Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code

Joel Brandt[1,2], Vignan Pattamatta[1], William Choi[1], Ben Hsieh[1], Scott R. Klemmer[1]

[1]Stanford University HCI Group
Computer Science Department
Stanford, CA 94305

[2]Advanced Technology Labs
Adobe Systems
San Francisco, CA 94103

{jbrandt, vignan, wchoi, srk}@cs.stanford.edu, bhsieh@stanford.edu

ABSTRACT

Instructive example code is a central part of programming. Web search enables programmers to quickly *locate* relevant examples. However, existing code editors offer little support for helping users *interactively explore* examples. This paper proposes that effective use of examples hinges on the programmer's ability to quickly identify a small number of relevant lines interleaved among a larger body of boilerplate code. This insight is manifest in *Rehearse*, a code editing environment with two unique features: First, Rehearse links program execution to source code by highlighting each line of code as it is executed. This enables programmers to quickly determine which lines of code are involved in producing a particular interaction. Second, after a programmer has found a single line applicable to her task, Rehearse automatically identifies other lines that are also likely to be relevant. In a controlled experiment, participants using visualization and highlighting adapted example code significantly faster than those using an identical editor without these features.

Author Keywords

Example-centric programming

ACM Classification Keywords

H5.2. Information interfaces and presentation: User Interfaces—*prototyping*.

INTRODUCTION

"I didn't know I needed *that* line!" exclaimed one participant in our need-finding study, as she re-examining a block of example code. She wasn't alone—all participants had difficulty adapting examples because they made mistakes in determining precisely which lines were relevant to their task. Instructive examples have long played a central role in programming practice [2, 13], and Web search tools help programmers to *locate* high-quality examples [1]. Despite examples' pervasiveness, current mainstream editing environments offer little specialized support for *understanding and adapting* examples. What interactions might assist programmers in using examples more effectively?

Previous research suggests that programmers prefer examples that are complete, executable applications [2, 13]. Examples in this form show relevant code in context, providing information about how it should be used. The downside of complete examples is that they necessarily contain a large amount of irrelevant "boilerplate" code with relevant lines interleaved throughout. The main insight presented in this paper is that *effective use of examples hinges on the programmer's ability to quickly identify a small number of relevant lines interleaved among a larger body of boilerplate code.*

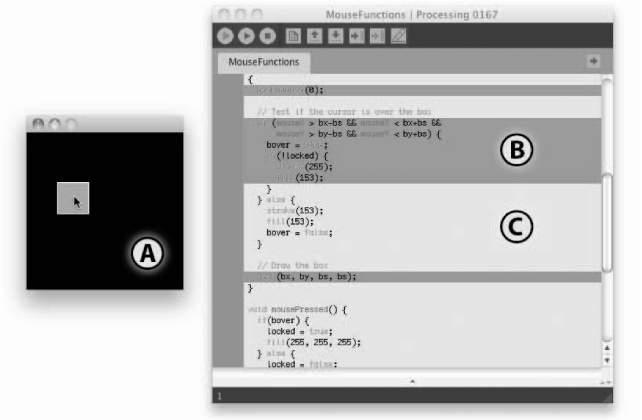

Figure 1. The Rehearse development environment, visualizing the execution of an example application. The user interacts with the running application (A). Lines that have recently executed are highlighted in dark green (B). As execution progresses, lines executed less recently fade to light green (C).

To explore this insight, we built *Rehearse*, which is an extension of the open source Processing development environment [3]. Processing uses a variant of the Java programming language and is completely interoperable with standard Java libraries. Rehearse enables two interactions not available in Processing. First, Rehearse links program execution to source code by highlighting each line of code as it is executed (see Figure 1). This enables programmers to quickly determine which lines of code are involved in producing a particular interaction. Second, after a programmer has found a single line applicable to her task, Rehearse automatically identifies other lines that are also likely to be related (see Figure 2).

We compared Rehearse to the unmodified Processing environment in the lab with 12 participants. We found that by

using these interactions participants were able to adapt example code significantly faster.

