Dinah: An Interface to Assist Non-Programmers with Selecting Program Code Causing Graphical Output

Paul Gross[1], Jennifer Yang[2], and Caitlin Kelleher[1]

[1]Washington University in St. Louis
St. Louis, MO, USA
{grosspa, ckelleher}@cse.wustl.edu

[2]University of Washington
Seattle, WA, USA
jyang88@u.washington.edu

ABSTRACT

The web holds an abundance of source code examples with the potential to become learning resources for any end-user. However, for some end-users these examples may be unusable. An example is unusable if a user cannot *select* the code in the example that corresponds to their interests. Research suggests that non-programmers struggle to correctly select the code responsible for interesting output functionality. In this paper we present Dinah: an interface to support non-programmers with selecting code causing graphical output. Dinah assists non-programmers by providing concurrency support and in-context affordances for statement replay and temporally based navigation.

Author Keywords

Non-programmer, end-user, search, navigation, selection, localization, Dinah, Looking Glass, Storytelling Alice

ACM Classification Keywords

H.5.2. Information interfaces and presentation: Graphical User Interfaces.

General Terms

Design, Experimentation, Human Factors.

INTRODUCTION

A wealth of freely available code resources exists on the web. These resources range from code snippets in API documentation to whole programs in code repositories. Some repositories exist exclusively for end-user environments (e.g., CoScripter [9]). In many domains, end-users attempt to learn from or reuse code from these code resources [2,3,13]. To effectively use examples from these resources, users must be able to *select* the code in an example that relates to their interests [4]. Research suggests non-programmers struggle to select the related code either unaided or with existing software support [4,5].

Software tools that enable non-programmers to select code from programs with graphical output may help non-programmers learn more effectively from code examples they find on the web (e.g., code reuse [6]). We have chosen to focus on graphical output from a program because our observations of inexperienced end-users indicate that they define their programming goals in terms of observable output rather than implementation details. Further, graphical output provides an approachable means by which non-programmers can determine whether a program is relevant to their needs.

In this paper we present Dinah: an interface which assists non-programmers in selecting the code causing graphical output. We first present Dinah's interface with an example usage scenario. We then discuss three guidelines for future code selection systems' design drawn from our formative studies. We conclude with the limitations to our approach.

CHI 2011, May 7–12, 2011, Vancouver, BC, Canada.

RELATED WORK

In software engineering, output localization [1] is concerned with correlating output to the code responsible, whether for a feature [14] or a fault [7]. Localization software support can use a combination of static (e.g., source code artifacts [11]) and dynamic information (e.g., execution traces [8]) to create assistive visualizations. Some tools use dynamic traces to create interactive graphical output timelines that enable indexing of active code sections at a point in time [4,8,10,12]. The Whyline [8] enables debugging by asking why and why not questions about program execution from recorded graphical output. ZStep95 [10] enables stepping of graphical output changes. The majority of these tools have been designed for experienced users and not focused on non-programmers.

Research shows that non-programmers struggle to find the code responsible for graphical output either alone [5] or with a debugger [4]. An output history tool [4, 6] enabling bi-directional search of a program's output and code significantly increased non-programmers' success relative to a debugger [4]. However, users took nine minutes on average to identify target code while struggling with unfamiliar code constructs and concurrent execution [4]. Dinah offers new supports to assist non-programmers with evaluating constructs and searching concurrent code.

LOOKING GLASS

We built Dinah into Looking Glass [6]: an IDE for creating interactive 3D animated stories. Looking Glass uses drag-and-drop based program construction to prevent users from

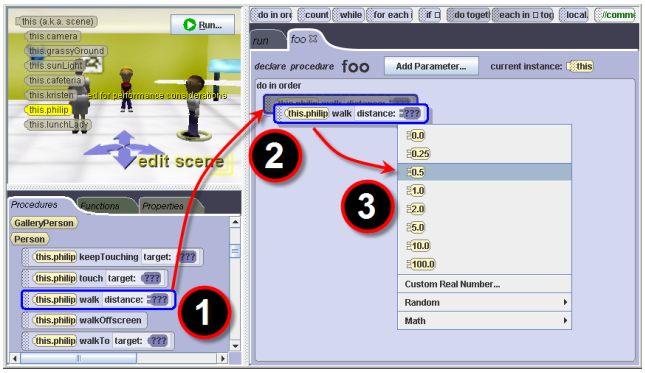

Figure 1. Looking Glass IDE where a user programs by (1) dragging a method, (2) dropping it into the code pane, and (3) selecting parameters.

making syntax errors (Figure 1). The environment supports many common programming constructs including methods, conditionals, and loops. Parallel execution is supported and frequently used in Looking Glass user programs [6].

