Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover

Joel Brandt, Philip J. Guo, Joel Lewenstein, and Scott R. Klemmer, *Stanford University*

Mira Dontcheva, *Adobe Systems*

Five principles of opportunistic programming can help guide the development of tools that explicitly support prototyping in code.

People often write code to prototype, ideate, and discover. To do this, they work opportunistically, emphasizing speed and ease of development over code robustness and maintainability. Quickly hacking a program together can provide both practical and learning benefits for novices and experts:[1] professional programmers and designers prototype to explore and communicate ideas,[2,3] scientists program laboratory instruments, and entrepreneurs assemble complex spreadsheets to better understand their business.[4] Their diverse activities share an emphasis on speed and ease of development over robustness and maintainability.

How do opportunistic programmers make these trade-offs between speed of development and maintainability, and how does the structure of their work compare to more formal software engineering practices? Through our recent research on how opportunistic programming enables prototyping and exploration, we've identified five traits of the opportunistic approach: Opportunistic programmers build software using high-level tools, and often add new functionality via copy-and-paste from the Web. They iterate rapidly, consider code impermanent, and find debugging particularly challenging. We're using these traits to guide the development of tools that explicitly support opportunism (for more about these tools, see http://hci.stanford.edu/opportunistic).

Hacking in the Wild and in the Lab

We became interested in opportunistic programming while conducting fieldwork with exhibit designers at the Exploratorium, a hands-on science and art museum in San Francisco. All the exhibits are developed in-house, and most have interactive computational components (see Figure 1).

Exhibit designers conceive and implement interactive exhibits that convey a particular scientific phenomenon. Many of these exhibits require custom software. For example, a microscopy exhibit required exhibit designers to retrofit a research-grade microscope with a remote, kid-friendly interface. Although designers must construct working exhibits, they have little responsibility for an exhibit's long-term maintainability or robustness. (A separate division of the museum commercializes successful exhibits and sells them to other

 0740-7459/09/$26.00 © 2009 IEEE

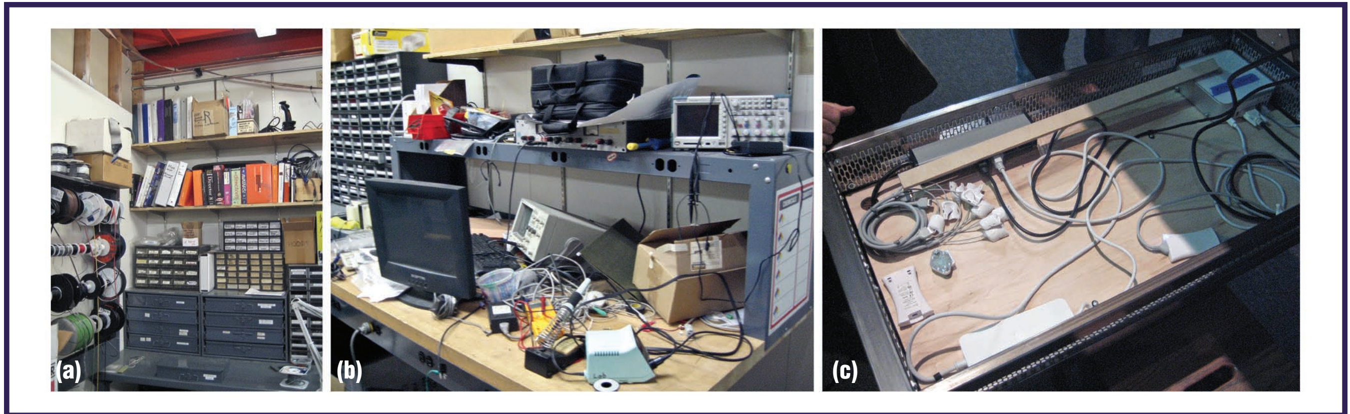


Figure 1. The Exploratorium, a hands-on science and art museum in San Francisco. Exhibit designers handle all phases of development. They design interactions, construct physical components, and develop software. (a, b) Their work environment is filled with computers, electronics equipment, and manuals for a diverse set of software. (c) A typical exhibit comprises many off-the-shelf hardware components hooked together using high-level languages such as Adobe Flash.

museums throughout the country.) They therefore focus on exploring many ideas as rapidly as possible.