RELATED WORK

Rehearse builds on a large body of work on code authoring and debugging tools. Many programming-by-demonstration (PBD) tools provide a visual link between source code and execution at runtime. For example, Koala [10] and Vegemite [9], two PBD tools for the Web, highlight lines of script before they execute and highlight the effect on the output as they execute. Similar visualizations are often provided in visual languages like d.tools [6] and Looking Glass (the successor to Storytelling Alice) [5]. In all of these systems, only a few "lines" of the user's code need to execute per second for the user's application to be performant. In contrast, with general-purpose languages like Java, the user's code often must execute at thousands of statements per second. One contribution of Reherase is extending this visualization technique to code that must execute much more rapidly.

An alternative to Reherase's realtime visualization is to record an execution history that can be browsed and filtered after execution completes. FireCrystal, for example, uses this technique to aid programmers in understanding and debugging JavaScript [12]. There are benefits and tradeoffs associated with both approaches. Offline browsing of execution history affords the programmer more time to explore an issue in-depth, but it necessarily requires an extra step of locating the portion of the execution trace that is relevant. The Whyline system offers an effective approach for browsing and filtering these execution traces [8]. Whyline allows users to ask "why" and "why not" questions about program output, which are used to automatically filter the execution trace for relevant data. We suggest that Reherase's realtime visualization allows users to ask similar questions implicitly. Simply by interacting with their running application, they are implicitly asking "what lines of code are responsible for creating this interaction?"

Rehearse also provides support for identifying lines of code that a *related to* a particular line of interest. This interaction was inspired by CodeTrail's insight that linking source code and documentation is beneficial [4]. CodeTrail links the Eclipse development environment and the Firefox Web browser to give users faster access to documentation and other resources. Rehearse builds on this to optimize one specific interaction with documentation: finding related API functions. Rehearse gives up the relatively general-purpose nature of CodeTrail in order to make one task highly efficient.

NEED-FINDING: OBSERVING EXAMPLE ADAPTATION

To inform the design of Rehearse, we observed five individuals in the lab as they searched for, evaluated, and adapted example code. Five university students participated in an hour-long unpaid lab study. All the participants had previous experience with Java; only one was familiar with Processing.

We first asked participants to follow a standard tutorial on Processing's Web site. Participants were then asked to perform two tasks: The first was to create an analog clock with numbers. We provided participants with two example applications: an analog clock without numbers and an application that drew text on a canvas. The second task was more open-ended. Participants were asked to create a custom paintbrush tool of their choice. We seeded them with ideas, such as "spray paintbrush" and "soft hair paintbrush." Participants were provided with a broad example database, including a few with functionality that was directly relevant to the task (such as mouse press and mouse drag).

In addition to the provided examples, participants were free to use any online resources. We encouraged participants to think aloud by asking open-ended questions as they worked.

Observations

Participants routinely executed examples before inspecting the source code. For example, one participant opened an example and immediately stated, "I'm going to run this and figure out what it does." We believe that this initial execution allowed participants to form a mental model of how the source code *should* be structured, which guided their subsequent inspection of the code itself.

We found that when participants read source code, they were very good at identifying a single "seed" line relevant to their task. For example, they could rapidly identify the line of code that actually drew text to the canvas because it contained a string literal. However, it took them much longer to identify related lines, such as those that loaded and selected a font or set the drawing position. Often, they would fail to indentify some relevant lines, which would lead to confusing bugs. In the provided example on drawing text, the line that set the font was in a setup function far away from the line that actually drew text. As a result, several participants did not see this line, and mistakenly assumed that there was a default font.

After participants found a potential "seed" line, they would frequently make small modification to that line and then re-execute the application. This modification was largely epistemic [7]: It wasn't in support of the eventual adaptation they needed to make to achieve their goal. Instead, it served as a way to confirm that they were on the right path. We hypothesized that by providing a more efficient way to confirm that particular lines of code were linked to desired output behavior, we could increase the utility of this epistemic action.

REHEARSE

Rehearse extends the Processing development environment [3] with two interactions designed to support understanding and adapting example code.

Execution Highlighting

During execution of the user's program, Rehearse highlights each line of code as it is executed (Figure 1). The line currently executing is highlighted in dark green. As execution progresses, the highlighting slowly fades to light green, which gives the programmer an overview of which lines

have executed most recently. Execution highlighting can be enabled or disabled using a toggle button in the toolbar.

