Source-Level Debugging with the Whyline

Amy Ko and Brad Myers
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213
{ajko, bam}@cs.cmu.edu

ABSTRACT

The visualizations of the Whyline are presented, which focus on supporting the exploration a source code and how it executes. The visualization is concise, simple to navigate, and mimics syntactic features of its target programming language for consistency. Two studies showed that users with the visualization completed a debugging task twice as fast as users without the visualization, partly due to features of the visualization. Applications of the visualizations to tasks other than debugging are discussed.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: User-centered design, interaction styles

General Terms

Human Factors

1. INTRODUCTION

Despite decades of research, debugging remains one of the most challenging and time consuming aspects of software development [9]. One promising solution to this problem is the idea of recording a program's execution and providing a user interface for exploring it. For example, some previous tools allow a developer to step through the recording event by event [6]; others visualize changes to data structures that are used in the program, to help the developer understand the relationship between the source code and its behavior [2]. Many show statistics about aspects of a program's execution, to help a developer isolate performance issues and other global aspects of a program's execution [6].

Although each of these tools is designed for a different task, they all move the developer's attention away from code and instead to *representations* of code. Yet, in practice, code is what developers fix, it what they understand, what they annotate, what they discuss: evidence shows that code is the artifact of interest and representations of code play supporting roles [1].

