Toward Transforming Freely Available Source Code into Usable Learning Materials for End-Users

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

ABSTRACT

The availability of example source code on the web presents an array of potential learning resources for any code consumer. However not all code consumers may find these resources usable. With end-user programmers increasingly relying on example code on the web, any difficulty can prevent these code resources from reaching their potential as learning materials for users who may see the greatest benefits: inexperienced end-users. In this paper, we discuss freely available source code's usability for end-users. We focus on one problem area: supporting inexperienced end-users in selecting relevant code sections from examples they find interesting. We discuss a user study to evaluate the adequacy of two tools that can support non-programmers in this code selection task, and highlight design guidelines for future tools. Finally, we identify further challenges in transforming example code into usable learning materials for all end-users.

Categories and Subject Descriptors

H.5.2 [**Information interfaces and presentation**]: Graphical User Interfaces

General Terms

Design, Human Factors, Languages.

Keywords

End-user, Non-programmer, Code Usability, Execution Explorer, Code Reuse, Looking Glass

1. INTRODUCTION

Transforming the ever-increasing wealth of source code on the web into useful materials for consumers of any experience level should demand the attention of software engineers. End-user programmers are increasingly relying on source code examples found on the web [11, 14, 26]. Unfortunately, the majority of these code examples may be unusable to precisely the users who could benefit the most from them: inexperienced end-users. The ultimate goal of research in this area should be to enable programmers of any experience level to take any code example and effectively use it to learn new skills, and reduce implementation time.

Freely available source code is an abundant resource on the web. Any user can find source code snippets in tutorials, in API documentation and in the source of web pages themselves. More code can be found in existing repositories for both open source code (e.g., [2, 4, 5]) and some end-user environments (e.g., [21, 22, 7]). Research suggests that end-user programmers are using these example code resources as learning aids or customizable examples [10, 14, 26].

There is great potential for end-user programmers of all experience levels to extend their programming skills with these resources. However, these code resources will not meet their full potential if they are unusable. To make effective use of available code, a user needs to be able to *evaluate* an example's relevance to their interests and *select* the code in the example which corresponds to those interests. Tools that enable efficient evaluation and selection of relevant code may increase the effectiveness of the estimated 22 million end-user programmers in the workplace [29] and the growing communities of recreational end-user programmers (e.g., mashups [33], stories and games [22], image tool scripting [14]) who are without formal training and teach themselves new programming skills. While this is an important problem at all skill levels, inexperienced end-user programmers may see the greatest benefits.

Many problems exist in transforming code examples into usable learning materials and software support for end-users may help tackle many issues. Our current focus is on helping inexperienced end-users find and select code in complete, executable programs that corresponds to desired program features. To this end, we are exploring tools that can map graphical program output to the executing program code.

In this paper, we focus on the challenges of designing software solutions to help transform the freely available source code on the web into usable learning materials for non-programmers. We first discuss our definition of usable learning materials relative to inexperienced end-users. We then explore one space of software support for helping inexperienced end-users select code corresponding to desirable program features. We present a user study evaluating the adequacy of two software tools for providing this support. Based on the study results we present design guidelines for future tools. Finally we present other major challenges in making arbitrary code examples usable.

PLATEAU October 18, Reno, NV USA

2. EXAMPLE CODE USABILITY

Although many code resources exist on the web, some are not usable examples for end-users of all experience levels. We consider a code resource to be a usable example if a user can *evaluate* the example's relevance to their interests and *select* the code in the example which corresponds to their interest. We are particularly interested in inexperienced end-users, and discuss the difficulty of these processes for inexperienced end-users in the following sections.

2.1 Evaluate Example Relevance

For an example to be usable, a user must first be able to *evaluate* whether the example contains any functionality relevant to his or her interests. An end-user's ability to determine relevance may be limited by the availability of affordances indicating the example's purpose, and his or her ability to read and understand program code. However, for some tasks, an end-user could evaluate an example's relevance by executing the example's code and observing it's graphical output.