Execution highlighting directly links what is happening in the program's output to the code responsible for that output. This link allows the programmer to use the running application as a query mechanism: to find code relevant to a particular interaction, the programmer simply performs that interaction. Because the visualization is produced in real-time, this makes it easy to answer questions such as "is the MouseDrag handler called only at the start of a mouse drag event, or continuously throughout the drag?"

Execution highlighting can help programmers find *some* of the lines of code that are relevant to their task. For example, it can help a programmer locate the line of code that draws text to the screen. It may not, however, help them find related, but infrequently executed lines of code such as those required for setup. When using execution highlighting alone, a programmer could easily miss an important line of code that, for example, loads a font.

Related Lines

Using Rehearse, the programmer can press a hotkey to identify lines of code that are likely related to the one she is currently editing (Figure 2). Related lines are demarcated by an orange highlight in the left margin of the editor. To determine which lines are related to the current line, the system examines all invocations of API methods on that line. The system then highlights any line that invokes a related method, as determined by a pre-computed mapping.

Implementation

The execution highlighting feature of Rehearse was implemented by adding a custom Java interpreter to Processing. Our interpreter is based heavily on BeanShell [11], which was modified to support the Processing language, and to provide tracing of execution. When execution highlighting is enabled, the user's code is executed in the interpreter. Calls to external methods (for example, those in the Processing API) are still executed as compiled bytecode inside Java's Virtual Machine, and the link between the interpreted and compiled code is handled through Java's reflection mechanism. This hybrid execution approach is crucial to achieving acceptable performance for most applications. It allows, for example, resource intensive API method calls to execute as fast as possible.

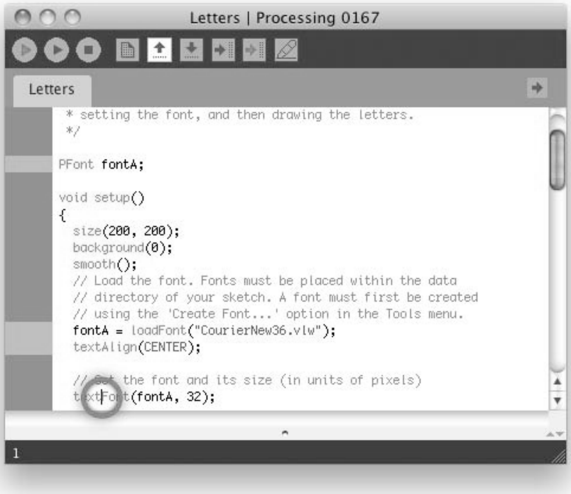

Figure 2. Rehearse indicating lines related to the line currently being edited. The user's cursor is circled in green; related lines are identified by orange highlights in the left margin. Display of related lines is triggered by a hotkey.

Determination of related lines is handled through a pre-computed mapping that specifies what API methods are related to each other. This mapping is taken directly from the Processing documentation; Java's API documentation provides a similar "related methods" paradigm.

EVALUATION: REHEARSE IN THE LAB

We hypothesized that Rehearse would help users understand and adapt example code more quickly because it would reduce the cost of identifying which lines are relevant to their task. To test this hypothesis, we ran a comparative lab study.

Method

We recruited 12 university affiliates for a 45-minute, unpaid study. We required all participants to have proficiency in Java (at least equivalent to what is taught in the first year of a typical undergraduate CS curriculum). No participants in our study had familiarity with Processing.

Participants were randomly assigned to a control or treatment condition. Control users were provided with the current Processing IDE and treatment users were provided with Rehearse. All participants completed a tutorial on Processing adapted from Processing's Web site, and treatment participants were introduced to Rehearse's features through this tutorial. We then provided participants with written instructions asking them to complete two tasks. For each task, we measured task completion time and recorded qualitative observations.

In the first task, participants started with an application that drew a rectangle on the screen each time the user pressed a key. The height of the rectangle varied by letter case: lower-case letters created rectangles half as tall as upper-case letters. Participants were asked to modify the height of rectangles created by lower-case letters. Completing this task required modifying one or two lines in an 89-line program, so participants were expected to spend the majority of their time identifying those lines.

In the second task was identical to the task used in our need-finding exercise: Participants were asked to add numbers to a provided analog clock application. Completing this task required integrating two existing applications, which necessitated writing or modifying approximately 10 lines of code in a 100-line application.

