Finding Causes of Program Output with the Java Whyline

Andrew Ko
The Information School, DUB Group
University of Washington
Seattle, WA 98195
ajko@u.washington.edu

Brad A. Myers
HCI Institute, School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu

ABSTRACT
Debugging and diagnostic tools are some of the most important software development tools, but most expect developers choose the right code to inspect. Unfortunately, this rarely occurs. A new tool called the Whyline is described which avoids such speculation by allowing developers to select questions about a program's output. The tool then helps developers work backwards from output to its causes. The prototype, which supports Java programs, was evaluated in an experiment in which participants investigated two real bug reports from an open source project using either the Whyline or a breakpoint debugger. Whyline users were successful about three times as often and about twice as fast compared to the control group, and were extremely positive about the tool's ability to simplify diagnostic tasks in software development work.

AUTHOR KEYWORDS
Debugging, Whyline, slicing, instrumentation.

ACM Classification Keywords
D.2.5 [Testing and Debugging]: Debugging aids, tracing; H. 5.2 [User Interfaces]: User centered design.

INTRODUCTION
In 2002, the National Institute of Standards and Technology estimated that testing and debugging account for 30-90% of software development costs, further finding that the average error takes 17.4 hours to find and fix [19]. According to the respondents in the study, the problem is the lack of effective tools. In effect, millions of developers work to improve the world's software infrastructure using little more than breakpoints and print statements.

A new approach to debugging tools, called the Whyline [8], has the potential help dramatically: by allowing developers to ask questions about program output, in one study it reduced debugging time by a factor of 8. It achieved this through a simple insight. Given a failure, there must be some *observable symptom* of failure, such as a suspicious value or a lack of feedback. When a developer sees such a symptom, they must *guess* about its cause and then test their hypothesis using tools, perhaps setting a breakpoint or writing a print statement. Unfortunately, developers usually guess incorrectly, spending considerable time exploring unrelated code before eventually finding the cause [9]. By supporting questions about *output* instead of *code,* the Whyline not only avoids speculation about the causes of a failure, but also simplifies the exploration of code *responsible* for the output.

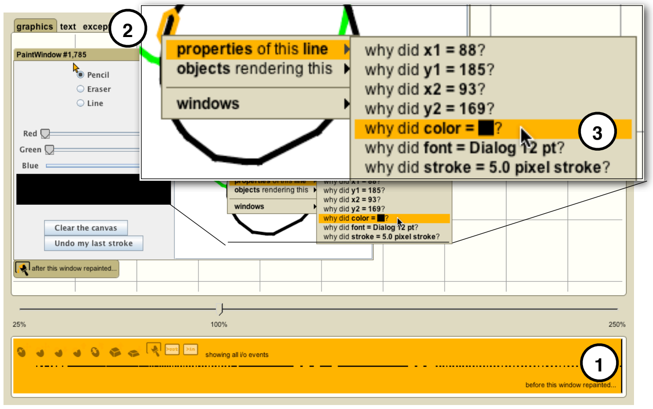


Figure 1. After the recording loads, the developer can choose a time in the history as if it were a digital movie (1), then click the desired output (2) to see questions (3).

Unfortunately, the original Whyline [8], designed for Alice (*www.alice.org*), did not scale to the types of programs developed by professional developers. It supported a limited set programming language features, ignored functions, included a global question menu that quickly grew in size, and presented answers that were highly tailored to Alice animation primitives. To support complex modern languages such as Java, the Whyline user interface had to be reinvented.

In this paper, we contribute a new Whyline user interface, designed to support Java programs of widely varying complexity, while preserving the benefits of the original. The Java Whyline allows developers to select "why did" and "why didn't" questions that are extracted from the program and a recording of its execution (Figure 1). The tool then answers the question using automated program analyses, helping the developer explore the causal relationships between the output and the program's execution. We have described the data structures, algorithms, and technical scope of our prototype in a prior publication [13]. In this paper we focus on the interaction design of the Whyline's user interfaces. For example, we contribute:

- Direct manipulation queries about program output ranging from graphical primitives to the program abstractions they represent.
- An interactive timeline visualization that serves as (1) an answer to questions, (2) an interface for *followup questions*, (3) a navigational aid, and (4) and an information gathering bookmarking tool.
- A familiarity heuristic for filtering and ranking questions and annotating views throughout the UI.

In addition to these contributions, we also provide empirical evidence that the Java Whyline helps developers avoid speculation and explore more relevant code. Our results show that Whyline users were successful about three times as often, were about twice as fast compared to the control group, and were extremely positive about the tool's ability to simplify diagnostic tasks in software development work.

In the rest of this paper, we discuss related work on debugging and diagnostic tools, provide an example of the Java Whyline in use, and discuss the design and design rationale of the Java Whyline user interface. We end with our experimental results.

RELATED WORK

Debugging as a human activity has been studied for decades. There is considerable evidence that debugging is hypothesis-driven [7]. Descriptions of how developers test their hypotheses vary, but generally show that successful developers are more systematic and objective [17]. Studies have also considered the types of questions that developers ask to test their hypotheses. Weiser showed that developers trace backwards to understand "slices" of program behavior [21]. Sillito extended this finding [18], showing that developers seek "focus points," or places in code related to the developer's goal. A similar study framed program understanding as "fact finding" [14], driven by efforts to discover properties of a program.

Research on debugging *tools* have taken a different trajectory. Many help developers watch execution, by setting breakpoints, or sampling events in program execution [15]. Others still help developers analyze causality in program execution, most notably slicing approaches [2]. The common feature of these approaches is that they require a developer to first speculate about what code is related to a failure. Other approaches compare events within [6] and between [22] program executions to find anomalies or trends. These help avoid speculation, but also severely limit the conditions in which the tools can identify bugs.

There are also several question-asking tools in non-programming domains. One major difference between these and the Whyline is the type of *dependencies* that are used to explain causality. The ACT-R framework [3] supports "why not" questions about production rule systems, giving answers in terms of rules that did not fire. Some AI systems support "why not" questions about why some data was not used in answering queries to a knowledge base [4]; here, answers consist of domain-specific dependencies from an ontology. Lieberman explored "why" questions about e-commerce processes [20], with answers constructed from business transactions. The Whyline concept has also inspired projects looking at constraints in UI design [5], spreadsheets [1], and application state in word processors [16]. Each of these present answers in ways tailored to their domain, but none have dealt with artifacts as complex as large Java programs.

