ASKING AND ANSWERING QUESTIONS ABOUT THE CAUSES OF SOFTWARE BEHAVIORS

THESIS PROPOSAL

Amy J. Ko

May 5th, 2006

Human Computer Interaction Institute
School of Computer Science
Pittsburgh, PA 15213
ajko@cs.cmu.edu

COMMITTEE

Brad A. Myers, Carnegie Mellon (Chair)
Bonnie John, Carnegie Mellon
Jonathan Aldrich, Carnegie Mellon
Gail Murphy, University of British Columbia

ABSTRACT

Most useful software undergoes a brief period of rapid development, followed by a much longer period of maintenance and adaptation. As a result, software developers spend most of their time exploring and analyzing a system's underlying source code in order to determine the parts of the system that are relevant to their tasks. Because these parts are often distributed throughout a system's modules, and because they can interact in complex and unpredictable ways as a system executes, this process of understanding a program and its execution can be extremely difficult.

The primary cause of this difficulty is that developers must answer their questions about a system's behavior by essentially guessing. For example, a developer wondering, "Why didn't this button do anything after I pressed it?" must form an answer 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 the explanation. Not only is this testing poorly supported by current tools, but worse yet, there are a vast number of potential explanations for a system's behavior, and so developers rarely formulate a valid explanation on the first attempt.

To address this problem, I propose to design and implement a new kind of program understanding tool called a Whyline, which allows a developer to ask questions about a system's behavior using direct manipulation. In response, the tool will accurately determine which parts of the system and its execution are related to the behavior in question, while also identifying any false assumptions the developer might have about what occurred during the execution of the program. Such a tool can be used for any activity that requires a developer to form a precise understanding of a program's execution, including debugging and reverse engineering.

A Whyline prototype has been shown to reduce debugging time by a factor of 8 when used to write interactive simulations in the Alice development environment. Given these encouraging results, I propose to scale the Whyline to support a more general-purpose language, larger programs, more specific questions, and more concise and helpful answers. The design of these improvements will be guided by a series of exploratory studies of software developers, which have already contributed several findings to theories of program understanding and debugging. To evaluate the effectiveness of the final prototype, I will compare it experimentally to conventional debuggers and identify features that are central to its effectiveness.

1. INTRODUCTION

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 [6, 32]. For example, a 2002 study by the National Institute for Standards and Technology found that software engineers in the U.S. spend 70-80% of their time testing and debugging, with the average error taking 17.4 hours to find and fix [19]. One reason that these debugging and maintenance activities take so long 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 [13, 31].

In order to understand these complexities, developers must map their questions about a system's *behavior* onto their tools' limited support for analyzing *code*. Currently, developers perform this mapping by essentially *guessing* the cause of a system's behavior. For example, suppose a developer is testing the user interface for a text editing application, and clicks on a button to print a document, but nothing seems to happen. The developer may wonder, "Why didn't the print button do anything after I pressed it?" To answer this question, he has to form a hypothetical answer to this question, such as, "Maybe the button's event handler isn't executing properly." To test this hypothesis, he might try to find the event handler in the source code, perhaps by searching for the text "print" in the source code. Assuming that such an event handler even exists and that it has the work "print" in its name, he could then set a breakpoint on the handler and run the application to see if it executes. If this is not the problem, he would then have to form a new hypothesis, repeating this process until an explanation for the system's behavior is confirmed. The developer can then use this newly formed understanding of the program's execution to design an appropriate modification to the system's source code.

There are two fundamental bottlenecks in this process of hypothesis formation and testing. First, hypothesis testing is poorly supported by current tools. For example, to determine if an event handler is executing, the developer must first find the handler, and then set a breakpoint, or add a print statement, and then re-execute the application. The second bottleneck is that even in simple systems, there are enough ways in which a system may function or fail that developers are unlikely to formulate a correct explanation of its behavior on the first attempt. The print button may not have had an effect because the handler was broken; it may not have a handler at all; perhaps the handler was not *attached* to the button; maybe it did print, but the printer was not properly configured; perhaps the printer *was* properly configured, but the developer was looking at the wrong printer. No matter how well tools help developers *test* their explanations, developers will always struggle to first *form* accurate explanations of software behavior.

In this proposal, I will discuss a new kind of program understanding tool called the *Whyline*, which addresses both of these bottlenecks by enabling developers to ask *why did* and *why didn't* questions about a system's behavior using direct manipulation. In response, the tool provides answers in terms of the parts of the system and its execution that were responsible for causing or preventing the behavior in question. The central thesis of this approach is:

> *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.*

In addition to helping developers more quickly form a correct understanding of a system's execution, the approach of the Whyline also allows the tool to inspect questions for discrepancies between what the developer believes a system has done at runtime, and what a system has *actually* done. For example, if a developer were to ask why some button had no effect on-screen, when in fact it did, but in some subtle or non-visible way, the tool can reveal this assumption, synchronizing the developer's understanding of a system's execution with its actual execution.

This work has several technical, theoretical and human-computer interaction contributions:

- Evidence that developers at all levels of expertise have difficulty forming accurate explanations of system behavior because they base their explanations on surface features of applications that do not correlate well with the actual causes of a program's behavior.
- Execution history data structures that facilitate the efficient creation of a question menu, and efficient analyses for answering questions.
- Automatic determination of the entities at a particular location on-screen that may have caused output or behavior that developers want to ask about.
- Analyses that determine relevant questions based on the spatial and temporal context of the developers' question.
- Interaction techniques for asking questions about software behavior by selecting objects and behavior in a reproduced history of a program's output. These techniques prevent developers from having to use unreliable natural language interfaces or from having to learn a custom query language.
- Interaction techniques that solicit developers' assumptions about a program's execution, enabling a tool to point out discrepancies between the developers' assumptions about a program's execution and the program's actual execution.
- Incremental algorithms for analyzing causality in a program's execution history, enabling the tool to provide immediate feedback in response to developers' questions.
- Analyses for answering "why didn't" questions about output statements that were not executed, output statements that were executed, but not with appropriate arguments, and even output statements that are missing from a program.
- Interactive visualizations of a programs' execution history, designed to facilitate developers' exploration and understanding of a program's execution and underlying causality, relative to the program's source code and corresponding output.
- User studies that demonstrate that a *Whyline* prototype increases developers' productivity and the quality of their solutions, relative to conventional program understanding tools.

In the next section, I review related work on behavioral research in program understanding, and then discuss the major types of tool support for this activity. In Section 3, I present a series of studies that I have conducted, which add several insights to theories of program understanding. In Section 4, I describe a Whyline prototyped for the Alice language and development environment, and the results of an experiment evaluating its effectiveness. I then propose several generalizations to the prototype in Section 5, which focus on supporting a more general purpose language, larger programs, more specific questions, and more concise answers. In the remaining sections, I present my evaluation plans, the contributions of my thesis, and my proposed schedule.

2. RELATED WORK

The related work falls into two categories: studies about human factors in program understanding, and tools to help with program understanding and debugging.

2.1 HUMAN FACTORS IN PROGRAM UNDERSTANDING

There is a long history of empirical research on debugging and program understanding, dating back to the 1950's. This body of work largely focuses 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 [14]. Subsequent studies at the time focused on novice mistakes, but this focus moved to experts as software became more ubiquitous. For example, Knuth recorded all of the debugging he performed in the development of TeX [23], revealing that the majority of his mistakes were due to oversights, which he labeled "surprise scenarios." Eisenstadt interviewed industry experts 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 [15].

Many researchers studied program understanding from a more 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 [8]: 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. [35] and Gugerty and Olson [19] found that expert 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 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 [51].

In addition to studying hypothesis *formation* in program understanding, a number of studies characterized developers' strategies for hypothesis *testing*. For example, Koenemann and Robertson [30] 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. [35], who argued that expert programmers followed a much more systematic strategy than novices, characterized by concrete plans and guided navigations of a program's dependencies. It has since been shown that both experts and novices use a combination [4, 43], but that systematic strategies are generally more productive than "as-needed" strategies [7, 39, 43]. Katz and Anderson identified other less common strategies for hypothesis testing, including hand-simulation of a program's execution and more rigorous causal reasoning [20].

