An Investigation of Non-Programmers' Performance with Tools to Support Output Localization

Paul Gross and Caitlin Kelleher
Dept. of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO, USA
{grosspa, ckelleher}@cse.wustl.edu

Jennifer Yang
Computer Science and Engineering
University of Washington Seattle
Seattle, WA
jyang88@u.washington.edu

*Abstract***— The wealth of code available through the web has the potential to dramatically change the way we learn to program. This includes inexperienced programmers, who may struggle to find code in example programs that relate to observable program features. We present a comparative study of three tools for assisting non-programmers with finding program code corresponding to a program's graphical output. From this study we also identify a model which captures the goals inherent in non-programmers' code search processes for this type of search task. Our results suggest a global pause marker may be an effective tool to support non-programmers' search.**

Keywords-component; Code Search, Non-programmer, Localization, Strategy, Looking Glass, End-User programming

I. INTRODUCTION

The wealth of code available through the web has the potential to dramatically change the ways we learn to program and the ways we construct new programs. Already, end-user programmers rely on example source code they find on the web to learn new skills and construct new programs [1, 2, 12]. While experienced end-user programmers can make use of found source code, non-programmers may struggle to locate the code responsible for observable functionality of interest [5]. Tools that can help these users to easily identify the code responsible for target functionality may help transform freely available code into usable learning materials [4].

In this paper, we describe the results of a study comparing the performance of non-programmers attempting to find code causing graphical output in the Looking Glass IDE [6]. Participants completed tasks assisted by either one of three code search support tools [4, 6] or unassisted. We analyze participants' common usage patterns to identify an underlying strategy model common to all conditions. This model may help to guide the design of future tools to support code search.

II. RELATED WORK

Many tools that support output localization utilize run-time information to create visualizations for users. ZStep95 [9] enables stepping of recorded graphical output changes while highlighting the executed code. The WhyLine [8] answers users' *why* and *why not* questions about a program's recorded execution behavior. FireCrystal [10] highlights the code that executed in a webpage when the user exhibited an interactive behavior. Two tools evaluated in this paper, Hastings [4] and Dinah [6], use recorded program output to help non-programmers localize code within animated stories.

Novice output localization strategy research encompasses investigations into novice strategies for finding either features or faults in localization tasks. Katz [7] identified two general novice debugging strategies: forward reasoning (i.e., step-by-step in code first), and backward reasoning (i.e., searching from incorrect output into code). Gross [5] noted non-programmers employed a similar bi-directional search. Romero [11] observed a novice strategy based on stepping execution and while checking a visualization. Fern [3] found end-user problem-solving strategies by using data mining techniques to identify frequent user interaction sequences. We employ a similar method to identify strategies by computing the frequency of interaction sequences that support user goals.

III. METHODS

We conducted a between-subjects study of non-programmers performance on output localization tasks either without code search support or using one of three different code search tools (see Fig. 1 and Instruments for descriptions). Forty-nine adults (university students and staff) participated in the study. None had prior programming experience.

A. Instruments

To ensure environment consistency in our study, we added statement markers, and implemented three code search tools in Looking Glass. Looking Glass (see Fig. 2) is a novice programming environment that includes support for common constructs (e.g., loops, methods, etc.) and parallel execution.

We added first and last *statement markers* in Looking Glass to enable participants to mark the lines or blocks of code that they believed corresponded to target output functionality. Users could drag-and-drop the markers onto any statement in Looking Glass' editing mode. When the user ran the program, the markers served as global breakpoints by pausing the entire program. The program paused before the statement marked first executed and after the statement marked last executed. We chose to include markers because we noticed in pilot studies that users in all conditions spent a long time trying to determine