To get a more fine-grained understanding of how people work opportunistically, we brought 20 programmers into our lab. They prototyped a Web-based chat room using HTML, PHP, and JavaScript. We gave them five specifications, such as "the chat room must support multiple concurrent users and update without full page reloads." For details of this lab study, see "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code."[5] For the five principles we uncovered from the field and lab, read on.

Glue Together High-Level Components

At the Exploratorium, designers select task-specific building blocks and build systems largely by writing "glue" code. For example, the nature observation exhibit Out-quiet Yourself teaches visitors how to walk quietly. In this exhibit, museum visitors walk over a bed of gravel. During the walk, the total amount of sound produced is measured and displayed on a large screen. From a performance perspective, all of the audio processing necessary for this exhibit could have been easily done on a single computer. To do this, however, the exhibit designer would have had to write a large amount of custom code. Instead, he used a series of hardware audio compressors and mixers to do most of the processing. He only needed to write two pieces of glue code: a small Python script to calculate the sum, and a simple Adobe Flash interface to display that sum. Our group has witnessed similar behavior in other domains.[3]

We observed that participants were most successful at bricolage development when components were themselves fully functioning systems. For example, we asked the Out-quiet Yourself designer why he used specialized audio hardware instead of a software library. He explained that he could experiment with the hardware independently from the rest of the system, which made understanding and tweaking the system much easier.

In general, gluing together fully functioning systems helps reduce several of the barriers that less-experienced programmers face.[6] First, because whole systems are easy to experiment with, programmers can more easily understand how the pieces work and can immediately intuit how to use them. Second, because a clear boundary between each piece exists, programmers avoid coordination barriers. There's exactly one way to connect the pieces, and it's easy to see what's happening at the connection point.

Familiarity and fitness to task are two important considerations when selecting components. What factors affect these considerations' relative weight? At the Exploratorium and in our lab study, composition and reuse occurred at multiple scales, and a component's scale played an important role in determining whether it would be used. Specifically, successful opportunistic programmers valued fitness over familiarity when selecting tools for large portions of the task. For example, an exhibit designer who was an excellent Python programmer learned a new language (Max/MSP) to build an exhibit on sound because the new language was better suited to audio processing than Python.

At smaller scales of composition, the familiarity/fitness trade-off shifts to favor the familiar. For

Figure 2. A typical snippet of PHP code (querying a database and iterating through returned values) that nearly all lab study participants copied from examples found on the Web. Most participants reported that they could have written the code from scratch, but it was faster to copy and paste.

```
<?php
$res = mysql_query("SELECT id, name FROM table");

while ($row = mysql_fetch_array($res)) {
    echo "id: ".$row["id"]."<br>\n";
    echo "name: ".$row["name"]."<br>\n";
}
?>
```

example, when we asked one participant in our lab study if he knew of libraries to make Ajax calls easier, he responded, "Yes … but I don't understand how Ajax works at all. … If I use one of those libraries and something breaks, I'll have no idea how to fix it." Only three participants in this study used external Ajax libraries, and these individuals already had significant experience with them.

An alternate approach to gluing a system together from scratch using high-level components is to find and tailor an existing system that almost does the desired task. In our Web programming lab study, three participants did this, and two of them failed to meet some of the specifications. Leveraging an existing system let them make quick initial progress, but the last mile was difficult. For example, one participant built upon an existing content-management system with a chat module that met all but two specifications. He spent 20 minutes finding the system and 10 minutes installing it, meeting the first three specifications faster than all other participants. However, he took an additional 58 minutes to meet one more specification (adding time stamps to messages), and he was unable to meet the final specification (adding a chat history) in the remaining hour. The other two participants who modified existing systems faced similar, although not as dramatic, frustrations.

