Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice

Joel Brandt, Philip J. Guo, Joel Lewenstein, Scott R. Klemmer
Stanford University HCI Group
Computer Science Department, Stanford, CA 94305
{jbrandt, pg, jlewenstein, srk}@cs.stanford.edu

ABSTRACT

At times, programmers work opportunistically, emphasizing speed and ease of development over code robustness and maintainability. They do this to prototype, ideate, and discover; to understand as quickly as possible what the right solution is. Despite its importance, *opportunistic programming* remains poorly understood when compared with traditional software engineering. Through fieldwork and a laboratory study, we observed five characteristics of opportunistic programming: Programmers *build software from scratch using high-level tools*, often add new functionality via *copy-and-paste*, *iterate more rapidly* than in traditional development, consider code to be *impermanent*, and face unique *debugging challenges* because their applications often comprise many languages and tools composed without upfront design. Based on these characteristics, we discuss future research on tools for debugging, code foraging and reuse, and documentation that are specifically targeted at this style of development.

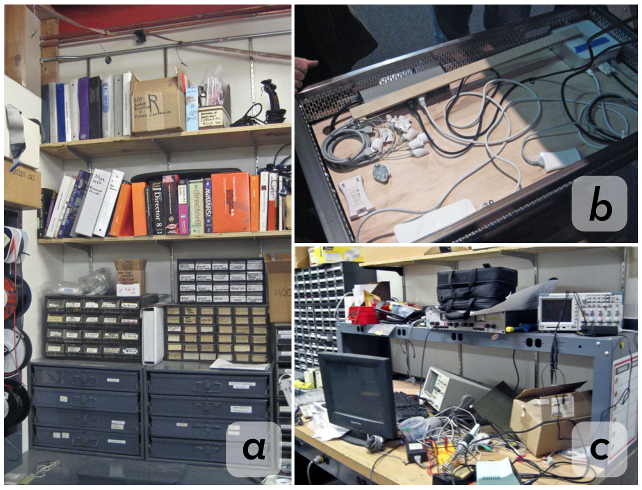

Figure 1: The Exploratorium Museum in San Francisco, California, where all exhibits are created in-house. Exhibit designers are responsible for all phases of development: designing interactions, constructing physical components, and developing software. They are jacks-of-all-trades, their work environment (a,c) filled with computers, electronics equipment, and manuals for a diverse set of software. A typical exhibit (b) comprises many off-the-shelf components hooked together using high-level languages such as Adobe Flash.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Prototyping*; D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Design, Experimentation

Keywords

Opportunistic Programming, Prototyping, End-User Software Engineering

1. INTRODUCTION

Hacking helps people prototype, ideate, and discover: An interface designer creates as many prototypes as possible before the user test next week. A museum exhibit designer explores many different directions to find the best way to convey a scientific concept. A physicist writes a complex real-time system to collect massive amounts of data during an experiment that she can only run once. A professional C++ programmer decides to create a simple web application in his spare time to share bookmarks with his friends. While these examples are diverse, there is a great deal of commonality in how each of these individuals approach the programming process. All of them emphasize speed and ease of development over robustness and maintainability of code — in many cases their code will only be run a few times, or used for days or weeks at the longest. Similarly, each is unlikely to invest a great deal of effort into learning complicated libraries or tools to help them in their development process — they may only be using those tools during the afternoon that the development takes place. In this paper, we refer to this approach as *opportunistic programming*.

Motivated by Clarke's persona of the "opportunistic developer" [1], we define opportunistic programming as follows: It is an activity where non-trivial software systems are constructed with little to no upfront planning about implementation details, and ease and speed of development are prioritized over code robustness and maintainability. This is not simply "sloppy programming"; instead, this approach enables prototyping, ideation, and discovery, tasks that benefit significantly from being done rapidly, and are often best accomplished by building a functional piece of software.

This paper presents five characteristics of opportunistic programming that we have identified in our ongoing research and suggests directions for further research on tools that support this practice. We hope that this paper will generate discussion on the position of opportunistic programming within the software engineering community.