To address this need, we present a code-centric visualization intended as a means of viewing code, rather than as a replacement for it (Figure 1). Our visualization is compact, supports random-access, combines control and data flow data, and is even aware of its user's familiarity with specific code. We also present evidence of the visualization's effective support of debugging tasks.
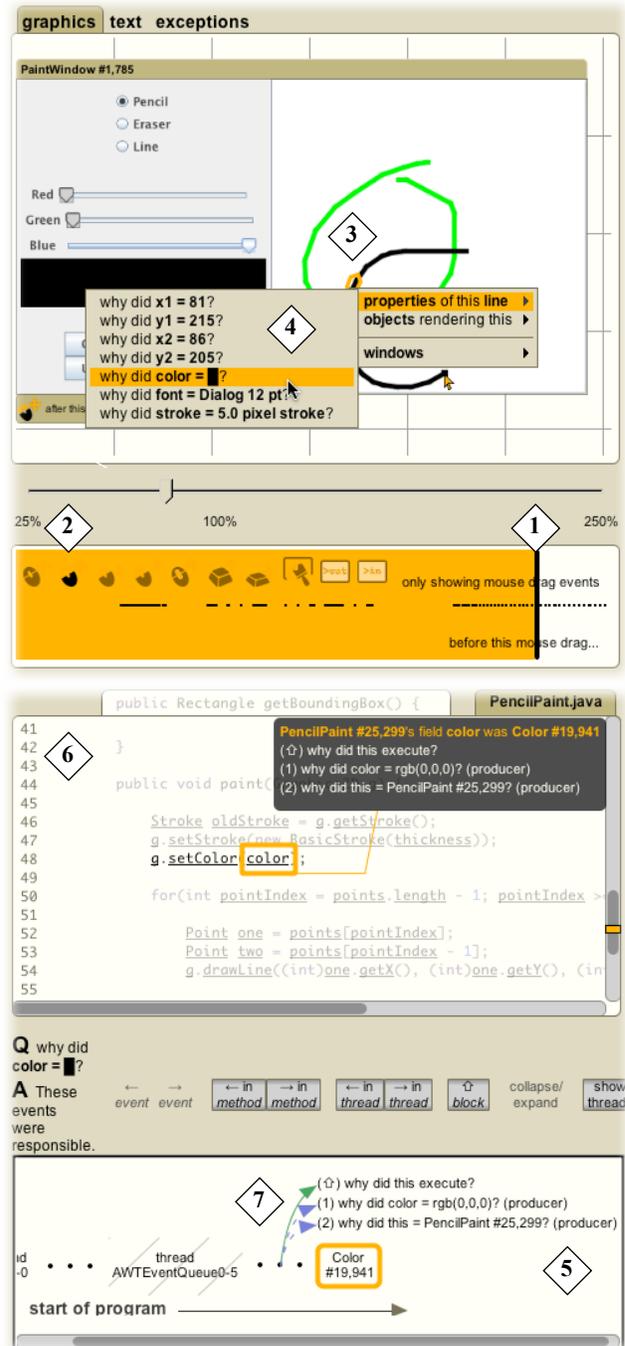
Figure 1. Question (top) and answer (bottom). The numbered areas are described in the text.
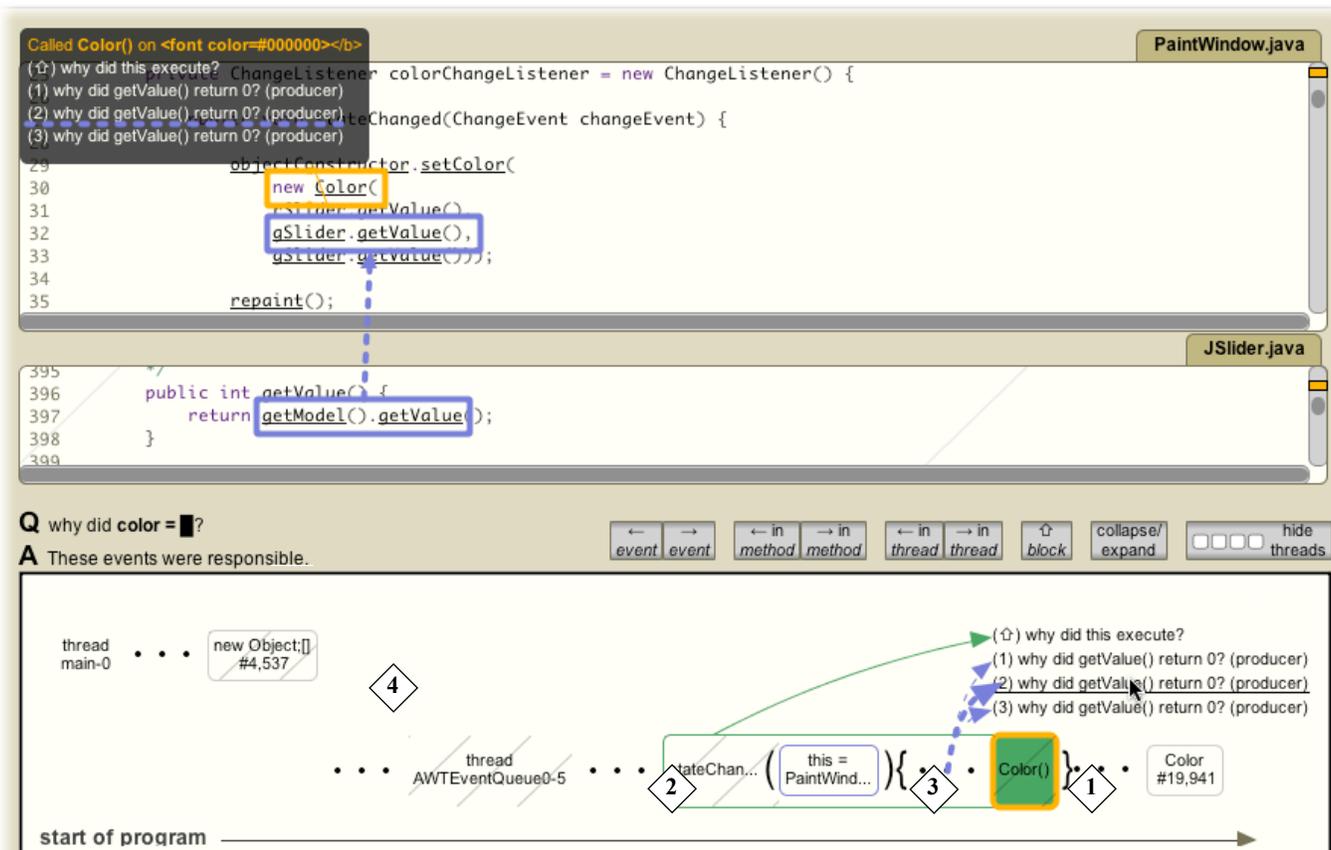
Figure 2. The Whyline's answer to "why did this line's color = ▪?", showing (1) the current selection, (2) the use of Java syntax to segment events, (3) two separate threads of execution, and(4) a call container.

