of these is constrained by two types of scope. *Temporal scope* affects what events it considers. For example, a developer may ask about something that did not occur after a specific event, but may have occurred in other situations. Therefore, “why didn’t” analyses only search through events that occurred *after* the event selected by the time cursor (see Figure 10.9) and before the end of the program. This omits other executions of events and reduces the amount of information to process. The tool could have supported scopes that end at a time different than the end of the program, but developers are notoriously bad at predicting precisely when something should have happened in the future; including the whole scope ensures that they make no false assumptions. Developers are fine at knowing the time after which something should happen, since most things happen as the result of a user action. *Identity scope* is the second kind of scope, which considers what object(s) the developer has expressed interest in. For example, if they have asked why the “hidden” field of a button did not change, the analyses are restricted to events on that specific button instance. This *calling constraint* is propagated through the algorithms discussed next.

Why was this instruction not executed? To explain why an instruction was not executed, the first thing the Whyline does is check if it did execute. The studies in Chapter 4 and 5 found that developers are often prone to misperceiving output, and believe something has occurred when it has not (for example, believing that something did not change color, when it did, but then changed back). By supporting “why didn’t” questions about things that did happen, the Whyline can reveal these

assumptions. If the instruction did *not* execute, the Whyline uses an algorithm (called `whynotreached`) to explain why not. If the method of the instruction being analyzed was not executed after the specified time, there are few possible reasons:

- It has no known callers (the tool has to say “known callers” because a call may exist, but its class may not have been loaded at runtime).
- A caller of the instruction’s method *did* execute, but on a different instance (if the instance is relevant).
- None of the method’s callers executed; the algorithm then recursively explains why each potential caller was not reached.

If the instruction’s method *was* executed, there are many possible reasons why the instruction was not reached:

- A caught exception jumped over the instruction of interest, or the method exited because of an uncaught exception.
- None of the instruction’s control dependencies executed (such as an if or switch); the algorithm recursively explains why none of these control dependencies executed (there is typically only one control dependency, except in certain exception handling situations).
- One of the instruction’s control dependencies did execute, but jumped to the wrong target, skipping over the instruction.
- The method executed, but the instruction of interest had not yet (or did not because the thread or program halted).

For most questions, there are one or more objects of interest (for example, the button clicked on or the button represented by a rectangle). In the algorithm above, if a call to a method is found, it is only analyzed if it executed using the object of interest as the instance invoked on or as an argument. For example, if the user has asked why a method did not execute on a particular button, and some upstream caller did execute, the algorithm checks to see if the specific button was referenced. As the algorithm traverses calls, the local variable that would reference the object of interest is tracked through invocations (for example, in one call the object may be the instance, and then inside the method, the instance may be passed as an argument; the algorithm tracks the flow of the object through the calls). If at any point the local variable is not a method argument, the algorithm stops tracking

identity, since there is no where else to back-propagate the calling constraint. In this case, the algorithm continues analyzing callers independent of the object of interest.

The result of the algorithm is a directed graph (not a tree, because of recursion), with nodes consisting of invocations and conditional instructions that were not reached. Nodes that involve an invocation on a different object or a conditional branching in the wrong direction also have a causal chain of events attached, explaining the source of the wrong object, or the values of the conditional's expression, respectively. When a question refers to multiple potentially unexecuted instructions, a single answer containing the union of these graphs is presented.

Why was the wrong value used? Questions that ask about potential values of fields or primitive properties compare the expected dynamic dependency path to the actual dynamic dependency path at runtime. The former is obtained by tracking the path followed by `getSources` in Figure 10.20. the latter comes from the dynamic slice on the event that actually occurred, whether it was a field assignment or argument of an output instruction. (These are lists because the algorithm only analyzes unmodified values passed through intermediaries.) To illustrate, consider the following code, which controls a text field's background based on various state.

```

draw()
1 color = getBack()
2 setColor(color)
3 fillRect()

4 setBack(newColor) color = newColor
5 getBack() return color
                                         determineColor()
6 if(invalid)
7   if(enabled)
8     setBack(red)
9   else
10    setBack(gray)
11 if(override)
12  setBack(black)

```

Imagine that the user expected the background to be red (line 8) and expected a question “why didn't this `TextField`'s color = red?” The expected dependency path from 2 would be `2,1,5,4,8`. Then imagine that instead, the background was gray (line 10), with actual dependency path `2,1,5,4,10`, or `black`, with path `2,1,5,4,12`. In both cases, the point of deviation was 4: the program called `setBack()` with some color other than red. To explain why, the Whyline then checks if the expected line (8) did execute. If it did and the other call to `setBack()` occurred after, then the color was overridden. If line 8 did *not* execute, then the tool uses the `whynotreached` algorithm to determine why the instruction did not execute (in this example, it would be because enabled and/or invalid were false, or `determineColor()` was not called). This algorithm is shown in Figure 10.24.

```

whynotvalue(List of instructions expected, List of events actual)

co-iterate through expected and actual, comparing
instructions and finding point of deviation

if deviation was not found, reason = value was used
let exp be instruction after deviation in expected
let act be event after deviation in actual

if exp executed within temporal scope
  reason = value was used, but then overriden
else
  whynotreached(exp)           // Algorithm described in text above

```

Figure 10.24. Algorithm `whynotvalue`, which explains why a certain dynamic data dependency did not occur.

10.4. EVALUATION

This section describes five evaluations, investigating performance characteristics of the Java Whyline, question coverage, and the effectiveness of the tool with novice and skilled Java programmers, as well as longitudinal use of the Whyline.

10.4.1. PERFORMANCE EVALUATION

In order to test the performance feasibility of the Whyline on modern hardware, four aspects of Whyline traces were tested empirically: *slow down* (comparing normal running time to tracing time, as well as to profiling time), *trace size*, *compressed trace size*, and *trace loading time*. The programs tested included five open source projects of various sizes and complexity including a binary clock (`binclock`), a command line HTML formatter (`jTidy`), a Java compiler (`javac`), a text editor (`jEdit`), and a diagramming tool (`ArgoUML`). For each, the test case listed in Table 10.4 was run without tracing, with Whyline tracing (classes pre-cached), and with a commercial profiler tracing (YourKit Java Profiler version 7.0, <http://www.yourkit.com>). Table 10.4 lists the resulting size of the Whyline trace (in terms of number of events and disk size). Each trace's folder of files was compressed into a `ZIP` archive using the standard `DEFLATE` algorithm. Finally, the loading time for each trace was recorded. All tests were run on a 2GHz Intel Core Duo MacBook Pro with 2GB of RAM, using the standard OS X JVM, given a 1 GB heap. Time was measured to the tenth of a second using the Unix `time` command (on OS X), and reported at one-second precision. All tests were run five times and the averages are reported in all cases.

Program	LOC	Test case	Execution time (sec)			Slowdown (ratio)			total # of events	Trace Size (mb)			Loading events/sec
			normal	YourKit	Whyline	normal	YourKit	Whyline		original	zip	% original (sec)	
Binclock	177	Run clock for five seconds	5.7	9.1	9.8	1.6	1.7	140.268	4.7	1.6	34%	2.5	56,107
jTidy	12,258	Clean html of NY Times front page	0.9	3.9	13.8	4.3	15.3	16,504.866	118.1	13.7	12%	13	1,269,605
JEdit	66,403	Load, open file, type "Goodbye", quit	8.4	11.7	60.1	1.4	7.2	8,983,890	84.5	15.0	18%	17.5	513,365
javac	54,054	Compile 2,810 line Java source file.	2.0	3.7	17.0	1.9	8.5	35,193,667	283.6	40.2	14%	46.5	756,853
ArgoUML	113,117	Load to splash screen and quit	5.60	15.00	28.6	2.7	5.1	18,303,691	137.60	17.9	13%	14.20	1,288,992

Table 10.4. Statistics about tracing slow down, trace size with and without compression, and trace loading time, on five open source Java programs, averaged over ten runs. The profiling times were computed using the YourKit Java profiler with tracing mode on (rather than sampling). Lines of code for each program were computed omitting whitespace lines.

As the results show, the Whyline's tracing time is slower than the profiler. Once optimized, this should improve considerably. Trace sizes, especially compressed, compare favorably to those reported in dynamic slicing work [Baowen 2005] and this is without using the run length encoding to compress loops, as reported in [Wang 2004]. It is also clear that trace size depends less on the program complexity and more on the nature of the output. Command line programs that batch-process data have a higher rate of instruction execution than GUI applications. This is likely due to users' idle time in GUIs. Loading time is also an issue. Although several optimizations have been implemented, the biggest limiting factor during loading is memory. In the larger traces, there were delays due to garbage collection and virtual memory, which could be avoided with better memory management in the prototype.

10.4.2. QUESTION COVERAGE

Another aspect to consider is the degree to which users would be able to find a question that matches the question they *want* to ask. Of course, this is difficult to measure, as there are no complete classifications of the questions that people can ask about program output. The approach was to randomly sample three bug reports on three of the applications in Table 10.4, and check to see (1) whether any question seemed like a reasonable translation of the problem specified by the report, and (2) if so, how much translation was required. Of the nine reports randomly sampled (Table 10.5), all but one had a suitable question. Two questions were about console output, five were about primitive graphical output, and one was about an exception.

program	bug report title - description	whyline question
jTidy	Again DOM Parsing error - <i>[error message listed in report]</i>	Why did <i>[error message]</i> print?
	JTidy allows duplicate ID attributes - If you give the same ID value, should cause error...	-
	JTidy locks up in a never ending loop - it locks up with this...	Why didn't <i>[success message]</i> get printed?
jEdit	soft wrap, cut and null-pointer exception - This results in a BeanShell error dialog...	Why did this text = <i>[error dialog text]</i> ?
	File Open/Save dialog's directory - File/Save dialog should start in the directory last selected	Why did this text = <i>[current folder name]</i> ?
	Invalid screen line count - java.lang.RuntimeException: Invalid screen line count: 0...	Why did <i>[exception thrown]</i> occur?
	ArgoUML Autoresize triggers at wrong times - stretch any class to greater than it required size...	Why did this <i>[class's]</i> rectangle width = <i>[wrong size]</i> ?
ArgoUML	Invisible FigNodes are being saved - software just displays error and doesn't open project	Why did this text = <i>[error dialog text]</i> ?
	Can not parse import statement after javadoc comment - unexpected token "import" ...	Why did this text = <i>[error dialog text]</i> ?

Table 10.5. Nine bug reports and the Whyline questions that could be asked.

The one report for which there was no suitable question (the 2nd jTidy report) was a request for an unsupported feature, and so there was no obvious question available. The Whyline might still be useful to help find where to add this functionality. It was also clear that the Whyline required some translation of the bug reports into questions. In all cases, the difficulty in such translation was in finding a suitable subject for the question (listed in []'s in Table 10.5). It seems that the more difficult it was to find a subject, the more distant the Whyline's answer was from the cause. Of course, these tests do not show whether a person would find the question or whether the answer would make sense. It does provide a best case.

10.4.3. NOVICES WITH THE WHYLINE VERSUS SKILLED DEVELOPERS WITHOUT

As a pilot evaluation of the Whyline's utility, 9 people worked on the Paint slider bug (described at the beginning of this chapter) with the Whyline. These participants had a variety of backgrounds, with the least experienced having never seen a line of code and the most having programmed for more than a decade. The participants' backgrounds were in psychology, design, computer science, linguistics, food science, and engineering. These people's task performance was compared with that of 18 self-described expert Java developers from the study in Chapter 6, who used Eclipse 2.1 (in that study, participants were interrupted about every three minutes, but this time was removed from the analyses here).

Each of the participants worked through a 1-2 minute tutorial about how to use the Whyline, including information on how to ask questions and how to follow data dependencies, and then were shown the Paint program and the blue slider's incorrect behavior. Participants were then asked to find the cause of the behavior and tell the experimenter when they thought they had found it. As they worked, participants were allowed to ask questions about the user interface, but not about the task or code (the focus was on utility, and not on usability problems). For example, many participants asked, "what do these numbers mean again?" referring to the data dependency labels in Figure 10.1e). The experimenter also offered clarifications when the user expressed confusion about the user interface.

Overall, the participants with the Whyline completed the task in a median of 4 minutes, ranging from 1 to 12, significantly faster than the control group, which had a median of 10 minutes, ranging from 3 to 38 ($p < .05$, Wilcoxon rank sums test). The Whyline participants were more than *twice* as fast as the skilled developers without

the Whyline. This is despite the fact that most of the Whyline users were self-described novices and that many of the developers in the control condition had already spent time understanding the design of the application. In fact, in this pilot study, the novices tended to outperform the skilled developers for some interesting reasons. The novices tended to say aloud, “Why is the line black?” and then use the Whyline to ask that question directly, quickly finding the cause. One novice said that “It was like a treasure hunt! It was fun! I didn’t know debugging was like this.” The skilled developers asked the same question, but then rather than proceeding to ask it with the Whyline, speculated about the possible reasons (e.g., “Why didn’t this slider’s event get handled?”), and then looked for a question that allowed them to check their speculation. When they failed to find such a question, only then did they ask about the color. One skilled developer explained that they did not “expect the Whyline to be able to make the connection between the slider and the color” and so they thought they had to make the connection themselves. This led to a number of changes to the presentation of the data dependencies to make them appear as “followup question,” as shown in the examples throughout this chapter.

Another issue found in this early testing was the lack of support for questions about *data* that indirectly influence primitive output. At the time of this study, the Whyline only included higher-level questions about objects that indirectly rendered graphical output, but not questions about data that influenced this rendering. After discovering this, these questions were then added to the Whyline and turned out to be a natural companion to the more control dependency oriented questions already supported. As part of this change, the top-level “why” menu no longer contained the list of objects under the mouse, but instead three top-level menus about the primitive output, data affecting the primitive output, and objects drawing the primitive output.

10.4.4. SKILLED DEVELOPERS WITH THE WHYLINE VERSUS WITHOUT

In this more formal study, the goal was to compare skilled Java programmers using the Whyline to similarly skilled Java programmers using conventional debugging tools. The study had a between-subjects experimental design, with the independent variable of “debugging approach” and dependent variables of task completion time and task success. The goal was to determine whether the Whyline would significantly impact success at program understanding compared to modern

debugging tools. To increase confidence that any observed differences were due to the experimental factor, the group that used conventional debugging tools used a version of a breakpoint debugger that was built on the same platform as the Whyline for Java. This way, they had comparable user interfaces and performance characteristics, with the only differences in the type of debugging support. The control group could set breakpoints and step through code, like any other debugger, but neither condition was allowed to edit code. This also meant that the control condition could not insert arbitrary print statements (but they were able to insert pseudo print statements that had no side effects). The Whyline group, in contrast, did *not* have access to breakpoint features, since the study was focusing on the effectiveness of each approach and not on which type of support developers would choose.

Participants. The sample of study participants consisted of 10 people in each group for a total of 20. Participants were all students in a masters program in software engineering, but had a median of 1.5 years of industry software development experience before coming back to school, ranging from 0-10. (The one developer who reported an industry experience of zero had worked on several large projects in an industry setting, but was never paid for the work and thus did not count it as industry experience). All rated conventional breakpoint debuggers as “important” to their work or higher on a scale of “useless” to “essential.” The participants also rated themselves with average or higher Java expertise on a scale of “beginner” to “expert”. There were no statistically significant differences in these measures between the two conditions.

Tasks. Participants worked on two tasks adapted from real bug reports of ArgoUML, which is a 150,000 line open source Java application for designing Java applications themselves using UML diagrams (it is also listed in Table 10.4). The task descriptions given to participants appear in the Appendix.

The first bug (shown in Figure 10.25) involved removing a particular checkbox from the user interface. The typical strategy of search for the label of the checkbox in the source code did not work well in this task because the application used localized strings for different languages that were stored in a compressed file on disk. The checkbox label did appear in the command line help, and the command line help text did appear in the source files, so if one formulated a search with part of the

checkbox label, one could make the connection between the two and eventually find the right source file, but few made this connection in the actual study.

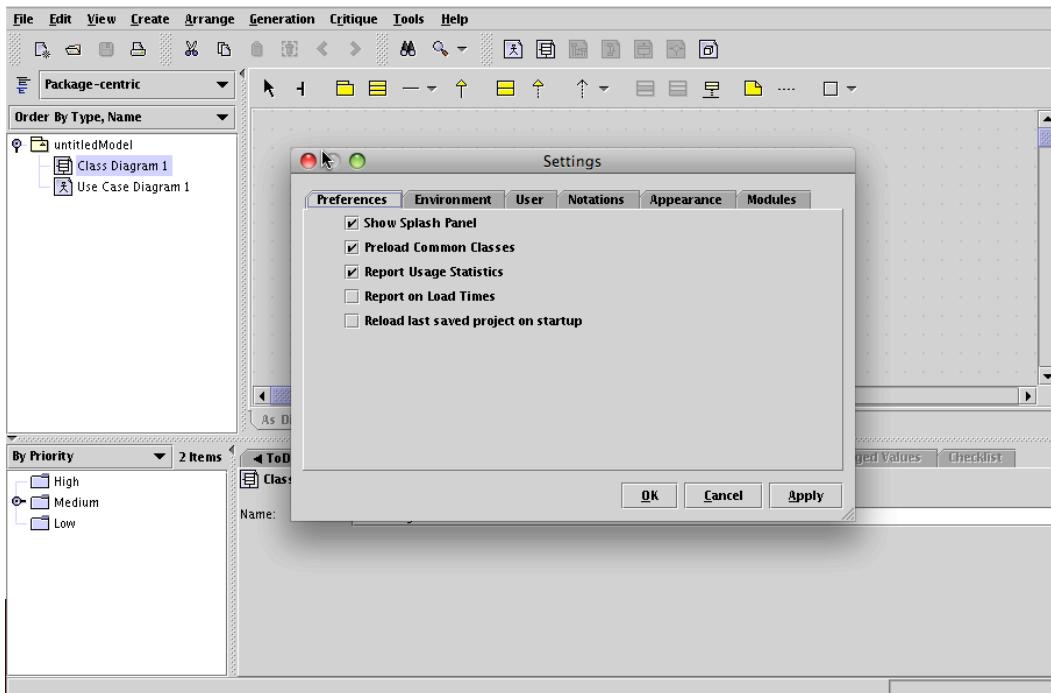


Figure 10.25. ArgoUML bug 3121, titled “Remove ‘Report Usage Statistics’ since it does not do anything.”

The second bug (shown in Figure 10.26) involved investigating a drop down list of Java types that was supposed to contain all legal Java classes for Java field, but was for some reason excluding classes in different packages that had equivalent names. The problem was that a class responsible for aggregating these class names collected the names in an ordered set, whose equality operator was only comparing the unqualified names of classes. These comparisons excluded the second class with the equivalent name. The challenge of the task was to identify the class that was aggregating these names and filling the drop down menu and understand precisely how it was gathering the items for the menu.

In order to determine success on a task, it was important to consider that there may be many possible correct explanations for a program’s failure. This is because there are many possible correct changes. For example, task two could have been fixed by qualifying the name of the type entities used to construct the list, or by making the set comparison more sophisticated, among other solutions. To determine which of

these was “correct,” the actual change suggestions made in the ArgoUML project were used as a guide. If a participants’ explanation of the cause was subjectively similar to that in one of the bug report discussions or check-in comments, then it was deemed correct. This actually turned out to be less of an issue than anticipated, since task 1 only had a single correct solution and task 2 was so difficult that few got near a correct solution.

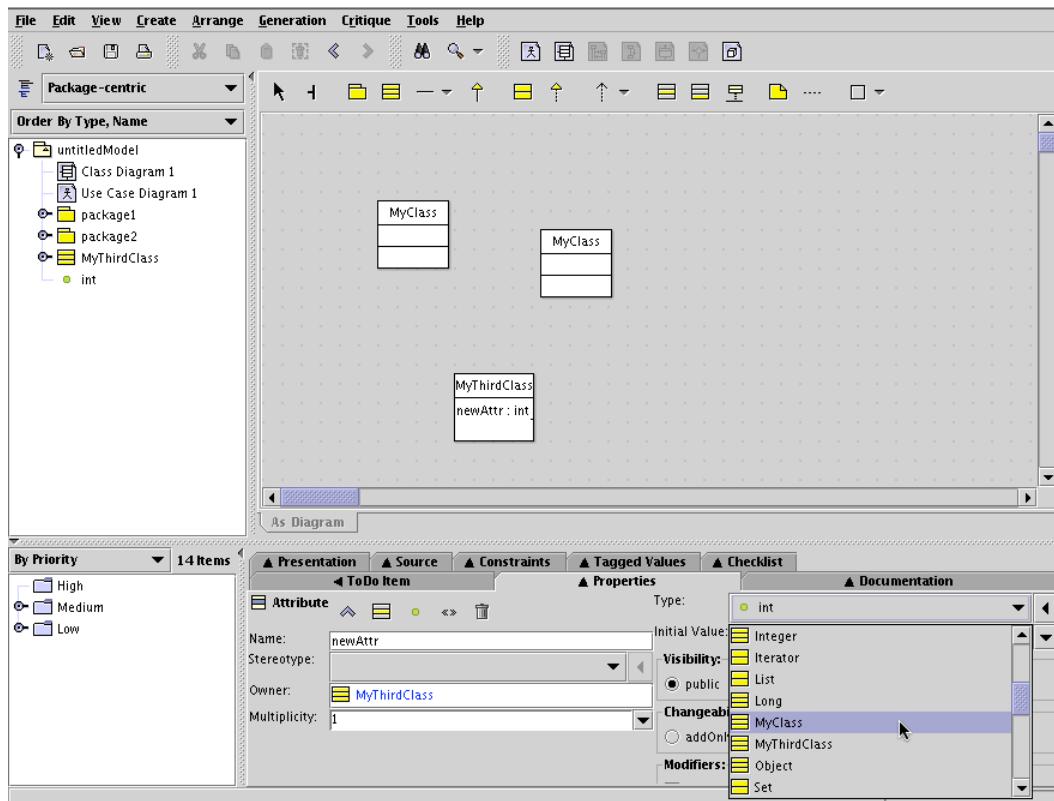


Figure 10.26. ArgoUML bug 3128, titled “Problems with two classes with the same name in different packages”.