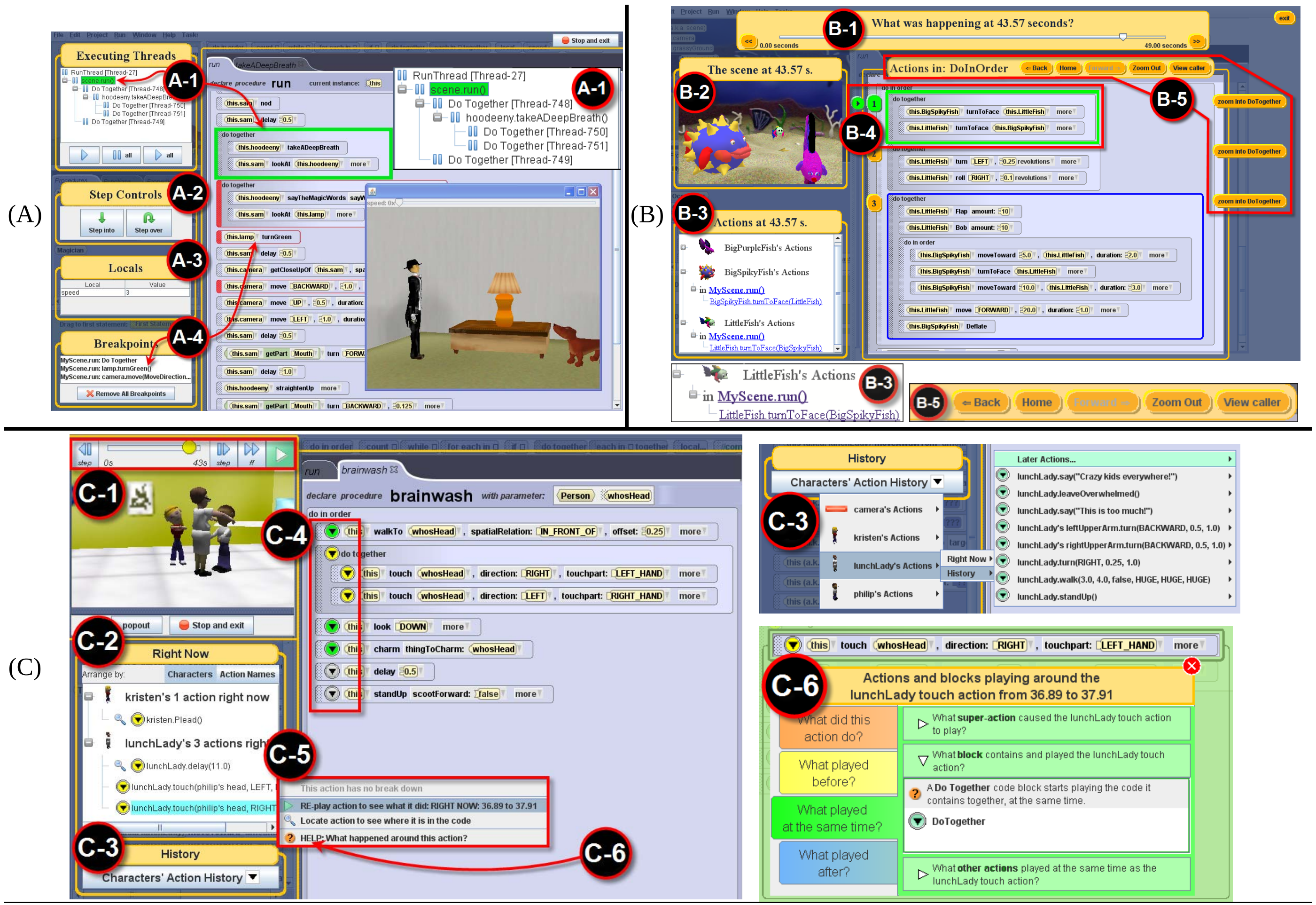

Figure 1. Interfaces for the Debugger **(A)**, Hastings **(B)** and Dinah **(C)** output localization support tools.
(A) The Debugger interface includes an **(A-1)** Executing threads pane, **(A-3)** a list of in-scope local variables, and **(A-4)** a list of current breakpoints. Users can click on a paused thread frame in A-1 to highlight its active line of code in green. **(A-2)** The Step Controls allow users to step into or over an active line of code.
(B) The Hastings interface has a **(B-1)** Time Slider for scrubbing through the recorded program time, **(B-2)** a view of the scene, and **(B-3)** a list of the actions characters performed at the selected time. Hastings annotates executing line(s) of code with **(B-4)** controls to replay or **(B-5)** navigate in the syntax tree.
(C) The Dinah interface has **(C-1)** program playback controls, **(C-2)** a Right Now pane showing that executed in the selected time, **(C-3)** action history, **(C-4)** Statement Buttons that indicate methods are executing (yellow), completed (green), or not started (gray), **(C-5)** menus that enable users to *breakdown* a statement (i.e., show its implementation), *replay* a statement, *locate* a statement, and **(C-6)** *help* users to find concurrent executing code not visible in the current code view

the correctness of a solution in the presence of interactions between threads. These markers enabled users in all conditions to more easily compare their selected code to the target code.