2. EXAMPLE

Our visualization is situated in the context of our work on the Whyline, a debugging tool for asking why and why not questions about program behavior. Our original work [3] supported a simple language, but our current work supports Java programs [4].

Interaction with the tool is simple. The user performs the behavior they want to ask about while the Whyline records a trace. After the program stops, the Whyline loads the recording and presents an I/O history; the user finds the time they want to ask about by dragging a cursor through this I/O history (Figure 1.1), optionally filtering events (such as drag events, as in 1.2) to simplify the selection of certain types. As they do so, the output are reproduced to show what the output looked like at the selected time.

Once the user finds the time they are looking for, they can click on something related to the behavior that they want to ask about (1.3). This shows questions about the selection (1.4); in this example, the user clicks on the line and asks "why did this line's color = ▪?" (in this example, the color should have been blue, but was not). The Whyline then finds the cause of the color and shows a visualization explaining the execution events that caused the stroke to have its color (1.5). When the user selects an event, the Whyline shows the corresponding source file (1.6). In this case, the Whyline selects the most recent event in the answer, which was the color used to paint the stroke. (Technical details about how the Whyline generates and answers questions are available in [4]).

To find out where the color came from, the user selects the label "❶ color" (1.7). This causes the view to select the instantiation (2.1) and its corresponding code (not shown). When the the code for the instantiation appears, the user would likely notice that the *green* slider was used for the blue part of the color.

3. DESIGN

The design choices behind our visualization focus on simplifying tasks such as the example above: a user needs to find some information, and the visualization helps by showing information in multiple forms and in relation to other information. These goals are reflected in both the design of our notation and in its interactivity.

3.1. Notation

We designed our notation to mimic Java syntax. For example, an invocation (such as `stateChanged` in 2.2), occurs just before the `()`'s, which contain method arguments. Everything between the `{}`'s occurred within the method (2.3). The visualization is two-dimensional: the x-axis contains the sequence of execution events and the y-axis indicates the thread in which an event occurred. For example, the two rows at 2.4 show separate threads.

Each event is placed in a container that represents a *control dependency*. For example, the event at 2.1 is contained in the call to `stateChanged` (2.2), which is surrounded by the rectangle in Figure 2. These containers contain all of the event caused by a particular control decision (such as invoking a method or

evaluating a condition). Similarly, an `if` might lead to a series of method calls; the `if`-container would contain all of these calls. It is important to note that we only show a border if an event in a container is selected; if we *always* showed the border, the visualization in Figure 2 would have dozens of rectangles surrounding each container, making it difficult to read.

Generally, the visualization only shows events that could not be determined by reading code. For example, we show invocations and instantiations, but we do not show intermediate values computed in expressions (instead, we just show the results). Our visualization also attempts to omit unfamiliar details. For example, it minimizes events that occurred in unfamiliar code (Figure 2.1)—code for which the user has no source code. It collapses containers of events that occurred in unfamiliar code. These are shown filled, unlike other events, indicating it is a "black box," whose contents are not shown. Finally, if familiar code executed as a result of unfamiliar code (for example, in the case of a call back procedure called by an API), the visualization hides the calling context, but shows the familiar events that occurred as a result. The dashes indicate "hidden."

We explored the possibility of a vertical layout for the event visualization, as most of the visualization contains horizontal text. This worked well in our prototypes, except for programs with multiple threads of execution, which required the second dimension. Collapsing all threads together (which our prototype supports) is also practical.

3.2. Interaction

A major design goal for the design of our visualization is to minimize the amount of interaction necessary to jump from event to event and to provide several ways of navigating between events. To support this, the UI maintains single selection (as in Figure 2.1), which determines what else is shown on screen.

The user can navigate from the selection using a variety of commands. They can click on an event in the history; they can also use the left and right arrow keys to jump between adjacent events in the history. This is useful for seeing the order in which things occurred. The up arrow key jumps back to the most recent control dependency before the selection (for example, at Figure 2.1, pressing up would jump back to the call to `stateChanged`). The down arrow key jumps ahead to the most recent control dependency *after* the selection (for example, at Figure 2.1, pressing down would jump to the next method invocation, the call to `setColor`). The user can also step to the previous or next event in the method or thread. These interactions are similar to the regular debugger commands.