Procedure. Prior to participation, participant were randomly assigned to one of the two debugging conditions. Participants were greeted and then asked to read and sign an informed consent form. They then completed a one page questionnaire about their prior programming experience. After this, the experimenter helped the participant complete a 10-minute tutorial on features common to both conditions, including the source code navigation tools such as “go to declaration” and “show callers” commands. Then, the experimenter walked the participant through a tutorial specific to the condition (the tutorial materials appear in the Appendix). Whyline participants learned how to ask questions and navigate answers, whereas control group participants learned how to set breakpoints, step through code,

inspect the call stack and local variable state, and insert pseudo print statements. After completing the tutorial, the experimenter read the first task description and provided a copy to the participant to follow.

Participants were told to isolate the cause of the specified problem and then write a change recommendation to a fictional boss in a text file. Participants were also told to emphasize speed over correctness, since the code they were understanding was unfamiliar and their fictional boss would know if their answer was on track. The participant was asked if they had any questions and then told to begin. Participants were allowed to ask for clarification about any of the tutorial content, but other questions were not answered. Participants were given 30 minutes to complete the task; at 10 minutes and 5 minutes remaining the experimenter warned the participant of the time remaining. After the first task, this process was repeated for a second task and then the participant was debriefed about the purpose of the study.

In designing the study, there were many possible times at which participants could have stopped their work. Allowing participants to work as long as they liked on a problem was not feasible, therefore, a time limit of 30 minutes was chosen (by piloting the tasks with several representative developers). However, even with a time limit, there could be unpredictable variations in when developers chose to stop, depending on how confident of their solution they wished to be. Therefore, participants were encouraged to trade confidence for speed, with the hope of unifying their performance tradeoffs.

Results. The results for task 1 are summarized in Figure 10.27. All 10 Whyline participants completed task one, compared to only 3 control participants ($\chi^2 = 10.6$, $p < .05$). Whyline participants also completed task 1 twice as fast ($t = 4.5$, $p < 0.05$). The control participants who did finish the task explored hundreds of files, but got lucky in their searches, whereas the Whyline participants only explored a median of three.

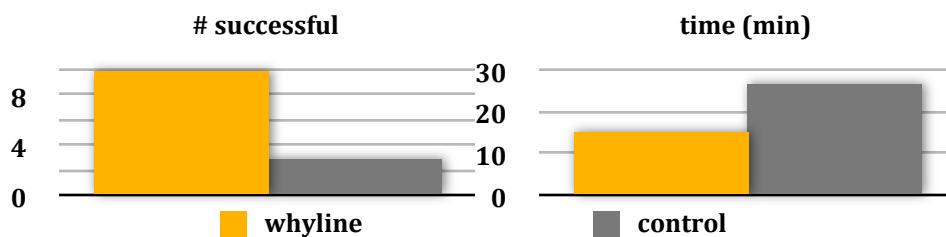


Figure 10.27. For task 1, the number of successful participants and the time on task.

The results for task 2 are summarized in Figure 10.28. Of the 10 participants, 4 Whyline participants were successful, compared to none in the control group ($\chi^2 = 5$, $p < .05$). This task was considerably more difficult; the successful Whyline participants spent all thirty minutes on the task, but much of it was in order to understand some of the Java APIs used in constructing the list for the drop down menu. Given the difficulty of the two tasks and the sheer size of the application, that anyone was able to solve the tasks, even with the Whyline, is a testament to the effectiveness of the Whyline approach and of the Whyline's user interface.

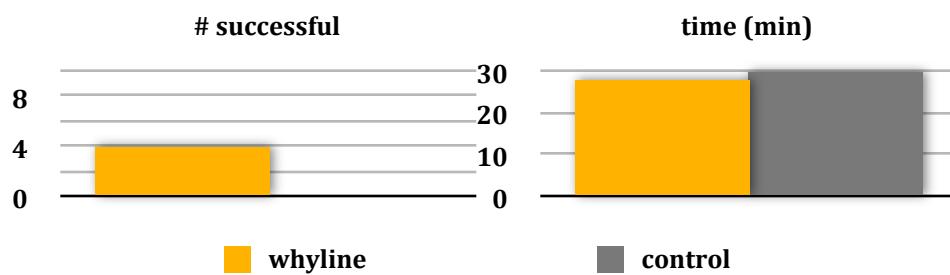


Figure 10.28. For task 2, the number of successful participants and the time on task.

There was no significant relationship between number of years in the software industry and success for either task (though this was probably because the sample was too small or the developers had too little industry experience).

Aside from the success of participants, it is also worth mentioning the success of the Whyline user interface. All Whyline participants asked at least one question with a maximum of three and all relied exclusively on the data presented in the Whyline's answers rather than doing manual searches through source code. There was a split in whether participants focused on the visualization or the code. Some used the visualization as the primary data, asking followup questions from the visualization and using them to drive their search. Others focused on the source code and asked questions from there. Whyline participants in general used Whyline questions and followup questions to guide their search almost exclusively; the control participants used manual searches and static navigations, namely "show declaration" and "show callers" for methods.

Unfortunately, there was not time for a followup questionnaire to assess user's overall opinions of the Java Whyline, but 8 of the 10 Whyline participants offered a their thoughts unprompted:

"This is really great!"

"I love it!"

"This is really going to reduce the burden on programmers."

"I think this will really help."

"It's so nice and straight and simple..."

"My god, this is so cool."

"This is great, when can I get this for C?"

"This is very nice."

The enthusiasm of participants was clearly evident and all participants asked to be notified of the tool's availability.

Discussion. There are a few interesting impressions that can be gleamed from the participants' work with the Whyline. For example, participants seemed to treat answers like they treat the first page of results of a web search: if they saw nothing in the first few events of a Whyline answer, they would try asking a different but related question. Similarly, there seemed to be a reluctance to follow data dependencies perhaps because other tools they were familiar with only allow one to navigate control dependencies (such as with a call stack). This means that it was crucial for the Whyline find relevant code to show immediately. Either that, or some additional training may have been necessary to emphasize the importance of looking deep within the results of the answer, and in particular, understanding the data dependencies in the answer.

Another observation was that there was some variation in the specificity of questions that the Whyline participants used, suggesting that users of the Whyline still need to use some caution in which questions they choose to explore. Some chose questions directly relevant to the symptoms of the failure, and as a result, found answers that were directly relevant. Others chose more generic questions only tangentially related to the failure, probably because they could not find other more relevant questions. They still tended to find the answers, but only with more work. Therefore, people still need to be systematic and cautious in their program understanding, but the Whyline helps by making the choice of question more explicit.

By making dependencies between code explicit through questions and answers, the Whyline seemed to influence the participants' confidence in their understanding of causality in the program. Although there are currently no numbers to support this claim, Whyline participants seemed to require less time to decide that they had found a buggy method and were generally right when they had decided so. Control group participants often read a method and after some time understanding related code, deemed it "unimportant" by never returning to it, even when it was precisely the method that contained the bug.

It was also clear that there is a subtle difference between types of program failures. Some bugs are obvious once one sees them and the challenge is purely in locating it (such as task 1 in this experiment). Other bugs are similarly challenging to locate, but even once found, it may take some time to understand the problem (task 2 in this experiment). The results of the study suggest that the Whyline is quite helpful at "finding the buggy method" but not for explaining "why the method is buggy." This is related to the fact that the Whyline is only providing causal explanations of output and not making change recommendations. It has no special knowledge of the intended behavior of the program, other than the implicit expectations from the users' questions.

Aside from these higher level issues, there are number of reflections on the Whyline user interface worth mentioning. Although there was a split in how people used the visualization, some to guide their search and others as a bookmarking tool, the central benefit of the visualization seemed to be as a place to gather relevant code. All of the Whyline users relied on the events in the visualization as a way to get back to recently viewed and relevant code and many complained that there was no way to remove events from the visualization once they had been added. This suggests that they intended to use the visualization as an important encapsulation of their discoveries about the problem they were investigating.

The tutorial was an important part of helping participants understand the visualization; without it, as found in earlier pilot studies, the notation itself and its similarity to Java syntax, was not enough to understand the semantics behind the events. This is not all that surprising, as no notation is arguably "natural" without some prior knowledge. The best approach to the training would probably be to integrate the explanation of the visualization notation into the tool itself through tooltips or other means, rather than as a separate introduction.

Whyline participants tended not to ask “why didn’t” questions and when they did, they tended to get frustrated at the extra time it took to answer some “why didn’t” questions (this was because the analyses involved exhaustive searches through potentially large and complex call graphs and were on the order of a minute or less for the tasks in this study). This problem is inherent to the static analyses required to generate these answers. It is not clear whether participants would have asked more “why didn’t” questions if they had been faster to produce. It is also unclear whether participants *avoided* “why didn’t” questions or just were not as aware of their presence. This raises the larger issue of whether participants were even able to find the questions they wanted to ask. Unfortunately, this was not what the study was designed to investigate, since it only involved two tasks. Exploring this question in depth would require a study that investigates a broad sample of programs and bugs. This is something I plan on investigating as part of the Java Whyline deployment by giving users an opportunity to send feedback about questions they wanted to ask but could not find.

Another potential issue regards the size of the Whyline’s answers. Because information was computed on demand, users only saw as much information as they requested, so there were no apparent difficulties with the amount of information presented. The challenge was instead in choosing the right data to request.

10.4.5. LONGITUDINAL AND EXPERT USE

To date, I am the only expert user of the Java Whyline as it has not been widely deployed, but I have a number of perspectives to offer on how it has influenced my own development work on Java software. Obviously, all of my comments come from a sample of one and are subject to many kinds of confounds, so they should be taken at face value.

First, I feel more careful in my own reasoning and skeptical of my hypotheses when using the Whyline. Something about having to explicitly choose and formulate a question makes me carefully consider precisely what I want to know about and whether any of my past experiences will be useful in predicting what happened during a program’s execution. I have learned over time that the cost of gathering and analyzing a trace usually pays off, even if at first it seems like lower overhead debugging tools such as print statements and breakpoints might be helpful. This is largely because it is nearly impossible to predict how long a bug will take to fix.

I have also found that the Whyline is useful as more than just a debugging tool. I have used it as a reverse engineering tool to trace back from output to code to understand the different architectural pieces of large applications I wanted to understand. For example, when I was trying to debug the tasks used in the ArgoUML study, I could simply click on output and gather concise lists of the data structures that influenced the output, providing me directly relevant lists of classes to understand. Even by simply scanning the mouse across the screen, I can see just how many source files are used to implement the user interface and within seconds, get a sense of the complexity of the application's design.

Another important observation about question asking is that having knowledge of how the Whyline answers questions helps me choose more relevant questions because I can predict the content of the answers. I know that a question about why a field was not assigned will implicitly analyze all of the potential assignments to the field and how they can be reached; this is an extremely helpful way of finding out how encapsulated the field is. This kind of advanced query selection is similar to expert users of search engines. Once a Google user knows that the order of the search terms influences the order of the results, they can more effectively control the quality of the search results.

The only things that is currently stopping me from using it for *all* of my debugging are the bugs in the Whyline itself. I would find a robust implementation extremely useful to my work.

10.5. DISCUSSION

Though there are numerous issues to discuss about the Whyline concept in general, this section will be limited to issues regarding the *Java* Whyline. One central issue is how much the design of the Java Whyline is specific to Swing and Sun's Abstract Windowing Toolkit (AWT). The Java Whyline assumes that particular rendering classes are used. Supporting these classes required considerable effort, because the rendering interfaces are quite complex. Many of the functions in `Graphics2D` take images and it was not feasible performance-wise to record the whole image. There were many other esoteric features of the class that were not supported, such as variations on the cap styling of pixel strokes. Offering full support for all of these details may be important in a proper release of a Java Whyline. There is also the

issue of how easily the Java Whyline could be adapted to other rendering toolkits, such as SWT (the toolkit used to create Eclipse and its plug-ins) or other web standards like Adobe's Flash or Microsoft's Silverlight. There are many similarities in the way that these platforms render output, but because of subtle architectural differences, it would require significant work to record output and provide user interfaces for exploring it. For example, much of the output in Flash is vector-based, so the output recording would need knowledge of the vector-based graphical data structures, which the Java Whyline does not have. Each platform's output rendering interfaces would require custom handling to support well and some platforms are decidedly less cleanly designed than AWT.

One issue related to the rendering interfaces used is whether the Java Whyline could be adapted to support questions in the context of a running application, rather than in a postmortem fashion. Pausing the program would not be difficult using conventional breakpoint debugging frameworks, but all of the work that the Java Whyline must do to track the location of primitive output onscreen would have to be done during execution. It could be a significant performance bottleneck to maintain and update this information as the program executes, without having to give the Whyline special knowledge about the user interface toolkit used by the program. If one took this approach, it may be possible to support higher level questions about user interface components and their state, but not about lower level output primitives like rectangles and lines. This could be a useful middle ground to explore, while streamlining the interaction with the Whyline.

This raises the issue of having question plug-ins for the Whyline, to support higher-level questions about specific types of output. The Java Whyline has no special knowledge of user interface toolkits or other APIs, other than knowledge about primitive I/O, meaning that the specificity of the questions and answers is often lacking. For example, if a user is wondering, "*why didn't this window change?*" users must choose a suitable substitute, such as "*why didn't this JFrame's repaint() method get called?*" It might be helpful if one could write plug-ins for the Whyline to add special knowledge and heuristics for certain APIs, to improve the specificity of questions and answers, and even offer recommendations about potential fixes for errors.

Aside from the rendering interfaces, another issue is how specific the Java Whyline is to Java itself. Could it be easily adapted to support other object-oriented languages

such as C# or C++? The Java Whyline does have a strong reliance on object-orientation, because it relies on the fact that all of a program's domain concepts have been subdivided and named in individual data structures. The challenge of supporting other language paradigms is discussed in the next chapter. However, there is the practical issue of being able to write a Whyline that covered a whole range of object-oriented languages with the same tool. Since Java, C++, and C# all use different instruction sets, one way to accomplish this would be to compile such instructions into some intermediate language, much like modern compilers can do, and then reason about the code at the level of this common representation. The tradeoff would be the lack of precision in relating these abstracted instructions back to the actual source files used to generate them. To ensure this was possible, it would be important for the compilers to generate tables of a precise mapping between the tokens in the source files and the instructions in the compiled code.

Java, among other languages, has certain features that make analyzing causality inherently incomplete. For example, languages that support reflection can result in calls to methods that cannot be identified statically. Therefore, when answering questions that have to reason about potential callers, the analyses are bound to miss certain possibilities. One approach to this problem is to be conservative and include all possible calls, even searching potential reflection calls; this may result in too many possibilities being provided. Another approach is to communicate the confidence that the answer is complete, just as the Java Whyline does by saying, "this method has no *known* callers."

10.6. SUMMARY

The Java Whyline was successful on many levels, contributing:

- A data representation for capturing the execution of Java programs and their output that computes properties of the recording on demand and efficiently caches them on disk.
- Algorithms for marking classes, fields, and methods that indirectly effect primitive output.
- Algorithms for identifying potential sources of values for a given Java variable.

- Algorithms for determining why a particular instruction was not reached, accounting for constraints on the time at which it should have been reached and the object context in which it should have executed.
- User interfaces for exploring graphical, textual, and exception output of Java programs.
- User interfaces for navigating the I/O history of a Java program.
- Algorithms for identifying “why did” and “why didn’t” questions about data and objects indirectly influencing a given output primitive.
- Heuristics for filtering question menus by a notion of familiarity, defined by code ownership.
- Support for questions about code that *did* execute, despite users’ beliefs, helping to reveal false assumptions about program execution.
- Timeline visualizations of code execution separated by thread and connected by control and data dependencies.
- A workspace that closely relates execution history, output, and code in a through a single unifying user selection.
- Evidence that the study participants liked the Whyline and want a version to support their favorite language.
- Evidence that Java Whyline users are significantly more successful and efficient at solving debugging tasks in a large open source system.

Overall, the Java Whyline addressed many of the issues of scale that were concerns with the Alice Whyline and also extended our understanding of how people conceive of program output and how tools can relate those conceptions to code. It is a solid example of how the Whyline concept can be supported for complex programming languages and could be quite helpful in inspiring Whyline tools for other widely used languages and platforms.

11.

LIMITATIONS AND FUTURE WORK

How well does the Whyline generalize? What are its limits? How can the concept be broadened to other languages and contexts of use? This section discusses these issues, exploring the rich design space of question asking tools.

11.1. LIMITATIONS

11.1.1. PROGRAM QUALITY AFFECTS QUESTION AND ANSWER QUALITY

Because the Whyline extracts all of the knowledge about a program from the program itself, any limitation on the knowledge encoded in a program limits a Whyline's utility. For example, the Whyline uses identifiers in the code to phrase questions, therefore, if the quality of the program identifiers is low, the quality of the question phrasing will be low. Were a Whyline in wide use, this dependency may have interesting social side effects. By making class and field names inherently public to the rest of the software development organization, the Whyline could incentivize more descriptive names for code constructs. It could also incentivize descriptive comments for fields and classes, which could be extracted from source code and shown to Whyline users to help them choose appropriate questions.

Another interesting form of program quality is the degree to which the concepts defined in a program faithfully represent the domain concepts they intend to represent (what might be called “type fidelity”). For instance, the Java Whyline relies heavily on Java’s object-oriented and statically-typed nature. Object-orientation

compels developers to declare classes and fields that separate distinct behaviors and state in these classes. The Java Whyline relies on this conceptual organization to provide conceptually organized menus of questions. There are a number of language paradigms that compel different forms of conceptual organization of domain concepts. For example, a purely procedural C program that nevertheless supports GUI components like buttons and menus, may not have a collection of clearly defined structures to render the components. Instead, there may be highly parameterized procedures for doing so, and a Whyline would have to do extra work to identify and organize these procedures, before using the techniques discussed in Chapter 10 to derive questions from these procedures.

11.1.2. TYPE INFORMATION AFFECTS QUESTION AND ANSWER PRECISION

The Whyline approach relies on rich static type information in order to extract, present and answer “why didn’t” questions. Therefore, dynamically-typed programming languages pose interesting challenges for all of these analyses.

One challenge is in the difficulty of building a precise call graph. Dynamically-typed languages such as Javascript are most problematic: even with runtime data, one cannot find all possible calls to a method without being conservative and losing precision. This makes it more difficult for the Whyline to answer “why didn’t” questions. Even statically typed late binding languages like Objective-C pose problems: when analyzing why an instruction did not execute, it is necessary to know all of the feasible callers to a particular method. Another problem is if the call graph is incomplete: if the Java class containing the invocation that needed to be called was never loaded, the call will not be known, and will not be part of the Whyline’s answer. This can be mitigated by actively loading referenced classes, but traversing too many levels of depth in such a call graph becomes impractical.

Another challenge with dynamically typed languages is in identifying code structures that indirectly render or affect primitive-level output. This is a side-effect of imprecise call graphs. For example, it would be difficult for a Whyline to determine precisely what variables could affect output produced by a JavaScript function, unless there was runtime information to detect such data dependencies. Even then, this would not reveal other possible data dependencies about which a user might want to ask negatively phrased questions. The consequence of this

imprecision is that it may be difficult to identify good names to use in “why didn’t” questions, since so many different functions may be relevant.

11.1.3. LIMITATIONS OF TRACING

Execution traces pose several limitations. It is not practical to record executions that span more than a few minutes because the amount of data captured is too much to load and process in a reasonable amount of time. Programs and test cases that process and produce substantial amounts of data also pose a similar problem since so much intermediate state is captured in the process. Of course, what is feasible depends highly on the context: if a bug is particularly difficult to find, it may be worth the time and space necessary to capture and analyze a trace. Tracing also limits Whyline support to program behavior that is reproducible while probing. Multi-threaded programs are often problematic in this sense since instrumentation can affect indeterminacy of multi-threaded bugs. Programs that rely on real-time behavior may also behave differently when instrumented, making it difficult to reproduce a problem that relies on real-time performance.