A DINAH USAGE SCENARIO

To illustrate Dinah (Figure 2), we present a scenario where a non-programmer selects code causing graphical output. We present two solutions based on search strategies employed by roughly half of each of our non-programmer participants during our formative evaluation study (see Formative Evaluation): bottom-up to select the code for the output's start and top-down to select the code for the end.

Scenario Problem

Imagine Sam, a non-programmer who wants to create a story where a boy shakes a gift to find out what is inside. Sam finds a program on the web where a lunch lady "brainwashes" a student by raising her arms to his head and then shaking it. Sam believes this example is relevant because he can use the same brainwashing action to make his boy shake the gift. Sam now wants to select the code for the arms raising and the shaking to use it in his program.

Bottom-Up Solution for the Start

When Sam runs the brainwash program, Dinah appears over the code. Sam pauses the program (Figure 2-1) when the lunch lady begins raising her arms.

Sam looks at the Right Now pane (Figure 2-2) to see the actions (i.e., methods) the lunch lady is doing. He notices three actions: a *delay* and two *touch* actions. Sam clicks on a *touch* action to open a statement context menu (Figure 2-5), and replays the action. Replay shows the lunch lady lifting her right arm, not both arms. Because this is part of what Sam wants, Sam clicks on the action again to open the context menu and locate the action in the program code.

Now Sam is looking at the code for the *brainwash* method. Sam notices another *touch* action below the *touch* action he located. He clicks on its Statement Button (Figure 2-4) for replay which shows the lunch lady lifting only her left arm.

Sam wants both *touch* actions, but he does not know how to replay them both at the same time. He clicks on the code button for a *touch* action and chooses the HELP operation (Figure 2-5). In the Help panel (Figure 2-6) Sam sees a tab for what played at the same time. Sam chooses the section referring to a block playing the *touch* action, and reads the description of a *Do Together* block. Sam clicks on the

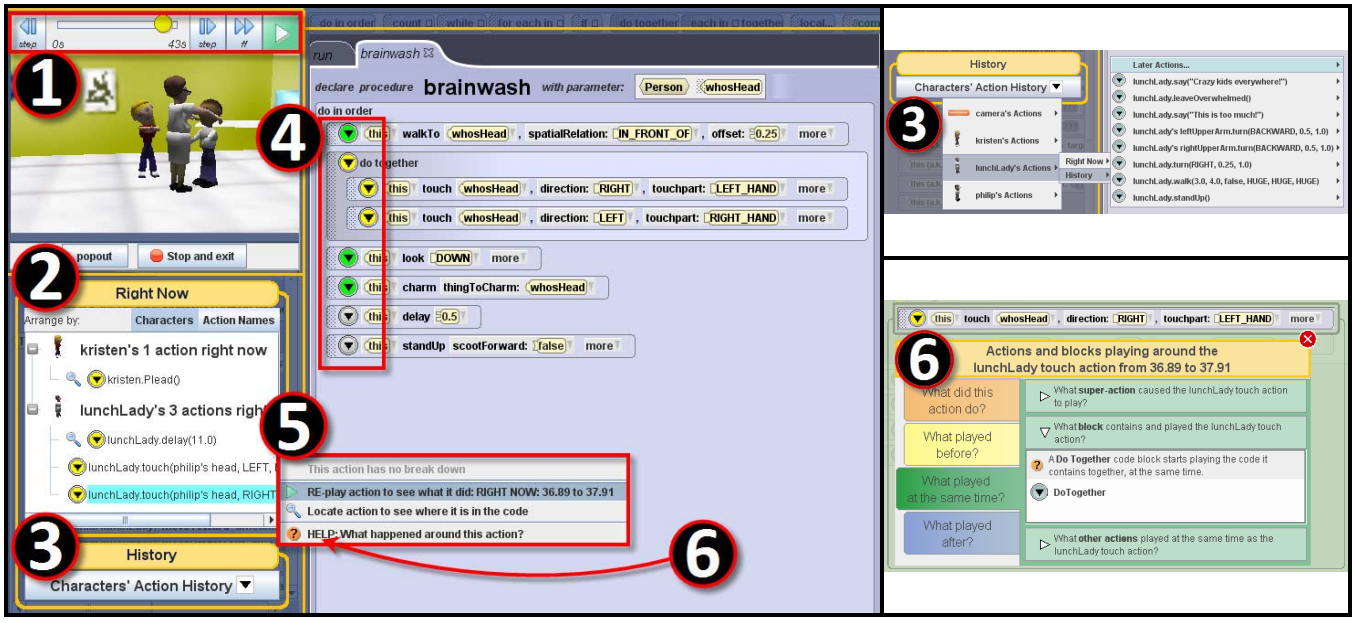

Figure 2. Dinah's interface: (1) program playback controls, (2) Right Now pane showing currently executing methods by character (i.e., object) or action name (i.e., method name), (3) History Pane indexing all executed methods by a target object, (4) Statement Buttons indicating what is executing (yellow), has executed (green) and has not executed (gray), (5) Statement Context Menus to *breakdown* a super-action (i.e., show method implementation), *replay* a statement, *locate* a statement (useful from Right Now, History, and Help views), and (6) *help* to show the Help pane which explains execution semantics temporally around the statement.

statement button in the help panel and replays the block to see what it did. This shows the lunch lady raising both arms. He uses the code button to locate the *Do Together* block and now has the start of the code he wants.