THE WHYLINE FOR JAVA

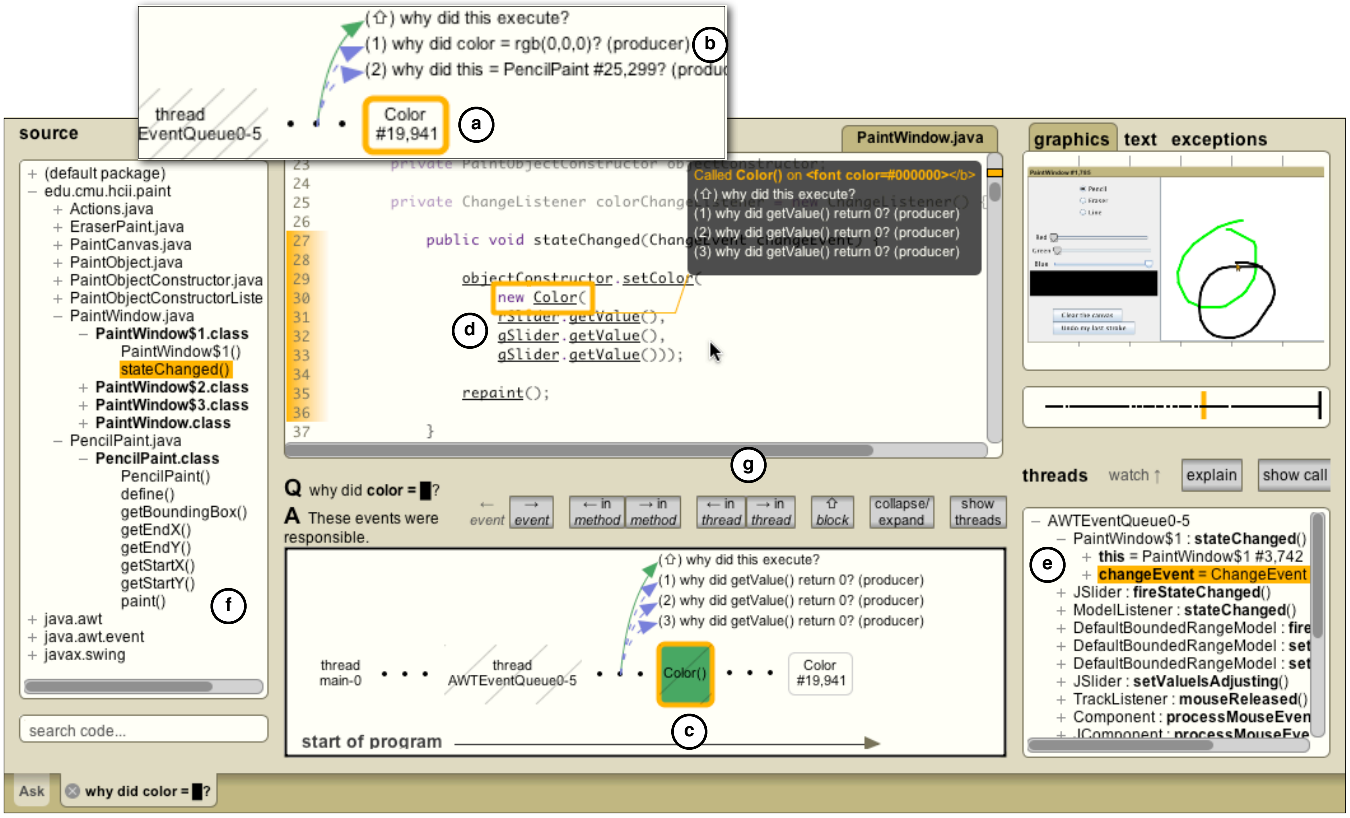
To describe the Whyline, let us begin with an example. A prior study [10] involved a painting program, which supported drawing colored strokes (see Figure 1a). Among the 500 lines of code, there were a few bugs in the program unintentionally inserted, which were left in for the study. One problem was that the RGB sliders did not create the right colors. In the study, users took a median of 10 minutes to find the problem, mostly using text searches for "color" to find relevant code.

With the Whyline, the process is simpler and faster (see Figures 1 and 2). The user launches their program and demonstrates the problem, in this case by drawing a stroke with an unexpected color. After quitting the program, the Whyline reads the recording from disk, opening a Whyline window. The user then moves the time controller (a), selecting the time that the problem occurred. Then, the user clicks on anything related to the problem to see questions (b). In this case, the click is on the stroke with the wrong color, showing 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 (Figure 2a,c). 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 (d), along with the call stack and locals at the time of the selected execution event (e). In this case, the Whyline selects the most recent event in the answer, which was the color object used to paint the stroke (a). To find out where the color came from, the user could find the source of the value by selecting the question "why did color = rgb(0,0,0)" (b). This causes the selection to go to the instantiation event (c) and the corresponding instantiation code (d). Here, the user notices that the green slider was used for the blue part of the color; it should have used the blue slider.

Asking Questions

One difference between the Java Whyline and other tools (including the Alice Whyline), is that it analyzes a *recording* of a program rather than a live program. This was a conscious choice: supporting live debugging *and* recording incurs too much performance overhead, and