Tracing also makes the approach feel ‘heavier’ than tools like breakpoint debuggers, which require virtually no setup time compared to the time spent waiting for a Whyline trace to load. All of these issues are worsened by the fact that the memory demands on a developer’s machine grows with the size of the trace. At a certain size, performance becomes an issue as the Whyline begins to rely on virtual memory. Better disk bandwidth would alleviate this. Also, there may also be ways to utilize multi core or distributed CPUs to provide dedicated support for trace capture and processing. Another possibility is that there may be ways to only trace at certain times, like today’s performance profilers; the challenge would be that the causes of events might not be captured, even if the effects were.

Outside of these issues, it is possible to support Whyline-like questions without capturing a whole trace. For example, the Whyline could capture inputs and outputs to support questions and then use this I/O information to re-execute the program to capture only the information in a static program slice on the queried code. The resulting recording would be smaller and incur less overhead during capture, possibly eliminating many of the probing problems in multi-threaded and real-time programs. The downside would be that it would take extra time to answer the question, adding a whole process of static analysis and program re-execution, in

addition to the loading and answering. Another possibility is to just support static questions with no trace information at all. Such an approach would be devoid of dynamic information and would probably require custom support for interrogating actual program output of a live program, rather than a reproduction of program output. The questions and answers would also be much less precise.

One possible remedy to the overhead imposed by tracing is to simply instrument the virtual machine rather than the program itself. This would reduce the number of instructions needed to manage the instrumentation. However, the tradeoff to this approach is the loss of platform-independence; furthermore, such an approach may have the same space overhead, and may in fact be just as I/O bound as an approach instrumenting the program.

11.1.4. LIMITATIONS OF QUESTIONS

By relying on a program as the source of questions, the Whyline will rarely perfectly match the questions that the user has in mind. People will phrase questions differently than the Whyline and there will be other context that the user may wish to specify in a query that the Whyline may not support (such as questions relative to multiple objects, as in “why didn’t the Menu appear next to the Button?”). Furthermore, the Whyline will not always describe output at the level of granularity that the user thinks of it. If the Whyline cannot extract a “Button” concept from the program’s code, the user will have to ask about whatever concept the Whyline is able to find as a proxy for “Button.” The consequence of these limitations is that users will have to learn about how the Whyline extracts questions in order to know what questions to expect and where to find them. The advantage over conventional program understanding tools is that the distance between the desired question and the supported analysis is usually far shorter for the Whyline (bridging Norman’s gulf of execution [Norman 1988]). Consequently, the potential for mistakes in this translation is correspondingly less.

There are other aspects of program output that are at a higher level of abstraction than an output primitive, but may not have any corresponding program entity to represent it. Imagine a menu with a list of items with varying degrees of padding between items in the list and the margins of the menu boundaries. To ask about the whitespace in the menu if the whitespace was defined internally through constants, one would have to just ask about the position of one of the item’s text label and hope

that it was related to the whitespace the user really wants to ask about (another option is to add domain-specific support for such concepts like Crystal did, as described in Chapter 9).

Another interesting limitation in deriving questions lies in potential distinctions between “program output” and “program behavior.” Up to this point, this dissertation has used these phrases interchangeably, but there are subtle differences in their meaning. *Output* can be thought of in a static way, devoid of time, such as a snapshot of a particular entity onscreen. Think of a pressed button, frozen in time. *Behavior*, in contrast, can be thought of inherently temporal, something that can only be seen through a sequence of changes to these individual snapshots, such as the before and after appearance of a button in response to a click. There are characteristics of such *behaviors* that no Whyline prototype currently supports. Why did the button depress so *slowly*? Why did the sound play so *soon* after the other sound? Currently, the only way to ask questions about these temporal aspects is to choose some characteristic of one of the states of the program behavior and then “manually” relate it to the other state.

This distinction between output and behavior raises the general issue of *software qualities*. The Whyline is designed to support questions about *functional correctness*, and explicitly avoids support questions about other forms of correctness. Profilers are good at answering questions about performance. Usability testing is a better way of diagnosing usability. Code reviews are a good way of assessing maintainability. Even within functional correctness, there are certain types of correctness that are better supported by static and dynamic analyses, such identifying potential deadlocks situations with static analyses. In imagining other types of questions that the Whyline might support, it is important to keep these other kinds of correctness in mind.

11.1.5. LIMITATIONS OF ANSWERS

The most important limitation of Whyline answers is that they only reason about causality. They do not offer change suggestions, they do not isolate bugs, and they will not guide the user in interpreting the consequences of the answer to the debugging problem. All of these things are still the developer’s responsibility. Therefore, although the Whyline will help users get closer to a fix than they would have otherwise, users must still cautiously, objectively and systematically explore

the answers provided by the Whyline in order to determine a reasonable fix for a bug. The reason for this limitation has less to do with the Whyline approach and more to do with the slippery notion of a bug. Bugs are really just undesirable behaviors; even a program crash can be an expected and desired behavior under the right circumstances, for example when it prevents a more serious failure from occurring. As a consequence of this notion of a bug, any undesirable behavior has many possible solutions. The Whyline has no knowledge about the desirability of these various solutions and so it is still up to the user to decide what of all that did or did not happen in a program’s execution needs to change. Of course, these points are true not only for the Whyline, but also for other debugging tools. The only kinds of techniques that can get around this problem are those that rely on specifications of a program’s intended behavior, such as model checking systems, because they can compare *intended* execution with *actual* execution.

The kinds of answers that the Whyline gives have their own limitations. The causal answers provided in response to “why did” questions and some “why didn’t” questions can be quite large, since they are based on dynamic slicing. What determines how “large” these answers are depends on several factors. The more complex the program design and execution, the more complex the Whyline answer. The more users explores causality, the more information they will have to consider in assessing the cause of a problem. As they explore the answers, users will have to work around incompleteness in the Whyline’s recording. Native calls and other procedures may not be amenable to instrumentation or even available for static analysis. This will result in a loss of precision for reasoning about the inputs and outputs of these calls, so Whyline prototypes will have to support ways for users to know when such information is missing and help them work around it. In general, the size of the answers was not a serious issue in the user studies described in Chapter 10, but the tests were limited to only two tasks.

The answers for “why didn’t” questions have other limitations. Because the “why didn’t” answering algorithm uses a constrained traversal of a program’s control flow graph, the completeness of the control flow graph greatly influences how much the user can trust the Whyline’s answers to “why didn’t” questions. For example, if the Whyline determines that there are no callers to a method and answers so, it may be that all known calls occur through reflection or other mechanisms that the control flow graph construction algorithm has overlooked. If the Whyline determines that there are callers to a method, it may be that none of the calls can feasibly be made at

runtime (a precision issue). These precision issues can be dealt with by incorporating other research on creating precise control flow graphs, but with a performance cost [Milanova 2002].

Another limitation about Whyline answers is that they are most helpful at finding particular kinds of bugs, but not others. A simple way to state this scope is that the Whyline will help *find* the buggy algorithm, but not *explain* why the algorithm is buggy. For example, if a value is computed from a complex machine learning algorithm or other complicated logical reasoning, the Whyline will show the user so, but it will provide no support in explaining mistakes in such algorithms. This is precisely because the Whyline has no knowledge of the intended behavior of these algorithms. Such issues are best left to the experts who wrote the algorithms and perhaps model-based techniques to help identify these mistakes before they manifest into failures.

The failures discussed in Eisenstadt’s “My Hairiest Bug War Stories” [Eisenstadt 1997] are also unlikely to be solved with the Whyline. These kinds of bugs, the ones that developers remember after even 10 years, often have particularly obscure causes, such as hardware failures and problems deep within an API or operating system. What the Whyline can do in these scenarios is help a developer *isolate* the problem to such modules, by ruling out problems in the developer’s main program.

There are also situations in which Whyline answers can lead to dead ends. There are at least two kinds, both having to do with developers’ navigation of the control and data dependencies in a Whyline answer. First, if the Whyline has not instrumented some function and therefore cannot reason about it precisely, a developer must understand the function from its code, rather than its execution, and resort to other methods. The other kind of dead end is where the Whyline answer *does* contain the relevant execution events, but the developer does not believe the code—and in particular, the names in the code—to be relevant to the problem, they may overlook a relevant chain of causality. This did occur in some cases in the lab study in Chapter 10, although these developers eventually returned to the unexplored dependencies after exhausting other possibilities.

11.2. FUTURE WORK

There are a number of practical future work items, such as releasing the Java Whyline to the public and writing a widely requested Eclipse plug-in to help integrate the tool into developers' workflow. However, this section focuses on issues that require more research to implement well.

11.2.1. REAL TIME DEBUGGING

A natural extension of the postmortem version of the Java Whyline is to support debugging of a live Java program, without having to quit the program and load a recreation of its output. Aside from the challenge of tracking dynamic dependencies in real time, one significant challenge with this is in allowing the developer to actually point to output in the running application and have the Whyline actually relate it to live objects in the Java heap. To do so would require the Whyline to do the same I/O tracking that is done when a Whyline recording is being loaded, but instead doing it at runtime. This could incur significant overhead. One way around this problem is to have special toolkit support. For example, one could imagine a JVM debug mode which maintains a history of output, providing a hook for whatever debugging tool wanted to relate output to an execution history. Another alternative is special toolkit support for asking questions about UI components and their state, rather than output primitives. This would limit the generality of the tool, but make it more feasible to ask questions in a live program. Of course, support for debugging a live program would come with limitations of its own. If the program freezes, the question asking support might freeze as well unless it was in an independent process.

11.2.2. OTHER OUTPUT MODALITIES

Current Whyline prototypes only support questions about graphical and textual output, but there are many other popular forms of program output (and thus program failures), including sound, network traffic, disk activity, and others. The central challenge in supporting these other modalities is in finding effective ways to inquire about features of the the output and also in choosing the appropriate output primitives in each. For example, what characteristics of sound are important to interrogate: just the presence or absence of sound, or detailed properties of its pitch

and modulation? Writing to disk can often entail large amounts of data and one often only notices a failure in output after a file is completely written. How can tools effectively present this output in a manner that makes it easy to find the subject of the question amid so much information? It is possible that there could be a visible proxy for such modalities.

This issue is related to the limitation of question mismatch, mentioned earlier. One could imagine implementing toolkit-specific plug-ins for certain collections of user interface components, bringing the supported questions much closer to the content that developers might want to ask about. Given the sheer diversity in types of program output, this might be the most effective way to more closely align the questions that developers want to ask with the questions supported by tools.

11.2.3. OTHER DOMAINS

As discussed in Chapter 2, a number of researchers have explored question asking tools in other domains. The ACT-R cognitive framework [Bothell 2004] and cognitive tutoring tools that used this framework [Aleven 2006] both support “why not” questions about production rule systems. AI knowledge base systems support “why not” questions about why certain data was not used in answering queries to a knowledge base [Chalupsky 2002]. Lieberman explored “why” questions about e-commerce transactions [Lieberman 2003]. The Whyline concept has directly inspired projects looking at one-way constraints in user interface design [Clark 2007] and spreadsheets [Abraham 2005]. Outside of these examples, there are a number of other domains in which to apply the Whyline concept. In terms of end user applications, how would question support generalize for applications other than word-processors? Home networking poses interesting challenges for computer users to diagnose. As hardware prototyping becomes more prevalent, finding ways to diagnose failures has become more of an issue. Another budding area of research is in helping to understand failures in software that uses machine learning and AI techniques, since the indeterminacy and data set dependencies of such software can be difficult to explain. As software becomes more pervasive, so will difficulty with understanding why it misbehaves.

11.2.4. COLLABORATION SUPPORT

At least for Whylines intended for teams of software developers, collaboration support is a central design challenge for future Whyline prototypes. Debugging is an inherently collaborative activity in these contexts, requiring the knowledge of multiple people over the course of many days or more. This places a number of constraints on the Whyline design.

First, Whyline traces should be small and easy to share. This allows traces to be shared along with bug reports and other software development artifacts. Related to this support is the need to annotate a Whyline trace, as developers discuss a program failure and move towards a solution. These annotations might relate to particular events in a trace or particular parts of the software's code. It is also possible that a Whyline trace could replace the modern notion of a "bug report," capturing information about who the trace is assigned to along with a discussion of other aspects of the trace. It may also be possible to define a bug report as a collection of traces, all potentially demonstrating the same failure.

Another issue, unexplored in the Java Whyline, is the problem of *versions* of code. The current prototype just records the Java classes, but without any notion of which version of each class is stored in a version control system. Version information will be important in relating the Whyline trace to a particular bug report, which is typically related to a particular release. By explicitly relating a failure to a particular set of versioned source files, there may be other opportunities to detect the same failure in other versions of these source files as well. The Whyline could also integrate well with unit testing systems that explore changes to source that lead to unit test failures [Xie 2007].

The notions of familiarity and ownership used in the Java Whyline might need to be more elaborate in collaborative settings. For example, the current definition defines familiarity by *access to editable source*. Software development teams typically have much more complicated notions of ownership and certainly of familiarity. One might be whether a developer has checked in code personally or whether the particular source file in question is managed by the developer's team. Ultimately, all of these determinations are used to reduce the size of question menus, so this particular issue may require the ability to customize this definition depending on the particular software development context.

11.2.5. LANGUAGE INDEPENDENCE

Given some of the complexities of designing a user interface for Whyline recordings, there may be some benefit in thinking about how to architect the Whyline to support multiple languages through a common tool (aside from simply adopting a common language runtime, such as Microsoft’s CLR). For example, it may be possible to find an intermediate language to compile to and reason about, allowing the reuse of slicing algorithms and the Whyline visualization, isolating language-dependent parts of the Whyline system to question derivation algorithms and source code highlighting. One benefit of such a system, beyond the usual benefits of code reuse, would be that it may be easier to support the analysis of programs that execute with multiple languages. It is increasingly common to see programs with three or four languages, glued together with sockets or scripting languages such as Javascript or Python.

11.2.6. INTEGRATION WITH OTHER DEBUGGING TECHNIQUES

The Whyline concept utilizes modifications of a number of well-known software engineering techniques including static and dynamic slicing and other methods for constructing and analyzing call graphs and data dependence graphs. There are, however, a number of other techniques that could be useful to integrate into the Whyline approach.

Unit testing, for example, provides a natural avenue for asking questions and gathering data automatically. When a unit test fails, for example, the unit test engine could automatically rerun the test with instrumentation on and gather data in the background for the user to analyze. A user could then select the failed test and immediately start understanding the cause of the failure.

One challenge with navigating a Whyline’s answer is that the user has little guidance beyond their own experience to know what control and data dependencies to follow. Researchers have looked at ways to provide such guidance. For example, static checkers [Bush 2000][Cole 2006] could provide cues about which dependency chains have the most potential problems, leading the user to fault-prone code. Another more interactive approach would be to allow the user to explicitly validate values as in the WYSIWYT testing and fault localization approach [Ruthruff 2005].

These annotations of correct and incorrect values could be propagated through the dynamic slice, highlighting contributors to incorrect values.

Another promising approach is to use multiple traces or multiple slices to identify differences in test cases, isolating the failure (the approach used by [Zeller 2002b], for example). Beyond just using the Whyline traces for determining differences between test cases, another approach would be to have the user specify multiple relevant questions related to the failure, rather than a single question. The system could then identify the intersection of the answers for the set of questions, potentially finding a smaller subset of the program's execution history related to the failure. This is similar to the notion of a "chop" described in [Gupta 2005], but unique in that it would be straightforward for the user to express.

11.2.7. INTEGRATION WITH VERIFICATION TOOLS

One of the limitations of the Whyline is that it does not help to explain why complex algorithms are "buggy." One way to remedy this problem would be to integrate the Whyline with model checking systems based on specifications of a program's intended behavior. This way, rather than doing a heavyweight batch model checking analyses, a Whyline answer might be a convenient context for invoking a more lightweight model checking analysis of specific methods within a slice. The results of these analyses could be portrayed in the context of the Whyline's answer, allowing the Whyline to both isolate the *location* of the bug, as well as give some indication of the *nature* of the bug at that location.

The Whyline might also integrate well with other kinds of static verification tools [Cole 2006], which apply a range of static checks as heuristics to find common problems with code. Rather than applying these checks in batch mode, again, the Whyline might be a more helpful context in which to make these checks, using the slice as a filter on which code to analyze. Not only would such information be more helpful contextually, but it may reduce the number of false positives that the systems report, because there would be much more dynamic data for the system to use in its static checking. Conversely, the Whyline user interface might be used to help explain the answers that static verification tools provide.

11.2.8. SUPPORTING OTHER PROGRAM UNDERSTANDING TASKS

The notion of “program understanding” is quite a broad one. There are actually a number of different types of program understanding tasks, driven by different motives, which affect how people might use a tool like the Whyline. Debugging, which is the Whyline’s primary focus, is a very focused kind of activity, driven by the goal of finding the cause of some behavior. This contrasts with reverse engineering activities, which are often more global and architectural in nature, and also differs from feature enhancement tasks, which often have the goal of finding integration points in a software architecture for some modification or new behavior. It is worth considering how the Whyline might differ for these other types of program understanding activities. For example, for feature enhancement tasks, there may be new types of questions such as “What code contributes to this functionality?” which is a broader question than the data driven ones supported by the Whyline. Such a question might be easier to ask using the output history user interfaces provided in the Whyline, but might use static analyses more heavily, using dynamic analyses only to generate the query for a static analyses about code influence. Some of these queries might come from studies of feature enhancement and reverse engineering tasks [LaToza 2007].

Reverse engineering tasks, which often have the goal of understanding the relationships between different components, might be supported by allowing users to select different parts of the program output to invoke static analyses that determine dependencies between the components. The benefit of asking these queries in terms of output instead of code might be that developers could more accurately identify the code structures by their appearance than by their name.

11.2.9. TEACHING DEBUGGING

The issue of debugging skills is also an interesting area of research to consider. For instance, the Whyline might be a useful way to teach some reusable debugging strategies, such as that of working backwards from program output and exploring data dependencies. In fact, many of the participants in the evaluation studies, after getting Whyline answers about things that they thought did not happen, but actually *did*, commented to themselves about needed to be more cautious about assumptions. One participant in the Whyline for Alice evaluation, after making such a comment, even hovered over subsequent questions in the “Why” menu, saying,

“Let me see, do I think this actually happened?” In the Whyline for Java evaluation, other participants would hover over questions about particular data, ask questions like, “Is this the data I really want to ask about?” These anecdotes suggests that it may be possible to train developers to be more objective and careful about their debugging efforts by using the tool. An interesting research question is whether such strategies would then persist, even if the Whyline was not available, and whether such strategies are the same strategies that skilled developers use.

11.3. SUMMARY

In general, the Whyline is less a competitor to other debugging tools and techniques and more of a platform. It is an effective way to *begin* the search for the cause of a failure, but any number of other tools can come into play after asking the first question to help find the ultimate cause. Aside from simply implementing Whyline tools for other languages and contexts, much of the future work is in finding ways to adapt the many ideas proposed in the past into the interaction framework envisioned for the Whyline concept. The hope is that future work will investigate ways of removing the limitations discussed in this section and finding ways to integrate other powerful techniques with those of the Whyline.

12.

CONCLUSIONS

In the brief history of computing, developer productivity has been woefully understudied. Statistics show that program understanding and debugging dominate a software developer's time, but there has been little investigation into precisely why. Looking back on the history of research on these topics, the vast majority of effort was put towards inventing new techniques and algorithms for analyzing program execution, with little understanding of what made debugging difficult and how tools might eliminate these difficulties. Considering this approach, it is not surprising that debugging tools have changed so little since in the history of computing.

The goal of this dissertation was to look take another look at debugging tools, this time from a *human* perspective, by first gaining a deeper understanding of what makes debugging challenging, and then designing tools around this understanding. This HCI approach worked: the studies in this dissertation (Table 12.1) explored debugging in a variety of contexts, finding that current tools force developers to speculate about what code causes a particular program behavior. Because developers usually guess wrong, much of the time spent understanding a program's execution involves reading irrelevant code. Worse yet, some of this speculation goes unchecked, leading developers to form inaccurate notions about a program's execution, which can cause later misunderstandings and even result in further errors. The studies in this dissertation documented the consequences of this speculation on productivity and software quality (Table 12.1) and showed that these problems exist not only for inexperienced developers, but also seasoned experts with decades of programming experience in the software industry.