We implemented three code search tools: a Debugger, Hastings, and Dinah (see Fig. 1), to explore how these different

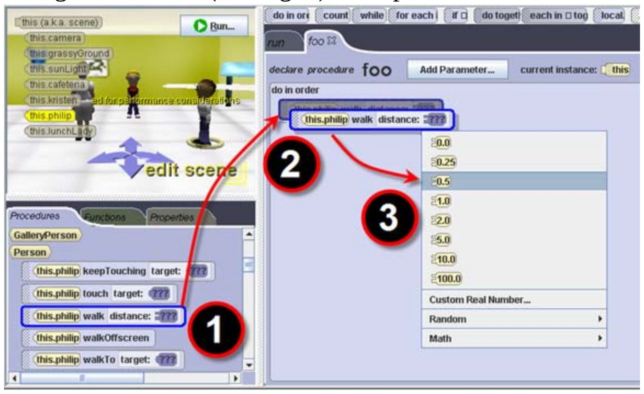

Figure 2. Looking Glass IDE where a user programs by (1) dragging a method, (2) dropping it into the code pane, and (3) selecting arguments.

tools can support non-programmers in finding code responsible for graphical output. We included a debugger because debuggers are the most commonly available tool to localize features in an unfamiliar program. Hastings (previously the Output History Explorer Tool [4]) supports non-programmers' natural search strategies to overcome search barriers [5]. Dinah [6] addresses Debugger and Hastings usage barriers [4] by utilizing non-programmers language for describing programs.

To ensure balance in our tasks, we created three programs that vary along dimensions identified in previous research [5] to adequately approximate Alice programs created by novice programmers. We created thirteen total tasks in these programs.

B. Study Sessions

The study took place in single, two-hour long sessions. We randomly assigned participants to one of four conditions: Control (markers only), Debugger, Hastings, or Dinah. Users completed a survey and then an in-software tutorial for basic Looking Glass skills. Users next followed instructions for an example task that explained both how to drag-and-drop statement markers and their effect on program execution. Then, users in the tool groups viewed an overview of their tool that focused purely on presenting tool features to explicitly avoid suggesting strategies. Finally, all participants completed three training tasks and then as many evaluation tasks as possible.

C. Study Tasks

Each study session included two types of tasks: training tasks and evaluation tasks. We presented tasks with a short video of the target output to find. In each video a red box highlighted the target objects and actions. By presenting tasks through videos, rather than verbal descriptions, we avoided linguistic cues that could bias participants' search strategies.

For each task, participants watched the target output video and attempted to find the code responsible for the output using their assigned tool, if any. Users marked possibly responsible code by dragging first and last markers onto statements. The markers' global breakpoint effect provided the only correctness feedback; we provided no other feedback.

1) Training Tasks: We asked participants to complete three training tasks to verify they understood the task completion process, had necessary skills (e.g., marking code in different contexts), and were familiar with their tool (if any).

2) Evaluation Tasks: After completing the training tasks, participants completed as many code selection tasks as possible within the two hour session. To minimize potential learning effects, we randomized the order of all tasks, ensuring the same source program was not used in consecutive tasks.

D. Data

We collected a demographics and computer history survey, video recordings of participants' task attempts, and logs of tool users' interface actions. After each answer submission, we asked participants to describe how they came to their solution. When the session ended, we asked participants how they would instruct a friend to find a line of code. This encouraged participants to verbalize their code search strategies.

IV. RESULTS

We exclude two of our 49 participants who failed to understand our tasks during training. We present our results in two parts: user task performance and task search processes.

A. User Task Performance

To detect instruction time differences, for 44 users (three outliers removed, one control and two Debugger based on box plots), we performed a one way ANOVA which revealed a significant instruction time effect [F(3,40) = 12.13; *p < .001*]. A post-hoc Tukey HSD test showed significant differences between Control users and Hastings and Dinah users (*p* < *.05*), with a marginally significant difference between Control and Debugger users (*p=.06*). We detected no significant differences in instruction time between the treatment groups. This is not surprising since participants in code search tools conditions needed to review more material. A one-way ANOVA found no significant differences in the amount of time needed to complete the practice tasks [F(3,41)=.78; *p=.512*].