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.

2.2 PROGRAM UNDERSTANDING TOOLS

Given the complexity of modern software, and the difficulty of understanding these complexities, researchers have developed a number of tools to support program understanding.

Some of the earliest forms of program understanding tools include the *dump* and the *trace*, both developed on the EDSAC in the 1940's [46]. 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 in the program's execution. Dumps also contain a lot of information, much of which is irrelevant to a problem.

In a *trace*, part of the machine state such as the line of code and the values of variables is printed to a display every time certain machine instructions are executed, such as the reading of a memory location or executing a line of code. The modern equivalents of traces are logging mechanisms, commonly known as "print statements" or "debug statements". These allow developers to instrument a program in order to print out information about a program's control and data flow during execution. One reason for these tools' popularity is that they allow developers to print exactly the information they want and nothing more. One tradeoff, of course, is that developers are often wrong in what they want when they are testing an inaccurate explanation of a system's behavior. Furthermore, because these tools require developers to heavily instrument a program, they can also incur "cleanup" costs when removed, and performance costs if not.

Breakpoint debuggers have been available since at least the 1960's [49], but work has continued to improve their utility [22]. 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 [34], while others have devised methods of undoing a program's execution [2, 50]. Despite these advances, breakpoint debuggers still they require developers to guess what code is causing a behavior in order to decide where to place a breakpoint, and then what information to inspect.

Algorithmic debugging [17] is technique in which the tool steps through the execution of some part of the program and asks the developer to verify the values of variables. The central limitation of this approach, in addition to the sheer number of user interactions required, is that developers are sometimes poor at knowing whether an intermediate value in a program is correct [40]. Furthermore, it 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.

Several researchers have created visualizations and animations of a program's data structures and execution to make it easier to reason about changes to data during a program's execution. For example, the Incense system [37] visualizes complex data structures and their relationships, to help developers detect problems in the data in memory. Mukherjea and Stasko [36] describe a variety of algorithm animations and animation authoring tools, which allow developers to see operations on data structures as a program executes. Other visualization tools abstract some of these details, providing a higher-level perspective of all of a program's data. For example, Baecker [3] describes 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 to learners, there is a general consensus that these visualizations are only helpful when directly associated with the source code corresponding to the animated behavior [21]. Furthermore, such animations must be hand-coded for each situation.

Another approach to supporting program understanding is to allow developers to write a set of conditions for a program's behavior and have a computer notify the developer when they are violated. Some of these take the form of *assertions*, which are inserted into the code, halting the program whenever the assertion is violated [45]. Others take the form of *queries* about the data structures and objects in memory [33], which notify the developer when the query results change during the program's execution. The primary issue with these approaches is that developers must translate their hypothesis about the cause of a program's behavior into code, which can be error prone and imprecise. 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.

Program slicing tools automatically determine which statements in a program could ("static slicing") or did ("dynamic slicing") affect the value of a variable in a program, and typically highlight the relevant statements in a developer's code editor [5]. 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 [52]. Unfortunately, even dynamic slicing, which was designed to produce a smaller, more specific subset of a program's statements for investigation, can select a up to a third of a program's statements for inspection, and cannot rank them in any particular order of relevance. Furthermore, slicing tools are only helpful if the developer is asking about a relevant variable in the program; 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 [16].

A number of tools specifically support developers' efforts to find and gather code that is relevant to some aspect of a software system. Robillard and Murphy describe a tool that helps developers navigate static dependencies in code, and combine them into a *concern graph* [41]. Robillard and Murphy also describe techniques for inferring potentially relevant code based on a developer's navigation through a program's source code [42], and Robillard describes a technique for inferring other relevant code based on a developer's current location in the source code [44]. While all of these techniques can be helpful when the developer is investigating relevant code, if the developer is investigating irrelevant code, such tools could potentially mislead developers in their efforts to find the relevant parts of a system.

Another approach proposed by Cleve and Zeller [10], called *delta debugging*, requires a developer to supply a program, two sets of input on which the program succeeds and fails, and a function that determines whether the program has succeeded. If possible, it returns a description

of the events that occurred in the failing execution that did *not* occur in the successful execution, by comparing the executions in a experimental manner. 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.

Relative debugging [48] is an approach similar to Cleve and Zeller's delta debugging, but instead of empirically testing the program and checking for failures, relative debugging determines the difference between two different versions of a program in order to help developers find problems as a program evolves. The central limitation of this approach is that the developer must specify the expected correspondences between their execution states, by deciding which data structures are important, and how they should be related. Not only does this require developers to write a program to test a program, but they must again guess what structures are relevant, and what relationships should be maintained. Such tools proceed with their analyses, even if developers' guesses are incorrect, possibly leading to overconfidence in the program's correctness.

Program analysis tools, such as ESC/Java [12], Fluid [18], and PREfix [9], while not directly applicable to debugging and program understanding, have the similar goal of helping developers identify errors in programs. The central difference between program analysis tools, and program understanding tools, is that program analysis tools aim to *detect* errors before they are found through testing, by verifying particular properties of programs that are indicative of errors. Program understanding tools, on the other hand, serve to facilitate a developer's understanding a *specific* program behavior. One advantage of program analyses 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 [12] 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.

Unlike the tools and analyses described here, my approach aims to prevent developers from guessing, by instead allowing them to analyze information that they can reason about accurately and objectively—namely, programs' observable output and behavior. Furthermore, my approach aims to elicit developers' incorrect assumptions, so that they may be detected and explained.

3. EXPLORATORY STUDIES OF HYPOTHESIS FORMATION AND TESTING

Although much of the research on program understanding describes it as a process of hypothesis formation and testing, there is still little knowledge of how people *form* their hypotheses, of what factors affect their formation, and to what degree the quality of developers' hypotheses affect productivity. Furthermore, all of the prior studies were performed with a number of artificial controls on developers' work, potentially limiting the generalizability of their findings. To address these limitations, I have started on a series of exploratory studies, in a variety of contexts and of several developer populations, in order to reproduce earlier findings, discover new trends, and elicit design requirements for better program understanding tools.

3.1 Developing a Pac Man Game in Alice

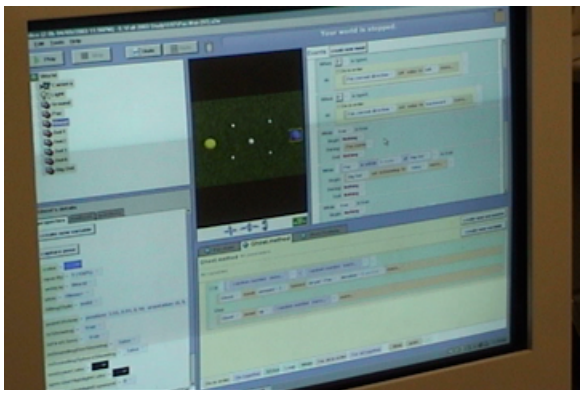

Figure 1. A user creating a Pac Man game in Alice 2.

In this study [27], six participants of varying expertise ranging from novice developer to industry expert were asked to use the Alice 2 programming environment (www.alice.org) to design and develop a 3D Pac Man game. Participants were videotaped and asked to think aloud while they worked, and were given two hours to complete the task (and more if they wished). The goal of the study was to study cycles of implementation, testing, and debugging, in order to understand the causes of errors and developers' strategies for finding them. There were a number of useful findings (see [27] for full details):

- An average of 46% of participants' time was spent debugging.
- 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).
- All of the time that each developer spent debugging was the result of an average of just 2 or 3 false hypotheses about the cause of the program's behavior.
- About half 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.
- About 85% of participants' questions involved a single object in the program's output.