1.1 Our current research

In this section, we introduce two of our ongoing projects designed to better understand opportunistic programming. We will present preliminary results in Section 2.

First, we are conducting fieldwork with exhibit designers at the Exploratorium Museum in San Francisco, California. The Exploratorium is a museum of science, art, and human perception. All exhibits are developed in-house (see Figure 1), and the majority of these exhibits have interactive computational components. Exhibit designers are responsible for coming up with and implementing the interactions that will best convey a particular scientific or perceptual phenomenon. Many of these interactions can only be achieved by developing custom pieces of software. For example, an exhibit on microscopy required exhibit designers to retrofit a research-grade microscope with a remote, kid-friendly interface. While exhibit designers need to have the skills to make these exhibits work — indeed, several of them have training as computer scientists or electrical engineers — they have little responsibility for the long-term maintainability or robustness of an exhibit. (If an exhibit is successful, it is commercialized by a separate division of the museum and sold to other museums throughout the country.) As such, they emphasize exploring many ideas as rapidly as possible over ensuring robustness and maintainability. Our fieldwork thus far has consisted of several open-ended interviews with two of the exhibit designers to better understand their work practice. We are currently beginning work on a retrospective of how several of the exhibits were built.

Second, we recently conducted an exploratory laboratory study with 20 individuals who were all competent programmers. The goal of our study was to understand what programmers do when asked to complete a task that encourages opportunistic programming practices. We asked participants to build a web-based group chat room with HTML, PHP, and JavaScript. When recruiting, we specified that participants should have basic knowledge of PHP, JavaScript, and the AJAX paradigm. However, almost all participants were novices in at least one of the technologies involved. We gave the participants several specifications for the chat room (*e.g.*, must support multiple concurrent users and update without full page reloads) but encouraged them to otherwise try to implement as much functionality as possible without regard to code efficiency or programming style. We asked them to approach this task as though they were working on a hobby programming project, not on a class assignment, in order to encourage opportunistic programming practices. Participants were given 2.5 hours to complete this task, and 15 out of 20 met all of the specifications. We provided the participants with a working execution environment within Windows XP (Apache, MySQL, and a PHP interpreter) with a "Hello World" PHP application already running, and allowed them to use *any* resources (Internet, print, etc.) that they wished during development. The data collection phase of this study is complete, but the analysis is ongoing.

2. OPPORTUNISTIC PROGRAMMING IN PRACTICE

In order to understand how to build better tools for opportunistic programming, it is important to understand how it happens in practice. This section presents five common characteristics we have identified in our ongoing research.

2.1 Build from scratch using high-level tools

Both our work at the Exploratorium and a prior study of three other disciplines (web programming, toy development, and ubiquitous computing design) [3] indicate that individuals engaging in opportunistic programming have the freedom to select the tools they use. They choose to use high-level tools that map closely to the task at hand and build their systems from scratch by "gluing" these tools together. For example, Exploratorium exhibit designers use user interface tools such as Adobe Flash and Director, sound processing tools such as Max/MSP, and high-level general-purpose languages such as Python as glue. Additionally, they appear to be more interested in selecting tools that map closely to the task at hand than selecting tools that they already know how to use.

At first blush, it may seem that an optimal strategy would be to find and modify an existing system that almost does the desired task. Even with the assumption that such an existing system is easy to find and open to modification, this approach can prove quite problematic: First, modifying an existing system requires first building a mental model of that system. This can be difficult and time-consuming even in the best of circumstances. Research at Microsoft [5] shows that even when the code is well documented, the programmer is familiar with the tools involved, and the authors of the original code are available for consultation, proper mental model formation can still take a considerable amount of time.