Affordances indicating an example's purpose, such as a description of an example's functionality from API documentation or comments in an example program, may help a user determine an example's relevance. However, we cannot assume resources will provide explanations or any other annotations assisting example evaluation.

A user's ability to evaluate an example's relevance is partially dependent on his or her programming experience. For shorter examples, an experienced programmer could evaluate an example's relevance by simply reading the code, without needing to execute it. In contrast, an inexperienced end-user may lack the skills required to use an experienced programmer's evaluation process. An end-user may need to execute the example and observe the example's output to evaluate its relevance.

Our observations of end-users indicate that they define their goals in terms of observable output products, such as graphical changes, rather than programming constructs or ideas. End-users can then evaluate an example's relevance by determining if an example displays any similarities to their goal product. For instance, suppose an end-user wants to enlarge an image on their website when a visitor clicks on the image. The end-user may be able to learn from another website which shows an enlarged image in a frame on top of the page when a user clicks on an image.

2.2 Select Relevant Example Code Sections

A relevant example is usable once a user can *select* the code sections relevant to their interest. This requires a user to read and navigate the code in search of relevant code sections. With little or no experience programming, an end-user may be unable to find and select relevant code sections [15]. This identifies a need for tools that help end-users efficiently make correct connections between interesting output and its corresponding code.

Research suggests that non-programmers struggle to correctly attribute given graphical output to the program code responsible for the output, completing only 41% of selection tasks correctly [15]. Participants encountered three major barriers: (1) mapping their descriptions of output actions to lines or sections of code, (2) fully navigating all relevant program code, and (3) recognizing language constructs and their effects on execution flow.

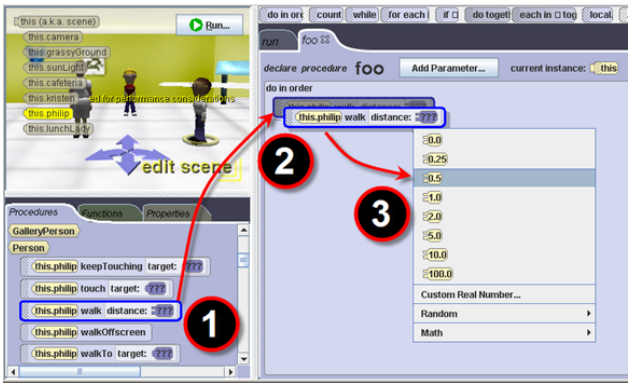

Figure 1: Looking Glass where a user programs by (1) dragging a method, (2) dropping it into the code pane, and (3) selecting parameters.

To overcome these barriers, we are exploring tools to help inexperienced end-users with selection tasks. If we assume a user relies on graphical output to determine an example's relevance, then tools that connect program output to executing code may provide adequate selection support. We refer to this class of tools as *execution explorer* tools. We explore the adequacy of these tools in later sections.

2.2.1 Related Work

Output Localization (e.g., feature localization [32], fault localization [16]) is a well studied area of software engineering. The goal of localization is to find code responsible for some functionality [9, 12]. Techniques to support localization can use both static (e.g., source code artifacts [23]) and dynamic (e.g., execution traces [19]) information to provide software visualizations [16]. Recent work uses program output to help localize responsible code in web pages [25] and user interfaces [20].

Some fault localization work focuses on end-user debugging. The Whyline [19] enables users to pose "why" and "why not" questions about program behavior and receive answers from runtime behavior. The Whyline is designed for debugging expected output which relies on knowledge of a program's construction. Non-programmers exploring unfamiliar code would lack this knowledge. WYSIWYT [27] is a spreadsheet program that visualizes the "testedness" of a given cell to help locate possible fault points. Further work identifies challenges in end-user debugging support [28].

3. EXECUTION EXPLORER TOOLS

To transform arbitrary code examples into usable learning materials for users of all experience levels, support may be necessary for inexperienced end-users to select relevant code sections from an example program. *Execution explorer* tools, or tools that map graphical program output to concurrently executing code, may provide this support.