3.2 Learning Visual Basic.NET over Several Weeks

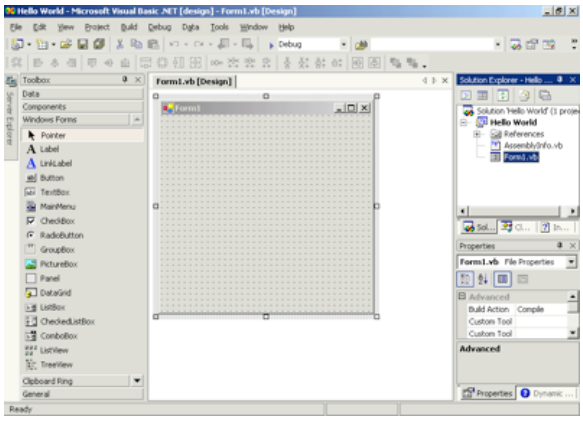

Figure 2. Visual Basic.NET, showing a new form behing created.

In the second study [25], the students of an Human-Computer Interaction class were learning to use Visual Basic.NET to prototype user interfaces. They were told that if they had questions about anything, they could ask the teaching assistants for aid. When they did, they were asked to describe what they were "stuck on," how they became "stuck," and what they had tried to become "unstuck." These descriptions were recorded by the teaching assistants on a form. There were a number of interesting trends in students' strategies and difficulties:

- 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.
- In most situations, 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...?"
- About 20% of the reported problems involved multiple objects not working together appropriately (for example, information from one window not being sent to another).
- In about 11% of the reported problems, 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. 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.
- Many students had spent considerable time investigating problems that did not exist, because they had misinterpreted or misperceived their program's output and feedback.

3.3 Repairing and Enhancing a Painting Application using Eclipse

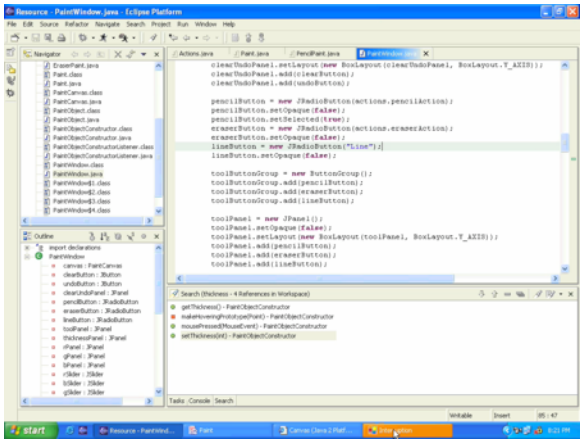

Figure 3. The Eclipse 2.0 development environment, showing the code editor and other views of a Java program.

In the third study [26], 31 expert Java developers were asked to work for 70 minutes on three debugging tasks and two feature enhancement tasks for a simple painting application written in Java. They were given the Eclipse development environment (www.eclipse.org), the Java API documentation, and access to the internet, and were paid $10 for each correctly completed task. Their work was screen-captured as a full-screen video, and then transcribed in terms of various developer actions. There were a number of interesting trends in developers' program understanding strategies:

- About 88% of developers' hypotheses about the causes of a program behavior of were false, causing them to spend an average of 49% of their time investigating irrelevant code.
- Developers used words that appeared in the running application to find relevant source code. For example, developers working on a problem with the button labeled "undo" typically searched for the word "undo" in the source code. Although such strategies sometimes led to relevant code, they often failed to result in anything useful.
- Developers' false hypotheses about the causes of a behavior often went untested, because their hypothesis was "correct enough" to solve their particular problem. This caused the developers to have an inaccurate understanding of the program's execution, which affected the correctness of their future hypotheses and implementation solutions.

3.4 LINGUISTIC TRENDS IN DESCRIPTIONS OF SOFTWARE BEHAVIOR

Figure 4. The Mozilla bug database.

The previous studies considered a small sample of questions about software behavior. In order to study a larger sample of issues with software behavior, I obtained about 180,000 bug reports on a web browser, a web server, a suite of office applications, a software development environment, and an operating system kernel. I performed linguistic analyses on the reports' *titles* (such as `crash if I try to clear cookies`), in order to assess how people describe software problems. There were several interesting trends [29]:

- There were three types of problems identified: unanticipated feedback, (such as error messages being displayed), wrong feedback (such as obtaining a result with an incorrect number), or the lack of feedback (such as nothing happening after pressing a button).
- Problems generally referred to visual feedback, but also included auditory feedback ("text edit beeps missing") and temporal feedback ("hangs" and "takes forever").
- About 95% of noun phrases in the report titles referred to visible entities, physical devices, or user actions, suggesting the feasibility of users selecting these entities in a tool.
- Temporal context was frequently specified using words such as *when*, *during*, and *after*, in order to indicate the situation in which a problem occurred. The context supplied was almost exclusively user input or program output.
- Many of the behaviors represented computations or system actions that executed over time, or repeatedly, but such behaviors were usually described by referring to a particular instance of the behavior (such as the most recent search results) rather than generally.

4. A WHYLINE FOR ALICE

The results of my studies and the prior work suggest that there are two major bottlenecks in understanding software behavior: (1) developers formulate several inaccurate explanations of a system's behavior before converging on a correct one; (2) current tools provide little support for testing these explanations efficiently. The type of tool that I propose, which I call a *Whyline*, addresses both of these problems by enabling developers to ask questions explicitly about software behavior, allowing the system to accurately determine the parts of a program and its execution history that caused or prevented the behavior in question (*Whyline* is an acronym for a ***w**orkspace that **h**elps **y**ou **l**ink **i**nstructions to **n**umbers and **e**vents*).

To test the feasibility of this approach, I created a Whyline [24] for Alice (www.alice.org). Alice is an object-oriented, multithreaded, imperative language and development environment for creating 3D simulations and games. Figure 5 illustrates how the Whyline for Alice works. On the right is a flowchart of the sequence of interactions between the developer and the tool, and on the left are two screenshots, showing a developer asking a question about a Pac Man game he is creating, and the resulting answer. In this example, `PacMan` (the yellow sphere) is supposed to resize when he touches the `Ghost` (the blue sphere), but does not.