The distinction between co-opting and tailoring an existing system is subtle but important. To co-opt a system, the programmer must understand only how to use its interface; to tailor a system, the programmer must understand how it's built. The main challenge of tailoring an existing system is building a mental model of its architecture. This can be difficult and time-consuming even in the best of circumstances. Even when the code is well documented, the programmer is familiar with the tools involved, and the original code's authors are available for consultation, mental-model formation can take considerable time.[7] Large software is inherently complex, and trying to understand a system by looking at source code is like trying to understand a beach by looking at a grain of sand.

Add Functionality via Copy-and-Paste from the Web

Even when programmers build software from existing components, they must write some glue code to hook these pieces together. *Copy-and-paste programming*—writing code by iteratively searching for, copying, and modifying short blocks of code (fewer than 30 lines) with desired functionality—is a staple of opportunistic programming. An earlier study by our research group observed students learning to use a new programming framework. One-third of participants' code consisted of modified versions of examples found in the framework's documentation.[8]

Copy-and-paste programming is most beneficial when the programmer is working in an unfamiliar domain. For example, most participants in our lab study who were unfamiliar with Ajax chose to copy and paste snippets of Ajax setup code rather than learn to write it from scratch.

However, copy-and-paste isn't simply for novices; several participants were expert PHP programmers and still used this practice for some code pieces, such as the snippet in Figure 2. When one participant searched for and copied a piece of PHP code necessary to connect to a MySQL database, he commented that he had "probably written this block of code a hundred times." Upon further questioning, he reported that he always wrote the code by copy-and-paste, even though he fully understood what it did. He claimed that it was "just easier" to copy-and-paste it than to memorize and write it from scratch.

This observation brings up interesting questions about how programmers locate promising code. In opportunistic programming, the primary source is through Web search.[5] Indeed, in our lab study, each participant spent on average 19 percent of his or her programming time on the Web, spread out over 18 distinct sessions. These sessions occurred throughout development and varied greatly in length. Figure 3 summarizes participants' Web access behavior.

How do these Web sessions differ? At one end of the spectrum, participants spent tens of minutes learning a new concept (for example, by reading a tutorial on Ajax-style programming). On the other end, participants delegated their memory to the Web, spending tens of seconds to remind themselves of syntactic details of a concept they knew well (for example, by looking up the structure of a `foreach` loop). Between these two extremes,