(⇧) why did this execute?
(1) why did color = rgb(0,0,0)? (producer)
(2) why did this = PencilPaint #25,299? (produ
thread EventQueue0-5
Color #19,941
a
b
source
(default package)
edu.cmu.hcii.paint
Actions.java
EraserPaint.java
PaintCanvas.java
PaintObject.java
PaintObjectConstructor.java
PaintObjectConstructorListe
PaintWindow.java
PaintWindow$1.class
PaintWindow$1()
stateChanged()
PaintWindow$2.class
PaintWindow$3.class
PaintWindow.class
PencilPaint.java
PencilPaint.class
PencilPaint()
define()
getBoundingBox()
getEndX()
getEndY()
getStartX()
getStartY()
paint()
java.awt
java.awt.event
javax.swing
f
search code...
PaintWindow.java
Called Color() on <font color=#000000></b>
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
d
g
Q why did color = ■?
A These events were responsible.
← event
→ event
← in method
→ in method
← in thread
→ in thread
⇧ block
collapse/ expand
show threads
(⇧) why did this execute?
(1) why did getValue() return 0? (producer)
(2) why did getValue() return 0? (producer)
(3) why did getValue() return 0? (producer)
thread main-0
thread AWTEventQueue0-5
Color()
Color #19,941
c
start of program
graphics
text
exceptions
threads
watch ↑
explain
show call
AWTEventQueue0-5
PaintWindow$1 : stateChanged()
this = PaintWindow$1 #3,742
changeEvent = ChangeEvent
JSlider : fireStateChanged()
ModelListener : stateChanged()
DefaultBoundedRangeModel : fire
DefaultBoundedRangeModel : set
DefaultBoundedRangeModel : set
JSlider : setValueIsAdjusting()
TrackListener : mouseReleased()
Component : processMouseEven
JComponent : processMouseEve
e
Ask
why did color = ■?

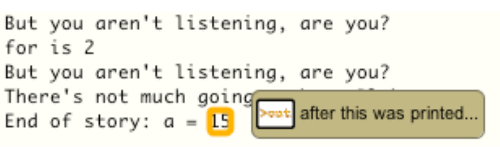
```
    private PaintObjectConstructor objectConstructor;

    private ChangeListener colorChangeListener = new ChangeListener() {

        public void stateChanged(ChangeEvent changeEvent) {

            objectConstructor.setColor(
                new Color(
                rSlider.getValue(),
                gSlider.getValue(),
                gSlider.getValue()));

            repaint();

        }
```

Figure 2. After choosing a question, the Whyline provides an answer (a), which the developer navigates using a followup question (b), revealing the color creation event (c). The code for this event (d) is the cause of the problem. Also shown are the call stack (e), source (f), and navigation buttons (g).

supporting the collaborative nature of debugging [11] by producing shareable recordings was deemed more important.

The Whyline window represents all the data in a Whyline recording. *Question asking mode* only shows program output and a time slider (Figure 1), and not code, to help developers avoid presuming that any particular code was responsible for a failure. The *answer mode*, which shows a visualization, source code and many other types of information (Figure 2), was designed to help developers juxtapose code and the *execution* of code, as we shall discuss later.

The Alice Whyline only supported questions about animations, because there were few other forms of output. For Java, however, output is more complex. Therefore, we chose to support primitives common to all Java programs, namely *graphical*, *textual*, and *exception* output (corresponding to the tabs in Figure 2's top right) and base higher-level questions on these primitives. When mousing over output, the Whyline highlights *primitive outputs*. For example, Figure 1 shows a line segment and Figure 3 shows a variable value printed to a console. Clicking on primitive output shows a menu of questions about attributes of the primitive output (as in Figure 4).



```
But you aren't listening, are you?
for is 2
But you aren't listening, are you?
There's not much going
End of story: a = 15
```

Figure 3. Mousing over textual output. The popup shows the question's temporal context.

Our studies show that most questions are about conceptual entities perceived on-screen, rather than just lines and text. Therefore, for graphical output, users can also ask questions about why a field that *affects* output has its current value, why such fields were not assigned a value, and also why a method was not executed after a certain time (Figure 5). All questions contain names extracted from code (e.g., `PaintCanvas` "canvas"). These were included to provide cues about the relevance of the contents of each sub-menu.

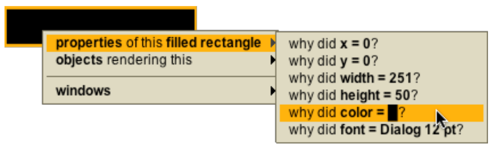


Figure 4. Questions about a rectangle's properties, derived from a `drawRect()` call.

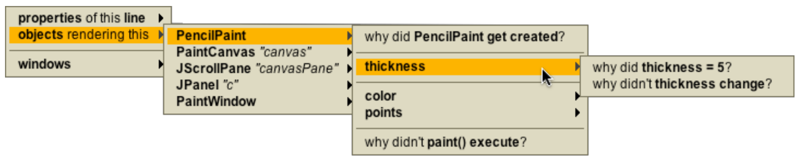


Figure 5. Questions related to a line, containing objects that the line represents, such as `PencilPaint` and data and methods that affect the line.

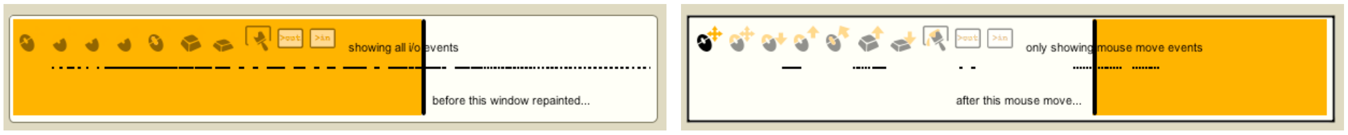

showing all i/o events
before this window repainted...
only showing mouse move events
after this mouse move...

Figure 6. The two modes of the output history timeline. "Why did" questions (left) analyze the past; "why didn't" questions (right) analyze the future. On the right, the view is filtered to only show mouse move events (as the left-most icon is selected to indicate).

These question types were all designed to be as close as possible to visible entities on-screen. For text, 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 algorithms for extracting questions from code are described in [13].

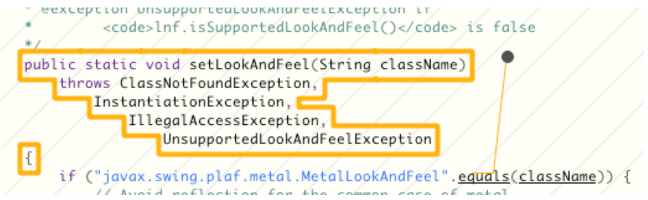
In addition to selecting the subject of a question, users must also select its *temporal context*. The Alice Whyline only supported questions relative to the end of the program, making it difficult to ask questions about earlier executions of the same code. The Java Whyline removes this limitation by providing a time slider (Figure 6) to allow the user to explore the output history of the program (by simply dragging or using the keyboard). Our study showed that developers' questions tend to be relative to a specific user input event [10], therefore, each black dot in the time slider represents an I/O event, such as a mouse click, keyboard press, or window repaint. The icons at the top of the time slider each represent a kind of I/O, allowing the user to filter events (e.g., the right of Figure 6 shows a mouse event filter selected, so that only mouse move events are shown).