TABLE I. TASK RESULTS FOR FIRST 17:23[a] OF TASK TIME

Group	**n**	**Avg. (SD) Tasks Attempted**	**Avg. (SD) Tasks Correct**
Control	10	2.60 (1.43)	1.90 (1.45)
Debugger	9	1.33 (0.84)	0.67 (0.70)
Hastings	12	1.67 (0.78)	0.83 (0.72)
Dinah	12	2.00 (1.35)	1.75 (1.14)

a. 17:23 is the minimum time a non-outlier user had to complete tasks

Because this significant difference in training time enabled control users to attempt more tasks in a session, we consider 43 participants performance (one Control outlier removed based on box plot) on evaluation tasks only for the shortest time (17 min., 23 sec.) that a participant had to attempt these tasks.

To assess the performances between our four groups, we performed a one way MANOVA on the number of tasks completed (regardless of correctness) and the number of correct tasks completed during the comparison period (see Table I). We found a significant main effect for group [Wilks' λ=.67; F(6,76)=2.81, *p<.05*]. Using post-hoc contrasts, we found a significant difference between the code search performance of participants in the Control and Dinah groups and participants in the Debugger and Hastings groups [F (2,38) =0.331; *p<.01*]. We found no significant performance differences between Control and Dinah participants [F(2,38)=.09; *p*=.17] or Debugger and Hastings participants [F(2,38)=0.01; *p*=.73].

B. Task Search Processes

After each session we asked participants to describe, step-by-step, how they would instruct a friend to complete tasks like the study tasks. We anecdotally noted similar strategy points across all conditions. The similarities amongst descriptions led us to hypothesize that non-programmers progress through a sequential series of goals when attempting a code search task:

1. Isolate when the target output functionality occurred.
2. Identify candidate method calls at the selected time.
3. Locate candidate method calls in the program code.
4. Evaluate whether the located code is responsible for the target functionality.
5. If necessary, search related (e.g., nearby) code.

To test for an underlying search process driven by the hypothesized goals, we investigated how tool participants' tool usage aligned with the goals. In Table II we outline a mapping of specific tool interactions that correspond to our hypothesized goals. We briefly summarize each goal and give an example of its tool support below.

1) Identify Temporal Location: Users begin code search by attempting to identify the point in time when the target graphical output occurred. For instance, Hastings and Dinah users can scrub through a program's recorded execution.

2) Evaluate Temporal Location Execution Information: Once the user has isolated a point in time, the user can view information describing the program's execution state, external to the program code, at that point in time. For example, Hastings users can view a characters' actions at a point in time by expanding the character in the current actions pane.

3) Identifying Corresponding Code Location: After identifying a promising action to explore, users next identify

TABLE II. TOOL FEATURES SUPPORTING AND INDICATING HYPOTHESIZED NON-PROGRAMMERS'SEARCH GOALS

Supported and Indicated Search Goal	**Features and Feature Sequences Supporting and Indicating Search Goal (A > B indicates use of feature A followed by use of feature B)**			
	Debugger	***Hastings***	***Dinah***	***Control/Default for All***
1. Identify Temporal Location	* Run Prog. > Pause All Threads * Resume All > Pause All Threads	* Time Scrub	* Run Prog. > Pause * Resume > Pause * Time Scrub	* Reading Prog. Code * Using statement markers as global breakpoints
2. Evaluate Temporal Location Execution Info.	* Select Thread	* Expanding Current Character Actions	* Click Action in Right Now	
3. Identify Corresponding Code Loc.		* Click Current Action	* Locate Action	
4. Evaluate Code Location	* Step Over * Resume Thread	* Replay	* Click Code Button > Replay	
5. Explore Code Location Context	* Step Into * Set Breakpoint > Play All Threads or Restart Prog.	* Navigation Controls * Click on Stmt. or Stmt. Index	* Click Code Button > Breakdown * Help	* Reading Prog. Code * Edit method to see implementation

where that action occurs within the code. This location is what the user will mark if the action relates to the desired output.

4) Evaluate Code Location: Once a user located a candidate action, the user needs to determine whether it is responsible for the target output effect. For example, Debugger users can use step over or resume a single thread to execute a statement and observe its output effect.

5) Explore Code Location: If the located action is not responsible for the target functionality, the target is often in a contextually related location (e.g., inside a method or block statement) which the user explores. For instance, Dinah users can breakdown statements to view their implementation. Additionally, Dinah's help operation can aid users in identifying temporally and spatially related code.