Top-Down Solution for the End

Sam resumes the execution of the program and pauses once the lunch lady begins shaking the student's head.

Sam looks in the program code for yellow Statement Buttons (Figure 2-4) to see what statements are currently executing. Sam notices a *brainwash* action first and replays the action. The replay begins too early with the lunch lady walking toward the student before shaking his head. Sam decides the code he is looking for is in the *brainwash* action. He clicks on the *brainwash* code button and chooses breakdown (Figure 2-5) to see the actions inside (i.e., the implementation of the *brainwash* method).

Inside the *brainwash* action Sam sees the yellow code button only on the *charm* action. Sam replays the *charm* action and sees only the head shaking, not any other actions that also occurred concurrently. Convinced, Sam decides he has found the end of the code he wants.

FORMATIVE EVALUATION

Twenty-six non-programmers (university students and staff) participated in our formative evaluation. Users first completed two tutorials explaining basic software operation and an example selection task. We provided documentation to eighteen users. Finally we asked users to watch a video showing graphical program output and asked participants to mark the first and last statements causing the output.

Future Selection Tool Guidelines

In the following sections we present three guidelines drawn from our formative evaluations and describe how they are supported in Dinah.

Use alternative views to indicate concurrency happened and enable independent thread replay to tease it apart.

Research suggests that non-programmers incorrectly attribute output to concurrently executing code (i.e. "magic code") [5]. Prior guidelines propose independent thread replay to overcome this problem [4]. We support reasoning about concurrency in two ways: 1) we provide a *replay* operation to help users isolate the output for a selected statement and 2) we provide navigational affordances to help users identify concurrent methods.

Dinah enables users to *replay* any execution of a statement. Replay shows a statement's output effect independent of any other concurrent statements' output. Dinah implements replay by storing all graphical state changes and their source execution thread. When a statement is replayed, Dinah shows graphical state changes from the statement's execution thread, and any child threads for concurrent blocks, over the statement's execution period. Users often utilized the replay operation (Figure 2-5) during their search process. One user described its value: "you can see exactly what each command means."

We provide two navigational affordances to help users identify concurrent methods. The *Right Now* pane (Figure 2-2) shows all methods executing in the running program or at a selected point in time, organized either by 3D object or by method names. One user described the *Right Now* pane as: "kind of representative of all the code that's going on, all at once." For concurrency identification in the program code, when a user has identified a particular statement that executes near the same time as their target output, they often search for ways to navigate based on that statement. *Statement context menus* include an item: *HELP: What happened around this action?* (Figure 2-5). This operation opens the *Help* pane (Figure 2-6) which includes tabs to help users find methods that executed *before*, at the *same time*, and *after* the selected statement. Users often begin the search for a concurrent method by looking for statements that executed "right before" or "next after" a statement. If users fail to find their statement in the before or after tabs, they often look at the methods occurring at the same time. In the *same time* tab, the *other actions* section shows all methods that executed concurrently, organized by 3D object, during the selected statement's execution period.

To implement these features Dinah maintains a dynamic trace of the running program. The trace organizes statement executions into a hierarchical execution tree (i.e., a block is an internal node and a method executing in the block is a child). Parent nodes sort children by execution period for easy execution time search and temporal neighbor location.

Represent execution flow directly in the program code with three states.

Previous research suggests non-programmers naturally focus on reading code to search a program [5]. To accommodate this focus, research also suggests providing direct code interactions, such as thread independent replay, to precisely evaluate statements [4]. To this end, we built a *replay* operation (Figure 2-5) accessible through a button displayed on each program statement (Figure 2-4). In the initial design, all these buttons had the same appearance, leading users to struggle to determine which statements they could replay. As one user stated: "I was assuming I could just play anything, like whatever anytime I want."

To address this struggle, we experimented with variable button colors based on execution status. All statement buttons began as red. As a program ran, the buttons for in-progress and completed statements became green. Although a user correctly interpreted the colors, that user could not identify why replay was disabled on red buttons. In the final design there are three color states (Figure 2-4). Statement buttons start gray. While a statement is running, the button is yellow. When the statement completes, the button is green. One user summarized this as "I think it's kind of showing how much has been completed." In subsequent

user tests, we observed that the three-color status helped users to understand which statements were replayable.