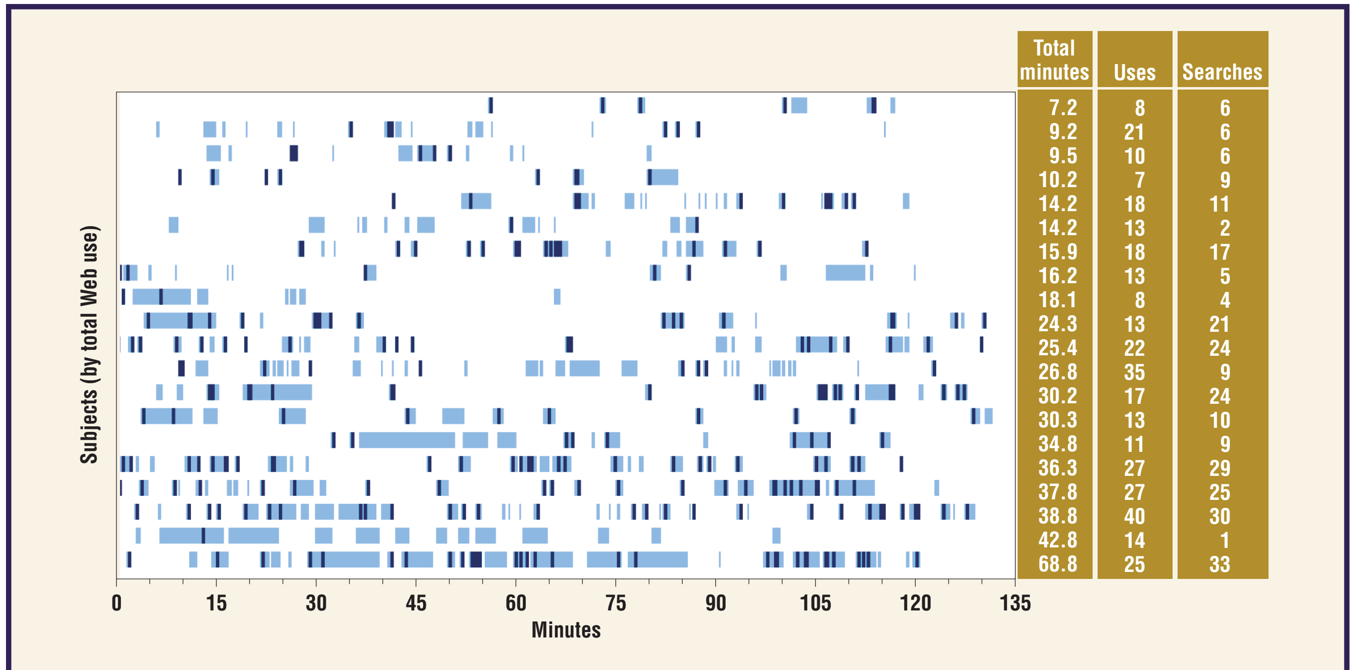


Figure 3. An overview of participants' use of the Web during the laboratory study. Subjects are sorted by total amount of time spent using the Web. Light-blue bars indicate Web use sessions; dark bars indicate Web search instances.

participants used the Web to clarify their existing knowledge (such as by viewing an HTML form's source to understand the underlying structure). Understanding these intentions is crucial to designing tools that help programmers better leverage the Web.

Iterate Rapidly

Successful opportunistic programmers in our lab study favored a short edit-debug cycle. Figure 4 gives an overview of the length of each participant's edit-debug cycles. For the vast majority of subjects, 50 percent of the cycles were less than 30 seconds long; for all subjects, 80 percent of the cycles were less than 5 minutes long. These times are much shorter than those commonly reported during traditional software engineering. In a 2006 O'Reilly technical blog entry, a Java developer estimated that an average cycle takes 31 minutes and a short cycle takes 6.5 minutes.[9]

Frequent iteration is a necessary part of learning unfamiliar tools and understanding found code. So, successful opportunistic programmers select tools that speed up iteration. For example, programmers prefer interpreted languages over compiled languages because they emphasize human productivity over code execution speed.[10]

Consider Code Impermanent

Programmers often use code written opportunistically to ideate and explore the design space when prototyping. It's a kind of breadth-first programming in which many ideas are thrown away early. Because developers throw away much of the code they write opportunistically, they often consider code impermanent. This perception affects how they write code in two important ways.

First, programmers spend little time documenting and organizing code that they write opportunistically. An Exploratorium exhibit designer remarked that it simply wasn't worth his time to document code because he "ended up throwing so much away." Instead of documenting their code, successful opportunistic programmers document their process. For example, one designer keeps a project notebook for each exhibit. In this notebook, he documents important knowledge gained through the design process, such as a particular tool's strengths and weaknesses, or why a user interface was unsuccessful. Programmers rarely reuse code written opportunistically. Another exhibit designer reported that he only reuses code when he had written it "for the last project [he] worked on. … Otherwise, it is just too much trouble." However, both designers reported that with the right kind of documentation, process reuse is common and invaluable.

Second, the perceived impermanence of code written opportunistically leads to *code satisficing*. Programmers often implement functionality suboptimally during opportunistic development to maintain flow. For example, a participant in our