The type of question determines how time is treated. For "why did" questions, the user needs to select the time at which the output in question is *visible*. "Why did" questions then reason *backwards* about the cause of the output prior to the time the output was rendered (as indicated by the highlighting in Figure 6). Conversely, "why didn't" questions ask about why something did not occur or change *after* a particular point in time. This differs from the Alice Whyline [8], which required the user to pause the program at the desired time. Also in Alice, "why didn't" questions reasoned about the whole execution history, rather than being scoped after the paused time; this was only adequate because Alice programs execute far less code.

Viewing Code

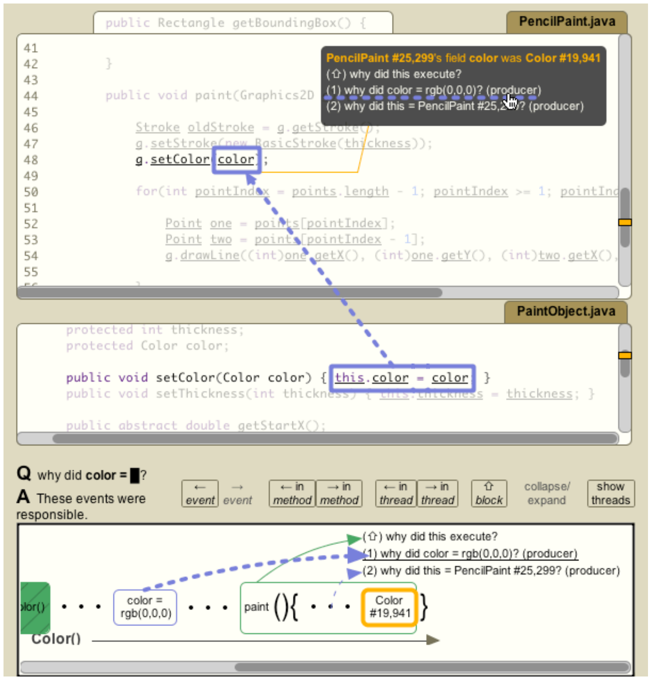
Several studies [12][16][17] have shown that *reading* code and understanding its relationships is a crucial part of program understanding. Therefore, rather than using a conventional editor, we designed a custom code *viewer* for the Whyline (with a standard syntax-colored, fixed-width layout). Source files are broken down into interactive *lines*, *tokens*, and *syntactic structures* for rendering onto the screen. This enables the Whyline to highlight complex token sequences, such as the header in Figure 7. Users can also click on individual tokens, lines, methods and fields to ask content-specific questions. For example, clicking on an *identifier* shows questions about the variable's current value (based on the time slider position) as well as questions about references to in the program. Clicking in the whitespace of a method allows users to ask about its callers and callees.

Also, for the selected event in an answer (e.g., Figure 2c), the Whyline automatically arranges relevant source files, rather than having the user manually open and arrange files. For example, Figure 8 shows two files arranged by the Whyline, with an arrow between two related elements in the files, with the rest of the code faded. This optimizes the readability and highlighting of relevant information about the user's selection. The file views also show followup questions about the highlighted relationship (as discussed next). The experience of navigating code is thus greatly simplified, only requiring a developer to change a selection in an answer to change the whole code view.



```
 *       <code>lnf.isSupportedLookAndFeel()</code> is false
 */
public static void setLookAndFeel(String className)
    throws ClassNotFoundException,
        InstantiationException,
            IllegalAccessException,
                UnsupportedLookAndFeelException
{
    if ("javax.swing.plaf.metal.MetalLookAndFeel".equals(className)) {
```

Figure 7. Precise highlighting in the Java Whyline source viewer and crosshatching over an unfamiliar source file.



PencilPaint.java

```
public Rectangle getBoundingBox() {

        }

        public void paint(Graphics2D

            Stroke oldStroke = g.getStroke();
            g.setStroke(new BasicStroke(thickness));
            g.setColor(color);

            for(int pointIndex = points.length - 1; pointIndex >= 1; pointInd

                Point one = points[pointIndex];
                Point two = points[pointIndex - 1];
                g.drawLine((int)one.getX(), (int)one.getY(), (int)two.getX(),
```

PencilPaint #25,299's field color was Color #19,941
(⇧) why did this execute?
(1) why did color = rgb(0,0,0)? (producer)
(2) why did this = PencilPaint #25,2??? (producer)