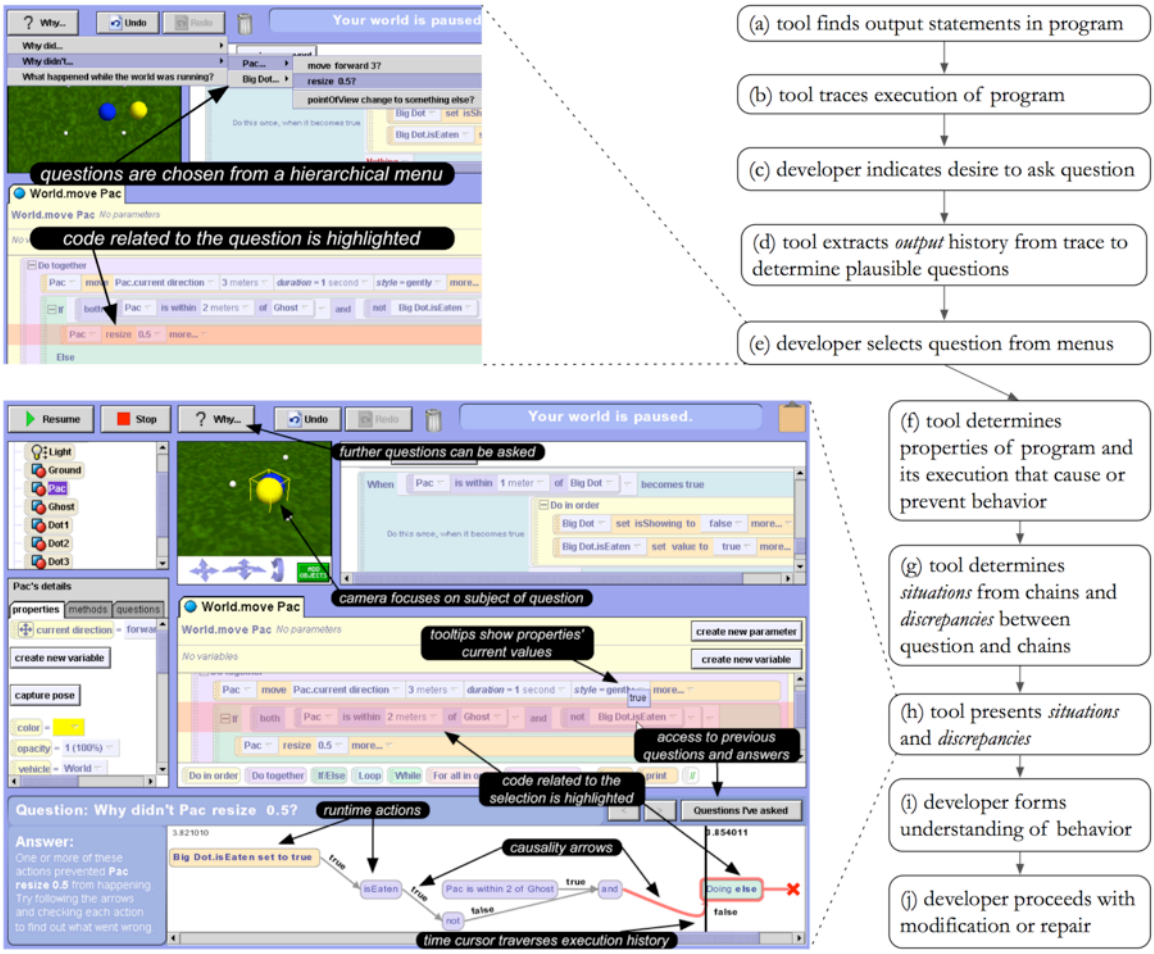

Figure 5. The interaction between the developer and the Alice Whyline. In this example, a developer working on a Pac Man game notices that Pac Man does not resize after touching the ghost, as expected. He asks, "*Why didn't Pac Man resize 0.5?*" and the Whyline explains that the *isEaten* flag prevented the resize.

The approach of the tool is as follows. The developer asks a question by selecting the behavior that did or did not occur from a menu that the Whyline constructs by analyzing the program and its execution. The Whyline then determines if there is a discrepancy between what the developer believed occurred and what actually occurred, and if so, reveals it. Otherwise, the system uses static and dynamic program analyses to determine what execution events caused or prevented the behavior in question, and then presents these events and their corresponding code.

4.1 Tracing Execution

To allow developers to ask questions about a program's output, the Alice Whyline determines all of the "output" statements in the program before it executes (Figure 5a). These include animations, such as *resize*, *move*, and *rotate*, and assignments to visible object properties such as *isShowing* and *color*, which have an effect on-screen when changed. During a program's execution, the Whyline records a history (Figure 5b), which includes the statements executed, the values computed, and what statements assigned or used these values. Although the history can be large, the tool cannot be sure beforehand what information the developer may ask about.

4.2 Extracting the Output History

While the program executes, the developer can ask a question by pressing the "Why" button, which pauses the program (Figure 5c). In order to allow developers to ask a question without having to use a natural language interface or a custom query language, the Whyline for Alice uses the history to construct two types of questions (Figure 5d). A *why didn't* question is created for each *output statement* in the program (independent of the execution history) and a *why did* question is created for each unique *execution* of these output statements (where unique is defined by the set of arguments used to execute the statement). An important aspect of the *why didn't* menu is that it also contains questions about behaviors that *did* occur. This allows the tool to elicit the developer's assumptions and misperceptions about the execution of the program, so that they may be corrected.

4.3 Choosing a Question

After the system determines the available questions, it displays the question menu (Figure 5e) so that the developer may choose a question. The questions are sorted by the objects that they refer to and the arguments they use. For example, if an Alice program were to have to resize statements, one with an argument of 0.5 and another with an argument of 0.25, there would be two separate questions for each. As the developer hovers over questions in the menu, the Whyline highlights the statement in the program that corresponds to the question, to indicate to the developer which statement is being questioned. In this example, the developer expected `PacMan` to resize after intersecting the `Ghost`, so he selects, "Why didn't `PacMan` resize 0.5?"

4.4 Analyzing Causality

Once the developer has selected a question, the goal is to produce as little information as possible for the developer to understand, while still providing a valid and helpful explanation for the system's behavior. To do this, the Whyline for Alice determines one or more *causal chains* of execution events that explain what caused or prevented the behavior in question (Figure 5f).

There are three possible answers in this prototype. The first is an *invariant* answer, which explains that the behavior in question either executes unconditionally (for example, the first line of a program will always execute), or is unreachable (for example, a procedure that is never called). This is determined statically by analyzing the call structure of the program. The second type of answer is a *assumption* answer, which occurs when there is a discrepancy between the developer's question and the execution history. For example, one potential answer to the question posed by the developer in Figure 5 is that Pac Man *did* resize, but the developer did not notice it, possibly because the camera was oriented far above the object, or the character did not resize by much. In this case, the Whyline determines the execution events that caused the resize to occur, using a standard dynamic slicing algorithm [53], and explains the discrepancy in a textual answer. The final type of answer is a *causal* answer, which determines what caused or prevented the behavior in question, also using dynamic slicing. This is the type of answer shown in the screenshot for Figure 5h. For *why didn't* questions, there are likely to be several conditions that may prevent a particular output statement from executing, and these conditions may refer to several variables; therefore, *why didn't* answers may involve several explanations for why a behavior did not occur. In order to reduce the amount of information for the developer to understand, these various situations are isolated and presented independently.

4.5 PRESENTING THE ANSWER

Once the Whyline has determined all of the relevant situations and discrepancies, its goal is to present the situations in a manner that helps developers form an accurate explanation of the system's behavior, so that the developer may conceive of an appropriate modification to the program. The Whyline for Alice presents the situations as a data and control flow graph of the execution events that caused or prevented the behavior in question, alongside a textual answer to the question (Figure 5h). In the example in Figure 5, the `BigDot.isEaten` flag was true, causing the condition's expression to evaluate to false, and preventing the resize animation from executing.

The Whyline for Alice has several interactive features to help a developer understand the information. The time cursor, the black vertical line in the timeline, allows developer to select events in the timeline, which causes the Whyline to highlight the code that caused the event, and changes the state of the output to match the state of the program at the time of the selected event. This allows developers to "scrub" the output history, helping them to associate the code, the execution, and the corresponding output in a single gesture. Developers can also hover over any variable in the code to see its current value, based on the current position of the time cursor. Developers can ask further questions about the output, or anything displayed in the answer. These answers are then combined with the answers to the previous questions, in order to help the developer understand the relevant execution context.

4.6 EVALUATION

In a study comparing a version of Alice with the Whyline to a version of Alice without, developers with the Whyline spent a factor of 8 less time debugging and got 40% further through their task of developing a Pac Man game [24]. These results are portrayed in Figure 6. These gains were due largely to the fact that developers *without* the Whyline generated multiple false hypotheses about what code was causing failures, and spent considerable time investigating them, often inserting new errors in the program as they attempted to repair existing errors. Developers with the Whyline asked about the behavior they did or did not expect, and were shown the code responsible. As a result, they spent less time investigating false hypotheses, inserted fewer errors, and formed more accurate explanations of their programs' behavior. Developers who used the Whyline had very positive opinions about its usefulness, and repeatedly expressed their desire to have such a tool for the languages they used regularly.

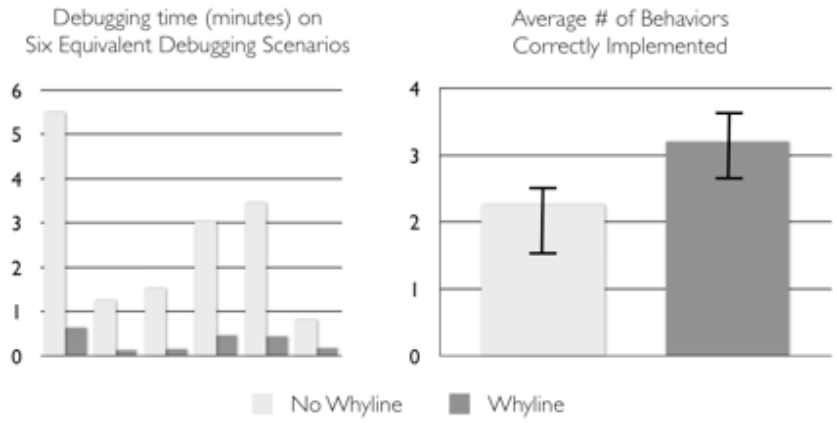

Figure 6. On the left, a comparison of debugging times with and without the Whyline on six debugging scenarios, and on the right, a comparison of the number of behaviors correctly implemented.