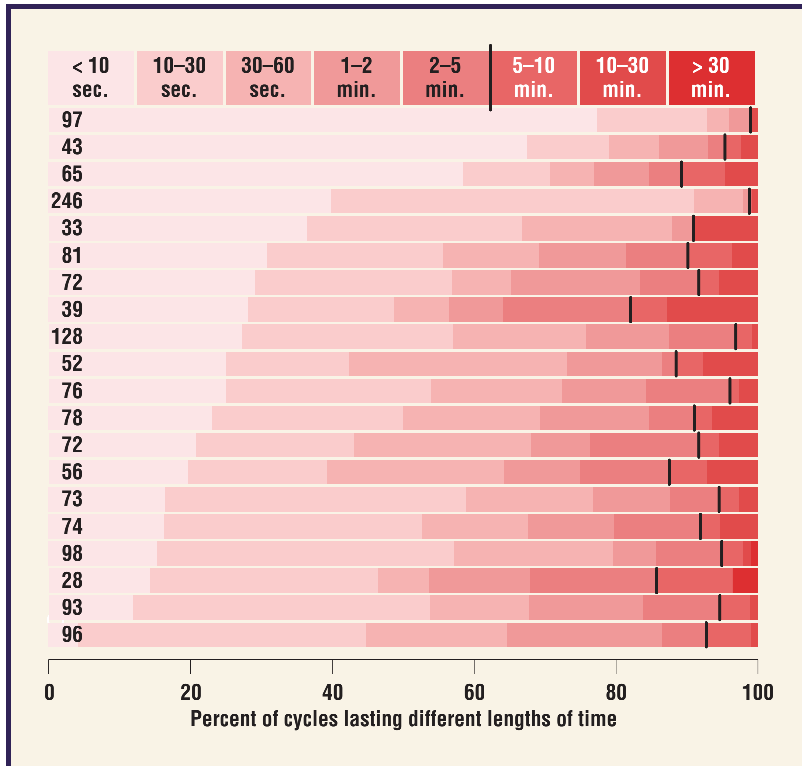


Figure 4. A histogram of per-subject edit-debug cycle times in our laboratory study. The black numbers on the left are the total number of edit-debug cycles for each subject. Bar lengths are normalized across subjects; a black line separates cycles shorter and longer than 5 minutes.

lab was attempting to implement a fixed-length queue using an array to store chat history. She was a novice PHP programmer but an experienced programmer overall. She guessed at PHP array notation, and guessed wrong. Instead of looking up the notation, she created 10 global variables, one for each array element. She commented that although she knew "there was a better way to do this," she "didn't want to be interrupted." Initially, it appeared she made the right decision, because she was able to test the history functionality only seconds later. However, this led to problems down the road. When implementing the dequeue operation, she made a typographical error that took more than 10 minutes to debug and clearly broke her flow.

As this example illustrates, code satisficing can be good and bad. Successful opportunistic programmers are good at weighing the trade-offs between implementing something correctly and implementing something quickly.

Face Unique Debugging Challenges

Opportunistic programming leads to unique debugging challenges. First, as we mentioned earlier, programmers often glue together many disparate components. Development therefore often occurs in multiple languages. For example, a typical museum exhibit consists of a Flash user interface that controls several stepper motors by communicating with an Arduino microcontroller via TCP/IP code written in Python! When projects use a federation of languages, programmers often can't effectively use sophisticated debugging tools intended for a single language. Instead, they must make state and control flow changes visible through mechanisms such as print statements. During our laboratory study, we observed that experienced opportunistic programmers would take proactive steps to make state visible while adding new functionality. For example, they would insert print statements preemptively "just in case" they had to debug later. Less-experienced programmers would have to make state visible retroactively (for example, insert print statements) after a bug occurred, which was much more time-consuming. Interestingly, the less-experienced programmers spent significant time trying to determine whether a block of code they had just written was even executing, let alone whether it was correct!

Second, because there's little or no up-front design, system pieces often don't have clean interfaces (for example, communication between functions might occur via a global variable). This makes debugging more difficult, because programmers must maintain a mental model of the entire system, not just of the particular component they're currently debugging.

Looking Forward

Guided by these five traits, we're building tools that explicitly support opportunistic programming. We've identified four broad areas that could benefit from better tool support.

Code Foraging and Reuse

The Web has made a wealth of example code available, but finding and understanding relevant code still remains challenging. Our group's recent work on d.mix explores a potential solution to this problem.[11] The d.mix tool makes it easier for programmers to find and experiment with Web APIs by letting them "sample" user interfaces that already use these calls and then experiment with the resulting code inside a wiki-like sandbox.