Multiple examples of execution explorer tools exist (e.g., [20, 25]). The most common is a debugger. Using a debugger's breakpoint and step features, a user can set a breakpoint at the beginning of an example, step into new code sections and step over each line of code to watch for output changes as each line of code executes. These features

Figure 2: The Looking Glass Debugger Interface. (1) Executing threads pane for viewing and selecting threads, (2) Step Controls for stepping execution, (3) Local Values lists locals in scope, (4) Breakpoints Panel lists breakpoints (statements outlined in red).

can support finding code responsible for interesting graphical output. Although novices' struggles with debuggers for debugging tasks are known [24], the selection task is simpler and a debugger may be adequate.

We set out to discover how well execution explorer tools support non-programmers in the selection process. We investigated the adequacy of two execution explorer tools for selection support in a user study of non-programmers. We describe this user study in the following sections.

4. EVALUATING EXECUTION EXPLORER TOOLS ADEQUACY FOR SELECTION SUPPORT

We conducted a user study to evaluate the adequacy of two execution explorer tools for supporting non-programmers in selecting the code responsible for graphical output. We developed two tools to evaluate: a *debugger*, the most widely available execution explorer tool, and our *Output History Explorer* tool, which was designed to address barriers non-programmers experienced in these tasks [15].

To maintain consistent programming contexts, we built both of these tools within the Looking Glass IDE. Looking Glass is the successor to Storytelling Alice [17]. Like Storytelling Alice, Looking Glass enables users to create interactive 3D animated stories. Looking Glass uses drag-and-drop based program construction to prevent users from making syntax errors (see Figure 1). The environment supports a range of programming constructs including methods, conditionals, and loops. Parallel execution is also supported and frequently used in programs [15].

4.1 Debugger

Debugger Design: While debuggers are designed primarily for fault localization, they can also be used for output feature localization. Because debuggers are available in many contexts, we wanted to explore their potential use in helping inexperienced programmers localize functionality and identify any barriers preventing their effective use.

To design our debugger, we identified common features and terminology used in the debuggers present in widely used novice programming environments (BlueJ [18], JGrasp [13], DrJava [8]) and professional IDEs used in first courses (Eclipse [1], NetBeans [6], JCreator [3]).

Debugger Interface: shown in Figure 2.

The Executing Threads Pane (1) displays a fine-grained thread tree. A user can view all current stack frames with concurrent threads under their invoking frame. The user can select a thread and then pause its execution. Clicking on a paused thread highlights the line of code the thread was executing when paused in a green box. Other controls enable a user to pause and play all threads.

Note that this executing thread representation differs from typical novice debuggers. Typical novice debuggers show executing thread lists with ambiguous thread names. Our pilot users struggled to understand and use that representation. Instead, we chose to show the concurrent stack frames which enable users to see hiearchical execution information and make more informed choices about which threads to investigate. This view to helped non-programmers move on to using all debugger features.

The Step Controls (2) enable users to step into an executing method or step over a statement.

The Local Values (3) show a user any local variables in scope of the selected thread and their current value.

The Breakpoints Panel (4) lists the breakpoints currently set by the user, which are shown in the code by a red outline. Clicking a breakpoint in the list shows its location. Users