Across all studies	<p>Created a framework for modeling the cognitive causes of software errors.</p> <p>Designed a methodology for reconstructing the causes of errors from video and verbal data.</p>
Alice (Chapter 3)	<p>Detailed the relationship between different types of breakdowns in Alice programming.</p> <p>Developers ask “why did” and “why didn’t” questions about failures, the majority “why didn’t”.</p> <p>Developers tended to form false hypotheses about the causes of program failures.</p> <p>Developers tend to insert new errors while debugging.</p>
The classroom (Chapter 4)	<p>There are at least six types of barriers in learning to use programming systems that span across implementation, APIs, testing, debugging, and design.</p> <p>Most student debugging problems occur because particular behaviors did not occur, even though the students had implemented code for the behavior.</p> <p>Students struggle to form hypothesis about the causes of a problem, with many getting help from their more experienced peers.</p> <p>About 20% of the reported problems involved multiple objects not working together appropriately (for example, information from one window not being sent to another).</p> <p>Many students spent considerable time investigating problems that did not exist, because they had misinterpreted or misperceived their program’s output and feedback.</p> <p>In 11% of the reported problems, students could not find a tool that would help answer their question, or could not understand how to use a tool that they had found.</p>
Skilled Java developers (Chapter 5)	<p>Developers generally form hypothetical explanations of program execution and then use a variety of tools to verify or reject their explanations.</p> <p>Developers based their guesses about the cause of program execution on surface features of its output, such as text labels found in user interfaces.</p> <p>About 88% of developers’ hypotheses about the causes of a program behavior were false.</p> <p>The consequences of guessing incorrectly caused developers to spend an average of 36% of their time investigating irrelevant code.</p> <p>Developers tend to form <i>task contexts</i> of relevant code in order to capture the information necessary to find a bug or add a new feature.</p> <p>Testing a hypothetical explanation of a program’s execution is more difficult for larger programs because there are a larger number of dependencies and possibilities to consider.</p> <p>The information in developers’ task contexts can vary considerably on the same task, likely due to differences in experience and in the actual process of gathering relevant code.</p> <p>Information foraging theory can be used to help understand the information cues that developers use to guide their search for relevant information.</p>
The software industry (Chapter 6)	<p>Developers’ work is highly fragmented, with interruptions an average of every three minutes.</p> <p>Software development is a highly social activity involving communication and collaboration.</p> <p>Developers have at least 21 observable types of information needs, spanning implementation, design, testing, and collaboration.</p> <p>Some information needs are easy to satisfy accurately (awareness) but others with only questionable accuracy (the value of a fix and the implications of a change).</p> <p>Some information needs are deferred often (knowledge about behavior and design), whereas some were impossible to satisfy in certain cases (reproduction steps).</p> <p>Information needs regarding debugging and program understanding, especially those regarding the causes of program behavior and conceiving of <i>potential</i> causes of program behavior, are particularly difficult to satisfy.</p> <p>Even after finding the cause of a particular bug, there is often the more daunting task of deciding what to do about it by collaborating with coworkers and uncovering the design rationale underlying particular code fragments.</p>

Table 12.1. Knowledge contributions from empirical studies of developers.

With this new understanding of debugging as a human activity, the fundamental problem with today’s debugging tools became clear: developers reason about the *visible symptoms* of a program’s failure, but tools force developers to speculate about the *causes* of these symptoms. The Whyline concept addresses this limitation by providing developers a way to choose questions about a program’s symptoms of

failure. By starting from output and working backwards, the Whyline helps developers avoid this costly speculation, leading them to the direct and indirect causes of the symptom that indicated the problem. Furthermore, by starting from output, the Whyline approach can even help developers quickly overcome misperceptions of their program's output, preventing them from spending precious time investigating non-issues. This dissertation shows that the Whyline approach can significantly reduce time spent debugging and also can significantly increase success on debugging tasks (Table 12.2). It also shows ways of adapting these ideas to different types of programming languages and applications (Table 12.2).

Alice (Chapter 8)	<ul style="list-style-type: none"> Techniques for deriving questions from source code to represent program output and behavior. Adaptions of backward dynamic slicing to generate concise answers about causality. Algorithms for answering "why didn't" questions using notions of reachability algorithms. A visualization of program execution and interaction techniques for combining the control and data flow causes of a program event. Evidence that the Alice Whyline increased people's productivity at debugging tasks by up to a factor of 8. Evidence that most of the questions that participants wanted to ask were supported by the Whyline.
Word Processors (Crystal), with David Weitzman and Brad Myers (Chapter 9)	<ul style="list-style-type: none"> Answering algorithms that explain causality in terms of user modifiable document and application state. Interaction techniques for asking questions about document and application entities, whitespace, and global events. An application framework for supporting questions by augmenting a conventional undo stack with information about command histories and the data they depended on for execution. Evidence that when using Crystal, users are significantly more effective at resolving common issues with complex and automated features of a word processor than when using conventional online help tools.
The Java Whyline (Chapter 10)	<ul style="list-style-type: none"> A data representation for capturing the execution of Java programs and their output that computes properties of the recording on demand and efficiently caches them on disk. Algorithms for identifying classes, fields, and methods that indirectly effect primitive output. Algorithms for identifying potential sources of values for a given Java variable. Algorithms for determining why a particular instruction was not reached, accounting for constraints on the time at which it should have been reached and the object context in which it should have executed. User interfaces for exploring graphical, textual, and exception output of Java programs. User interfaces for navigating the I/O history of a Java program. Algorithms for identifying "why did" and "why didn't" questions about data and objects indirectly influencing a given output primitive. Heuristics for filtering question menus by a notion of familiarity, defined by code ownership. Support for questions about code that <i>did</i> execute, despite users' beliefs, helping to reveal false assumptions about program execution. Timeline visualizations of code execution separated by thread and connected by control and data dependencies. A workspace that closely relates execution history, output, and code in a single unifying user selection. Empirical evidence that Java Whyline users are significantly more successful and efficient at solving debugging tasks in a large open source system. Subjective evidence that the study participants liked the Whyline and want a version to support their favorite language.

Table 12.2. Technical contributions across three Whyline prototypes.

It is my hope that the background studies and tool ideas presented in this dissertation will inspire a new generation of developer-centric tools for increasing productivity and software quality. I also hope that my research stands as proof of the value and effectiveness of designing technology from a human-centric perspective.



APPENDIX

This appendix contains most of the study materials for the studies reported on in Chapters 3 through 6, as well as the evaluation studies reported on in Chapters 8, 9, and 10. Some materials could not be recovered and other details were never committed to paper.

Tutorial for the Alice lab study in Chapter 3

A Brief Tutorial

This is a brief tutorial on the basic features of the Alice language and environment. We'll read through it together, while you work with Alice. Feel free to ask any questions that come to mind.

Alice is a 3D programming language and environment, which has a world object and any number of objects within the world. All of the variables in the world are global. Even still, Alice objects, including the world, have properties (like Color and Shape), methods (like procedures, but they can't return values), and questions (which are procedures which must return a value).

I'll go ahead and load a small world.

Changing Objects' Properties

- This world has a sphere named *Ball*, as you can see in the object view.
- You can change all objects' properties by click on the "properties" tab in the "details" area in the lower left.
- Try changing the color of *Ball* to red.
- Also try setting the *isShowing* property to false, and back to true.
- You can change the location of *Ball* by clicking on the object in the worldview and dragging it around the world.
- If you have trouble seeing the object, you can use the arrows directly below the worldview to move the camera to any position and orientation. Try getting an profile view of the ball so that the camera looks straight into the horizon.
- If you lose sight of an object, you can right click on the object in the object list and select "get a good look at". But beware: you'll lose your camera position.
- If you want to move the objects on a different axis, hold the control or shift buttons and drag the mouse. Try moving the ball up and down using shift.
- In addition to locations, objects also have orientations. In Alice, forward, backward, up, down, left and right all have meanings relative to an object's orientation.

Methods

- Objects have many methods that come for free, such as move and resize. Click on the "methods" tab to see these methods.
 - "move" moves an object in one of six directions
 - "turn" changes the orientation left, right, forward, and backward
 - "move toward" moves toward another object
 - "move at speed" moves at a certain meters/second

Tutorial for the Alice lab study in Chapter 3

- You can use these methods while you're programming to affect objects. Try dragging the *Ball.resize* method into the worldview, and select a number from the context menu.
- You can also use methods to affect objects during runtime. Let's create a new method that makes the sphere move up, then move down, as if it were hopping. Create a new method by click on the "Create new method" button, and name it *hop*. A new code view appears in the code area, and the tab is named *Ball.hop*.
- We can parameterize the method by creating a variable to store the height of the hop. Click the properties tab, click the "Create new variable" button, and name it *height*. Notice that variables can have types, including a list type. Height should be of type *number*. Set *height* to 5 by clicking on the drop down menu and selecting the other menu item; then enter 5 into the calculator.
- Let's make the Ball move up first, go back to the methods tab, and drag the "move" method onto the code area and select *up -> expressions ->ball.height*.
- Now we want to make the ball move down. Drag another move from the methods tab and make the Ball move *down* by *ball.height*
- Notice that tooltips appear over the variables showing you what their current value is, without navigating to the properties panel.

Events and the Whyline

- To have *Ball.hop* be called, we need to make an event. Try making a *When the world starts* event, by clicking on the *Create new event* button. Then drag the *hop* method on top of the event.
- Press the play button in the upper left to test the world. When the worldview is enlarged, that means you're in play mode.
- Your only options are to pause by clicking the play button again, so try clicking the pause button.
- Notice now the button says resume, and an area has appeared below. This area is called a Whyline, for a few reasons:
- Like a timeline, it helps you see what happened while the world was running (note the question right above it). You can click and drag the time cursor, the vertical bar, to control the time.
- Notice how *events* are selected, depicting things that happened while the world was running. Try selecting the *Starting Ball move down 5* event. The code that caused the event is highlighted above.
- The other reason it is called a Whyline is because it allows you to ask questions about what happened while the world was running. For example, try clicking on the why button.
- There are two types of questions: *why did*, which allows you to ask a question about something that happened. There are also *why is* questions, which allow you

Tutorial for the Alice lab study in Chapter 3

to ask questions about object's properties. Try asking why the *ball moved with direction down*.

- The Whyline answers by showing you what caused what. As you already know, the Ball moved down because Ball.hop was called, and it was called because the world started.
- The curved arrow shows you that *Ball move down 5* used Ball's *height property*.
- You can navigate back to other questions you've asked with the arrow buttons and the *Questions I've asked* button.

Questions

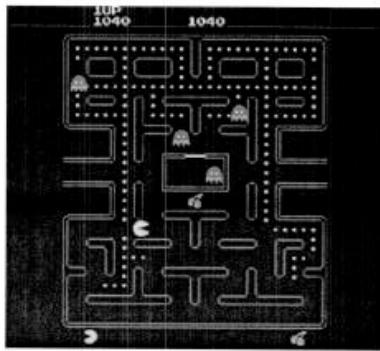
- Objects also have questions, which are methods that must return data. You can write your own, and you can also use many of the provided questions. Go to the questions tab of the ball.
 - *Is within threshold of object* returns true if one object is specified number of meters of another object
 - Other questions return numerical values, such as *ball's width*
- Let's make the ball move up and down random amounts by setting *height* to a random number each call to hop.
- You can set the value of the height variable by dragging the height tile to the hop method area. Put it above the two move commands, and select one of the numbers as a placeholder.
 - Notice that because you changed the world, the world had to stop playing.
- To get a random number, find the world's questions by selecting the world object in the object list, and then clicking on the *questions* tab.
- Then, drag the *random number* question onto the *set height to* tile.
- Set the *minimum* and *maximum* numbers by clicking *more* on each of the random tiles, choosing -5 to 5 as a good range.
- Then press the *play* button again to see the results.

Tutorial for the Alice lab study in Chapter 3

Tutorial for the Alice lab study in Chapter 3

A Simplified Pac-Man for Alice

In the original Pac-Man game (shown below), Pac-Man moved around a maze while ghosts chased him, and collected all of the big and small dots. When he collected all of the dots, the level was complete. If he ate a big dot, the ghosts would run away for a short time, and Pac-Man could eat the ghosts, sending them to the area in the center of the maze.



Your job is to create a simplified version of Pac-Man using the Alice environment, according to the specifications provided on the next page.

Here are some ground rules:

- Feel free to ask questions while you're working. I'll answer anything about the problem description, anything about the tutorial (which you can feel free to use), and anything about programming in Alice that you don't know how to do.
 - You can think of me as an Alice manual. Ask for whatever knowledge you think you need.
- I won't provide answers to questions about how to program, test, and debug your Pac-Man world.
- Most importantly, feel free to think aloud about your decisions. I hope to learn about how you program, test, and debug your Alice world.
- Also, if I'm curious about anything you're doing, I might interrupt and ask you to explain what you're doing.

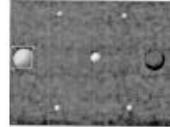


Tutorial for the Alice lab study in Chapter 3

Pac-Man Specification

Layout

Although I've already provided you with the objects and layout, the objects and camera should be arranged and oriented as in the image to the right when the world is running.



Pac-Man

- Pac-Man should be self-propelled at 3 meters per second, and move in the same direction until the player changes his direction.
- The player should be able to change Pac-Man's direction using the up, down, left, and right keys (there's an event for keyboard keys that you may use).
 - *Note: There is currently no way to change Pac-Man's orientation to a specific direction. Instead, you can create a variable for Pac-Man that stores one of four directions (there is a direction type, much like a number or Boolean type you may use), and use the direction stored in that variable to determine which way Pac-Man moves.*

The Ghost

- When the Ghost is *chasing* Pac-Man:
 - 50% of the time the Ghost should move up, down, left or right (chosen randomly with the equal likelihood) at 3 meters per second.
 - The other 50% of the time, the ghost should move *directly towards* Pac-Man at 3 meters per second.
- When the Ghost is *running from* Pac-Man, the ghost should move *directly away* from Pac-Man at 3 meters per second.

Events

- If Pac-Man touches a dot, the dot should become "eaten" by disappearing (all objects have an *isShowing* property you may use).
- If the big dot is eaten, the Ghost should begin *running from* Pac-Man for about 5 seconds, then return to *chasing* Pac-Man.
- If all five dots are eaten, Pac-Man should bounce up and down in place and the Ghost should stop moving completely.
- If the Ghost touches Pac-Man while *chasing* him, Pac-Man should flatten into a yellow disc and stop moving. The ghost should continue to chase flattened Pac-Man, even though Pac-Man isn't moving.
- If the Pac-Man touches the Ghost while the Ghost is *running from* Pac-Man, the Ghost should flatten into a disc and stop moving for 5 seconds, then return it's original shape and start *chasing* Pac-Man again.

Questionnaire for the Alice lab study in Chapter 3

ID _____

Thank you for allowing me to observe and inquire. This survey attempts to get some simple background information. Complete it at your own pace, but please remember:

- DO NOT WRITE YOUR NAME ON THIS DOCUMENT
- Once you've completed the survey, please seal it in the self-addressed envelope provided and drop it in a campus mail box or return it to me in person

Personal Background

What is your age?	____ Years
Since completing high school, what degrees have you completed and in what areas? <i>i.e., BS, Computer Science</i>	
What degree are you currently working on?	
How many hours a week do you spend programming?	____ Hours
How much programming have you done in industry?	____ None ____ Years

Questionnaire for the Alice lab study in Chapter 3

ID _____

Programming Background

Languages known. In the first column, please list all of the programming languages you know (including Alice), whether remotely or as an expert. If you know more than 10, choose the 10 languages you are most familiar with.

Environments. In the second column, list the *environments* you have used for each programming language. For example, if you have used Java in Unix, Code Warrior, and Emacs, write those three environments in the Java row.

Expertise. In the third column, score each environment on a scale from 1 to 10, where 1 is an environment you only know remotely and 10 is an environment you are expert with. *This is not a ranking, so environments can have the same expertise score.*

Recent Use. In the last column, score each environment based on how much you've used it in the past year on a scale from 1 to 10, where 1 is an environment you haven't used in a year and 10 is an environment you have used a lot recently.

These are examples of what your table entries should look like.

If you aren't sure what score to pick, do you best to pick a number and put a "?" next to it.

Questionnaire for the Alice lab study in Chapter 3

ID _____

Using Alice

Please list below at least 3 difficulties you've had with *creating code* in Alice and *some change to Alice or a new tool* that would alleviate the difficulty. Creating code would be considered writing a method, dragging variables and code around, importing characters, and other ways of creating code in Alice. *Please try to think of at least 3, and include more if you can on the back of this paper. For this question, assume that Alice is bug-free, stable, and fast.*

1. _____

_____2. _____

_____3. _____

Please list below at least 3 difficulties you've had with *testing and debugging code* in Alice and *some change to Alice or a new tool* that would alleviate the difficulty. Testing and debugging would include finding objects in the world, checking to see if an animation is correct, fixing a bug in an external script, and other tasks. *Please try to think of at least 3, and include more if you can on the back of this paper. For this question, assume that Alice is bug-free, stable, and fast.*

1. _____

_____2. _____

_____3. _____

Questionnaire for the Alice lab study in Chapter 3

ID _____

Do you have any other ideas for improving the Alice language or environment?

Questionnaire for the Alice lab study in Chapter 3

ID _____

Opinions on Alice

For each statement below, simply circle the degree to which you disagree or agree. *For this question, assume that Alice is bug-free, stable, and fast.*

	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Object and world data is difficult to see when programming in Alice							
Modifying code is easy to do in Alice	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Creating code in Alice is straightforward	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
I find myself debugging in different ways in Alice than other languages	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
I like to use Alice to create worlds	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
I have to think differently about programming when using Alice	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Learning Alice was easy	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Reusing code from other worlds is straightforward	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Programming in Alice is tedious	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Alice is good for virtual worlds, but nothing else	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
It is easy to access object and world data when running a world	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Alice would be a good environment for beginning programmers	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
Major design changes in Alice are difficult to accomplish	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
I use Alice features for unintended purposes	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
I never want to use Alice after this course	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree
When my program doesn't work, it is easy to see what's wrong	Absolutely Disagree	Disagree	Somewhat Disagree	Neutral	Somewhat Agree	Agree	Absolutely Agree

Coding sheet for the VB.NET classroom study in Chapter 4**Programming System:** *Visual Basic.NET* *Flash***Date:**

Explain as many details about the problem as possible, including:

- Interface involved,
- Roles of variables,
- Error responsible for failure
- Behavior desired
- Language construct involved,
- Strategies used

Source of observation→	<input type="checkbox"/> E-mail	<input type="checkbox"/> Face-to-face	<input type="checkbox"/> Over the shoulder
------------------------	---------------------------------	---------------------------------------	--

Explain as many details about the problem as possible, including:

- Interface involved,
- Roles of variables,
- Error responsible for failure
- Behavior desired
- Language construct involved,
- Strategies used

Source of observation→	<input type="checkbox"/> E-mail	<input type="checkbox"/> Face-to-face	<input type="checkbox"/> Over the shoulder
------------------------	---------------------------------	---------------------------------------	--

Explain as many details about the problem as possible, including:

- Interface involved,
- Roles of variables,
- Error responsible for failure
- Behavior desired
- Language construct involved,
- Strategies used

Source of observation→	<input type="checkbox"/> E-mail	<input type="checkbox"/> Face-to-face	<input type="checkbox"/> Over the shoulder
------------------------	---------------------------------	---------------------------------------	--

Explain as many details about the problem as possible, including:

- Interface involved,
- Roles of variables,
- Error responsible for failure
- Behavior desired
- Language construct involved,
- Strategies used

Source of observation→	<input type="checkbox"/> E-mail	<input type="checkbox"/> Face-to-face	<input type="checkbox"/> Over the shoulder
------------------------	---------------------------------	---------------------------------------	--

Task descriptions for the Eclipse study in Chapter 5

Paint Application Agenda

User Complaints

- Users have complained that scroll bars don't always appear after painting outside the canvas. However, when they do appear, the canvas doesn't look right. Fix *Paint* so that (1) the scroll bars appear immediately when painting outside the visible canvas and (2) the canvas is correctly rendered when using the scroll bars to navigate the canvas.
- Users have complained that they can't select the color yellow (a mix of red and green). Fix *Paint* so that users can paint with the color yellow.
- Users have complained that the *Undo my last stroke* button doesn't always work. Fix *Paint* so that the *Undo my last stroke* button undoes the last stroke or clear of the canvas.

User Requests

- Users have requested a line tool. We have created a radio button for the line tool, but it doesn't work yet. Create a line tool for *Paint* so that users can select the line tool, and click and drag from one point to another to create a line stroke. Users should be able to see the line as they are dragging.
- Users have requested control over the stroke thickness of the pencil and eraser tools (and line tool, if you've created it). Create a line thickness slider for *Paint* that sets the line thickness from values 1 to 50, which controls the thickness of the stroke for all of the tools.