Another approach is to integrate Web search into the development environment. Doing this could improve search by leveraging the programmer's current context (for example, languages,

libraries, and frameworks being used). Additionally, we might be able to guide the user in adapting found code by collecting information on how others have used that code. For example, if the last 10 programmers to use an example all changed a particular literal, the 11th programmer probably should as well. We're exploring these ideas through Blueprint (http://hci.stanford.edu/blueprint), a plug-in for Adobe Flex Builder.

Code Authoring and Debugging

Debugging in opportunistic programming is difficult for three reasons:

- A single project often uses many languages.
- Code satisficing leads to code that isn't well encapsulated.
- Developers often refuse to invest time in learning complex (but powerful) tools.

There's significant value in building authoring and debugging tools that embrace how opportunistic programmers already work. For example, print statements could be made a first-class tool. A development environment could make inserting or removing a print statement as easy as setting a breakpoint. The debugger could then capture a wealth of context at each of these print points: the call stack, the value of all local variables, and a snapshot of the program's output. Similarly, development environments could exploit the rapid iteration inherent in opportunistic programming—code that was written 30 seconds ago is likely the code the programmer wants to test and debug. Simply indicating which lines of code were executed during the program's last run would help programmers avoid time-consuming debugging mistakes. Our group is exploring new editing and debugging interactions using the Rehearse development environment (http://hci.stanford.edu/rehearse).

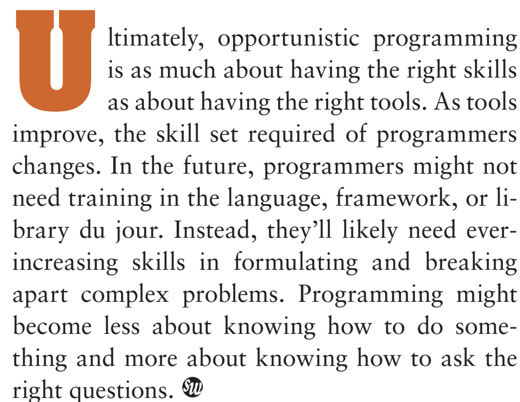
Alternatively, tools might eliminate the need for rapid iteration in specialized cases, such as parameter tuning. Juxtapose, for example, lets programmers easily tune parameter values at runtime.[12] Interactive tuning is particularly valuable for exploring user interface variations, because programmers can consider alternatives without having to stop execution, edit, compile, execute, and navigate to the previous state.

Version Control

Current version-control systems have large upfront setup and learning costs, and aim to support the development of large systems by many developers over months or years. What might version control look like for opportunistic programming? Our observations suggest that programmers would benefit from version control designed for a 10-minute scale. Participants often wished that they could revert to the code they had, for example, two tests ago, or quickly branch and explore two ideas in parallel. Perhaps we could bring single-user version control inside the editor, eliminating the setup burden of current tools. Such a system could perform code committal automatically each time the code is executed, reducing the need for programmers to think proactively about version management. Finally, perhaps users could browse past versions by viewing snapshots of the execution, removing the burden of explicitly specifying commit messages or applying tags.

Documentation

Although opportunistic programmers throw much of their code away, the insights gained during the entire design process are extremely valuable. An Exploratorium exhibit designer commented that whereas he rarely looked at code from prior projects, he often reviewed his process. Right now, however, the tools for documenting process (such as a notebook) are independent of the tools being used (such as Adobe Flash). Bridging this divide is a valuable path for future research.



Ultimately, opportunistic programming is as much about having the right skills as about having the right tools. As tools improve, the skill set required of programmers changes. In the future, programmers might not need training in the language, framework, or library du jour. Instead, they'll likely need ever-increasing skills in formulating and breaking apart complex problems. Programming might become less about knowing how to do something and more about knowing how to ask the right questions.

There's significant value in building authoring and debugging tools that embrace how opportunistic programmers already work.