In addition to navigating temporally, the user can also navigate events based on data dependencies. In our example, the visualization started in Figure 1.5 by showing the `Color` value used; this particular use depended on a two prior events: the instantiation of the `Color`, and the object that happened to store the color in its field (obtained by a call to `next()`). Choosing the first dependency (by clicking or typing the number in the circle) allows the user to quickly jump to the instantiation of the color (Figure 2.1), which then shows *its* data dependencies, namely the three `getValue()` calls on the color sliders. The source file shows the same data dependencies as in the visualization, with the same numbers (compare Figure 1.6 and 1.7).

No matter what navigation is performed, the user may always press backspace in order to go back to the previous selection. This allows users to peek ahead at a related event without fear of losing their place.

3.3. Revealing Dependencies

The other views in our environment are closely connected with the visualization. For example, the event highlighted at Figure 1.5 shows the source code for both the event's line of code (1.6), and the code for that line's data and control dependencies. Rather than having the user navigate to these dependencies, the Whyline automatically shows the relevant files, scrolls to the relevant lines of code, and highlights each of them by showing the surrounding context (de-emphasizing the rest with a gradient), and underlining the lines of code involved.

We also use animation throughout these changes to help reify relationships between code and events. For example, as the selection changes and source files rearrange, those files that are common to the new and previous selection stay in place, while unneeded windows shrink away and new windows appear. This helps the user know which files are still relevant, without causing them to lose their focus. Figure 3 shows an example of this animation, in which the file in the window in the upper left animates to take up the whole screen. At the same time, the window also animates scrolling to the new line of code.

On new selections, the Whyline also updates the call stack to show the current state of the thread and the current values of the current method's local variables (Figure 4). Clicking on any of the call stack entries jumps to the particular call in the event visualization; clicking on any of the local variables jumps to the place in the history where the local variable was assigned. To keep the user oriented with the program's execution, the Whyline also updates the state of the output history (Figure 1.3) for every new selection. Therefore, if the user was debugging code about shapes being drawn on screen, they would be able to see the shapes appear while navigating through events.

4. USABILITY EVALUATION

As a preliminary evaluation of our visualization, we performed a usability test with nine users of varying backgrounds (with the least experienced user having never seen a line of code and the most having programmed for more than a decade). We gave each participant a two minute tutorial about how to use the Whyline, including information on how to ask questions and follow data dependencies, and then showed the paint program from the example and the program's blue slider's incorrect behavior. We then asked participants to find the cause of the behavior and tell us when they thought they had found it. As they worked, participants were allowed to ask questions about the user interface, but not about the task or code.

Many participants asked, "what do these numbers mean again?" referring to the data dependencies in Figure 1.4. Users also inquired about the meaning of the various types of notation in the visualization, and in each case, the experimenter asked them to guess. For example, several asked, "what do these curly braces mean?"; their guesses were of the form, "I guess they're like the values passed to the methods," which was correct.

Some participants had trouble understanding why the visualization was showing a particular source file for a given selection (as in Figure 1.6), saying, "wait, what are all these for?" Some spent the time to find out, and usually found the visualizations arrows and labels helpful in establishing the connection between the files. Others would navigate between the events in order to see how the arrangement of source files changed, until they understood the meaning of the changes.

We compared the performance of these participants to that of users from a prior study working on the same task using Eclipse [5]. Whyline users completed the task in a median of 4 minutes,

ranging from 1 to 12, twice as fast as the control group, which required a median of 10 minutes, ranging from 3 to 38 (p < .05, Wilcoxon rank sums). Based on participants' comments, part of this was due to the Whyline's answers and part was due to the ability to explore the history at the granularity of the code.

5. EXPERIMENTAL EVALUATION

In a second evaluation, we designed an experiment to compare the performance of Whyline users against conventional debugger users (in order to make stronger claims, the conventional debugger was simulated using a Whyline trace, ensuring all other tool features were equivalent). Our sample consisted of 10 participants in each group for a total of 20. Participants were all students in a masters program in software engineering, but had a median of 1.5 years of industry software development experience.