Provide intuitive program navigation affordances, or views and operations to avoid navigating the call hierarchy.
A study of non-programmers naturally searching and selecting code suggested that non-programmers do not infer program structure, and consequently fail to fully navigate programs [5]. We addressed this by enabling users to directly *locate* statements from program output. Additionally, we provide two navigation operations, *breakdown* and *HELP* (Figure 2-5), because correct selection may require navigating to find parent method calls or blocks.

The *locate* operation (Figure 2-5) avoids navigational difficulties by directly navigating to, and highlighting, a statement in the program code. The *Right Now* (Figure 2-2), *History* (Figure 2-3), and *Help* (Figure 2-6) panes all summarize methods that are executing or have executed in panes outside the program code. When a user chooses a method execution from any of these panes, the locate operation becomes available to show the corresponding statement in the program code. This enables users to find the statement without navigating the call hierarchy. We observed many users incorporating *locate* in their searches.

The *breakdown* operation (Figure 2-5) navigates down the call hierarchy by showing a method's implementation. We originally labeled this operation *show details in tab* because previous users described an implementation as an "action's details." However, users mentioned a desire to find a way "to breakdown this task into the sub-tasks." These users dismissed *show details in tab* as unrelated to their goals. We incorporated this intuitive explanation and later a user expressed that they used *breakdown* very early in the session because "it made sense at the time."

The *Help* pane (Figure 2-6) offers time-based contextual navigation for a selected statement and the ability to navigate up the call hierarchy. We commonly observed users using the *locate* operation and then getting stuck. Moving up the call hierarchy or considering a parent block was unintuitive. As one user stated, "it's hard to think of things happening on top of things... you think things happen sequentially." To support reasoning in the presence of parental relationships, the Help pane presents temporal execution information in three tabs: what executed *before*, at the *same time*, and *after* the selected statement. The *same time* tab includes sections for the parent method (i.e., *super-action* as many users referred to its implementation as the "sub-tasks" or "sub-code") and one for the parent construct block. The super-action section explains the super-action execution relationship and enables a user to move up the call hierarchy by using the locate operation. The parent block section similarly explains the block's execution semantics (e.g., a loop) and offers the locate operation.

LIMITATIONS AND FUTURE WORK

Dinah's features are limited to applications that can be visualized graphically. Dinah's approach and affordances should scale to any other graphical environments with appropriate execution time management mechanisms (e.g., graphical change stepping [10]), detection of code sections related to graphical change, and efficient trace storage.

ACKNOWLEDGMENTS

The NSF funded this work through grant #0835438.

REFERENCES

1. Biggerstaff, T.J., Mitbander, B.G., and Webster, D. The concept assignment problem in program understanding. In *Proc. ICSE,* IEEE (1993), 482-498.
2. Brandt, J., Guo, P., Lewenstein, J., Dontcheva, M., and Klemmer, S. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. CHI*, ACM (2009), 1589-1598.
3. Dorn, B. and Guzdial, M. Graphic designers who program as informal computer science learners. In *Proc. ICER*, ACM (2006), 127-134.
4. Gross, P. and Kelleher, C. Toward transforming freely available source code into usable learning materials for end-users. In *Proc. PLATEAU*, ACM (2010), In Press.
5. Gross, P. and Kelleher, C. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *JVLC 21*, 5 (2010), 263-276.
6. Gross, P., Herstand, M., Hodges, J., and Kelleher, C. A code reuse interface for non-programmer middle school students. In *Proc. IUI*, ACM (2010), 219-228.
7. Jones, J.A., Harrold, M.J., and Stasko, J. Visualization of test information to assist fault localization. In *Proc. ICSE,* ACM (2002), 467-477.
8. Ko, A.J. and Myers, B.A. Extracting and answering why and why not questions about Java program output. *ACM Trans. on Soft. Eng. and Met. 20*, 2 (2010), 1-36.
9. Leshed, G., Haber, E., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proc. CHI*, (2008), 1719-1728.
10. Lieberman, H. and Fry, C. ZStep 95: A Reversible, Animated, Source Code Stepper. In J. Stasko, ed., *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.
11. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., and Sergeyev, A. Static techniques for concept location in object-oriented code. In *Proc. IWPC,* (2005), 33-42.
12. Oney, S. and Myers, B. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Proc. VL/HCC,* IEEE (2009), 105-108.
13. Rosson, M.B., Ballin, J., and Rode, J. Who, What, and How: A Survey of Informal and Professional Web Developers. In *Proc. VL/HCC,* IEEE (2005), 199-206.
14. Wilde, N. and Scully, M.C. Software reconnaissance: Mapping program features to code. *J. of Soft. Maint.: Research and Practice 7*, 1 (1995), 49-62.