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();
```

Q why did **color** = █?
A These events were responsible.

Figure 8. Automatic arrangement and dependency highlighting of multiple files related to the selection.

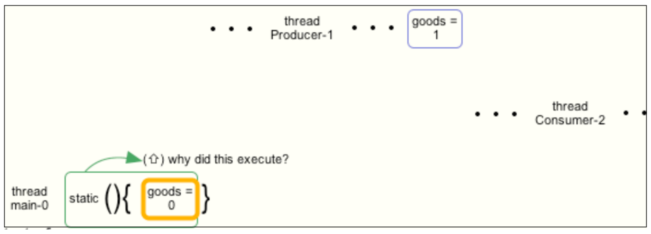


Figure 9. Threads separated along the y-axis.

Viewing Answers

What makes reading code difficult in a debugging task is that programs are written in a fixed and sequential way, but they execute in a dynamic, non-sequential fashion. An effective debugging tool needs views to expose both of these perspectives. The Java Whyline's *timeline visualization* was designed to portray the dynamic characteristics of program execution. The visualization uses a notation that was designed not to be understood in isolation, but alongside the code to which the visualization's events correspond. *Execution events*, the small labeled boxes as in Figure 9, are organized temporally along the x-axis and non-overlapping (since code does not technically execute in parallel). The events are separated by execution thread along the y-axis (as in Figure 9).

There are several types of events that appear in a Whyline visualization, each distinguished by a color. The two most important are *control* and *data* events, which are fundamental to the operation of computer programs. Events with *green* borders refer to *control* events, such as method calls and branches (such as `if` statements and loops). Events with *blue* borders refer to *data* events, such as assignments to different types of Java variables. *Orange* represents information about the event selected in the visualization (orange also highlights the corresponding code in the source files). *Grey* events are code that was not recorded by the Whyline (Figure 10a) and events with cross hatching (slanted vertical lines) refer to API calls (code for which editable source is unavailable, also Figure 10a).

We designed the notations in the visualization to mimic Java syntax. For example, in Figure 9, parentheses () are used to group arguments passed to method calls ("static" refers to a class initialization method, which has no arguments). Curly braces {} group events that occurred within a method call, as in Figure 9; these can be nested if the visualization contains nested method calls (a call stack depth greater than one). The elision • •• in Figure 9 indicates hidden events. Clicking on these reveals the most recent of the elided events.

There are a number of things that do *not* appear in the visualization, with the rationale that any event that corresponds to a single token in the source code was simple enough to understand from the code alone. Therefore, variable references are *not* included, but variable *assignments* are. In general, expressions do not appear in the visualization, but the values *computed* by expressions do. This differs from the Alice Whyline, where all events were included, since its average user was less experienced at programming.

Exploring Answers

The Whyline's answers are more than just static visualizations. In addition to representing an answer to a why question, they are also designed to be temporally organized *bookmarking* tools. As the user navigates the visualization, events are automatically added to the view, allowing the user to accumulate a collection events and code (unlike the Alice Whyline, where *all* events were included by default). This not only allows users to easily revisit important places in the source, but after some exploration, it results in a concise collection of the events that occurred in the program to cause a failure. Therefore, it is an explanation, navigational aid and bookmarking tool rolled into one.

There are many ways to navigate a Whyline answer, all designed to facilitate developers' *mental simulation* of a program's execution [7]. Users can change the event selection by clicking on another event or using the left and right arrow keys to go to previous and next events visible in the visualization. The Whyline also supports common breakpoint debugger commands (but in reverse as well). For example, less-than ‹ and greater-than › 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 add the new event to the visualization and select it; this immediate feedback is intended to reinforce the relationship between the visualization and the code.

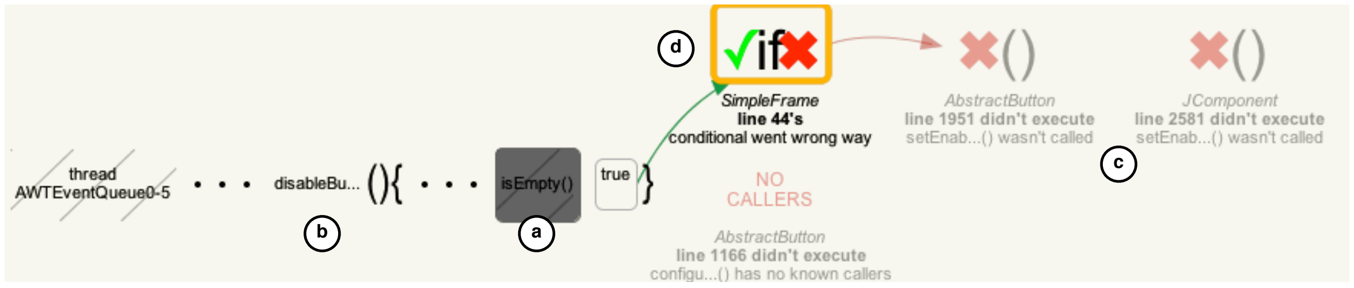


Figure 10. An answer showing (a) a collapsed invocation, (b) a hidden call context, (c) several unexecuted instructions, and (d) a conditional that evaluated in the wrong direction, preventing the desired instruction from executing.

To support the "peeking" behaviors observed in a study of code navigation [10], every action affecting the event selection can be undone with `backspace`, giving users confidence that they will be able to return to their previous location if a navigation is not fruitful. These types of interactions do not change the visualization in any way and can be used to navigate between code and execution events that one has already explored.

A fundamental feature of the Whyline's answers is its *followup questions*. Figures 2b and Figure 8 show followup questions for an event that refers to a reference to a `color` field. The first question, 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 8 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, choosing these questions shows the *origin* of the value mentioned. The origin is where the value is *computed*, skipping over method calls that simply pass the value unmodified. (Of course, one of these calls could be in error, this default behavior is based on the heuristic that most Java bugs occur in data dependencies, such as computing or passing the wrong value, not calling the wrong method). If the user desires to view the skipped calls, they can hold `shift` to see *direct* data dependencies. All followup questions appear in both the source and the timeline (Figure 8).

For "why didn't" answers, the Whyline also includes code that was *not* executed (Figure 10c), but is needed for the output in question to occur. When selected, the Whyline shows the unexecuted code and draws arrows from the code that would have caused the selection to execute. The Whyline also includes events when the answer includes a branch in the wrong direction. For example, in Figure 10d, the Whyline shows that a statement was not executed because the conditional evaluated to true instead of false. As the user clicks on or uses the keyboard to navigate the unexecuted code, the code view above the timeline us automatically updated to show the corresponding code.

Using Familiarity for Filtering and Annotating

Asking questions about a particular primitive output (or some concept related to it) dramatically reduces the amount of information that must be analyzed by the Whyline and the developer. However, we found it necessary to design additional measures to keep question menus and answers a reasonable size. Our most effective idea was to define a *familiarity* measurement and use it to filter and annotate information throughout the Whyline user interface.