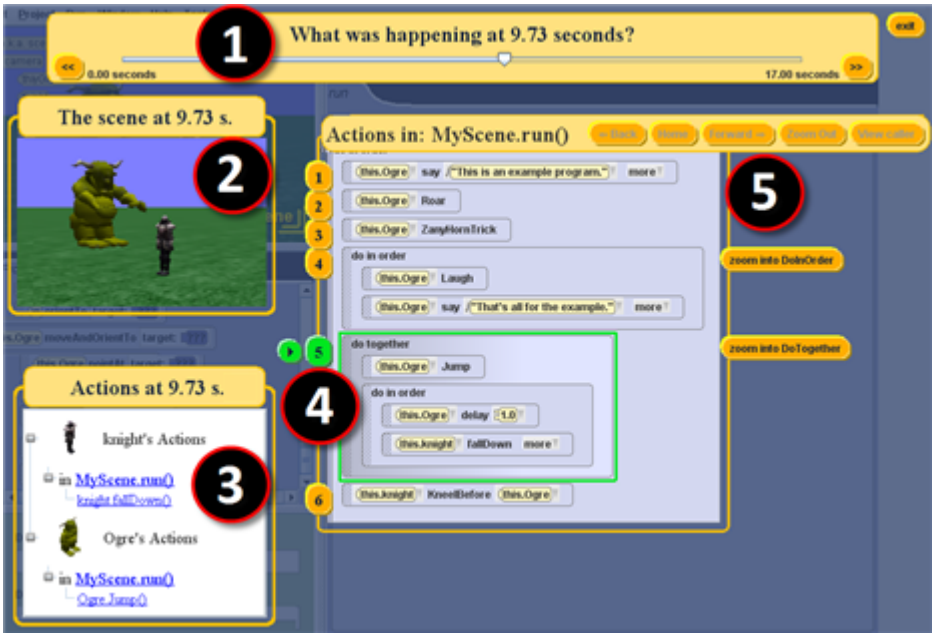

Figure 3: The Output History Explorer Tool Interface. (1) Time Slider for scrubbing through time, (2) Scene Viewer shows scene at the selected time, (3) Current Actions Pane shows what actions characters did at the selected time, (4) Annotated Code View highlights the executing line(s) of code and affords block and statement playback, and (5) Navigation controls for zooming blocks and methods.

can set breakpoints by right-clicking on statements. The panel also allows users to remove all breakpoints.

4.2 Output History Explorer Tool

Output History Explorer Tool Design: Previous research identified three barriers encountered by non-programmers in finding feature code [15]: (1) mapping output action descriptions to code, (2) fully navigating program code, and (3) recognizing language constructs' consequences. We designed the history tool to address these by (1) connecting output screenshots to the code that executed when the shot was taken, (2) affording controls for navigating directly to action code or incrementally zooming into blocks and methods, and (3) offering replay for individual statements and construct code blocks.

The history tool dynamically captures screenshots of a program's graphical output and a dynamic trace of a running program. To implement statement replay, the history tool shows the screenshots captured during the execution period of a statement. Although it is arguably more robust to implement replay through substituting previous program states (e.g., [31]), we chose to use screenshots because it provided a low-cost approach to evaluating the promise of output driven selection.

History Tool Interface: Figure 3 shows the History Tool interface. A user starts the history tool by first running a program. At any point the user can choose to "stop and explore" the output and execution history which ends the program and opens the tool.

The Time Slider (1) enables users to scrub forward and backward through the program's recorded history while the Scene Viewer (2) displays the scene's appearance at the selected time. A user can scrub though the recorded history until he or she sees an action of interest in the scene viewer.

The Current Actions Pane (3) shows all methods executing at the selected time, organized by character. Users expand a character's actions to see the individual statements executed at that time and which methods invoked them. Users can select and navigate to a statement or method call in the Code View Pane (4) by clicking on it.

The Code View Pane highlights statements executing at the selected time in green and offers a play button to replay the images captured while the selected element executed.

The Navigation Controls (5) offer means to navigate the code by zooming into and out of block statements and editable methods. A user can also navigate to a parent invocation, the main run method, and previous selections.

4.3 Methods

We randomly assigned users to a tool and provided a series of tasks in unfamiliar programs. Our design attempted to emulate a non-programmer, armed only with documentation, who finds a relevant example and needs to select the code causing the relevant graphical output. Each task asked users to mark the first and last line of code responsible for

User Tool	Avg. # Correct Tasks (9 users)	Avg. % Correct Tasks (9 users)	Avg. Task Time (8 users)
Debugger	1.22	40.2%	15:07
History	3.33	72.8%	8:58
	$p < .01$	$p < .05$	$p = .06$

Table 1: User performance results.

a given output. In some tasks, we distributed the responsible code throughout the program to simulate multiple code sections being responsible.

Eighteen adults, university students and staff with no prior programming experience, participated in the study.