References

1. S. Clarke, "What Is an End-User Software Engineer?" *End-User Software Eng.*, Internationales Begegnungs- and Forschungszentrum für Informatik, 2007; http://drops.dagstuhl.de/portals/index.php?semnr=07081.
2. S. Houde and C. Hill, "What Do Prototypes Prototype?" *Handbook of Human–Computer Interaction*, M. Helander, T.Ê. Landauer, and P. Prabhu, eds., Elsevier Science, 1997, pp. 367–382.
3. B. Hartmann, S. Doorley, and S.R. Klemmer, "Hacking, Mashing, Gluing: Understanding Opportunistic Design," *IEEE Pervasive Computing*, vol. 7, no. 3, 2008, pp. 46–54.

About the Authors

Joel Brandt is a PhD candidate in Stanford University's Human-Computer Interaction Group and a research intern at Adobe Systems. His research interests include opportunistic programming, an approach to building software often undertaken by nonprofessional programmers such as designers, scientists, and business professionals. Brandt has an MS in computer science from Washington University. Contact him at jbrandt@cs.stanford.edu.

Philip J. Guo is a PhD candidate in Stanford University's Computer Science Department. His research interests include how computer programmers work, how to automatically find bugs in the code they write, and how programmers deal with these bugs. Guo has an MEng in electrical engineering and computer science from the Massachusetts Institute of Technology. Contact him at pg@cs.stanford.edu.

Joel Lewenstein is a software engineer at GoodGuide.com. His research interests include the use of external resources during programming, specifically during opportunistic programming. Lewenstein has a BS in symbolic systems, with a focus on human-computer interaction, from Stanford University. Contact him at jlewenstein@cs.stanford.edu.

Mira Dontcheva is a research scientist in computer graphics and human-computer interaction at Adobe Systems. Her research interests include building tools that improve information foraging and sense-making on the Web. Dontcheva has a PhD in computer science from the University of Washington. Contact her at mirad@adobe.com.

Scott R. Klemmer is an assistant professor of computer science at Stanford University, where he codirects the Human-Computer Interaction Group. His research focuses on understanding and building tools that support the prototyping process. Klemmer has a PhD in computer science from the University of California, Berkeley. Contact him at srk@cs.stanford.edu.

4. M. Schrage, *Serious Play: How the World's Best Companies Simulate to Innovate*, Harvard Business School Press, 1999.
5. J. Brandt et al., "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code," *Proc. 27th Int'l Conf. Human Factors in Computing Systems* (CHI 09), ACM Press, 2009, pp. 1589–1598.
6. A.J. Ko, B.A. Myers, and H.H. Aung, "Six Learning Barriers in End-User Programming Systems," *Proc. 2004 IEEE Symp. Visual Languages—Human-Centric Computing* (VLHCC 04), IEEE CS Press, 2004, pp. 199–206.
7. T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), ACM Press, 2006, pp. 492–501.
8. R.B. Yeh, A. Paepcke, and S.R. Klemmer, "Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces," *Proc. 21st Ann. ACM Symp. User Interface Software and Technology* (UIST 08), ACM Press, 2008, pp. 111–120.
9. T.M. O'Brien, "Dead Time (... Code, Compile, Wait, Wait, Wait, Test, Repeat)," blog, 30 Mar. 2006; www.oreillynet.com/onjava/blog/2006/03/dead_time_code_compile_wait_wa.html.
10. J.K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *Computer*, vol. 31, no. 3, 1998, pp. 23–30.
11. B. Hartmann et al., "Programming by a Sample: Rapidly Creating Web Applications with d.mix," *Proc. 20th Ann. ACM Symp. User Interface Software and Technology* (UIST 07), ACM Press, 2007, pp. 241–250.
12. B. Hartmann et al., "Design as Exploration: Creating Interfaces through Parallel Authoring and Runtime Tuning," *Proc. 21st ACM Symp. User Interface Software and Technology* (UIST 08), ACM Press, 2008, pp. 91–100.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Engineering and Applying the Internet

IEEE Internet Computing

IEEE Internet Computing reports emerging tools, technologies, and applications implemented through the Internet to support a worldwide computing environment.

For submission information and author guidelines, please visit www.computer.org/internet/author.htm