In Fig. 3 we show the transitions between the hypothesized user search goals as evidenced by tool usage patterns. Percentages on transitions indicate the frequency an interaction sequence supporting a goal was followed by an interaction sequence supporting another goal. We computed these frequencies by mining user interaction log data for the tool feature sequences shown in Table II. For clarity, we removed edges between the states which account for fewer than 10% of transitions. We note that the most common sequence through the goals states, ignoring cycles, directly corresponds with our hypothesized goal sequence.

V. CONCLUSION AND FUTURE WORK

Regardless of condition, non-programmers in our study used a common process for code search. This process is suggested by participants' verbalizations and validated through participants' use of tools. It is notable that none of the tools we evaluated performed significantly better than a global pause marker. This finding suggests two possibilities:

1. Much of the difficulty in code search is evaluating whether the correct code is selected.
2. Having multiple tasks in the same program may have created a memory effect that was more pronounced among control users due to an increased need for program comprehension.

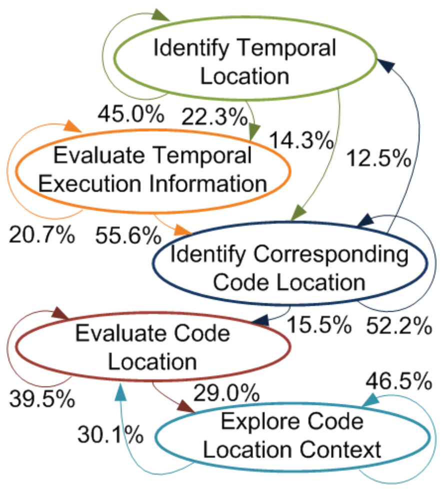
Identify Temporal Location
45.0%
22.3%
14.3%
12.5%
Evaluate Temporal Execution Information
20.7%
55.6%
Identify Corresponding Code Location
15.5%
52.2%
Evaluate Code Location
39.5%
29.0%
46.5%
30.1%
Explore Code Location Context

Figure 3. Flow diagram between user search goals. Percentages represent the frequency one goal follows another in users' tool interactions.

We note this study took place in a short time period and, due to the study structure, some groups had more task time.

Additional research is necessary to determine whether our code search model generalizes to longer usage times and other programming domains. However, we believe this model may be a useful design aid for future code search tools.

REFERENCES

[1] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," *Proc. of CHI*, 2009, pp. 1589-1598.

[2] B. Dorn and M. Guzdial, "Graphic designers who program as informal computer science learners," *Proc. of ICER*, 2006, pp. 127-134.

[3] X. Fern, C. Komireddy, V. Grigoreanu, and M. Burnett, "Mining problem-solving strategies from HCI data," *ACM TOCHI*, vol. 17, Apr. 2010, pp. 3:1–3:22.

[4] P. Gross and C. Kelleher, "Toward Transforming Freely Available Source Code into Usable Learning Materials for End-Users," *Proc. of PLATEAU*, ACM, 2010.

[5] P. Gross and C. Kelleher, "Non-programmers identifying functionality in unfamiliar code: strategies and barriers," *JVLC*, v. 21, Dec. 2010, pp. 263-276.

[6] P. Gross, J. Yang, and C. Kelleher, "Dinah: an interface to assist non-programmers with selecting program code causing graphical output," *Proc. of CHI*, 2011, pp. 3397–3400.

[7] I.R. Katz and J.R. Anderson, "Debugging: an analysis of bug-location strategies," *Hum.-Comput. Interact.*, vol. 3, 1987, pp. 351-399.

[8] A.J. Ko and B.A. Myers, "Extracting and answering why and why not questions about Java program output," *ACM TOSEM*, vol. 20, Sep. 2010, pp. 4:1–4:36.

[9] H. Lieberman and C. Fry, "ZStep 95: A Reversible, Animated, Source Code Stepper," *Software Visualization: Programming as a Multimedia Experience*, J. Stasko, Ed., MIT Press, 1997.

[10] S. Oney and B. Myers, "FireCrystal: Understanding interactive behaviors in dynamic web pages," *Proc. of VL/HCC*, 2009, pp. 105-108.

[11] P. Romero, B. du Boulay, R. Cox, R. Lutz, and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *Int. J. of Human-Comp. Stud.*, vol. 65, Dec. 2007, pp. 992-1009.

[12] M.B. Rosson, J. Ballin, and J. Rode, "Who, What, and How: A Survey of Informal and Professional Web Developers," *Proc. of VL/HCC*, 2005, pp.199-206.