5. GENERALIZING THE WHYLINE

Although the Whyline prototyped for Alice demonstrated the feasibility of the Whyline approach, it only did so for a simple language, a small program, and a simplified set of questions about program behavior. When designing a Whyline for more general-purpose languages and more complex systems, there are a number of issues of scale that must be further investigated, which I discuss in detail below:

- Support for a more complex language (Java).
- Techniques for tracing programs efficiently, in both space and time.
- Algorithms for determining structural granularities of program output.
- User interfaces for specifying the context of a behavior in a question.
- Support for asking three types of *why didn't* questions, and algorithms for answering each.
- Incremental analyses and interaction techniques that help developers explore an answer.

5.1 SUPPORTING A MORE COMPLEX LANGUAGE

There are several languages that I could attempt to support, but I chose Java mainly because there are several open source projects that I can use to help implement the prototype, and Java has a more consistent and simpler design than other widely used languages such as C++ and C#.

Besides these pragmatic reasons, Java also poses a number of interesting technical and interaction design challenges. The majority of a Java program's execution consists of method invocations, whereas Alice programs typically have very few invocations. Furthermore, Alice's method invocations are statically bound (the method being called is known before runtime), and Java's invocations are dynamically bound (the method being called is only known at runtime). This makes some of the static analyses used in the Whyline for Alice less precise, since there may be several *potential* invocations of a method that may never actually invoke the method. Furthermore, there are also technical challenges in handling Java's support for multithreaded applications and synchronization. Although I anticipate having to address these challenges in novel ways, there is prior work to guide my solutions [52].

5.2 TECHNIQUES FOR TRACING PROGRAMS EFFICIENTLY

A central issue in supporting complex Java programs is the feasibility of recording an execution trace sufficient for answering why questions. The Whyline for Alice used dynamic slicing algorithms, which have been successfully implemented for Java by recording only information about a program's execution that cannot be determined statically [52, 53]. These techniques have generally resulted in traces of several hundred megabytes or more for numerical programs on the order of a few thousand lines. I do anticipate needing to record more information than these techniques, particularly information about the values of variables at runtime. I have begun implementing a tool to investigate the feasibility of recording this information, and my preliminary results suggest that although the traces are as much as twice as large as those cited in the prior work, they are not so large to make the tool infeasible for programs of average size and complexity. One approach to limiting the size of the trace is to only record information about the execution of the developer's code, and not of the code executed through external APIs (except for output). This way, the data dependencies can be stopped at the invocation of an API method,

rather than having to record information about the API's execution. This would not be a severe limitation, since developers generally do not have access to the source code of an API anyway.

In addition to assessing the feasibility of tracing Java programs, there are also two possible approaches to implementing the tracing. One is to instrument Java bytecode, which would allow the prototype to be used on any platform, but would involve greater difficulty and performance overhead in logging of multithreaded applications. The alternative would be to modify a Java virtual machine to performing the logging outside of the program's execution. The limitation of this approach is that the prototype could not be used on other platforms. Others have considered these issues [47], and so I anticipate finding at least one workable solution.

5.3 Algorithms for Determining Structural Granularities of Program Output

For the Alice Whyline, it was obvious what parts of an Alice program constituted "output," because the only type of output involved changes to visible properties of 3D objects. In general, however, a Java program may have a variety of output, including graphical primitives and text rendered onto a rectangular canvas, textual output displayed in a console, network activity, sound, and writing to recordable media, among others.

I propose to focus the Whyline's support on questions about graphical primitives such as rectangles, lines, ellipses, and text, and their attributes (such as color, size, location, font, etc.), and aggregates of these primitives. To implement this, I will trace all calls on instances of `java.io.PrintStream` and `java.awt.Graphics2D` (such as `println` and `fillRect`), and log the arguments used to execute them. I will then use this history of output to reconstruct an interactive history of the graphical and textual output produced by the program. These reconstructed histories will allow developers to freely navigate the output history produced by the program, simulating reverse execution, and provide a user interface foundation for querying output by direct manipulation. These histories are portrayed in Figure 7, which shows mockups of a developer asking questions about a graphical and textual output history.

According to my studies, many of the developers' questions will not be at the level of graphical and textual primitives, but at the level of user interface components, lists of objects, and other high-level software entities. To support questions about these, I will develop techniques of inferring these higher-level output structures from the execution history.

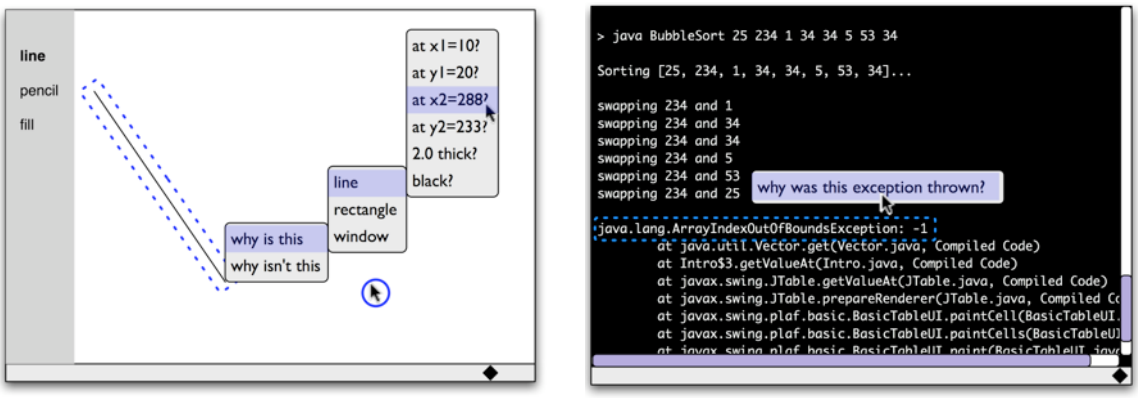

Figure 7. Mockups of graphical (left) and textual (right) output histories. Developers would use these interactive histories to select things to ask questions about. The position of the mouse cursor is shown in the graphical history, and circled to distinguish it from the live mouse cursor.

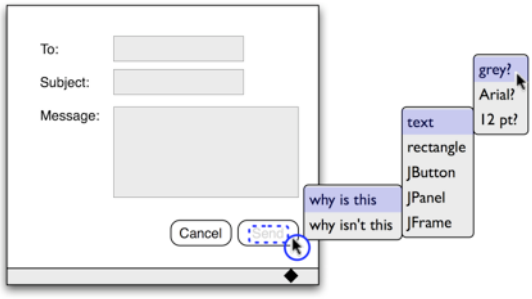

Figure 8. The Whyline for Java will determine many granularities of output structure to allow developers to clearly specify the output of interest.

One approach will be to aggregate output primitives that have a common execution context. For example, the `JButton` component in the Swing toolkit renders many graphical primitives in its `paintComponent()` method to create its appearance, and a Swing `JFrame` window indirectly renders all of the output in the window with invoking a single method. These groups of primitives could be referred to by the names of their originating object, such as `JButton` and `JFrame`, as shown in Figure 8. The Whyline might also detect changes to these higher-level objects over time; for example, even though a `JButton` may be rendered repeatedly as a window resizes, resulting in different instances of graphical primitives, the same object and method were responsible for rendering these primitives each time. These approaches may require the tool to have special knowledge about Swing in order to exclude invocations that do not have a proper on-screen representation.