Actions.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import javax.swing.*;
import java.awt.event.*;

public class Actions {
    public AbstractAction clearAction, undoAction, pencilAction, eraserAction;
    private PaintWindow paintWindow;

    public Actions(PaintWindow window) {
        this.paintWindow = window;
        clearAction = new AbstractAction() {
            public void actionPerformed(ActionEvent actionEvent) {
                paintWindow.clear();
            }
        };
        clearAction.putValue(Action.NAME, "Clear the canvas");
        undoAction = new AbstractAction() {
            public void actionPerformed(ActionEvent actionEvent) {
                paintWindow.undo();
            }
        };
        undoAction.putValue(Action.NAME, "Undo my last stroke");
        pencilAction = new AbstractAction() {
            public void actionPerformed(ActionEvent actionEvent) {
                paintWindow.setPaintObjectClass(PencilPaint.class);
            }
        };
        pencilAction.putValue(Action.NAME, "Pencil");
        eraserAction = new AbstractAction() {
            public void actionPerformed(ActionEvent actionEvent) {
                paintWindow.setPaintObjectClass(EraserPaint.class);
            }
        };
        eraserAction.putValue(Action.NAME, "Eraser");
    }
}
```

EraserPaint.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import java.awt.*;

public class EraserPaint extends PencilPaint {

    public void setColor(Color color) {
        this.color = Color.white;
    }

    public void setThickness(int thickness) {
        this.thickness = 25;
    }

}
```

PaintObject.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import java.awt.*;

public abstract class PaintObject {

    protected int thickness;
    protected Color color;

    public void setColor(Color color) { this.color = color; }
    public void setThickness(int thickness) { this.thickness = thickness; }

    public abstract double getStartX();
    public abstract double getStartY();
    public abstract double getEndX();
    public abstract double getEndY();

    public abstract Rectangle getBoundingBox();
    public abstract void paint(Graphics2D g);
    public abstract void define(Point[] points);

}
```

PaintObjectConstructorListener.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;

public interface PaintObjectConstructorListener {

    public void constructionBeginning(PaintObject temporaryObject);
    public void constructionContinuing(PaintObject temporaryObject);
    public void constructionComplete(PaintObject finalObject);
    public void hoveringOverConstructionArea(PaintObject hoverObject);

}
```

PencilPaint.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;

import java.awt.*;

public class PencilPaint extends PaintObject {

    Point[] points;

    public PencilPaint() {

    }

    public double getStartX() { return points[0].getX(); }
    public double getStartY() { return points[0].getY(); }
    public double getEndX() { return points[points.length - 1].getX(); }
    public double getEndY() { return points[points.length - 1].getY(); }

    public void define(Point[] points) {

        this.points = points;
    }

    public Rectangle getBoundingBox() {

        int minX = 100000, minY = 100000;
        int maxX = 0, maxY = 0;

        for(int pointIndex = points.length - 1; pointIndex >= 0; pointIndex--) {

            int x = (int)points[pointIndex].getX();
            int y = (int)points[pointIndex].getY();
                if(x - thickness / 2 < minX) minX = x - thickness / 2;
                else if(x + thickness / 2 > maxX) maxX = x + thickness / 2;
                if(y - thickness / 2 < minY) minY = y - thickness / 2;
                else if(y + thickness / 2 > maxY) maxY = y + thickness / 2;

        }

        return new Rectangle(minX, minY, maxX - minX, maxY - minY);
    }

    public void paint(Graphics2D g) {

        Stroke oldStroke = g.getStroke();
        g.setStroke(new BasicStroke(thickness));
        g.setColor(color);

        for(int pointIndex = points.length - 1; pointIndex >= 1; pointIndex--) {

            Point one = points[pointIndex];
            Point two = points[pointIndex - 1];
            g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(),
            (int)two.getY());

        }

        g.setStroke(oldStroke);
    }
}
```

PaintCanvas.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import javax.swing.*;
import java.awt.*;
import java.util.*;

public class PaintCanvas extends JPanel {

    Vector history;
    Vector paintObjects;

    private PaintObject temporaryObject;
    private PaintObject hoveringObject;

    public PaintCanvas(int initialWidth, int initialHeight) {
        setPreferredSize(new Dimension(initialWidth, initialHeight));
        paintObjects = new Vector();
        history = new Vector();
    }

    public void paintComponent(Graphics g) {
        ((Graphics2D) g).addRenderingHints(
            new java.awt.RenderingHints(
                java.awt.RenderingHints.KEY_ANTIALIASING,
                java.awt.RenderingHints.VALUE_ANTIALIAS_ON));

        Rectangle clipBounds = g.getClipBounds();
        g.setColor(Color.white);
        g.fillRect((int)clipBounds.getX(), (int)clipBounds.getX(),
                   (int)clipBounds.getWidth(), (int)clipBounds.getHeight());

        Iterator paintObjectIterator = paintObjects.iterator();
        while(paintObjectIterator.hasNext())
            try {
                ((PaintObject)paintObjectIterator.next()).paint((Graphics2D)g);
            } catch(Exception e) {
                System.err.println("The graphics context isn't a
Graphics2D. No anti-aliasing!");
            }
        if(temporaryObject != null) temporaryObject.paint((Graphics2D)g);

        if(hoveringObject != null) {
            Rectangle rect = hoveringObject.getBoundingBox();
            g.setColor(Color.black);
            g.drawRect((int)rect.getX() - 1, (int)rect.getY() - 1,
                       (int)rect.getWidth() + 2, (int)rect.getHeight() + 2);
            hoveringObject.paint((Graphics2D)g);
        }
    }

    public int sizeOfHistory() { return history.size(); }
```

PaintCanvas.java, of Paint, from the study in Chapter 5

```
public void setTemporaryObject(PaintObject temporaryObject) {  
    this.temporaryObject = temporaryObject;  
    repaint();  
}  
  
public void setHoveringObject(PaintObject hoveringObject) {  
    this.hoveringObject = hoveringObject;  
    repaint();  
}  
  
public void addPaintObject(PaintObject newObject) {  
    history.addElement(new Vector(paintObjects));  
    paintObjects.addElement(newObject);  
    repaint();  
}  
  
public void clear() {  
    history.addElement(new Vector(paintObjects));  
    paintObjects.removeAllElements();  
    repaint();  
}  
  
public void undo() {  
    paintObjects = (Vector)history.lastElement();  
    history.removeElement(history.lastElement());  
}  
  
}
```

PaintObjectConstructor.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import java.util.*;
import java.awt.event.*;
import java.awt.*;

public class PaintObjectConstructor implements MouseListener, MouseMotionListener {

    private Vector pointsGathered;
    private PaintObjectConstructorListener constructorListener;
    private Class paintObjectClass;
    private PaintObject temporaryObject;

    private Color color;
    private int thickness;

    public PaintObjectConstructor(PaintObjectConstructorListener listener) {
        this.constructorListener = listener;
    }

    public void setThickness(int thickness) { this.thickness = thickness; }
    public void setColor(Color color) { this.color = color; }
    public Color getColor() { return this.color; }
    public void setClass(Class paintObjectClass) { this.paintObjectClass = paintObjectClass; }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}

    public void mouseExited(MouseEvent e) {
        constructorListener.hoveringOverConstructionArea(null);
    }

    public void mouseMoved(MouseEvent e) {

        constructorListener.hoveringOverConstructionArea(makeHoveringPrototype(e.getPoint()));

    }

    public void mousePressed(MouseEvent e) {
        pointsGathered = new Vector();
        pointsGathered.addElement(e.getPoint());

        try {
            temporaryObject = (PaintObject)paintObjectClass.newInstance();
        } catch(Exception exception) {
            System.err.println("There was a problem making the paint object.");
        }

        temporaryObject.setColor(color);
        temporaryObject.setThickness(thickness);

        temporaryObject.define((Point[])pointsGathered.toArray(new Point[pointsGathered.size()]));

        constructorListener.hoveringOverConstructionArea(makeHoveringPrototype(e.getPoint()));
        constructorListener.constructionBeginning(temporaryObject);
    }
}
```

PaintObjectConstructor.java, of Paint, from the study in Chapter 5

```
public void mouseDragged(MouseEvent e) {  
    pointsGathered.addElement(e.getPoint());  
    temporaryObject.define((Point[])pointsGathered.toArray(new  
    Point[pointsGathered.size()]));  
  
    constructorListener.hoveringOverConstructionArea(makeHoveringPrototype(e.getPoint()));  
    constructorListener.constructionContinuing(temporaryObject);  
}  
  
public void mouseReleased(MouseEvent e) {  
    pointsGathered.addElement(e.getPoint());  
    temporaryObject.define((Point[])pointsGathered.toArray(new  
    Point[pointsGathered.size()]));  
    constructorListener.constructionComplete(temporaryObject);  
    constructorListener.hoveringOverConstructionArea(null);  
  
    pointsGathered = null;  
    temporaryObject = null;  
}  
  
private PaintObject makeHoveringPrototype(Point p) {  
    PaintObject prototype = null;  
    try {  
        prototype = (PaintObject)paintObjectClass.newInstance();  
    } catch(Exception exception) {  
        System.err.println("There was a problem making the paint  
object.");  
    }  
    Point[] points = new Point[2];  
    points[0] = points[1] = p;  
    prototype.define(points);  
    prototype.setColor(color);  
    prototype.setThickness(thickness);  
  
    return prototype;  
}  
}
```

PaintWindow.java, of Paint, from the study in Chapter 5

```
package edu.cmu.hcii.paint;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

public class PaintWindow extends JFrame implements PaintObjectConstructorListener {

    private PaintCanvas canvas;
    private JButton clearButton, undoButton;
    private JPanel clearUndoPanel;
    private JRadioButton pencilButton, eraserButton, lineButton;
    private JPanel toolPanel;
    private JPanel rPanel, gPanel, bPanel;
    private JSlider rSlider, bSlider, gSlider;
    private JPanel colorPanel;
    private JPanel controlPanel;
    private JScrollPane canvasPane;
    private Actions actions;

    private ButtonGroup toolButtonGroup;

    private PaintObjectConstructor objectConstructor;

    private ChangeListener colorChangeListener = new ChangeListener() {
        public void stateChanged(ChangeEvent changeEvent) {
            objectConstructor.setColor(new Color(rSlider.getValue(),
                gSlider.getValue(), bSlider.getValue()));
            repaint();
        }
    };

    private JComponent currentColorComponent = new JComponent() {
        public void paintComponent(Graphics g) {
            Color oldColor = g.getColor();
            g.setColor(objectConstructor.getColor());
            g.fillRect(0, 0, getWidth(), getHeight());
            g.setColor(oldColor);
        }
    };
}
```

PaintWindow.java, of Paint, from the study in Chapter 5

```
public PaintWindow(int initialWidth, int initialHeight) {  
    super("Paint");  
    actions = new Actions(this);  
    setResizable(true);  
    setBackground(new Color(128, 10, 160));  
  
    canvas = new PaintCanvas(initialWidth, initialHeight);  
    closeButton = new JButton(actions.clearAction);  
    closeButton.setOpaque(false);  
    undoButton = new JButton(actions.undoAction);  
    undoButton.setOpaque(false);  
  
    clearUndoPanel = new JPanel();  
    clearUndoPanel.setOpaque(false);  
    clearUndoPanel.setLayout(new BoxLayout(clearUndoPanel, BoxLayout.Y_AXIS));  
    clearUndoPanel.add(closeButton);  
    clearUndoPanel.add(undoButton);  
  
    pencilButton = new JRadioButton(actions.pencilAction);  
    pencilButton.setOpaque(false);  
    pencilButton.setSelected(true);  
    eraserButton = new JRadioButton(actions.eraserAction);  
    eraserButton.setOpaque(false);  
    lineButton = new JRadioButton("Line");  
    lineButton.setOpaque(false);  
  
    toolButtonGroup = new ButtonGroup();  
    toolButtonGroup.add(pencilButton);  
    toolButtonGroup.add(eraserButton);  
    toolButtonGroup.add(lineButton);  
  
    toolPanel = new JPanel();  
    toolPanel.setOpaque(false);  
    toolPanel.setLayout(new BoxLayout(toolPanel, BoxLayout.Y_AXIS));  
    toolPanel.add(pencilButton);  
    toolPanel.add(eraserButton);  
    toolPanel.add(lineButton);  
  
    rPanel = new JPanel(new FlowLayout());  
    rPanel.setOpaque(false);  
    rPanel.add(new JLabel("Red"));  
    rSlider = new JSlider(0, 255, 0);  
    rSlider.setOpaque(false);  
    rSlider.addChangeListener(colorChangeListener);  
    rPanel.add(rSlider);  
  
    gPanel = new JPanel(new FlowLayout());  
    gPanel.setOpaque(false);  
    gPanel.add(new JLabel("Green"));  
    gSlider = new JSlider(0, 255, 255);  
    gSlider.setOpaque(false);  
    gSlider.addChangeListener(colorChangeListener);  
    gPanel.add(gSlider);  
  
    bPanel = new JPanel(new FlowLayout());  
    bPanel.setOpaque(false);  
    bPanel.add(new JLabel("Blue"));  
    bSlider = new JSlider(0, 255, 0);  
    bSlider.setOpaque(false);  
    bSlider.addChangeListener(colorChangeListener);  
    bPanel.add(bSlider);
```

PaintWindow.java, of Paint, from the study in Chapter 5

```
colorPanel = new JPanel();
colorPanel.setOpaque(false);
colorPanel.setLayout(new BoxLayout(colorPanel, BoxLayout.Y_AXIS));
colorPanel.add(rPanel);
colorPanel.add(gPanel);
colorPanel.add(bPanel);
currentColorComponent.setPreferredSize(new Dimension(100, 50));
colorPanel.add(currentColorComponent);

controlPanel = new JPanel();
GridBagLayout controlPanelGridBag = new GridBagLayout();
GridBagConstraints constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.NONE;
constraints.gridx = 0;
constraints.gridy = 1;
constraints.insets = new Insets(5, 5, 5, 5);
controlPanelGridBag.setConstraints(toolPanel, constraints);
controlPanelGridBag.setConstraints(colorPanel, constraints);
controlPanelGridBag.setConstraints(clearUndoPanel, constraints);
controlPanel.setLayout(controlPanelGridBag);
controlPanel.setOpaque(false);
controlPanel.add(toolPanel);
controlPanel.add(colorPanel);
controlPanel.add(clearUndoPanel);

canvasPane = new JScrollPane(canvas);

getContentPane().setLayout(new BorderLayout());
getContentPane().add(canvasPane, BorderLayout.CENTER);
getContentPane().add(controlPanel, BorderLayout.WEST);

addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent event) {
        System.exit(0);
    }
});

objectConstructor = new PaintObjectConstructor(this);
objectConstructor.setClass(PencilPaint.class);
objectConstructor.setColor(new Color(0, 255, 0));
objectConstructor.setThickness(5);
canvas.addMouseListener(objectConstructor);
canvas.addMouseMotionListener(objectConstructor);

pack();
setVisible(true);
}

public void setPaintObjectClass(Class paintObjectClass) {
    objectConstructor.setClass(paintObjectClass);
}

public void undo() {
    canvas.undo();
    if(canvas.sizeOfHistory() == 0) actions.undoAction.setEnabled(false);
}
```

PaintWindow.java, of Paint, from the study in Chapter 5

```
public void clear() {
    canvas.clear();
}

public void constructionBeginning(PaintObject temporaryObject) {
    canvas.setTemporaryObject(temporaryObject);
}

public void constructionContinuing(PaintObject temporaryObject) {
    canvas.setTemporaryObject(temporaryObject);
}

public void constructionComplete(PaintObject finalObject) {
    canvas.setTemporaryObject(null);
    canvas.addPaintObject(finalObject);
    actions.undoAction.setEnabled(true);
}

public void hoveringOverConstructionArea(PaintObject hoverObject) {
    canvas.setHoveringObject(hoverObject);
}