Results

In task 1, Rehearse uses completed the task significantly faster than the control group ($p < 0.03$, 1-sided t-test). Control participants completed the task in 18.3 minutes on average; Rehearse uses spent 12.6 minutes on average, a 31% speed-up (see Table 1). One participant in the treatment group chose not to complete the task, and is not included in these statistics.

In task 2, Rehearse users completed the task faster than the control group—17.4 vs. 22.2 minutes, a 22% speed-up—but this difference was not statistically significant ($p \approx 0.15$). One participant in the treatment group chose not to complete the task, and is not included in these statistics.

Discussion

The execution highlighting feature appeared to have the biggest impact on participants' performance. This was most evident in Task 1, where the bulk of the task consisted of understanding *where* in the code to make a very simple change. One participant said, "First, I tried to hack around the example code to get it to work. When that did not work, I used execution highlighting to actually understand the code."

The related lines feature appeared useful for those participants who actually used it. Only 3 of the 6 participants in the treatment group did so, and these participants only used it on the second task. While it is not appropriate to draw conclusions from such a small sample, it is interesting to note that three of the four fastest participants on Task 2 were those who used the related lines feature. Additionally, the second fastest control participant on Task 2 used the "related methods" portion of the Processing documentation, which provides the same information in a less efficient manner.

The fact that the related lines feature was used infrequently suggests that it was not discoverable. We also believe that, as it is currently implemented, making use of this feature requires some skill at identifying when it *might* be useful. That is, the programmer has to have the foresight to predict that there may be related lines that she is not aware of. Improving this feature remains important future work.

CONCLUSION

Rehearse allows programmers to use examples more efficiently. The interactions supported by Rehearse stem from the insight that effective use of examples hinges on the programmer's ability to quickly identify a small number of relevant lines interleaved among a larger body of boilerplate code. Execution highlighting and automatic identification of related lines make it easier for programmers to focus their attention, leading to faster code understanding.

	Task 1		Task 2	
	T	**C**	**T**	**C**
1	14	22	18	18
2	15	23	21	16
3	—	20	16	23
4	7	14	17	16
5	13	21	15	41
6	14	10	—	19
Average	*12.6*	*18.3*	*17.4*	*22.2*

Table 1. Task completion times for treatment (**T**) and control (**C**) participants. Participants using Rehearse completed the first task significantly faster than those in the control condition ($p < 0.03$).

REFERENCES

1 Brandt, J., M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-Centric Programming: Integrating Web Search into the Development Environment. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2010.

2 Brandt, J., P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. pp. 1589-98, 2009.

3 Fry, B. and C. Reas, *Processing*. http://processing.org

4 Goldman, M. and R. C. Miller. Codetrail: Connecting Source Code and Web Resources. In *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. pp. 65-72, 2008.

5 Gross, P. A., M. S. Herstand, J. W. Hodges, and C. L. Kelleher. A Code Reuse Interface for Non-Programmer Middle School Students. In *Proceedings of IUI: International Conference on Intelligent User Interfaces*. pp. 219-28, 2010.

6 Hartmann, B., S. R. Klemmer, *et al.* Reflective Physical Prototyping through Integrated Design, Test, and Analysis. In *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. pp. 299-308, 2006.

7 Kirsh, D. and P. Maglio. On Distinguishing Epistemic from Pragmatic Action. *Cognitive Science* **18**(4). pp. 513-49, 1994.

8 Ko, A. J. and B. A. Myers. Finding Causes of Program Output with the Java Whyline. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. pp. 1569-78, 2009.

9 Lin, J., J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-User Programming of Mashups with Vegemite. In *Proceedings of IUI: International Conference on Intelligent User Interfaces*. pp. 97-106, 2009.

10 Little, G., T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. pp. 943-46, 2007.

11 Niemeyer, P., *BeanShell*. http://www.beanshell.org

12 Oney, S. and B. Myers. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. pp. 105-08, 2009.

13 Rosson, M. B. and J. M. Carroll. The Reuse of Uses in Smalltalk Programming. *TOCHI: ACM Transactions on Human-Compter Interaction* **3**(3). pp. 219-53, 1996.