4.4 Results

As shown in Table 1, non-programmers using the history tool correctly completed 2.7 times as many tasks as users of the debugger ($p < .01$). In addition to correctly completing more tasks, history tool participants had an overall success rate 81% higher than debugger participants and also tended to complete tasks more quickly ($p = .06$). Note that the task completion time excludes a user from each condition who did not finish the practice tasks.

To better explain these results we consider the barriers participants encountered with each tool.

4.4.1 Debugger Barriers

Debugger users experienced two barriers: users misunderstood threads, and misinterpreted step over.

Thread Misunderstanding: Misunderstanding of threads, and the executing threads interface, caused users to make poor code exploration choices. We did not observe users recognizing or acknowledging the concept of an execution stack, the main idea behind our thread tree and similar components in other debuggers (e.g., [13, 1]). Two users initially thought threads analogous to lines of code. One explained threads were "the programming parts that give specific commands", and watched for specific code statements, not invocation threads, to appear.

Step Over Misinterpretation: Five of the nine debugger users confused the step over function with a "skip over" function causing some users to not use it. The confusion came both from language, "I feel when you do step over you're skipping over an action," and from experience, "I think it just skipped ahead like in the code... step over seemed like a fast forward button to me." This confusion may have discouraged some users from using step controls.

4.4.2 History Tool Barriers

History tool users encountered two barriers: concurrency ambiguity, and playback fidelity issues.

Concurrency Ambiguity: Because the history tool's play feature relies on screenshots and time tagging, it cannot visually separate multiple actions that executed concurrently. Five history tool users struggled with this limitation and one noted "[I] can't really differentiate between the one that was move forward and the one that was turn forward, they both looked the same."

Playback Fidelity: Currently, the history tool captures screenshots as quickly as possible. This can result in a low frame-rate playback that makes it difficult to recognize short duration output actions. Five history tool users acknowledged the problem. Three users also had difficulty when the image displayed in the scene viewer appeared to show an action was occurring when it had just finished.

4.5 Discussion

Our results suggest that these execution explorer tools have potential to help inexperienced programmers select code. Although the history tool performed better, neither is perfectly adequate. Addressing these barriers could further improve support for identifying code sections corresponding to interesting graphical output. We suggest the following for future execution exploration tools.

1) *Do not rely on execution abstractions.* Our observations indicate tree based representations of code execution do not help non-programmers understand the overall organization and execution behavior of programs, leading to poor choices about what parts of the program code to explore.

2) *Provide affordances for replaying code elements.* History tool users frequently replayed individual statements and blocks. While the ability to replay statements and blocks was typically helpful, users struggled in situations in which multiple actions concurrently executed. This is a significant problem for environments where concurrency is commonly present, such as Looking Glass. Combining the simplicity of playing statements with the ability to separate concurrent actions seems a promising direction.

3) *Use direct code interactions.* Reading the code is the most common strategy for non-programmers. In the first practice task most users started by reading the code. Users later chose to read code when they found code sections likely related to their search target. This suggests features directly connected to lines or blocks of code, such as the play feature of the history tool, may be most natural for users.

5. FURTHER CHALLENGES

Although this paper focused on the challenge of supporting inexperienced end-user programmers in selecting example code, there are still many challenges in the space of transforming arbitrary code examples into usable learning materials. Challenges exist in searching for program examples, helping inexperienced users efficiently make relevant selections, and supporting complex selections.

5.1 Searching for Program Examples

The description an inexperienced end-user assigns to graphical output may not clearly map to an example's implementation. For instance, a user may want a fading animation on a web page and search for an example with a "fade" method. However, the only examples with a fading animation may use a loop iterating over transparency values without the word fade present. Concrete output concept descriptions may prevent end-users from adequately describing their search target or recognizing relevant code sections. After querying for an example, a user must evaluate the resulting example to determine their relevance. However, it is unrealistic to expect a user will execute 30 examples returned from a search engine. Tools making API search more usable (e.g., [10, 30]) may be useful in this context.