In addition to inferring high-level output *structures*, I will also determine high-level application *state* that affects a program's output. For example, the `JButton`'s `enabled` flag determines whether its text label is grayed out. I will develop techniques to analyze these dependencies to identify such application state, allowing developers to ask specifically about the state, rather than having to phrase questions in terms of low level primitives. One challenge of this approach is determining how far to follow these data dependencies; for example, a developer's code may have other state that determines the state of the `enabled` flag. I will use a heuristic that identifies state at the edge of interfaces between the developer's code and the program's output. These are usually found at the public interfaces of APIs, such as user interface toolkits.

5.4 Interfaces for Specifying the Context of a Behavior

The Whyline for Alice assumed that every question referred to the most recent execution of a output statement, limiting developers' ability to specify the temporal context of the behavior they wanted to question. However, our study of developers' descriptions of software problems suggest that the context of a problem is central to identifying and understanding the problem. To enable developers to specify this context, I propose to include *input* events as part of the reconstructed output history, and allow developers to select these input events in order to indicate the context of a question, as portrayed in Figure 9. Input events would be explicitly represented, as shown by the drag event in Figure 9 (this is the same visual approach used in Marquise [38], a programming by demonstration system). This would allow the Whyline to give more specific answers about the particular behavior, by focusing its analyses on a particular segment of the program's execution.

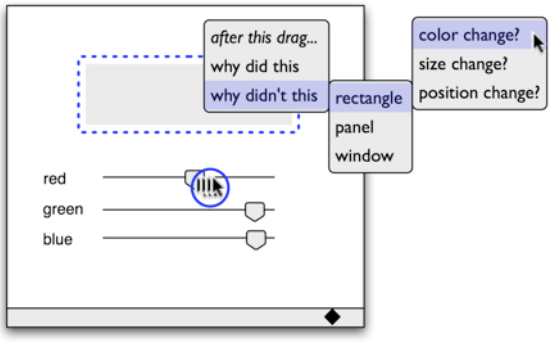

Figure 9. A developer moves the time cursor to when the slider drag event happens, contextualizing a question about the rectangle's color.

5.5 WHY DIDN'T QUESTIONS

In general, there are three types of *why didn't* questions that the Whyline could answer: (1) those which refer to some output statement that was not executed; (2) those that refer to some output that *was* executed, but with inappropriate arguments; and (3) those that refer to output for which *no* code has been written. I will support each of these to varying degrees.

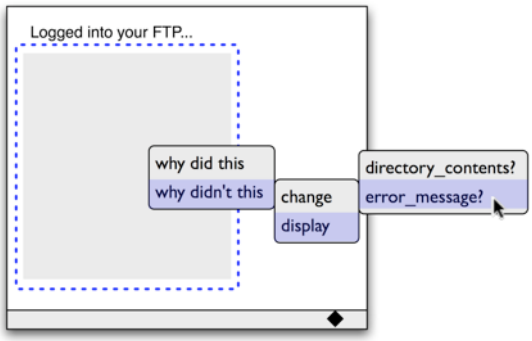

Figure 10. The *why didn't* questions available in a menu could be determined by what output statements the selected entity might execute.

For questions that refer to output that was not executed, the central challenge is providing a means by which the developer can ask about the output, since it will not be part of the reconstructed output history. The simple approach taken by the Alice Whyline was to include a menu with *all* of the program's output statements, but this likely will not be feasible for larger programs. One approach is to allow developers to refer to a specific entities in the output, and present *why didn't* questions only about the output statements that the particular entity might feasibly execute, as in Figure 10. Furthermore, some output statements can only execute as a result of particular input events (for example, any output statement in a mouse click event handler), so a question that specifies context as in the previous section would only have to display questions about behaviors that were possible after the input event. Another approach would be to determine what output statements could have feasibly affected a location that a developer specifies on-screen. For example, a dialog window may appear anywhere on screen, but changes to a text field are likely limited to a much smaller region. Once the developer chooses the desired question, the Java Whyline would use the same approach as in Alice, and determine the predicates that evaluated to false, preventing the execution of the output. In addition to predicates, the Java Whyline would also determine all of the potential invocations of the method containing the output statement, and determine what prevented those invocations from executing.

For questions that refer to an output statement that *was* executed, but not with the appropriate arguments, there is an issue of specificity. For example, one way of allowing developers to phrase such questions would be of the form "Why didn't this change to value V?" This specific phrasing would require the Whyline to determine why a particular variable did not have a particular value, which is likely intractable because of the combinatorial possibilities. However, there is evidence that while developers can sometimes recognize wrong values, they have more difficulty conceiving of appropriate values [40]. Therefore, I plan on supporting questions of the more general form, "Why didn't this change?", as in Figure 9. This type of question would be answered by determining the statements in the program that could have changed the argument's value, and determining why these did not execute.

To allow questions that refer to output for which *no* code has been written is generally intractable, because it would require a tool that could translate a description of the desired behavior and generate code for it. There are, however, some situations in which heuristics can be applied. For example, in any Java program, there are a limited number of ways to cause a user interface component to repaint itself; if such code is missing from a program in a particular context, the Whyline may be able to offer this solution as a possibility. I will identify other scenarios in which the Whyline can offer simple change suggestions, along the lines of Abraham and Erwig [1].

5.6 Incremental Analyses and Interaction Techniques for Exploring an Answer

Because Alice programs were generally small, presenting the full causal chain of execution of events was feasible, since the chains generally only spanned a few screens. More complex Java programs may have chains that are much larger, and so a central design challenge is to compute such chains efficiently, and present them in a manner that allows developers to understand the information without being overwhelmed.

The Java Whyline will have the time cursor and the output history, just like the Alice Whyline; these are portrayed in the mockup in Figure 11. However, the answers will differ. Much of a Java program's execution involves method invocations and computations, but showing sequences of invocations and arithmetic might not be helpful to developers, since these are somewhat straightforward for experienced developers to read directly from the code. Instead, I will focus on presenting information that can only be known at runtime, including the values assigned to variables and the decisions made at conditionals. These can be seen in the answer in Figure 11, but other information, such as the methods in which the events occurred and intermediate computations, would be shown in the code viewer once the event was selected.

I will also focus on helping developers manage the size of answers by developing incremental versions of the program analyses that I use, given empirical evidence that the time to compute full dynamic slices on programs of ten thousand lines or more is on the order of minutes, and hours for larger programs [53]. This way, as developers investigate the answer, the Whyline will incrementally compute information as requested, minimizing the amount of time that developers must wait for answers to their questions. The interfaces to invoke these incremental computations might be implemented as shown in the answer in Figure 11, in which the causes of the events can be shown when the developer clicks on the ellipses that are next to the causality arrows.

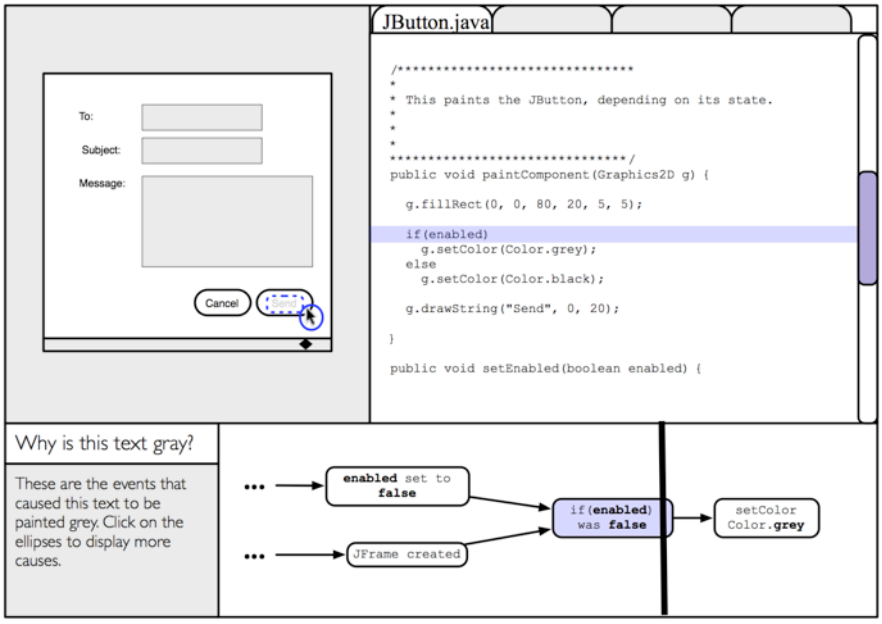

Figure 11. A mockup of a Java Whyline's answer to the question posted in Figure 8, explaining that the button's text was grey both because the enabled flag was false, and because the window was created. The causes of these two events can be inspected by clicking on the ellipses.