In our laboratory study, only three individuals chose to modify an existing system, and two of those failed to meet some of the specifications. Leveraging an existing system allowed them to make quick initial progress, but made it difficult to achieve the exact specifications. For example, one participant built upon an existing content-management system with a chat module that already met 4 of the 6 specifications. He spent 20 minutes finding and 10 minutes installing the system, thereby meeting those specifications faster than all other participants. However, it took him an additional 58 minutes just to add timestamps to messages, and he was unable to meet the final specification (adding a chat history) in the final hour. The other two participants who modified existing systems faced similar, albeit not as dramatic, frustrations.

Similarly, only three participants in our study used external libraries, and in all cases these individuals already had

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

while ($row = mysql_fetch_array($res)) {
    echo "id: ".$row["id"]."<br>\n";
    echo "id: ".$row[0]."<br>\n";
    echo "name: ".$row["name"]."<br>\n";
    echo "name: ".$row[1]."<br>\n";
}
?>
```

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.

significant experience with those libraries. When one participant who didn't use external libraries was asked if he was aware that libraries existed 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."

So why are those who engage in opportunistic programming willing to spend effort to understand new tools but not existing bodies of code or libraries? We suggest that good tools typically have a "closeness of mapping" [6] that makes learning the associated mental model much easier than constructing a mental model about an existing body of code. Additionally, programmers are likely to employ a federation of tools, one for each sub-task. A consequence of leveraging tools over existing code is that development often occurs in multiple languages. (*E.g.*, 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!)

This approach has many benefits. In addition to facilitating the process of mental model development, it allows developers to compartmentalize different components of the system, which is beneficial when opportunistic programming is used for ideation and exploration (discussed further in Section 2.4). Additionally, it makes debugging easier, as it is easy to make state visible "at the glue" (discussed further in Section 2.5).

Interestingly, this characteristic of opportunistic programming contrasts with both professional software engineering and end-user development: Professionals are often constrained by the tools that their team or organization have adopted and are, in fact, often recruited based on their knowledge of particular tools; end-user programmers might not have the expertise to be able to select and adopt the proper tools.

2.2 Add new functionality via copy-and-paste

"Copy-and-paste programming" — writing code by iteratively searching for, copying, and modifying short blocks of code (< 30 lines) with desired functionality [4] — appears to be a staple of opportunistic programming. In our laboratory study, all participants employed this practice extensively. One high-level reason for this is obvious: When people are working in a domain in which they are novices, copy-and-paste programming is simply easier than trying to come up with the code by oneself. For example, the vast majority of participants were novices with AJAX and copied-and-pasted snippets of AJAX setup code rather than try to learn to write it from scratch. However, copy-and-paste is not simply for novices; several participants were expert PHP programmers and still employed this practice for some pieces of code, like the one shown 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.

Cognitive science research on human performance and human error offer a deeper insight into why this may be true [10]. Psychologists divide human performance into three levels: skill-based (*e.g.*, walking), rule-based (*e.g.*, navigating to another office in a well-known building), and knowledge-based performance (*e.g.*, planning a route to a place one has never been). We believe that copy-and-paste programming allows developers to engage in rule-based performance regardless of whether or not they are experts with the tools they are using. Broadly speaking, the rule they follow to accomplish the goal of "implement functionality *foo*" is: 1.) search for code that does *foo*; 2.) evaluate quality of found code; 3.) copy code into project; 4.) modify as necessary; 5.) test code. Because the individuals doing opportunistic programming are *programmers*, it is easy for them to come up with the high-level goals, and copy-and-paste programming gives them a rule by which to meet those goals, regardless of familiarity with existing tools.