Participants worked on two tasks adapted from bug reports of *ArgoUML* a 150,000 open source Java application for designing Java applications themselves. The first bug involved removing a particular checkbox from the user interface. (The typical strategy of search for the label of the checkbox in the source code did not work well in this task because the application used localized strings, but the checkbox label did appear in the command line help). The second bug involved investigating a drop down list of Java types that was supposed to contain all legal types for a field, but was for some reason excluding classes in different packages but with equivalent names. Participants were given 30 minutes to work on each task and told to write a change recommendation to the code's owner for each bug rather than actual modify the code.

All 10 Whyline participants completed task one, compared to only 3 control participants (χ^2 =10.6, p<.05). Whyline participants also completed task one twice as fast (t=4.5, p<0.05). The control participants who did finish the task got lucky in their searches and explored hundreds of files, whereas the Whyline participants only explored a median of three. On task two, 4 Whyline participants were successful, compared to none in the control group (χ^2 =5, p<.05). This task was considerably more difficult; the successful Whyline participants spent all thirty minutes on the task, but much of it was in order to understand some of the Java APIs used in constructing the list for the drop down menu.

Given the difficulty of the two tasks and the sheer size of the application, that anyone was able to solve the tasks, even with the Whyline, is a testament to the effectiveness of the Whyline approach and of the Whyline's user interface. The participants concurred. Some unprompted quotes include "My god, this is so cool." and "When can I get this for C?"

6. DISCUSSION

Our visualization was designed specifically to support debugging and program understanding, but we believe our design ideas can be applied to tools with other goals as well. For example, algorithm animation tools with educational goals [2], which typically only support very small, single file programs, could use our techniques to help learners grasp the execution of programs that span multiple files and tens of thousands of lines. Performance analysis tools, which currently force users to analyze coarse ranges of program execution, could use our techniques to help users specify more precise ranges of times to analyze performance (i.e., after this input event, what took the most time). Our visualization would also improve tools that visualize concurrency events by providing a more concrete level of detail for users to explore. For example, users could check the behaviors of specific threads in specific methods relative to the code and freely navigate between these threads, without having to reason about such threads as abstract entities. Even traditional breakpoint debugging tools could use our visualizations in order to help users monitor and explore what a program has done (even if in a more limited way if they lack a full trace).

Of course, any design makes tradeoffs; ours' emphasizes certain kinds of information over others. For example, because our execution event visualization focuses on low-level code events, and not high level behaviors of a program, it makes it difficult to see such high level interactions, which are visualized by other tools [6]. Such views might be a good starting point for investigating these interactions, and then our visualizations could be used to "drill down" and explore the details.

Another limitation is that our event visualization cannot stand alone: is was designed to be used in conjunction with the other views of source and call stack state. This is largely because the representations of our events do not indicate to what code they refer. This is not the case for other types of visualizations, where the goal is often to aid the user in perceiving visual patterns in pixels, and do not require the user to see the related code until identifying a pattern.

In conclusion, we believe that our visualization not only provides an effective way to help with debugging tasks, but it also provides a flexible visual design that may help users understand other types of data from development tools. In the future, we hope to explore these possibilities in depth.

7. ACKNOWLEDGEMENTS

8. REFERENCES

1. Cherubini, M., Venolia, G., DeLine, R. and Ko. A. J. (2007). Let's Go to the Whiteboard: How and Why Software Developers Draw Code. ACM Conference on Human Factors in Computing Systems (CHI), April 28-May 3, 557-566.

2. Hundhausen, C.D., & Brown, J.L. (2007). What You See Is What You Code: A 'Live' Algorithm Development and Visualization Environment for Novice Learners. *Journal of Visual Languages and Computing* 18(1), pp. 22-47.

3. Ko, A.J. & Myers, B.A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, April 24-29, 151-158.

4. Ko, A.J. & Myers, B.A. (2008). Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. *Submitted for publication*.

5. 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 Transactions on Software Engineering*, 32(12), 971-987.

6. Lewis B. (2003). Debugging backwards in time, *International Workshop on Automated Debugging*, 225-235.

7. Reiss, S.P. (2007). Visual Representations of Executing Programs. *Journal of Visual Languages & Computing*, 18(2), 126-148.

8. Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, RTI Project Number 7007.011, 2002.