6. EVALUATION

Before performing a formal evaluation of the Java Whyline, I will perform several iterations of usability testing to improve the design of the prototype.

Once the prototype is sufficiently usable, my primary goal for the evaluating the Java Whyline will be to demonstrate its effectiveness relative to the most widely available program understanding tools: logging mechanisms and breakpoint debuggers. To do so, I will design a controlled experiment in which the control group will receive a modern software development environment such as Eclipse and the experimental group will receive the same environment, but with additional support from the Whyline. Because I am interested in evaluating the tools' perceived utility, developers will not be *required* to use any of the available tools. However, all developers will receive brief tutorials on using the tools, so that they can form initial perceptions of the tools' benefits and limitations. In addition to comparing the Whyline for Java to logging mechanisms and breakpoint debuggers, I also intend to include a third group, which will receive a Whyline prototype that only allows questions about *code* and *not* behavior, in order to test the central thesis that asking about *behavior* is central to helping developers form more accurate hypotheses.

The tasks that developers will attempt will be debugging and feature enhancement tasks on open source programs of various sizes. My current plans are to include a debugging and feature enhancement task on a smaller program on the order of 1,000 lines of non-comment, non-whitespace code, and a debugging and feature enhancement task on a medium sized program on the order of 10,000 lines. If feasible, I will also include a much larger system on the order of 100,000 lines, but the performance and robustness of the prototype may preclude a system of this size. To select these tasks, I will search the open source projects' bug report databases for a variety of representative tasks that involve many kinds of program output.

The dependent variables in the experiment will be *task completion time, quality of solution*, and a developers' *understanding* of the subject programs. This latter variable is included as a more sensitive measure of success, since it may be difficult to find tasks that all developers can complete, but are not so simple that they mask the effectiveness of the Whyline. Task completion time will be measured from the start of a task to the time at which developers believe they are finished. The quality of their solution will be measured by experts, relative to the degree to which it maintains the original design of the system. Developers' understanding will be measured by developing a test that samples developers' knowledge of various aspects of the systems' design and implementation. These tests will be modeled after tests developed in others' prior work on assessing developers' mental models of programs [11].

Developers will be recruited from the local community, and will have sufficient experience with the software development environment in which I perform the tests to minimize avoidable learning effects.

If the prototype is sufficiently robust for deployment, I may make a public, open source release of the prototype in addition to the more formal evaluation. As part of this deployment, I will solicit users' feedback and qualitatively assess the impact of the tool on their work through self-reports. This will also be an opportunity to identify the limitations of the tool's support for real-world Java programs, and allow developers from around the world to improve the prototype. The deployment will also allow users the option of sending low-level usage data from an instrumented version of the prototype to enable detailed analyses.

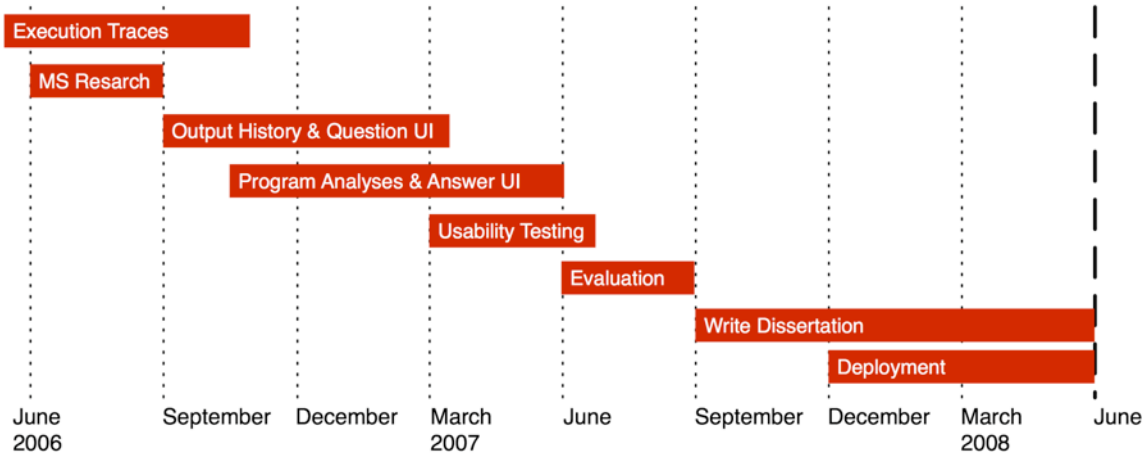

Figure 12. Proposed schedule, ending in June of 2008.

7. SCOPE

The proposed work could be extended in a number of directions that I may or may not pursue as part of my thesis work:

- The impact of the Whyline on the learning of programming and debugging strategies.
- The use of execution traces for other purposes, such model checking, anomaly and invariant detection, and other dynamic analyses.
- The feasibility of the tool in distributed, embedded, and collaborative contexts (for example, the feasibility of debugging code remotely or on mobile devices).
- The integration of a Whyline with other development tools, such as source code editors [28], and program understanding tools that support the navigation and analysis of source code [44].
- Other types of questions and queries, such as those that filter the execution trace for matching events, or perform clustering or other statistical analyses on the execution trace.

8. SCHEDULE

My proposed schedule is shown in Figure 12. I intend to spend the summer of 2006 at Microsoft Research investigating program understanding issues in an industrial, collaborative context with Rob DeLine. When I return from Microsoft, I will spend the 2006-2007 school year designing, implementing, and usability testing the Whyline prototype. In the following summer of 2007, I will perform the formal evaluation of the prototype's effectiveness. In the fall of 2007, I will begin writing my dissertation and publishing my results. For the remainder of the 2007-2008 school year, I will finish writing my dissertation and deploy the prototype as an open source project. My plans are to graduate at the end of the 2007-2008 academic year, utilizing the remaining support of my NSF Graduate Research Fellowship.

9. ACKNOWLEDGEMENTS

I would like to thank Brad Myers for advising me on this work. This work has benefited greatly from discussions with other faculty, students and staff, including Scott Hudson, Jeff Nichols, Jacob Wobbrock, Jeff Stylos, Andrew Faulring, Michael Coblenz, Htet Htet Aung, Duen Horng Chau, Elsabeth Golden, Karen Tang, Santosh Mathan, Donna Malyeri, and Thomas LaToza. I would also like to thank my undergraduate advisor Margaret Burnett, and her many students and colleagues, for encouraging me to pursue a Ph.D. I would also like to thank my family for their continuing support of my career, especially my wife, Katherine, my daughter Ellen, and my parents, Robert and Judith. This work was supported by the National Science Foundation under NSF grant IIS-0329090 and the EUSES consortium under NSF grant ITR CCR-0324770. The author is also supported by an NDSEG fellowship and by a NSF Graduate Research Fellowship.

10. REFERENCES

[1] R. Abraham and M. Erwig, Goal-Directed Debugging of Spreadsheets, IEEE Symposium on Visual Languages and Human-Centric Computing 2005, Dallas, Texas, 37-44.

[2] T. Akgul, V. J. M. III, and S. Pande, A Fast Assembly Level Reverse Execution Method via Dynamic Slicing, International Conference on Software Engineering 2004, Scotland, UK, 522-531.

[3] R. Baecker, C. DiGiano, and A. Marcus, Software Visualization for Debugging, *Communications of the ACM*, 40, 4, 44-54, 1997.