The Whyline defines *familiar code* as *any code that is either declared or directly referenced in code declared by the developer*. For example, if the developer defines a new kind of button class that extends the standard Swing button, both the custom class and the Swing class become familiar. The super class of the Swing button would *not* be familiar, however. This measure of familiarity is used throughout the Whyline UI. The Whyline excludes questions about code that is related to the output selected, but unfamiliar. For example, the second level menu of Figure 5 shows several objects related to a line; technically, an object of type `ScrollPaneUI` would also appear in this list, but since it is unfamiliar, the item is excluded.

Familiarity is also used in answers. For example, unfamiliar source files (Figure 7) and executions of unfamiliar code (Figure 10a) are crosshatched, to help users focus on events from code that they wrote or referenced. Also, when showing any event in a visualization, the Whyline collapses events that occurred in unfamiliar methods, effectively "black boxing" API calls and other code for which the developer has no source (Figure 10a). 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 10b). These mechanisms reduce the number of events presented in Whyline answers to those likely to be familiar.

THE WHYLINE VS. BREAKPOINT DEBUGGING

A central assumption underlying the Whyline's design is that asking about *output* first instead of *code* will prevent developers from speculating about the causes of a program failure, therefore saving them time in their investigations. This was true for the Alice Whyline [8], but needed to be tested for the Java version, because of its more sophisticated UI, support for more complex programs, the broadened scope of why questions supported. Therefore, we designed an experiment to compare skilled Java developers using the Whyline to those using conventional tools.

Study Design

The study used a between-subjects design to assess the influence of debugging approach on *completion time* and *task success*. The goal was to determine whether the Whyline would increase success when compared to conventional tools. To increase confidence that any observed differences were due to the Whyline, the control group used a version of a breakpoint debugger that was built using the *identical* Java Whyline UI, but with different debugging tools. This way, participants had identical user interface experiences, except for debugging interfaces. The control group could set breakpoints and step through code, like any other debugger. The downside of ensuring this UI equivalence across groups was that neither condition could edit code (the prototype had no editor or compiler). To make up for this, both groups could insert print statements with no side effects. The Whyline group did not have access to breakpoint features (since we did not focus on which type of support developers would *choose* given both options).

Participants

Both groups had 10 participants, all students in a masters program in software engineering with median of 1.5 years of industry software development experience (ranging from 0-10). (The one developer who reported zero experience had interned on industry projects but did not count it). All rated conventional breakpoint debuggers as "important" to their work or higher on a 5-point scale of "useless" to "essential." The participants also rated themselves with average or higher Java expertise on a 5-point scale of "beginner" to "expert." There were no significant differences in these measures between conditions.

Tasks

We adapted two real bug reports from the `ArgoUML` project, a 150,000 line application for designing Java programs with UML (*http://argouml.tigris.org*). We sought bugs that (1) had checked-in solutions with which to compare participants' solutions, (2) that varied in complexity and difficulty, and (3) that were representative of other bugs in the project. The following bugs were chosen after some initial piloting.

The first (Figure 11) involved removing a checkbox from the UI. The strategy of searching for the label of the checkbox in the code did not work because the application used localized strings for different languages stored in a compressed file on disk. The label *did* appear in the command line help, which did appeared in code, so if one searched with part of the checkbox label, one could make the connection between the two and find the right file. Few made this connection in the actual study.

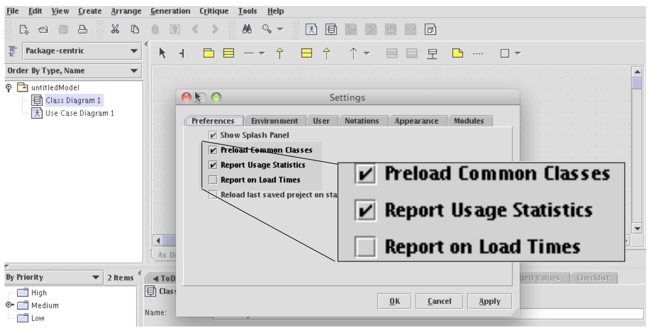
The second bug (Figure 12) involved a drop down list of class names that was supposed to contain all available classes in the project, but was for some reason excluding classes in different packages with identical names. The problem was that the code responsible for aggregating these class names collected the names in an ordered set of unqualifed names. It therefore excluded the second class with the same name. The challenge was to identify the class that was aggregating these names and identify the ordered set's equivalence operator.

Success for each task was evaluated based on the solutions committed to the `ArgoUML` code base. Task 1 had only one valid fix with slight variations in how the layout code was updated. Task two, however, had a larger space of solutions (qualifying the name of the type entities used to construct the list, making the set comparison more sophisticated, among others). In our study, however, the task 2 change recommendations were bimodal in their approach: they were either blind guesses, or reasonably close to the fix submitted for the task 2 bug in the actual `ArgoUML` project.

Procedure

After obtaining consent, participants completed a one page survey about their programming experience. The experimenter then guided the participant through a 10-minute tutorial on features common to both conditions, including the code navigation and call stack tools. The experimenter then trained the participant on features specific to the condition. The Whyline group learned how to ask questions and navigate answers; the control group learned how to set breakpoints, step through code, and insert print statements. After completing the tutorial, the experimenter read the first task description and provided a copy to the participant to follow. All of these materials appear in the appendix of [12].

For each task, participants were told to find the cause of the problem and write a *change recommendation* to a fictional boss. They were also told to emphasize speed over correctness, since the code they were understanding was unfamiliar and their boss would know if their recommendation was on track (this was also to help unify their productivity tradeoffs, so that their task completion times would be more comparable). Participants were allowed to ask for clarification about tutorial content, but other questions were disallowed. Participants were given 30 min. to complete the task; at 10 and 5 min. remaining the experimenter gave time remaining warnings.



☑ Show Splash Panel
☑ Preload Common Classes