This observation opens up interesting questions on *how* programmers locate "promising" code. In opportunistic programming, we believe the primary source is through web search. Indeed, while we encouraged participants in our lab study to bring any external resources they typically used while programming (*e.g.*, books), not a single participant did so. All participants used the Internet to locate code to copy, and all but three used this exclusively. The remaining three also copied from code they had written in the past. Strategies for locating code were quite consistent with Information Foraging Theory [9]: First, whenever possible, participants would look to their own code before searching on the Internet because the former was written in their own personal style and was perceived as being easier to comprehend. When searching the Internet, participants used a variety of clues to gauge information "scent" (the potential for finding high-quality information down a certain search path): attractiveness and professionalism of the website, the complexity and modularity of the code examples, and so forth. If the quality of a code snippet was not satisfactory, they would often look for confirmation by comparing against related snippets from other websites before pasting the code into their project. Finally, once a participant was satisfied with a snippet of code copied from a particular site, she would often remain "loyal" to that site, copying additional code from there without searching the whole web.

2.3 Iterate rapidly

Developers tend to favor a short edit-debug cycle when doing opportunistic programming. Figure 3 presents an overview of the length of edit-debug cycles in our laboratory study. The graph shows that for the vast majority of

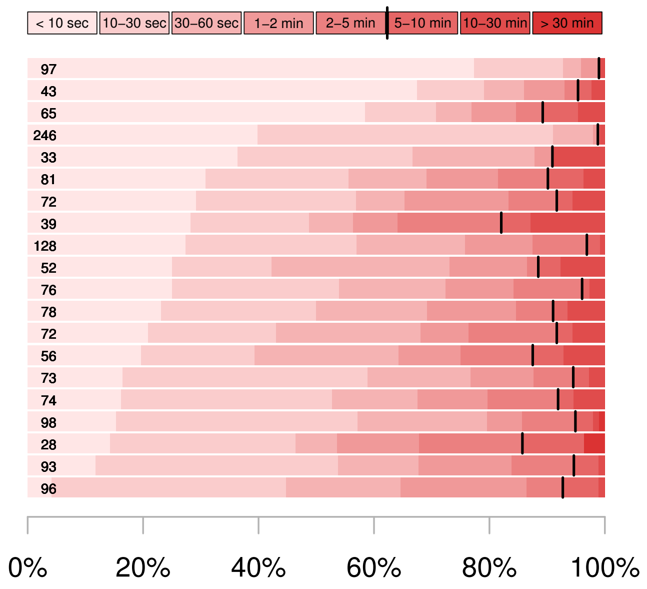

Figure 3: Histogram of per-subject edit-debug cycle times in our laboratory study. Total number of edit-debug cycles for each subject are given by the black number on each bar, and bar length is normalized across subjects. A black line separates cycles of less than and greater than 5 minutes. For all subjects, 80% of edit-debug cycles were less than 5 minutes in length.

subjects, 50% of their edit-debug cycles were less than 30 seconds in length, and for all subjects, 80% of their edit-debug cycles were less than 5 minutes in length. Only 2 subjects had edit-debug cycles of longer than 30 minutes, and each only underwent 1 such cycle. These times are much shorter than those commonly reported by professional programmers; in a 2006 Oriole technical blog article, a Java developer estimates that an average cycle takes 31 minutes and a short cycle takes 6.5 minutes [7].

We believe that this rapid iteration is a consequence of both copy-and-paste programming in general and programmers' tendencies to attempt rule-based performance using unfamiliar tools (both discussed in Section 2.2). This desire for rapid iteration has clear implications for tool selection as well. Programmers tend to, for example, prefer interpreted languages over compiled languages for opportunistic programming, favoring human productivity much more than code execution speed [7, 8].

2.4 Consider code impermanent

Developers often consider the code they write opportunistically to be impermanent, since often times the code will only be used once (*e.g.*, to run an experiment), and sometimes the code may not even be used at all! For example, one Exploratorium exhibit designer reported that the only time he reuses code is when "[he] wrote it for the last project [he] worked on". Otherwise, code reuse is "just too much trouble." Furthermore, opportunistic programming is often used to ideate and explore the design space when prototyping — it is a kind of "breadth-first" programming where many ideas are thrown away early.

More interesting, however, are the consequences of this perceived impermanence. First, code written opportunistically receives little to no documentation. An exhibit designer at the Exploratorium remarked that it simply wasn't worth his time to document code because "[he] ended up throwing so much away". During the development process, it isn't clear what code will be kept, so programmers choose to document nothing rather than to waste effort. At first, this lack of documentation doesn't seem like a problem. However, the code is only *perceived* as impermanent. During a talk in 2007, Joshua Schachter, creator of del.icio.us, described how the service had been written in a style that we would classify as opportunistic programming: The project started out as nothing more than a 100-line Perl script for sharing his bookmarks with his friends. His company recently decided to completely rewrite their software because the current system serving millions of users had become a mass of "terrifying Perl code" [11].