[4] E. L. A. Baniassad, G. C. Murphy, C. Schwanniger, and M. Kircher, Managing Crosscutting Concerns During Software Evolution Tasks: An Inquisitive Study, Aspect-Oriented Software Development 2002, Enscheda, The Netherlands, 120-126.

[5] X. Baowen, Q. Ju, Z. Xiaofang, W. Zhongqiang, and C. Lin, A brief survey of program slicing, *SIGSOFT Software Engineering Notes*, 30, 2, 1-36, 2005.

[6] B. W. Boehm, Software Engineering, *IEEE Transactions on Computers*, 12, 25, 1226-1242, 1976.

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

[8] R. Brooks, Towards a Theory of the Cognitive Processes in Computer Programming, *International Journal of Human-Computer Studies*, 51, 197-211, 1999.

[9] W. R. Bush, J. D. Pincus, and D. J. Sielaff, A Static Analyzer for Fnding Dynamic Programming Errors, *Software Practice and Experience*, 30, 7, 775-802, 2000.

[10] H. Cleve and A. Zeller, Locating Causes of Program Failures, International Conference on Software Engineering 2005, St. Louis, MI.

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

[12] D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe, Extended Static Checking, Compaq Systems Research Center SRC Research Report 159, December 18 1998.

[13] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, Does Code Decay? Assessing the Evidence from Change Management Data, *IEEE Transactions on Software Engineering*, 27, 1, 1-12, 2001.

[14] M. Eisenberg and H. A. Peelle, APL Learning Bugs, APL Conference 1983, Washington, D. C., 11-16.

[15] M. Eisenstadt, "Tales of Debugging from the Front Lines," in *Empirical Studies of Programmers: Fifth Workshop*, C. R. Cook, J. C. Scholtz, and J. C. Spohrer, Eds. Palo Alto, CA: Ablex Publishing Corporation, 1993, 86-112.

[16] M. A. Francel and S. Rugaber, The Value of Slicing While Debugging, *Science of Computer Programming*, 40, 2-3, 151-169, 2001.

[17] P. Fritzon, N. Shahmehri, and M. Karkar, Generalized Algorithmic Debugging and Testing, *ACM Letters on Programming Languages and Systems*, 1, 4, 303-322, 1992.

[18] A. Greenhouse, T. J. Halloran, and W. L. Scherlis, Observations on the Assured Evolution of Concurrent Java Programs, *Science of Computer Programming*, 58, 3, 384-411, 2005.

[19] L. Gugerty and G. M. Olson, "Comprehension Differences in Debugging by Skilled and Novice Programmers," in *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Washington, DC: Ablex Publishing Corporation, 1986, 13-27.

[20] I. R. Katz and J. R. Anderson, Debugging: An Analysis of Bug-Location Strategies, *Human Computer Interaction*, 3, 351-399, 1988.

[21] C. Kehoe, J. Stasko, and A. Taylor, Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study, *International Journal of Human-Computer Studies*, 54, 2, 265-284, 2001.

[22] P. B. Kessler, Fast Breakpoints: Design and Implementation, Programming Language Design and Implementation 1990, 78-84.

[23] D. Knuth, The Errors of TeX, *Software: Practice and Experience*, 19, 7, 607-685, 1989.

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

[25] A. J. Ko, B. A. Myers, and H. Aung, Six Learning Barriers in End-User Programming Systems, IEEE Symposium on Visual Languages and Human-Centric Computing 2004, Rome, Italy, 199-206.

[26] A. J. Ko, H. Aung, and B. A. Myers, Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, International Conference on Software Engineering 2005, St. Louis, MI, 126-135.

[27] A. J. Ko and B. A. Myers, 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, 2005.

[28] A. J. Ko and B. A. Myers, Barista: An Implementation Framework for Enabling New Interaction Techniques and Visualizations in Code Editors, ACM Conference on Human Factors in Computing 2006, Montreal, Canada, to appear.

[29] A. J. Ko, B. A. Myers, and D. H. Chau, A Linguistic Analysis of How People Describe Software Problems in Bug Reports, Submitted for publication 2006.

[30] J. Koenemann and S. P. Robertson, Expert Problem Solving Strategies for Program Comprehension, Conference on Human Factors and Computing Systems 1991, New Orleans, Louisiana, 125-130.

[31] T. LaToza, G. Venolia, and R. DeLine, Maintaining Mental Models: A Study of Developer Work Habits, International Conference on Software Engineering 2006, Shanghai, China, to appear.

[32] M. M. Lehman and L. Belady, *Software Evolution – Processes of Software Change*. London: Academic Press, 1985.

[33] R. Lencevicius, U. Holzle, and A. K. Singh, Dynamic Query-Based Debugging of Object-Oriented Programs, *Journal of Automated Software Engineering*, 10, 1, 367-370, 2003.

[34] B. Lewis, Debugging Backwards in Time, International Workshop on Automated Debugging 2003, 225-235.

[35] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, Mental Models and Software Maintenance, Empirical Studies of Programmers, 1st Workshop 1986, Washington, DC, 80-98.

[36] S. Mukherjea and J. Stasko, Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger, *ACM Transactions on Computer-Human Interaction*, 1, 3, 215-244, 1994.

[37] B. A. Myers, Incense: A System for Displaying Data Structures, Computer Graphics 1983, Detroit, MI, 115-125.

[38] B. A. Myers, R. G. McDaniel, and D. S. Kosbie, Marquise: Creating Complete User Interfaces by Demonstration, Human Factors in Computing Systems 1993, Amsterdam, The Netherlands, 293-300.

[39] N. Pennington, Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs, *Cognitive Psychology*, 19, 295-341, 1987.

[40] A. Phalgune, C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. R. Ruthruff, Garbage In, Garbage Out? An Empirical Look at Oracle Mistakes by End-User Programmers, IEEE Symposium on Visual Languages and Human-Centric Computing 2005, Dallas, Texas.

[41] M. P. Robillard and G. C. Murphy, Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, International Conference on Software Engineering 2002, 406-416.

[42] M. P. Robillard and G. C. Murphy, Automatically Inferring Concern Code from Program Investigation Activities, International Conference on Automated Software Engineering 2003, 225-234.

[43] M. P. Robillard, W. Coelho, and G. C. Murphy, How Effective Developers Investigate Source Code: An Exploratory Study, *IEEE Transactions on Software Engineering*, 30, 12, 889-903, 2004.

[44] M. P. Robillard, Automatic Generation of Suggestions for Program Investigation, ACM SIGSOFT Symposium on the Foundations of Software Engineering 2005, 11-20.

[45] D. S. Rosenblum, A Practical Approach to Programming With Assertions, *IEEE Transactions on Software Engineering*, 21, 1, 19-31, 1995.

[46] E. H. Satterthwaite, Source Language Debugging Tools: Stanford University Computer Science Department, 1975.

[47] V. Schuppan, M. Baur, and A. Biere, JVM Independent Replay in Java, Runtime Verification 2004, Barcelona, Spain, 76-94.

[48] R. Sosic and D. A. Abramson, Guard: A Relative Debugger, *Software Practice and Experience*, 27, 2, 185-206, 1997.

[49] T. G. Stockham and J. B. Dennis, FLIT - Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0, MIT, Cambridge, MA Memo 5001-23, July 1960.

[50] D. Ungar, H. Lieberman, and C. Fry, Debugging and the Experience of Immediacy, *Communications of the ACM*, 40, 4, 39-43, 1997.

[51] A. Vans and A. v. Mayrhauser, Program Understanding Behavior During Corrective Maintenance of Large-scale Software, *International Journal of Human-Computer Studies*, 51, 1, 31-70, 1999.

[52] T. Wang and A. Roychoudhury, Using Compressed Bytecode Traces for Slicing Java Programs, International Conference on Software Engineering 2004, Scotland, UK, 512-521.

[53] X. Zhang and R. Gupta, Cost Effective Dynamic Program Slicing, Programming Language Design and Implementation 2004, Washington, D.C.