☑ Report Usage Statistics
☐ Report on Load Times
Reload last saved project on sta

Figure 11. `ArgoUML` **bug 3121, titled "Remove 'Report Usage Statistics' since it does not do anything."**

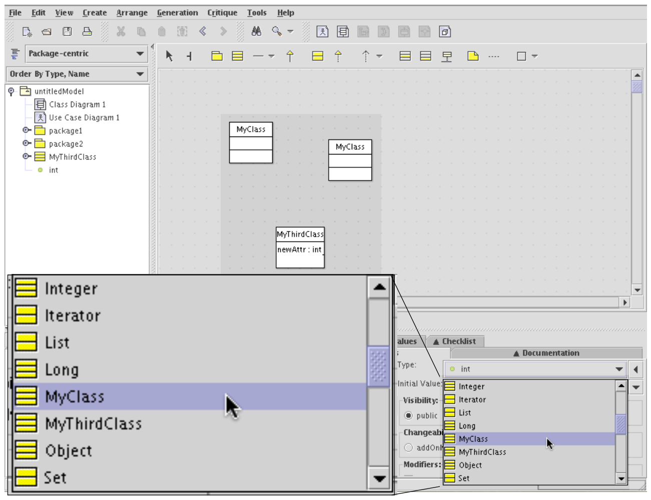


Figure 12. `ArgoUML` **bug 3128, titled "Problems with two classes with the same name in different packages." The name "MyClass" only appears once instead of twice.**

Results

The speed and success results for task 1 are summarized in Figure 13. All 10 Whyline participants completed task 1, compared to only 3 control participants, a statistically significant difference ($\chi 2 = 10.6$, $p < .05$). Whyline participants also completed task 1 twice as fast ($t = 4.5$, $p < 0.05$). As seen in Table 1, this was usually achieved using 1 or 2 "why did" questions, almost exclusively about the creation of the checkbox or the label drawn. The speed and success for task 2 are shown in Figure 14. Whyline participants were more successful ($\chi 2 = 5$, $p < .05$), with 4 of 10 Whyline users succeeding, compared to 0 in the control. Whyline users asked a median of 4 "why did" questions (see Table 1), usually starting on the "MyClass" label, eventually asking about the creation of the list of objects containing the labels (which was a few dependencies away from the bug). Because the task was more difficult, both groups experienced ceiling effects, causing no difference in speed. There was no relationship between industry experience and success for either task (though the sample was probably too small to detect such differences).

It is also informative to consider the information that participants explored. The tools were instrumented to capture data about source file views and navigations with both the keyboard and mouse, allowing us to see what lines of code participants were viewing and for how long. Table 1, for example, lists statistics about the number of files participants viewed per minute and overall, by task and condition. For task 1, Whyline participants viewed significantly *fewer* files per minute than the control group ($t = 22.6$, $df = 18$, $p < 0.0001$), but both groups viewed similar numbers of files overall. For task 2, Whyline participants viewed significantly *more* files per minute than the control group ($t = 2.2$, $df = 18$, $p < .05$). This discrepancy is consistent with the nature of the two tasks: task 1 involved changes to a single file, so viewing fewer files should relate to success; task 2 involved dependencies across many files, so viewing more files should relate to success.

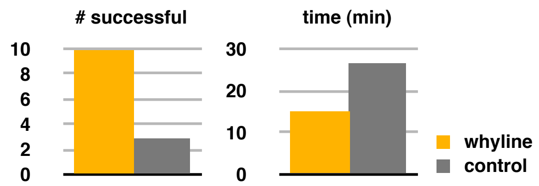


Figure 13. For task 1, the number of successful participants and the time on task.

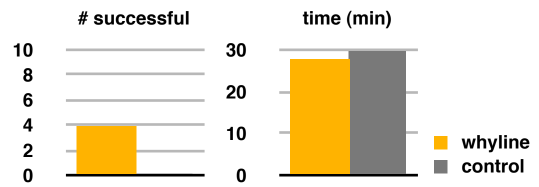


Figure 14. For task 2, the number of successful participants and the time on task.

To assess the *relevance* of the files they viewed, we selected a single function for each task that was key to solving each problem and, for each function visited, computed the distance from the visited function to the key function in the application's *program dependence graph* [2]. (For example, if a method was a single call or variable reference away from the key function, the distance of the method would be 1. The key function itself has a distance of 0). Using this metric, we computed each participant's median distance from the key function for each task. For task 1, Whyline participants were significantly closer to the key function than the control group ($t = 4.6$, $df = 18$, $p < .0002$). For task 2, there was no significant difference in distance (likely due to the low degree of success).

Another telling difference in participants' performance were the UIs used to debug. As seen at the bottom of Table 1, Whyline participants relied mostly on questions, avoiding the more common strategy of text searches for relevant content [9]. The control group, despite using breakpoints, relied more on text searches (which is to be expected [9]) and were far less successful. No participants had usability problems with the breakpoint features, likely due to our extensive 3-month period of user testing prior to the study.

Finally, 8 of the 10 Whyline users offered their opinions on the Whyline unprompted:

I love it!
This is really great!
I think this will really help.
This is really going to reduce the burden on programmers.
This is great, when can I get this for C?
It's so nice and straight and simple...
My god, this is so cool.
This is very nice.

The enthusiasm of participants was clearly evident and all asked to be notified of the tool's availability.

		task 1		task 2	
		whyline	*control*	*whyline*	*control*
# of unique source files viewed per minute	*mean*	1.8	13.3	1	0.6
	σ^2	1.4	0.8	0.5	0.4
range of files viewed		8 – 39	10 – 66	16 – 72	6 – 44
median distance to key function	*mean*	2.2	3.4	3.6	3.3
	σ^2	0.6	0.5	0.5	0.5
# *why did* questions (median, range)		2, 1–4	—	4, 1–8	—
# *why didn't* questions (median, range)		0, 0–0	—	0, 0–2	—
median # debugger steps taken		—	9	—	14.5
median # text searches		0.5	7	1	8

Table 1. Statistics about each condition per task, including files visited per minute and overall, the median distance to the solution, and the tools used to debug.

LIMITATIONS

Our study design had several limitations. Our sample was small and may not be representative of Java users, as many stated their primary expertise was in other languages. The debugger used in our control condition imposed limitations on participants' ability to control the live program and both conditions were disallowed from editing the program. While changing the program and the runtime can be counterproductive in debugging, it is a common strategy [9] and may have led to artificial differences in success. Furthermore, none of the participants were familiar with the `ArgoUML` source; caution should be exercised in applying our results to situations in which a developer is more closely familiar with the code being debugged.

DISCUSSION

The study revealed several important usage patterns for the Whyline UI that inform not only the design of debugging tools, but also the nature of debugging as an activity. For example, 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 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's bookmarking and history features to capture a *summary* of their discoveries about the problem they were investigating.

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, were not as aware of their presence, or just did not need them for the experiment tasks. This also raises the issue of whether participants were even able to find the questions they wanted to ask. We are investigating this as part of the Java Whyline deployment, by allowing users to send feedback about questions they want to ask but cannot find. Informally, it seems that participants were satisfied as long as they found a question that was close enough to the one in their mind, with participants generally starting with questions about surface-level output, and converging on a data structure related to the problematic symptom.