}
```

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

0. About this user study

This is a between-subject test with 2 groups of users:

- Group A: The “Why” people– 10 users
- Group B: The “Without why” people – 10 users

We try to keep the test script and instructions the same whenever possible for the 2 groups of users. The differences will be noted.

Will be using Camtasia Studio to record both screen and audio (what experimenter and user say, and the sound from computer)

Will also use stop watch to time during study (Use the stop watch on cell-phone, since my stop watch has beeping sound)

Table of Contents

0. About this user study.....	1
1. Preparation.....	2
2. Introduction.....	3
3. Training on using Help	4
4. Tasks	6
5. Pre-questionnaire.....	14
6. Post-questionnaire A.....	15
7. Post-questionnaire B	17

Time Allocation

Section	Time allocation breakdown	Total time for section
Introduction	Reading script 1.5 Pre-questionnaire 2 Read and sign consent form 1.5	5
Training on using Help		2
Tasks	Reading introduction script 1 Task 1: ounce 3 Task 2: teh 3 Task 3: alignment 3 Task 4: grayed out 3 Task 5: styles 6	18
Post questionnaire		5
		Total: min 30

Study materials for the Crystal evaluation study in Chapter 9Sept 2005Participant #**1. Preparation**

Crystal MUST BE ON TOP LEFT, so that other windows that pop up won't be hidden

1. Start Camtasia studio
2. There should be 3 instances of Crystal started before study begins. Press F8 for every Crystal instances started for *Group B: The "Without Why" people.*
 - a. One instance for demo (then restart for each of Task 1, 2, 5)
 - b. One instance for Task 3, with text already typed in Very important!!!
 - c. One instance for Task 4, with text already typed in Very important!!!
3. Consent forms (2)
4. Tasks on separate pages (5)
5. Pre-questionnaire
6. Post-questionnaire
 - a. A for Group A: "Why" people
 - b. B for Group B: "Without why" people
7. Subject payment record
8. \$15
9. Mouse

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

2. Introduction

Thanks for coming to this study. I'll be reading to you from a script so as to keep the information similar for all participants.

My name is ... and I'm a researcher from Carnegie Mellon University's Human Computer Interaction Institute. Our research team is investigating how to make it easier for people to understand and use the features of a word processor that we developed.

Today, you will be completing 5 small tasks using our word processor on this laptop computer. The whole study will take about 30 minutes. We'll record the audio and the computer screen; however, all this information will remain anonymous.

First, we need your consent to participate in this study. Here is the consent form. [Show consent form] Please read and sign it. If you have any questions, please ask me. [Let user read and sign consent form] [Give a blank copy to user]

Next, please complete this questionnaire about your background. [Show and let user complete pre-questionnaire]

Finally, there are two things we'd like you to remember. First, we are testing the word processor and not you. Second, please try to work out the tasks by yourself.

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

3. Training on using Help

Let us begin. [Show Crystal start up screen]

This is a word processor called Crystal. It lets you type and edit text, as well as format and apply styles to the text.

Only for the "Why People":

It also has a Help feature that lets you ask questions about things that have happened but you don't know the reason. Now let me demonstrate how to use the Help feature.

[Type abc]
[Hit "b" button in the toolbar to turn on bold]
[Type def]
[Hit "i" button in the toolbar to turn on italic]
[Type ghi]

You just saw how I made some of the text bold and italic. But just imagine you didn't know why that happened. You can find out why by clicking the Why? menu in the menu bar to ask the word processor to explain the last few things that you did. [Click Why? menu] In my case, I want to know why the text was made italic, so I click this. [Click "The text was made italic"] It explains to me down here why the text was made italic, and it also highlights the relevant part in the interface that caused what happened. In this case, it is the "i" button.

[Click "Close" to close explanation window] You can also ask questions by pointing your mouse and then hitting F1 on the keyboard. For example, I point to the character "d" [Point mouse to "d"] and hit "F1" [Hit "F1"] and ask why it is bold [Click "Why is the d bold" on the pop-up menu] You don't have to select the items that you're interested in; you just need to put your mouse cursor over it, and then hit F1.

You can point the mouse anywhere, even on blank areas and the menus, and then hit F1 to ask questions. For example, watch as I pop up the Edit menu, and hit the F1 key over "Copy", and it will tell us that Copy is grayed out because nothing is selected.

So, in short, the Help feature of Crystal can help you find out why something has happened. The Why feature also tries to be helpful and show you how to do things, especially when they get complicated. If you are trying to figure out how to do something, be sure to try getting help about why the things are the way they are.

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

4. Tasks

[Camtasia starts to record screen and audio]

Introduction

We would like you to do 5 small tasks with this word processor, one at a time. Let's start with the first task. [Show Task 1] The instructions tell you what you have to do to complete the task.

Please read out loud the instructions once. Then, if you don't have any questions, you may start.

[Let participant read out instructions]

Remember you can use the Help feature if you get stuck.

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 1 "ounce"

Time taken (from the moment the user intends to look for the check box to the moment he / she clicks on it): _____ seconds

Task 1

1. Type in the following sentence "The abbreviation fl. oz. stands for fluid ounce."
2. You notice that the word processor has capitalized some characters for you, but you don't want this to happen.
3. Your task is to make the automatic capitalization not happen again.
4. When you think you're done, type "fl. oz. stands" again to make sure it works.

Goals

1. Find out if users know what setting (auto capitalization) they are looking for
2. Find out how long it takes the users to realize they are looking for that setting
3. Find out how long it takes the users to bring up the interface (AutoCorrect dialog) and locate the widget (the check box "Capitalize first letter...") that will change the setting.
4. Compare the times used for the 2 groups of users

Measures

1. Length of time from the moment the user intends to look for the check box to the moment he / she explicitly locates it and clicks on it. If the user gives up, the experimenter will prompt them to try to keep looking. However, if they really want to give up, they will have to be allowed to do this, and the time will be a special value, like not-finished.
2. The number of irrelevant or incorrect widgets that the user clicks on before clicking on the correct check box
3. Whether the user is satisfied about all the interaction that he / she needs to perform in order to locate and click on the check box

Evaluating success or failure

Success:

- Get the right widget eventually (i.e. the check box)

Failure:

- not (Success)

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 2 "teh"

Time taken (from the moment the user intends to look for the check box to the moment he / she clicks on it): _____ seconds

Task 2

1. Type in the following sentence "This word processor helps me change teh into the."
2. You notice that the word processor has changed "teh" into "the" for you, but you don't want this to happen.
3. Your task is to make the correction not happen again.
4. When you think you're done, type "change teh into the" again to make sure it works.

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 3 “alignment”

Time taken (from the moment the user starts any action to text is correctly formatted):
seconds

[Bring up the Crystal instance with the paragraph already typed. It should look like Figure A.]

Task 3

1. Somebody has sent you a document that looks like Figure A, and you want it to look like Figure B.
2. Your task is to make it look like Figure B.

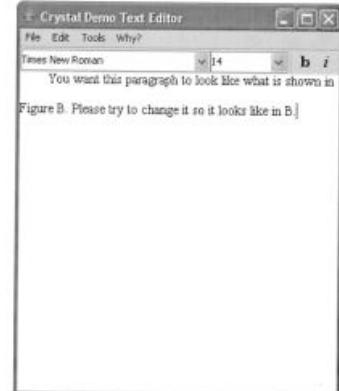


Figure A How it looks like now

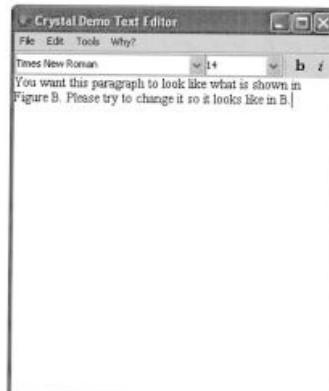


Figure B How you want it to look like

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 4 “grayed out”

[Bring up a new Crystal instance. Copied an image in XARA]

For this task, I'm going to ask you a question:

1. Can you figure out why the paste function under the “edit” menu is shown in gray?

Possible answer 1: “because you haven't copied anything”

Response 1: “That isn't right. Can you come up with another reason?”

Possible answer 2: “because what's been copied isn't text”

Response 2: “Right!”

If got stuck 1: “see if you can get the answer from the system”

If got stuck 2: “Just to remind you that you can use the Help feature if you get stuck”

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 5 "styles"**5A: Time taken** (from the moment the user starts any action to text is correctly formatted):
_____ seconds**5B: Time taken** (from the moment the user starts any action to text is correctly formatted):
_____ seconds

Style	Based on	Settings different from base Style
Header	Default	First-line indent = 0 Font size = 20 Bold = yes
Sub-Header	Header	Bold = no Italic = yes

[Bring up the Crystal instance with the text already typed. It should look like Figure C]

Here in the word processor, some text has already been typed in. Please take some time to skim through the text. [Allow about 15-20 seconds for users to skim through it] Let me know when you're done.

Now, here are the task instructions. Please read out loud the instructions once. Then, if you don't have any questions, you may start.

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 5A

1. Imagine this is part of a really long document, consisting of hundreds of paragraphs and headings.
2. And suppose you want to make all the sub-headings (like "Location" and "Size") not be italic, without changing the main headings (like "Geography of the USA") and without changing any of the words in the regular paragraphs.
3. Try to figure out how you can do this in one step

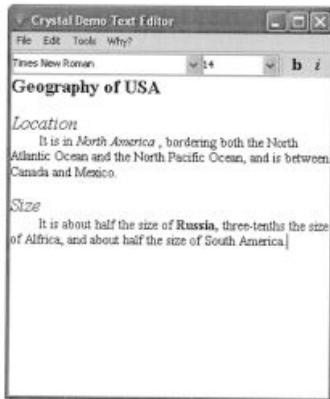


Figure C

If they just select all and change the italics, say: "Notice that you changed the italics for the contents. Please use UNDO to undo that change, and try to find a way to do it without changing the contents."

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

Task 5B

1. Now all of the headings are not italic, and they are all of size 20
2. You want to make all of the headings, both the main headings like “Geography of the USA”, and the sub-headings like “Location”, be of size 40
3. Try to figure out how you can do this in one step

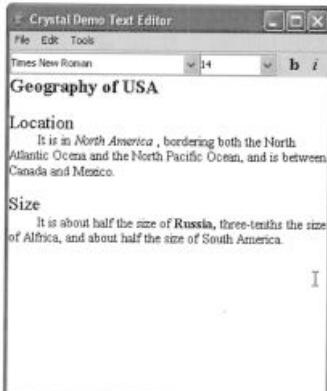


Figure D

If they change the size for sub-headings only, say: “that didn’t change the size of the main headings. Please UNDO that change, and try to find a way to change both kinds of headings in one step.

If they get stuck (either group A or B), then give them the hint: “Maybe looking at the Based On property of the styles will help you understand how to do this.”

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

5. Pre-questionnaire

1. What is your gender? Male / Female
2. What is your age? _____
3. What is your occupation? _____
If you are a student, what is your field of study? _____
4. Is English your first language? Yes / No
If No, how many years have you been spoken English? _____
5. Which computer platform do you use most often?
 - A. Windows
 - B. Macintosh
 - C. Linux
 - D. Unix
 - E. I've never used any
6. How often do you use a computer?
 - A. Daily
 - B. 1-5 times a week
 - C. 1-5 times a month
 - D. Other. Please specify: _____
 - A. Never
7. Which word processor do you use most often?
 - A. Microsoft Word
 - B. Corel WordPerfect
 - C. Sun StarOffice
 - D. Other. Please specify: _____
 - E. I've never used any
8. How often do you use a word processor?
 - A. Daily
 - B. 1-5 times a week
 - C. 1-5 times a month
 - D. Other. Please specify: _____
 - E. Never
9. If you have used a word processor, when was the last time you used it? _____
What did you do? _____
10. How would you rate your familiarity / expertise in using word processor programs?
 - A. No knowledge
 - B. Beginner
 - C. Intermediate
 - D. Advanced
 - E. Expert

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

6. Post-questionnaire A

Circle your level of agreement with each statement.

	Strongly disagree	Neutral	Strongly agree
1. I understand how to use the Help feature in Crystal	1	2	3
2. I found the Help feature easy to use	4	5	6
Task 1			
3. I was able to complete the task easily	1	2	3
4. The Help feature helped me in dealing with the task	4	5	6
5. I could have completed the task more quickly without using the Help feature	7	1	2
Task 2			
6. I was able to complete the task easily	1	2	3
7. The Help feature helped me in dealing with the task	4	5	6
8. I could have completed the task more quickly without using the Help feature	7	1	2
Task 3			
9. I was able to complete the task easily	1	2	3
10. The Help feature helped me in dealing with the task	4	5	6
11. I could have completed the task more quickly without using the Help feature	7	1	2
Task 4			
12. I was able to complete the task easily	1	2	3
13. The Help feature helped me in dealing with the task	4	5	6
14. I could have completed the task more quickly without using the Help feature	7	1	2
Task 5A			
15. I was able to complete the task easily	1	2	3
16. The Help feature helped me in dealing with the task	4	5	6
17. I could have completed the task more quickly without using the Help feature	7	1	2
Task 5B			
18. I was able to complete the task easily	1	2	3
19. The Help feature helped me in dealing with the task	4	5	6
20. I could have completed the task more quickly without using the Help feature	7	1	2

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

	Strongly disagree	Neutral	Strongly agree
21. The Help feature improved my word-processing experience	1	2	3 4 5 6 7
22. The answers provided by the Help feature were easy to understand	1	2	3 4 5 6 7
23. The answers provided by the Help feature were what I wanted to know	1	2	3 4 5 6 7
24. I was comfortable using the Help feature	1	2	3 4 5 6 7
25. I was curious about how to use the Help feature	1	2	3 4 5 6 7
26. Using the Help feature was intuitive	1	2	3 4 5 6 7
27. I felt confident in using the Help feature to learn about the questions I may want to ask	1	2	3 4 5 6 7
28. I would really like a help feature like this in the programs I use	1	2	3 4 5 6 7

Consider the following questions like a brainstorming session. Take as much time as you need to think about and write down anything relevant to the question.

29. What did you find most helpful about the word processor when dealing with the tasks?

30. What would you suggest to improve about the Help feature in Crystal?

Study materials for the Crystal evaluation study in Chapter 9

Sept 2005

Participant #

7. Post-questionnaire B*Circle your level of agreement with each statement.*

	Strongly disagree	Neutral	Strongly agree
Task 1			
1. I was able to complete the task easily	1 2 3 4 5 6 7		
2. I would really like a help feature to tell me why things didn't happen as I had expected during the task	1 2 3 4 5 6 7		
Task 2			
3. I was able to complete the task easily	1 2 3 4 5 6 7		
4. I would really like a help feature to tell me why things didn't happen as I had expected during the task	1 2 3 4 5 6 7		
Task 3			
5. I was able to complete the task easily	1 2 3 4 5 6 7		
6. I would really like a help feature to tell me why things didn't happen as I had expected during the task	1 2 3 4 5 6 7		
Task 5A			
7. I was able to complete the task easily	1 2 3 4 5 6 7		
8. I would really like a help feature to tell me why things didn't happen as I had expected during the task	1 2 3 4 5 6 7		
Task 5B			
9. I was able to complete the task easily	1 2 3 4 5 6 7		
10. I would really like a help feature to tell me why things didn't happen as I had expected during the task	1 2 3 4 5 6 7		

Questionnaire for the Java Whyline evaluation study in Chapter 10

Questionnaire



What is your age?
_____ years old

What is your native language?

Rate your experience with the **Java** programming language by *drawing a mark somewhere on the line.*
beginner _____ | expert

How many **Java** programs have you written?
1 5 10 100 250 500 1000 > 1000

How many total years have you held a job in the **software industry** involving programming, including internships at software companies?
Round up to the nearest whole number.

never worked in industry _____ years

In the past year, **what programming language** have you used the most?

In the past year, **what programming environment or IDE** have you used the most?

How important are the following types of debugging tools to your work? *Draw a mark somewhere on the line.*

never used useless _____ | essential

If there are other debugging tools that are essential to your work, write them here.

Think of the **most complex software project** that you've ever worked on. *Approximately how many classes did it contain (if not object-oriented, how many files?)*

classes _____ OR # files _____

Think of the **typical software project** that you work on. *Approximately how many classes did it contain (if not object-oriented, how many files?)*

classes _____ OR # files _____

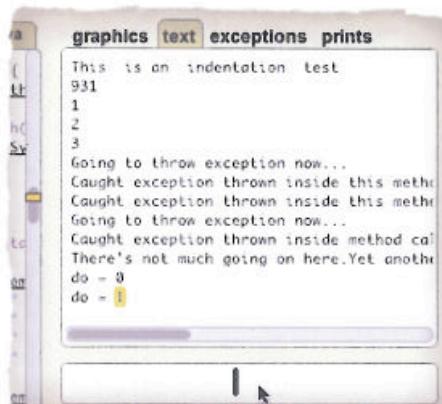
Describe the **kind of software** you typically work on, for example, *user interfaces, embedded software, server-side web development, client-side web development, cell phones, etc.*

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Recordings



This tool works by using a saved history of a program's execution.



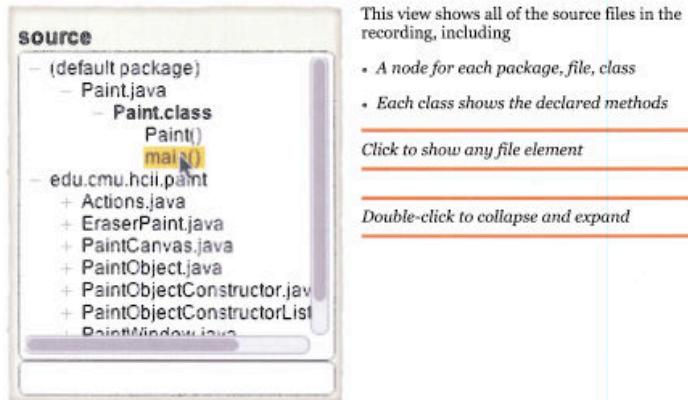
You can view the input/output history of the program using this time controller.

Drag to change the state of the program output to review what happened while the program executed.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Source Outline



next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

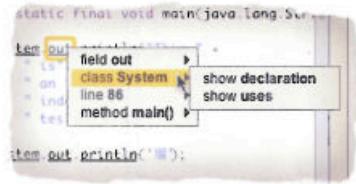
Source Files



Source files present a typical syntax-colored display of Java source code.

As you hover with your mouse, three types of things can be selected:

- *Tokens*
- *Lines*
- *Methods*



Click highlighted a file element to see its menu

The menu includes commands to

- *Go to declaration*
- *Show callers*
- *Bookmark line*
- *Copy line text*

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Searching

The screenshot shows the Java Whyline interface with a search results tab open. The tab title is "button". The search results list several file locations where the word "button" appears:

File	Line Number	Text
PaintWindow....	10	private JButton clearColorButton,
PaintWindow....	12	private JRadioButton
PaintWindow....	21	private ButtonGroup
PaintWindow....	63	clearButton = new
PaintWindow....	64	clearButton.setOpaque(false)
PaintWindow....	65	undoButton = new
PaintWindow....	66	undoButton.setOpaque(false)
PaintWindow....	71	clearUndoPanel.add(clearBu
PaintWindow....	72	clearUndoPanel.add(undoBu
PaintWindow....	74	pencilButton = new
PaintWindow....	75	pencilButton.setOpaque(false)

Below the results, a message says "Done searching... search for 'button'...". To the right of the search results, there are two explanatory text blocks:

Searches are **case insensitive** and will look inside of each token for matches to your query. Each search appears as a tab near the bookmarks and history.

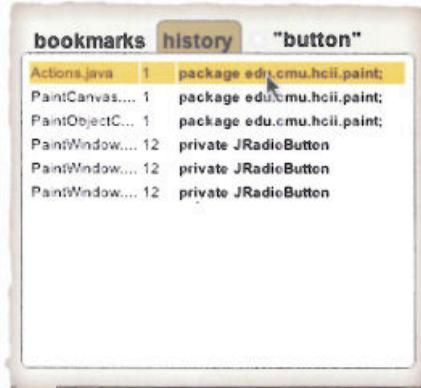
Searches will search all of the source in the recording and all of the available source for JDK libraries.

Click the "x" at the bottom of a tab to close a results tab.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Navigating



Every time you navigate to somewhere new, this navigation is recorded in the **history** tab.

To return to previously viewed locations, press the back button on the toolbar, the 'T' key, or select the location in the **history** tab. Return with the ']' key or the forward button on the toolbar.

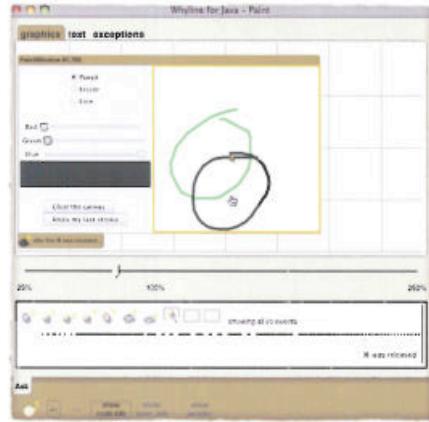
You can also **bookmark** lines to remember important places.

Type 'b' to bookmark the currently selected line, or choose the line and select the **bookmark** command.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Input and Output



The Whyline allows you to ask questions about **program output**.

Like in Google Maps, you can zoom in and out of the input with the **scroll wheel**. You can also click and drag to pan around the screen.

Part of asking questions about output is finding the output in the program's output history.

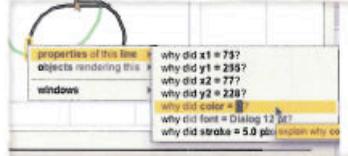
Drag the **time controller** in the history to see the program's output at different points in time.

Click the filtering buttons to find particular kinds of I/O events.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Asking Questions

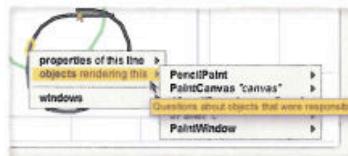


To ask a question, click on primitive output and a menu will appear.

You can ask questions about

- Primitive output (e.g. lines, circles, text)
- Objects that affected output

If you select the primitive output in the menu, you will find questions about each of the output's properties, such as color, position and font.



If you select an object, you will see questions about the values of the object's fields at the current time.

To ask why a particular object was instantiated, choose the "get created?" question.



If a field points to an object, you can see a menu of questions about the object as well.

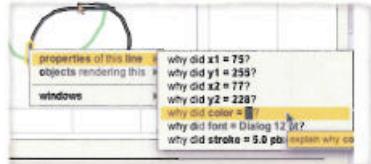
Fields are grouped by kind:

- Primitive type
- Common suffixes (e.g., "...Listener")
- Other fields

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Viewing Answers



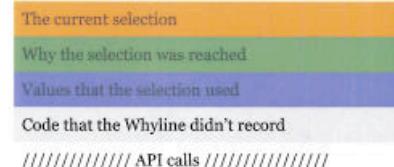
In this example, even though the blue slider is moved all the way up, this stroke is still black. Imagine you want to know why it was black.

To find out, you click on the black stroke and ask "why did this line's color = black?"



The Whyline responds by showing a **timeline visualization** of the events responsible for black being used.

- The Whyline only shows events relevant to your question
- The beginning of the program is on the left
- Time goes to the right



The Whyline uses particular colors for particular meanings.

The current selection
Why the selection was reached
Values that the selection used

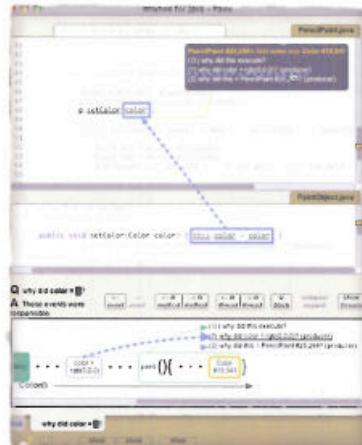
Code that the Whyline didn't record

/////////// API calls ///////////

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Data Dependencies



The blue arrows represent data dependencies.

In this example, we are curious about where the **black** color came from. This is dependency (1) in the answer.

Click or type the dependency number to show the value's origin.

The Whyline then jumps to the origin of the value and shows the corresponding code.

This value came from this instantiation, which depended on three slider values.

The arrows in the source file correspond to the arrows in the visualization. Clicking either has the same effect.

*If you don't want to jump to the source of a value, but instead want to see its **direct dependency**, hold down the **command** key.*

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Keyboard Commands

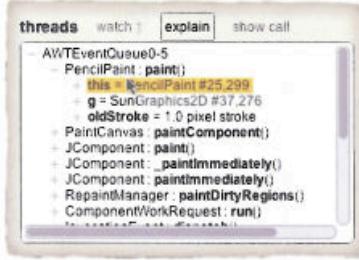
There are also several other useful keyboard commands that can be used to move the selection.

key	action
left	go to previous event
right	go to next event
<	go to previous event in method call
>	go to next event in method call
command-left	go to previous event in thread
command-right	go to next event in thread
up arrow	go to event that caused selection to execute
down arrow	go to event that selection caused to execute
[go to previous selection (back)
]	go to next selection (forward)
type a number	go to source of value
command + a digit	go to direct producer of value
escape	collapse/expand sequence of events
b	bookmark the line of the current selection

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Threads, Call Stacks and Locals



The **threads** view shows:

- All active threads
- Methods in progress at the current time
- Values of local variables within each method

Double-click to collapse or expand any of these

Clicking the **explain** button asks the question
"why did this local or field have its current value?

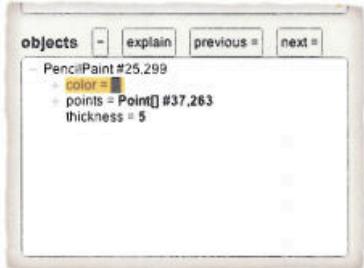
Clicking the **show call** button shows the event
corresponding to the selected method call.

If you find an object that you want to explore
further, click the **watch** button.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Objects



Once an object is placed in the **objects** view, you can explore its fields' current values. If a field points to an object, you can explore that object's fields too.

Values for fields are **not** shown immediately, since sometimes they can be time-consuming to determine.

Click on a field to see its current value.

Press the '-' button to remove an object from the objects list.

*Press the **explain** button to ask "why did the selected field have its current value?"*

Press the "previous =" and "next =" to see previous and next values for the selected field. This will show the corresponding events in the visualization.

next page...

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

**Please don't tell anyone
about the solutions to
the following tasks.**

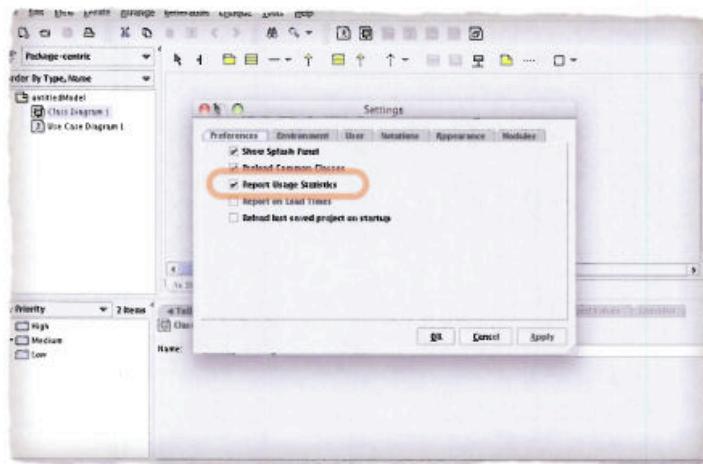
**That would ruin the
results of this study!**

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Task 1

Imagine you work on a team writing an application called *ArgoUML*, a diagramming tool for designing object-oriented software. **Sharon**, the lead developer in charge of user interfaces, is very busy, so she's asked you to investigate a few of the easier bugs in order to prepare for the upcoming release.

The first bug is a problem with a checkbox that has no actual purpose. It's labeled "Report Usage Statistics" and circled in red below. Although the system has support for this feature, it is currently not attached to the checkbox, and so the checkbox should be removed.



You have two responsibilities:

- (1) Find where this checkbox is added in the application's source code.
- (2) Write a change recommendation to Sharon about how to remove the checkbox. It should be detailed enough for her to evaluate and make the change. (If this was your code, you would normally make the change, but since this is Sharon's code, you simply want to find the relevant code for her). You do not need to find the other functionality related to this checkbox; you just need to find the code related to the checkbox.

You should be confident about the correctness of your recommendation (you don't want to waste Sharon's time), but you shouldn't spend too much time understanding the system, since you're on a tight schedule. You can afford to spend up to **30 minutes** on this bug.

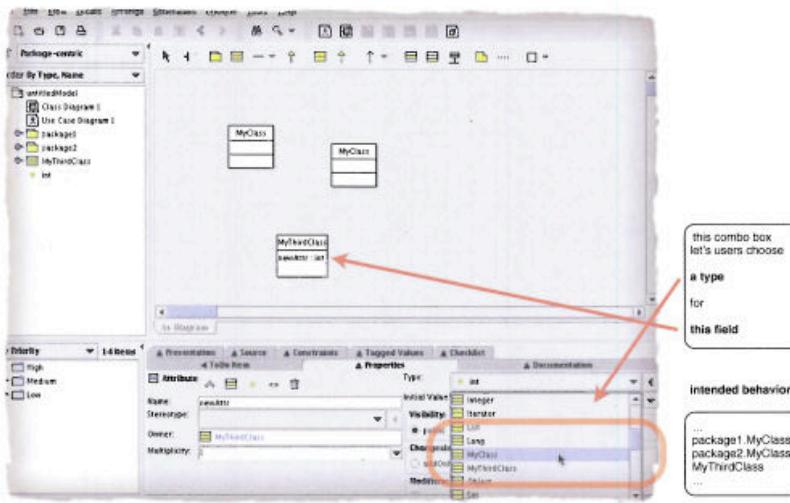
When **5 minutes** remain, I will remind you about writing the recommendation.

Tell the experimenter when you are done with your recommendation.

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

Task 2

Sharon needs your help again, this time on a more complicated bug. In the screen shot below, there are three classes. Two have the same name, but are in different packages. A third class has an attribute. The user is trying to select the attribute's type, but notice that the list only includes "MyClass". It should include two classes named "MyClass," one from each package.



Again, You have two responsibilities.

- (1) Find out why this menu only has one item labeled "MyClass".
- (2) Write a change recommendation, detailed enough for Sharon to understand the cause of the problem. You should also include at least one idea for a solution to the problem.

You should be confident about the correctness of your recommendation (you don't want to waste Sharon's time), but you shouldn't spend too much time understanding the system, since you're on a tight schedule and this is not your code. You can afford to spend up to **30 minutes** on this bug.

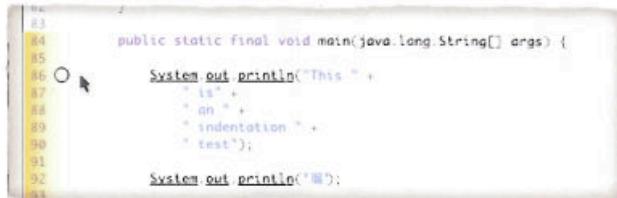
When **5 minutes** remain, I will remind you about writing the recommendation.

Tell the experimenter when you are done with your recommendation.

Tutorial and tasks for the Java Whyline evaluation study in Chapter 10

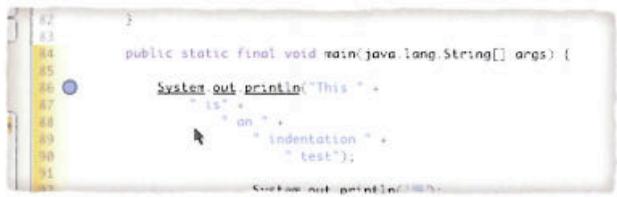
Breakpoints

If you have used a breakpoint debugger before, this one works mostly as you'd expect. When you find a line that you'd like to break on, click the line and choose "turn breakpoint on"