5.2 Helping Users Efficiently Make Relevant Selections

A primary issue for non-programmers selecting relevant code section is recognizing that code may be distributed into functions, classes, or files, and how these code containers relate to each other when a program is executing. This problem varies widely as languages and programming environments differ for each domain. One solution would be to provide an interface guiding inexperienced users through a good search strategy. Tools promoting an understanding of these concepts, or managing them, can help inexperienced users make better code exploration decisions when selecting relevant code.

5.3 Supporting Complex Selections

The code sections relevant to a user's interest may not always be a single sequential area of code. It is conceivable that a user finds interest in code spanning multiple files, or when using dynamic program tracing, multiple threads. Further, a user may perceive some code or abstraction of the code as unnecessary when in fact it is a hidden dependency of their primary interest. We must support users in making selection decisions and explaining their implications.

There is great potential for end-user programmers of all experience levels to extend their programming skills but, without tools to transform freely available code resources into usable materials, the potential of these resources code is unrealized.

6. ACKNOWLEDGMENTS

We thank Micah Herstand for thoughtful discussions and his help coordinating user studies. The NSF funded this work through grant #0835438.

7. REFERENCES

[1] Eclipse. http://www.eclipse.org.

[2] Google code search. http://google.com/codesearch.

[3] JCreator. http://jcreator.com/.

[4] Koders open source code search engine. http://www.koders.com.

[5] Krugle. http://www.krugle.com/.

[6] NetBeans. http://netbeans.org/.

[7] Userscripts.org: Power-ups for your browser. http://userscripts.org.

[8] E. Allen, R. Cartwright, and B. Stoler. DrJava: a lightweight pedagogic environment for java. In *Proc. of SIGCSE*, pages 137–141, 2002.

[9] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proc. of ICSE*, pages 482–498, 1993.

[10] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proc. of CHI*, pages 513–522, 2010.

[11] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. of CHI*, pages 1589–1598, 2009.

[12] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, Feb. 2009.

[13] J. A. Cross and T. D. Hendrix. jGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. *J. Comput. Small Coll.*, 23(2):170–172, 2007.

[14] B. Dorn and M. Guzdial. Graphic designers who program as informal computer science learners. In *Proc. of ICER*, pages 127–134, 2006.

[15] P. Gross and C. Kelleher. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *JVLC*, 21(5):263–276, Aug. 2010.

[16] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE*, pages 467–477, 2002.

[17] C. Kelleher, R. Pausch, and S. Kiesler. Storytelling alice motivates middle school girls to learn computer programming. In *Proc. of CHI*, pages 1455–1464. ACM, 2007.

[18] M. KÃűlling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249, 2003.

[19] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proc. of CHI*, pages 151–158, 2004.

[20] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *Proc. of CHI*, pages 1569–1578, 2009.

[21] G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proc. of CHI*, pages 1719–1728, 2008.

[22] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: a sneak preview [education]. In *Proc. of C5*, pages 104–109, 2004.

[23] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proc. of IWPC*, pages 33–42, 2005.

[24] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[25] S. Oney and B. Myers. FireCrystal: understanding interactive behaviors in dynamic web pages. In *Proc. of VL/HCC*, pages 105–108, 2009.

[26] M. B. Rosson, J. Ballin, and J. Rode. Who, what, and how: A survey of informal and professional web developers. In *Proc. of VL/HCC*, pages 199–206, 2005.

[27] J. R. Ruthruff and M. Burnett. Six challenges in supporting end-user debugging. In *Proc. of WEUSE*, pages 1–6, 2005.

[28] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. Burnett. Interactive, visual fault localization support for end-user programmers. *JVLC*, 16(1-2):3–40, Feb. 2004.

[29] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Proc. of VL/HCC*, pages 207–214, 2005.

[30] J. Stylos and B. Myers. Mica: A Web-Search tool for finding API components and examples. In *Proc. of VL/HCC*, pages 195–202, 2006.

[31] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson. Replay debugging of real-time systems using time machines. In *Proc. of IPDPS*, page 8 pp., 2003.

[32] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *J. of Soft. Maint.: Research and Practice*, 7(1):49–62, 1995.

[33] J. Wong and J. I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proc. of CHI*, pages 1435–1444, 2007.