Another interesting trend was that participants treated Whyline answers like they treat the results of a web search: if they saw nothing in the first few events of an 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 (i.e., a call stack); this result, seen early in pilot studies, motivated several changes to the visual presentation and phrasing of followup questions. In general, effective navigation of data dependencies likely depends on developers having some notion of what a data dependency is.

There was variation in the specificity of questions that the Whyline participants used, suggesting that users still need to use caution in which questions they explore. Some chose questions directly relevant to the failure, and as a result, obtained relevant answers. Others chose more generic questions only tangentially related to the failure. They still tended to find the answers, but only with more work. One challenge with choosing the right questions is that users may not find what they are looking for in the question menus. For example, because the questions and even the phrasing of the questions are derived from the program, the degree to which the questions match users' perception of program output depend greatly on the degree to which the *program* matches these perceptions. This is an inherent limitation of the Whyline approach.

The choice of exposing the concept of "dependencies" in the UI seemed to influence participants' confidence in their understanding of causality in the program. 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.

One potential downside of the UI's focus on dependencies is that users may not find important or relevant code serendipitously. Whyline users may be so focused on a particular subset of a system that they lose this coincidental knowledge. Future work should investigate whether such knowledge is obtained in debugging tasks and whether such knowledge later becomes important.

The study also revealed that the Whyline UI could help users *localize* bugs, but often failed to help users understand the exact nature of the bug. For example, task 1 was a relatively obvious fix once found, whereas task 2's cause, if found still took considerable time for participants to understand. This suggests that the Whyline is helpful at "finding the buggy method" but not for explaining "why the method is buggy." This is likely because the Whyline only provides causal explanations of output. It has no special knowledge of the program's intended behavior.

The issue of diagnostic skills is also an interesting area of research to consider. For instance, the Whyline might be a useful way to teach diagnostic strategies, such as that of working backwards from program output and exploring data dependencies. In fact, many of the participants in our study, 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. Participants would also mouse over questions about particular data and say, "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.

CONCLUSIONS

We have described a novel user interface that allows developers to ask "why did" and "why didn't" questions about program output and explore answers with a combined timeline visualization, bookmarking tool and navigational aid. We have shown that the UI helps developers avoid costly speculation about the causes of failures and that this translates into increased success and productivity. While a number of challenges remain in adapting the Whyline approach to other languages, software development platforms, and other types of software failures we are optimistic that it will be influential in the design of future debugging and diagnostic tools in software development.

ACKNOWLEDGEMENTS

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 first author was also supported by NDSEG and NSF Graduate Fellowships. This work is the culmination of the first author's dissertation work and so he would like to thank all of his former advisors, particularly Margaret Burnett, for commitment to fostering undergraduate research.

REFERENCES

1. Abraham R., and Erwig M. 2005. Goal-directed debugging of spreadsheets. *IEEE Visual Languages and Human-Centric Computing*, 37-44.
2. Baowen X., Ju Q., Xiaofang Z., Zhongqiang W., & Lin C. 2005. A brief survey of program slicing, *IEEE Software Engr. Notes*, 30(2), 1-36.
3. Bothell D. 2004. *ACT-R Environment Manual*, Version 5.0, *http://act-r.psy.cmu.edu/software/EnvironmentManual.pdf*.
4. Chalupsky H. and Russ T.A. 2002. WhyNot: Debugging failed queries in large knowledge bases. *Nat'l Conf. on Artificial Intelligence*, 870-877.
5. Clark P., Chaw SY, Barker K, Chaudhri V, Harrison P, Fan J, John B, Porter B, Spaulding A, Thompson J, & Yeh PZ 2007. Capturing and answering questions posed to a knowledge-based system. *Int'l Conf. on Knowledge Capture*, 63-70.
6. Ernst M.D., Czeisler A., Griswold W.G., & Notkin D. 2000. Quickly detecting relevant program invariants. *Int'l Conf. on Soft. Engr. (ICSE)*, 449-458.
7. Gilmore D.J. 1992. Models of debugging, *Acta Psychologica*, 78, 151-173.
8. Ko A.J. & Myers B.A. 2004. Designing the Whyline: a debugging interface for asking questions about program failures. *ACM Conf. on Human Factors in Computing Systems (CHI)*, 151-158.
9. Ko A.J., Myers B.A., Coblenz M. & Aung H.H. 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. on Soft. Engr.*, 32(12), 971-987.
10. Ko A.J., Myers B.A., Chau D.H. 2006. A linguistic analysis of how people describe software problems. *IEEE Visual Languages and Human-Centric Computing*, 127-134.
11. Ko A.J. DeLine R., & Venolia G. 2007. Information needs in collocated software development teams. *Int'l Conf. on Soft. Engr. (ICSE)*, 344-353.
12. Ko A.J. 2008. Asking and answering questions about the causes of software behaviors. *Human-Computer Interaction Institute* CMU-CS-08-122.
13. Ko A.J. & Myers B.A. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. *Int'l Conf. on Soft. Engr (ICSE).*, 301-310.
14. LaToza T.D., Garlan D., Herblseb J.D. & Myers B.A. 2007. Program comprehension as fact finding. *ACM Int'l Symp. on Foundations of Soft. Engr.*, 361-370.
15. Lewis B. 2003. Debugging backwards in time, *Int'l Workshop on Automated Debugging*, 225-235.
16. Myers B.A., Weitzman D., Ko A.J., & Chau D. H. 2006. Answering why and why not questions in user interfaces. *ACM Conf. on Human Factors in Computing Systems (CHI)*, 397-406.
17. Robillard M.P., Coelho W., & Murphy G.C. 2004. How effective developers investigate source code: An exploratory study, *IEEE Trans. on Soft. Engr.*, 30(12), 889-903.
18. Sillito J., Murphy G.C., & De Volder K. 2006. Questions programmers ask during software evolution tasks. *ACM Int'l Symp. on Foundations on Soft. Engr.*, 23-34.
19. Tassey G. 2002. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, RTI Project Number 7007.011.
20. Wagner E. & Lieberman H. 2003. An end-user tool for e-commerce debugging. *Int'l Conf. on Intelligent User Interfaces*, 331-331.
21. Weiser M. 1982. Programmers use slices when debugging, *Communications of the ACM*, 25(7), 446-452.
22. Zeller A. 2002. Isolating cause-effect chains from computer programs. *ACM Int'l Symp. on Foundations of Soft. Engr.*, 1-10.