```
86     public static final void main(java.lang.String[] args) {
87         System.out.println("This " +
88             " is" +
89             " on " +
90             " indentation " +
91             " test");
92         System.out.println("■");
93     }
```

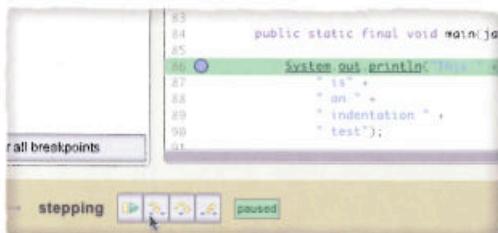
An icon appears to show that the breakpoint is on and it also appears in the "breakpoints" list



```
82
83
84
85
86     public static final void main(java.lang.String[] args) {
87         System.out.println("This " +
88             " is" +
89             " on " +
90             " indentation " +
91             " test");
92         System.out.println("■");
93     }
```

You can remove the breakpoint by clicking on the line and choosing "turn breakpoint off."

To run the program, click the "run" button at the bottom.



```
83
84     public static final void main(ja
85
86     System.out.println("This " +
87         " is" +
88         " on " +
89         " indentation " +
90         " test");
91
92
93
```

Use the "step into" button to go into method calls. Use the "step over" button to jump over a call. Use the "step out" button to jump out of a method call. Press the "stop" button to stop execution and "continue" to run to the next breakpoint, if any. When the program is done executing, the tool will note this on the status bar.

There is also limited support for printing the values of variables while the program executes. Choose a variable and select the "print value before this executes" command.

BIBLIOGRAPHY

- Abraham R. and Erwig M. (2005). Goal-Directed Debugging of Spreadsheets. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, 37-44.
- Akgul T., V. J. M. III, and Pande S. (2004). A Fast Assembly Level Reverse Execution Method via Dynamic Slicing. *International Conference on Software Engineering*, Scotland, UK, 522-531.
- Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. In M. Ikeda, K. D. Ashley, & T. W. Chan (Eds.), *International Conference on Intelligent Tutoring Systems*, Jhongli, Taiwan, 61-70.
- Altmann E. M. (2001). Near-term Memory in Programming: A Simulation-Based Analysis, *International Journal of Human-Computer Studies*, 54, 189-210.
- Anderson J. R. (2000), "Problem Solving," in *Cognitive Psychology and its Implications*, Fifth ed. New York, New York: Worth Publishers, 239-278.
- Anderson P. B., B. Holmqvist, and J. F. Jensen (1993), *The Computer as Medium*. Cambridge: The Cambridge University Press.
- Antoniol G., H. Gall, M. D. Penta, and M. Pinzger (2004). Mozilla: Closing the Circle, *Technical University of Vienna, Vienna, Austria TUV-1841-2004-05*.
- Anvik J., Hiew L., and Murphy G. (2006) Who Should Fix this Bug? *International Conference on Software Engineering*, Shanghai, China, 361-368.
- Auguston M., Jeffery C., and Underwood S. (2002). A Framework for Automatic Debugging. *IEEE International Conference on Automated Software Engineering*, Edinburgh, UK, 217-222.
- Baecker R. M. and A. Marcus (1990) *Human Factors and Typography for More Readable Programs*. Reading, Masschusets: Addison-Wesley.

- Baecker R., DiGiano C., and Marcus A. (1997). Software Visualization for Debugging, *Communications of the ACM*, 40(4), 44-54.
- Baniassad E. L. A., Murphy G. C., Schwanniger C., and Kircher M. (2002). Managing Crosscutting Concerns During Software Evolution Tasks: An Inquisitive Study, *Aspect-Oriented Software Development*, Enscheda, The Netherlands, 120-126.
- Baowen X., Ju Q., Xiaofang Z., Zhongqiang W., and Lin C. (2005). A Brief Survey of Program Slicing, *SIGSOFT Software Engineering Notes*, 30(2), 1-36.
- Begel A. and Graham S. L. (2005). Spoken Programs, *IEEE International Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, 99-106.
- Berlage, T. (1994). A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects, *ACM Transactions on Computer Human Interaction*, 1(3), 269-294.
- Berlin L. M. (1993). Beyond Program Understanding: A Look at Programming Expertise in Industry, *Empirical Studies of Programmers, 5th Workshop*, Palo Alto, CA, 6-25.
- Biehl J.T., Czerwinski M., Smith G., Robertson G.G., Bailey B. (2007). FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. *ACM Conference on Human Factors in Computing*, San Jose, California, 1313 - 1322.
- Blackwell A. (2002a). First Steps in Programming: A Rationale for Attention Investment Models, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, VA, Sept. 3-6, 2-10.
- Blackwell, A. and Burnett, M. (2002b). Applying Attention Investment to End-User Programming, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, VA, 28-30.
- [Blackwell 2003] Blackwell A. and Green T.R.G. (2003). Notational Systems—The Cognitive Dimensions of Notations Framework, in *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, J. M. Carroll, Ed. San Francisco, CA: Morgan Kaufmann.

- Boehm B. W. (1976). Software Engineering, *IEEE Transactions on Computers*, 12(25), 1226-1242.
- Boehm-Davis D. A., Fox J. E., and Philips B. H. (1996). Techniques for Exploring Program Comprehension, *Empirical Studies of Programmers, Sixth Workshop*, Washington D.C., 3-37.
- Boothe B. (2000). Efficient Algorithms for Bidirectional Debugging. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 299-310.
- Boren M. T. and Ramey J. (2000). Thinking Aloud: Reconciling Theory and Practice, *IEEE Transactions on Professional Communication*, 43(3), 261-278.
- Bothell, D. (2004) ACT-R Environment Manual, Version 5.0, April 22, <http://act-r.psy.cmu.edu/software/EnvironmentManual.pdf>
- Briggs, J.S., Jamieson S.D., Randall G.W. and Wand I.C. (1996). Task Time Lines as a Debugging Tool. *ACM SIGAda Ada Letters*, XVI(2), 50-69.
- Brooks R. (1972, published 1999), Towards a Theory of the Cognitive Processes in Computer Programming, *International Journal of Human-Computer Studies*, 51(2), 197-211.
- Brooks, F.P. Jr. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, Reading, MA.
- Brun Y., Ernst M.D. (2004). Finding Latent Code Errors via Machine Learning Over Program Executions, *International Conference on Software Engineering*, 480-490.
- Bureau of Labor Statistics, U.S. Department of Labor (2004), "Occupational Outlook Handbook," Dept. of Labor, <http://stats.bls.gov/oco>.
- Bush W. R., Pincus J. D., and Sielaff D. J. (2000). A Static Analyzer for Finding Dynamic Programming Errors, *Software Practice and Experience*, 30(7), 775-802.
- Calder, P.R. and Linton, M.A. (1990). Glyphs: Flyweight Objects for User Interfaces, *ACM Symposium on User Interface Software and Technology*, Snowbird, Utah, 92-101.

- Cataldo, M., Wagstrom P., Herbsleb J.D., Carley K. (2006). Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Computer Supported Cooperative Work*, Banff, Alberta, 353-362.
- Chalupsky, H. and Russ, T. A. (2002). WhyNot: Debugging Failed Queries in Large Knowledge Bases. *National Conference on Artificial intelligence (AAAI)*, Edmonton, Alberta, Canada, 870-877.
- Chong, J., Siino R. (2006). Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. *Computer Supported Cooperative Work*, Banff, Alberta. 28-39.
- Chu-Carroll M., Wright J., and Shields D. (2002). Supporting Aggregation in Fine Grained Software Configuration Management, *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 99-108.
- Clark, P., Chaw, S.Y., Barker, K., Chaudhri, V., Harrison, P., Fan, J., John, B., Porter, B., Spaulding, A., Thompson, J., Yeh, P.Z. (2007). Capturing and Answering Questions Posed to a Knowledge-Based System. *International Conference on Knowledge Capture (K-CAP)*, Whistler, British Columbia, Canada, 63-70.
- Clause J. and Orso A. (2007). A Technique for Enabling and Supporting Debugging of Field Failures, *International Conference on Software Engineering*, Minneapolis, MN, 261-270.
- Cleve H. and A. Zeller (2005). Locating Causes of Program Failures, *International Conference on Software Engineering*, St. Louis, MI, 342-351.
- Cole B., Hakim D., Hovenmeyer D., Lazarus R., Pugh W. and Stephens K. (2006). Improving Your Software Using Static Analysis to Find Bugs. *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, Oregon, USA, 673-674.
- Coblenz, M. J., Ko, A. J., and Myers. B. A. (2005). Using Objects of Measurement to Detect Spreadsheet Errors. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September 23-26, 314-316.

- Cook, C., Burnett, M., and Boom, D. (1997). A Bug's Eye View of Immediate Visual Feedback in Direct-Manipulation Programming Systems, *Empirical Studies of Programmers, 7th Workshop*, Alexandria, VA, 20-41.
- Cooper K.D., Harvey T.J. & Kennedy K. (2001). A Simple, Fast Dominance Algorithm. <http://www.hipersoft.rice.edu/grads/publications/dom14.pdf>.
- Cooper S., Dann W., and Pausch R. (2003). Teaching Objects-first in Introductory Computer Science. *SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, 191-195.
- Corritore C. L. and S. Wiedenbeck (2001), An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers, *International Journal of Human-Computer Studies*, 54, 1-23.
- Corritore C. L. and S. Wiedenbeck (1999). Mental Representations of Expert Procedural and Object-Oriented Programmers in a Software Maintenance Task, *International Journal of Human-Computer Studies*, 50(1), 61-83.
- Cubranic D. and G. Murphy (2000). Hipikat: Recommending Pertinent Software Development Artifacts, *International Conference on Software Engineering*, Portland, Oregon, 408-418.
- Curtis B. (1981). Substantiating Programmer Variability, *Proceedings of the IEEE*, 69(7), 846.
- Dann W., Cooper S., and Pausch R. (2003), *Learning to Program with Alice*: Prentice-Hall.
- Davies S. P. (1994). Knowledge Restructuring and the Acquisition of Programming Expertise, *International Journal of Human-Computer Studies*, 40(4), 703-726.
- Davies S. P. (1993). Models and Theories of Programming Strategy, *International Journal of Man-Machine Studies*, 39, 236-267.
- Davies, S.P. (1996). Display-based problem solving strategies in computer programming, *Empirical Studies of Programmers, 6th Workshop*, Washington, D.C., 59-76.

- DeLine R., Czerwinski M., and Robertson G. (2005). Easing Program Comprehension by Sharing Navigation Data. *IEEE Symposium on Visual Languages & Human-Centered Computing* (VL/HCC), September, 241-248.
- de Souza, C.R.B., Redmiles D.F., Mark G., Penix J., and Sierhuis M. (2003). Management of Interdependencies in Collaborative Software Development: A Field Study. *IEEE International Symposium on Empirical Software Engineering*, Rome, Italy, 294–303.
- Detlefs D. L., K. Rustan, M. Leino, G. Nelson, and J. B. Saxe (1998). Extended Static Checking, *Compaq Systems Research Center SRC Research Report 159*, December 18.
- Douce C. (2001). Long Term Comprehension of Software Systems: A Methodology for Study, *Psychology of Programming Interest Group*, Bournemouth, UK.
- Dourish, P. (1995). Accounting for System Behaviour: Representation, Reflection and Resourceful Action, *Third Decennial Conference on Computers in Context*. Aarhus, Denmark:
- Dworman, G. and Rosenbaum, S. (2004). Helping Users to Use Help: Improving Interaction with Help Systems. *ACM Conference on Human Factors in Computing Systems, Extended abstracts*, Vienna, Austria, 1717-1718.
- Eick S. G., Graves T.L., Karr A.F., Marron J.S., and Mockus A. (2001). Does Code Decay? Assessing the Evidence from Change Management Data, *IEEE Transactions on Software Engineering*, 27(1), 1-12.
- Eisenberg M. and H. A. Peelle (1983). APL Learning Bugs, *APL Conference*, Washington, D. C., 11-16.
- Eisenberg A. & De Volder K. (2005). Dynamic Feature Traces: Finding Features in Unfamiliar Code. *International Conference on Software Maintenance*, Budapest, Hungary, 337-346.
- Eisenstadt, M. (1997). My Hairiest Bug War Stories, *Communications of the ACM*, 40(4), 30-37.

- Engebretson A. and Wiedenbeck S. (2002). Novice Comprehension of Programs Using Task-Specific and Non-Task-Specific Constructs. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, VA, Sept 3-6, 11- 18.
- Ericsson K. A. and Simon H.A. (1984). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: MIT Press.
- Ernst M.D., Czeisler A., Griswold W.G., and Notkin D. (2000). Quickly Detecting Relevant Program Invariants. *International Conference on Software Engineering*, Limerick, Ireland, June 7-9, 449-458.
- Fischer M., Pinzger M., and Gall H. (2003). Analyzing and Relating Bug Report Data for Feature Tracking, *Working Conference on Reverse Engineering*, Victoria, British Columbia, Canada, 90-99.
- Fogarty J., Ko A.J., Aung H.H., Golden E., Tang K.P., and Hudson S.E. (2005). Examining Task Engagement in Sensor-Based Statistical Models of Human Interruptibility, *ACM Conference on Human Factors in Computing Systems*, Portland, Oregon, USA, 331-340.
- Fowler, M.; Highsmith, J., (2001) The Agile Manifesto. *Software Development*, 2001. August.
- Francel M. A. and S. Rugaber (2001). The Value of Slicing While Debugging, *Science of Computer Programming*, 40(2-3), 151-169.
- Fritzson P., Shahmehri N., and Karkar M. (1992). Generalized Algorithmic Debugging and Testing, *ACM Letters on Programming Languages and Systems*, 1(4), 303-322.
- Furnas G.W., Landauer T.K., Gomez L. M., and Dumais S. T. (1987). The Vocabulary Problem in Human-System Communication, *Communications of the ACM*, 30, 964-971.
- Gall H., Jazayeri M., and Krajewski J. (2003). CVS Release History Data for Detecting Logical Couplings, *International Workshop on Principles of Software Evolution*, Helsinki, Finaland, 13.
- German, D. M. (2006). An Empirical Study of Fine-Grained Software Modifications. *Empirical Software Engineering*, 11(3), 369-393.

- Gestwicki, P. and Jayaraman, B. (2002). Interactive Visualization of Java Programs, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, VA, 226-235.
- Gilmore, D. J. (1992). Models of Debugging, *Acta Psychologica*, 78, 151-173.
- Greenhouse A., Halloran T. J., and Scherlis W. L. (2005). Observations on the Assured Evolution of Concurrent Java Programs, *Science of Computer Programming*, 58(3), 384-411.
- Gonzalez V. M. and G. Mark (2004). "Constant, Constant, Multi-Tasking Craziness": Managing Multiple Working Spheres, *ACM Conference on Human Factors in Computing*, Vienna, Austria, 113-120.
- Gonzalez, V., G. Mark., Harris J. (2005). No Task Left Behind? Examining the Nature of Fragmented Work. *ACM Conference on Human Factors in Computing*, Portland, OR, 321-330.
- Green T.R.G. (1989). Cognitive Dimensions of Notations, in *People and Computers*, A. Sutcliffe and L. Macaulay (eds.), Cambridge, UK, Cambridge University Press, 443-460.
- Green T.R.G. and Petre M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework, *Journal of Visual Languages and Computing*, 7, 131-174.
- Groce A., Visser W. (2003). What Went Wrong: Explaining Counter Examples, *10th International SPIN Workshop on Model Checking of Software*, 121-135.
- Grove, D. and Chambers, C. (2001). A Framework for Call Graph Construction Algorithms. *ACM Transactions on Programming Languages and Systems* 23(6), November, 685-746.
- Gugerty, L. and Olson, G.M. (1986). Comprehension Differences in Debugging by Skilled and Novice Programmers, *Empirical Studies of Programmers, 1st Workshop*, Washington, DC, 13-27.

- Gupta N., He H., Zhang X. and Gupta R. (2005). Locating Faulty Code Using Failure-Inducing Chops. *IEEE/ACM International Conference on Automated Software Engineering*, Long Beach, CA, USA, 263-272.
- Gutwin, C., R. Penner, K. Schneider, K. (2004). Group Awareness in Distributed Software Development. *Computer Supported Collaborative Work*, Chicago, IL, 72-81.
- Halsted, K.L. and Roberts, J.H. (2002). Eclipse Help System: An Open Source User Assistance Offering. *SIGDOC: International Conference on Computer Documentation*, Toronto, Ontario, Canada, 49-59.
- Hertzum, M. (2002). The Importance of Trust in Software Engineers' Assessment of Choice of Information Sources. *Information and Organization*, 12(1), 1-18.
- Holtzblatt K. and Beyer, H (1998). *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann, San Francisco, CA.
- Jackson D. (2001). Visual Debugging of Multithreaded Java Programs, *IEEE Symposia on Human-Centric Computing Languages and Environments*, Stresa, Italy, 340-341.
- Kehoe C., Stasko J., and Taylor A. (2001) Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study, *International Journal of Human-Computer Studies*, 54(2), 265-284.
- Kelleher, C. and Pausch, R. (205). Stencils-Based Tutorials: Design and Evaluation, *ACM Conference on Human Factors in Computing*, Portland, Oregon, USA, 541-550.
- Kessler P. B. (1990). Fast Breakpoints: Design and Implementation, *Programming Language Design and Implementation*, 78-84.
- Kiczales, G. (1992). Towards a New Model of Abstraction in Software Engineering, *IMSA Workshop on Reflection and Meta-level Architectures*.
- Kranzlmuller, D., Grabner, S., and Vokert, J. (1996). Event Graph Visualization for Debugging Large Applications, *SIGMETRICS Symposium on Parallel and Distributed Tools*, Philadelphia, Pennsylvania, USA, 108-117.

- Knuth D. (1989). The Errors of TeX, *Software: Practice and Experience*, 19(7), 607-685.
- Ko A. J. (2003a). A Contextual Inquiry of Expert Programmers in an Event-Based Programming Environment. *ACM Conference on Human Factors in Computing*, Fort Lauderdale, FL, April 8-10, 1036-1037.
- Ko, A. J. and Myers, B. A. (2003b). Development and Evaluation of a Model of Programming Errors. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, October 28th-31st, 7-14.
- Ko A. J. and B. A. Myers (2004a). Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior, *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, 151-158.
- Ko, A.J., B.A. Myers, H.H. Aung (2004b). Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, 199–206.
- Ko A. J., H. Aung, and B. A. Myers (2005a). Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing, *ACM Conference on Human Factors in Computing Systems*, Portland, Oregon, USA, 1557-1560.
- Ko A. J., Aung H.H., and Myers B. A. (2005b). Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *International Conference on Software Engineering*, St. Louis, Missouri, 126-135.
- Ko A. J. and Myers B. A. (2005c). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *Journal of Visual Languages and Computing*, 16(1-2), 41-84.
- Ko, A.J. and Myers, B.A. (2005d) Citrus: A Toolkit for Simplifying the Creation of Structured Editors for Code and Data, *ACM Symposium on User Interface Software and Technology*, Seattle, Washington, USA, 3-12.

- Ko. A. J., Myers, B.A., Chau, D. H. (2006a) A Linguistic Analysis of How People Describe Software Problems. *Visual Languages and Human-Centric Computing*, Brighton, United Kingdom, September 4-8, 127-134.
- Ko. A. J., Myers, B.A., Coblenz, M. and Aung, H. H. (2006b). 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.
- Ko, A. J., Myers, B. A. (2006c) Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views for Code Editors. *ACM Conference on Human Factors in Computing Systems*, Montreal, Canada, April 24-27, 387-396.
- Ko, A.J. DeLine, R., & Venolia, G. (2007). Information Needs in Collocated Software Development Teams. *International Conference on Software Engineering*, Minneapolis, MN, 344-353.
- Ko, A.J. and Myers, B.A. (2008a). Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *International Conference on Software Engineering* (ICSE), Leipzig, Germany, May 10-18. To appear.
- Ko, A.J. and Myers, B.A. (2008b). Source-Level Debugging with the Whyline. *Cooperative and Human Aspects of Software Engineering*, co-located with the *International Conference on Software Engineering*, Leipzig, Germany, May 10-18. To appear.
- Koch S. and Schneider G. (2000). Results from Software Engineering Research into Open Source Development Projects Using Public Data, *Wirtschaftsuniversität*, Vienna, Austria 22.
- Koenemann J. and Robertson S.P. (1991). Expert Problem Solving Strategies for Program Comprehension, *ACM Conference on Human Factors and Computing Systems*, New Orleans, Louisiana, 125-130.
- Laird J., Bates C., Congdon, Altmann E., Swedlow K. (1990). Soar User's Manual: Version 5.2, *Carnegie Mellon Technical Report CMU-CS-90-179 and University of Michigan Technical Report CSE-TR-72-90*.

- LaToza T., G. Venolia, and R. DeLine (2006). Maintaining Mental Models: A Study of Developer Work Habits, *International Conference on Software Engineering*, Shanghai, China, 492-501.
- LaToza, T.D., Garlan, D., Herblseb, J.D., and Myers, B.A. (2007). Program Comprehension as Fact Finding. *Foundations of Software Engineering*, Cavtat near Dubrovnik, Croatia, September 3-7, 361-370.
- Lehman M. M. and Belady L. (1985). *Software Evolution – Processes of Software Change*. London: Academic Press.
- Lencevicius R., Holzle U., and Singh A. K. (2003). Dynamic Query-Based Debugging of Object-Oriented Programs, *Journal of Automated Software Engineering*, 10(1), 367-370.
- Lewis B. (2003). Debugging Backwards in Time, *International Workshop on Automated Debugging*, 225-235.
- Liblit B., Naik M., Zheng A., Aiken A. & Jordan M. (2005). Scalable statistical bug isolation. *Programming Design and Implementation*, Chicago, IL, USA, 15-26.
- Liblit B., Aiken A., Zheng A. X. , and Jordan M. I. (2003). Bug Isolation via Remote Program Sampling, *Programming Language Design and Implementation*, 141-154.
- Lieberman, H. and Fry, C. (1995). Bridging the Gulf between Code and Behavior in Programming, *ACM Conference on Human Factors in Computing Systems*, Denver, CO, 480-494.
- Lieberman, H. (1997). The Debugging Scandal and What to Do About It, *Communications of the ACM*, 40(4), 26-78.
- Lin, J., Quan, D., Sinha, V., Bakshi, K., Huynh, D., Katz, B., and Karger, D. R. (2003). The Role of Context in Question Answering Systems. *ACM Conference on Human Factors in Computing Systems, Extended Abstracts*, Ft. Lauderdale, Florida, USA, 1006-1007.

- Littman D. C., J. Pinto, S. Letovsky, and E. Soloway (1986). Mental Models and Software Maintenance, *Empirical Studies of Programmers, 1st Workshop*, Washington, DC, 80-98.
- Mayrhofer, A.V. and Vans, A.M. (1997). Program Understanding Behavior during Debugging of Large Scale Software, *Empirical Studies of Programmers, 7th Workshop*, Alexandria, Virginia, 157-179.
- McCarthy, J. (1981). History of LISP. In History of Programming Languages I, R. L. Wexelblat, Ed. *History of Programming Languages*. ACM, New York, NY, 173-185.
- McDonald, D.W., Ackerman M.S. (1998). Just Talk to Me: A Field Study of Expertise Location. ACM Conference on *Computer Supported Cooperative Work*, Seattle, WA, 315-324.
- Miara J. R., Musselman J. A., Navarro J. A., and Schneiderman B. (1983). Program Indentation and Comprehensibility, *Communications of the ACM*, 26(11), 861-867.
- Milanova A., Rountev A., and Ryder B. G. (2002). Precise Call Graph Construction in the Presence of Function Pointers. *IEEE International Workshop on Source Code Analysis and Manipulation*, October 1, Bratislava, Slovak Republic, 155.
- Miller R.C. and Myers B.A. Outlier Finding: Focusing User Attention on Possible Errors. *ACM Symposium on User Interface Software and Technology*, Orlando, FL, November, 81-90.
- Moskal, B., Lurie, D. Cooper, S. (2004). Evaluating the Effectiveness of a New Instructional Approach. *SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, 75-79.
- Mukherjea S. and Stasko J. (1994). Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger, *ACM Transactions on Computer-Human Interaction*, 1(3), 215-244.
- Murphy G. C., Kersten M., Robillard M. P., and Cubranic D. (2005). The Emergent Structure of Development Tasks, *European Conference on Object-Oriented Programming*, Glasgow, United Kingdom, 34-48.

- Myers, B.A. (1980). Displaying Data Structures for Interactive Debugging. *XEROX Palo Alto Research Center*, June.
- Myers B. A. (1983). Incense: A System for Displaying Data Structures, *Computer Graphics* Detroit, Michigan, USA, 115-125.
- Myers B.A., Giuse D., Dannenberg R.B., Vander Zanden B., Kosbie D., Pervin E., Mickish A., and Marchal P. (1990). *IEEE Computer*, 23(11), November.
- Myers B. A., McDaniel R. G., and Kosbie D. S. (1993). Marquise: Creating Complete User Interfaces by Demonstration, *ACM Conference on Human Factors in Computing Systems*, Amsterdam, The Netherlands, 293-300.
- Myers, B.A. and Kosbie, D. (1996). Reusable Hierarchical Command Objects, *ACM Conference on Human Factors in Computing (CHI)*, Vancouver, BC, Canada, 260-267.
- Myers B.A., McDaniel R.G., Miller R.C., Ferrency A.S., Faulring A., Kyle B.D., Mickish A., Klimovitski A. and Doane P. (1997). The Amulet Environment: New Models for Effective User Interface Software Development, *IEEE Transactions on Software Engineering*, 23(6), June, 347-365.
- Myers, B.A. (1998). Scripting Graphical Applications by Demonstration. *ACM Conference on Human Factors in Computing (CHI)*, Los Angeles, California, USA, 534-541.
- Myers, B. A., Weitzman, D., Ko, A. J., Chau, D. H. (2006) Answering Why and Why Not Questions in User Interfaces. *ACM Conference on Human Factors in Computing Systems*, Montreal, Canada, April 24-27, 397-406.
- Nanja, M. and Cook, C. R. (1987). An Analysis of the On-Line Debugging Process, in G. M. Olson, S. Shepard, and E. Soloway, (eds.) *Empirical Studies of Programmers: Second Workshop*, Norwood, NJ: Ablex, 172-184.
- Navarro-Prieto R. and Canas J.J. (2001). Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension, *International Journal of Human-Computer Studies*, 54, 799-829.

- Nichols D. M. and Twidale M. B. (2005). Usability Discussions in Open Source Development, *Hawaii International Conference on System Sciences*, Hawaii, USA, 198-207.
- Nielsen, J. and Molich, R. (1990). Heuristic Evaluation of User Interfaces. *ACM Conference on Human Factors in Computing Systems*, Seattle, Washington, USA, 249-256.
- Nistor E. C. and van Der Hoek A. (2006). Concern Highlight: A Tool for Concern Exploration and Visualization, *Workshop on Linking Aspect Technology and Evolution*, Bonn, Germany.
- Norman D. A. (1988). *The Design of Everyday Things*. New York, NY: Doubleday.
- Pirolli P. and Card S. K. (1999). Information Foraging, *Psychological Review*, 106(4), 643-675.
- Pane J. F. and Myers B. A. (1996). Usability Issues in the Design of Novice Programming Systems, *Carnegie Mellon University, Pittsburgh, PA CMU-CS-96-132*, August, <http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html>.
- Pane J. F., Ratanamahatana C. A., and Myers B. A. (2001). Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies*, 54(2), 237-264.
- Pennington N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs, *Cognitive Psychology*, 19, 295-341.
- Perlow L. (1999). The Time Famine: Toward a Sociology of Work Time, *Administrative Science Quarterly*, 44, 57-81.
- Perry, D.E., Staudenmayer N.A., Votta L.G. (1994). People, Organizations and Process Improvement. *IEEE Software*, July, 36-45.
- Petre M. and Blackwell A. F. (1997). A Glimpse of Expert Programmers' Mental Imagery, *Empirical Studies of Programmers, 7th Workshop*, Alexandria, Virginia, USA, 109-128.
- Phalgune A., Kissinger C., Burnett M., Cook C., Beckwith L., and Ruthruff J. R. (2005). Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User

- Programmers, *IEEE Symposium on Visual Languages and Human-Centric Computing* Dallas, Texas, 45-52.
- Podgurski A., Leon D., Francis P., Masri W., Minch M., Sun J., and Wang B. (2003). Automated Support for Classifying Software Failure Reports, *International Conference on Software Engineering*, Portland, Oregon, USA, 465-475.
- Potanin A., Noble J., & Biddle R. (2004). Snapshot Query-based Debugging. *Australian Software Engineering Conference*, 251.
- Pothier, G., Tanter, É., and Piquer, J. 2007. Scalable Omnipotent Debugging. *ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, Montreal, Quebec, Canada, 535-552.
- Quintana C., Krajcik J., and Soloway E. (2002). A Case Study to Distill Structural Scaffolding Guidelines for Scaffolded Software Environments. *ACM Conference on Human Factors in Computing Systems*, Minneapolis, Minnesota, USA, 81-88.
- Ramachandran, A. and Young, R.M. (2005). Providing Intelligent Help Across Applications in Dynamic User and Environment Contexts. *International Conference on Intelligent User Interfaces*, San Diego, California, USA, 269-271.
- Reichwein J., Rothermel G., and Burnett M. (2000). Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging. *Conference on Domain-specific Languages*, Austin, Texas, 25-38.
- Reiss S. P. (1996). The Design of the Desert Software Development Environment, *International Conference on Software Engineering*, Berlin, Germany, 398-407.
- Robertson, T. J., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J. R., Beckwith, L., and Phalgune, A. (2004). Impact of Interruption Style on End-User Debugging, *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, 287-293.
- Robillard M. P. and Murphy G. C. (2002). Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, *International Conference on Software Engineering*, 406-416.

- Robillard M. P. and Murphy G. C. (2003a). Automatically Inferring Concern Code from Program Investigation Activities, *International Conference on Automated Software Engineering*, 225-234.
- Robillard M. P. (2003b). Representing Concerns in Source Code, in *Department of Computer Science*. Vancouver, Canada: University of British Columbia, 2003.
- Robillard M. P., Coelho W., and Murphy G. C. (2004). How Effective Developers Investigate Source Code: An Exploratory Study, *IEEE Transactions on Software Engineering*, 30(12), 889-903.
- Robillard M. P. (2005). Automatic Generation of Suggestions for Program Investigation, *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 11-20.
- Romero P., Cox R., de Boulay B., and Lutz R. (2003). A Survey of External Representations Employed in Object-Oriented Programming Environments, *Journal of Visual Languages and Computing*, 14(5), 387-419.
- Rosenblum D. S. (1995). A Practical Approach to Programming With Assertions, *IEEE Transactions on Software Engineering*, 21(1), 19-31.
- Rothermel, G., Harrold, M.J., and Dedhia, J. (2000). Regression Test Selection for C++ Software. *Software Testing, Verification & Reliability*, 10(2), 77-109.
- Ruthruff, J.R., Prabhakararao S., Reichwein J., Cook C., Creswick E., Burnett M.M., Interactive, Visual Fault Localization Support for End-User Programmers. *Journal of Visual Languages and Computing*, 16(1-2), 3-40.
- Sandusky J., Gasser L., and Riponche G. (2004). Bug Report Networks: Varieties, Strategies, and Impacts in an OSS Development Community, *Workshop on Mining Software Repositories (MSR)*, Edinburgh, Scotland.
- Sandusky, R.J., Gasser L. (2005). Negotiation and Coordination of Information and Activity in Distributed Software Problem Management. *International ACM SIGGROUP Conference on Supporting Group Work (GROUP)*, Sanibel Island, Florida, USA, 187-196.

- Sarma, A., Z. Noroozi, A. van der Hoek (2003). Palantír: Raising Awareness among Configuration Management Workspaces. *International Conference on Software Engineering*, Portland, Oregon, USA, 444–454.
- Satterthwaite E. H. (1975). Source Language Debugging Tools. *Stanford University Computer Science Department*.
- Schuppan V., M. Baur, and A. Biere (2004). JVM Independent Replay in Java, *Runtime Verification*, Barcelona, Spain, 76-94.
- Seaman, C.B., Basili V.R. (1998). Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering*, 24(6), July.
- Sillito J., Murphy G.C., and De Volder K. (2006). Questions Programmers Ask During Software Evolution Tasks. *SIGSOFT Foundations on Software Engineering*, November 5-11, Portland, Oregon, 23-34.
- Singer J., Lethbridge T., Vinson N., and Anquetil N. (1997). An Examination of Software Engineering Work Practices, *Conference of the Centre for Advanced Studies in Collaborative Research*, 209-223.
- Sosic R. and Abramson D. A. (1997). Guard: A Relative Debugger, *Software Practice and Experience*, 27(2), 185-206.
- Spohrer, J. G. and Soloway, E. (1986). Analyzing the High Frequency Bugs in Novice Programs, *Empirical Studies of Programmers, 1st Workshop*, Washington, DC, 230-251.
- Sridharan M., Fink S.J., & Bodik R. (2007). Thin Slicing. *Programming Language Design and Implementation*, San Diego, California, USA, 112-122.
- Stockham T. G. and Dennis J. B. (1960). FLIT - Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0, *MIT, Cambridge, MA Memo 5001-23*, July.
- Stylos J. (2005). Designing a Programming Terminology Aid. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, 347-348.

- Sukaviriya, P. and Foley, J.D. (1990). Coupling A UI Framework with Automatic Generation of Context-Sensitive Animated Help. *ACM Symposium on User Interface Software Technology*, Snowbird, Utah, 152-166.
- Tan D.S., Gergle D., Scupelli P., Pausch R. (2006). Physically Large Displays Improve Performance on Spatial Tasks. *ACM Transactions on Computer-Human Interaction*, 13(1), 71-99.
- Tassey, G. (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011.
- Teasley B.E. (1994). The Effects of Naming Style and Expertise on Program Comprehension, *International Journal of Human-Computer Studies*, 40, 757-770.
- Tip, F. (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3, 121-189.
- Toutanova K., Klein D., Manning C., and Singer Y. (2003). Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network, *HLT-NAACL*, 252-259.
- Ungar D., Lieberman H. and Fry C. (1997). Debugging and the Experience of Immediacy, *Communications of the ACM*, 40(4), 39-43.
- Vander Zanden B.T., Baker D. and Jin J. (2004). An Explanation-Based, Visual Debugger for One-Way Constraints. *ACM Symposium on User Interface Software and Technology*, Santa Fe, NM, 207-216.
- Vans A. and von Mayrhofer A. (1999). Program Understanding Behavior During Corrective Maintenance of Large-scale Software, *International Journal of Human-Computer Studies*, 51(1), 31-70.
- Vessey, I. (1985). Expertise in Debugging Computer Programs: A Process Analysis, *International Journal of Man-Machine Studies*, 23, 459-494.
- Wagner E. and Lieberman H. (2003). An End-User Tool for E-Commerce Debugging. *International Conference on Intelligent User Interfaces*, Miami, Florida, 331-331.

- Wang, T. and Roychoudhury, A. (2007). Hierarchical dynamic slicing. *International Symposium on Software Testing and Analysis*, London, United Kingdom, July 09 - 12, 228-238
- Wallace C., Cook C., Summet J., and Burnett M. (2002). Assertions in End-User Software Engineering: A Think-Aloud Study, *IEEE Symposia on Human-Centric Computing*, Arlington, VA, 63- 65.
- Wang T. and A. Roychoudhury (2004). Using Compressed Bytecode Traces for Slicing Java Programs. *International Conference on Software Engineering* Scotland, UK, 512-521.
- Weiser M. (1982). Programmers Use Slices When Debugging, *Communications of the ACM*, 25(7), 446-452.
- Weiser, M. and Lyle, J. (1986). Experiments on Slicing-Based Debugging Aids, *Empirical Studies of Programmers, 1st Workshop*, Washington, DC, 187-197.
- White R.W., Ruthven I., and Jose J.M. (2002). Finding Relevant Documents using Top Ranking Sentences: An Evaluation of Two Alternative Schemes. *ACM SIGIR Conference on Research and Development in Information Retrieval*, Tampere, Finland, 57-64.
- Wiedenbeck S., Fix V., and Scholtz J. (1993). Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study, *International Journal of Man-Machine Studies*, 39(5), 793-812.
- Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., and Cook, C. (1997). Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? *ACM Conference on Human Factors in Computing*, Los Angeles, California, USA, 22-27.
- Xie T. and Notkin D. (2004). Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences, *International Conference on Software Maintenance*, 28-37.
- Xie T., Taneja K., Kale S., and Marinov D. (2007). Towards a Framework for Differential Unit Testing of Object-Oriented Programs. *International Workshop on Automation of Software Test*, Minneapolis, May 20-26, 202.

- Zeller A. and Hildebrandt R. (2002a). Simplifying and Isolating Failure-Inducing Input, *IEEE Transactions on Software Engineering*, 28(2), 183-200.
- Zeller, A. (2002b). Isolating Cause-Effect Chains from Computer Programs, *International Symposium on the Foundations of Software Engineering*, Charleston, South Carolina, USA, 1-10.
- Zhang, X. and Zhang, Y. (2003). Precise Dynamic Slicing Algorithms. *International Conference on Software Engineering*, Portland, Oregon, USA, 319-329.
- Zhang X. and Gupta R. (2004). Cost Effective Dynamic Program Slicing, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Washington, D.C., 94-106.
- Zhang X. & Gupta R. (2005). Whole Execution Traces and their Applications. *ACM Transactions on Architecture and Code Optimization*, 2(3), 301-334.