This perceived impermanence also leads to what we call *code satisficing*. Programmers will often implement functionality in a sub-optimal way during opportunistic development in order to maintain flow [2]. In one example of code satisficing from our laboratory study, the participant was attempting to implement a fixed-length queue using an array. She was a novice PHP programmer, but a very experienced programmer overall. She took a guess at PHP array notation, and guessed wrong. Instead of looking up the notation, she decided to create ten global variables, one for each element of the "array". She commented that "[she knew] there was a better way to do this" but "didn't want to be interrupted". However, this led to problems down the road. She made a typographical error when implementing the *dequeue* operation that took her over ten minutes to debug and clearly broke her flow.

2.5 Face unique debugging challenges

Opportunistic programming leads to unique debugging challenges. First, because programmers often employ a federation of languages (as mentioned in Section 2.1), they often cannot make effective use of sophisticated debugging tools intended for a single language. They are thus forced to make state and control flow changes visible through mechanisms like `print` statements. During our laboratory study, we observed that people who were better at opportunistic programming would do things to make state visible *while* adding new functionality. For example, they would insert `print` statements preemptively "just in case" they had to debug later. Individuals who were less experienced would have to do this after a bug occurred, which was much more time consuming. Interestingly, the less experienced programmers spent a significant amount of time trying to determine if a block of code they had just written was even executing, let alone whether it was correct! (We are currently analyzing data to make more precise claims about this behavior.)

Second, because there is little or no upfront design, pieces of the system often do not have clean interfaces (*e.g.*, communication between functions often might get done via a global variable.) This makes debugging more difficult, because the programmer must maintain a mental model of the entire system, not just of the particular component she is currently debugging.

3. FUTURE RESEARCH

We conclude with a brief discussion of our planned future research. Once we have completed our fieldwork and finished analyzing the data from our laboratory study, our next goal will be to understand how to better develop tools that are specifically intended to support opportunistic programming. So far, we have identified three broad areas that would benefit from better tool support:

Debugging — Debugging in opportunistic programming is made difficult for a number of reasons: Many languages are used in a single project, code satisficing leads to code that is not well encapsulated, and developers often refuse to invest time in learning complex (but powerful) tools. We believe that an ideal debugging solution would be language independent (or at the very least, work for many languages), and work across control-flow boundaries when multiple languages are "glued" together. Additionally, we believe that debugging tools could better leverage the rapid iteration inherent in opportunistic development — *e.g.*, code that was written 30 seconds ago is likely the code that the programmer wants to test and debug.

Code Foraging and Reuse — While there has been a large amount of recent research on tools to aid searching for code, the vast majority of these tools focus on finding and presenting these code snippets. We believe there are significant opportunities to support the process of comparing, reasoning about, integrating, and modifying found code. Additionally, we believe that there may be opportunities to combine the process of searching one's past projects and searching the Internet for functionality.

Documentation — Although much of the *code* that is written during opportunistic programming is thrown away, the process itself is extremely valuable. An exhibit designer at the Exploratorium commented that while he rarely wanted to go back and look at code from prior projects, he often wanted to review the *process* by which he did something. We believe there are many interesting questions regarding both what should be documented during opportunistic programming, and how best to produce that documentation.

There has, of course, been a great deal of research on these topics in both the fields of traditional software engineering and end-user software engineering. We believe that by building upon the research from both of these communities as well as leveraging what we are currently learning through our fieldwork and exploratory studies, we can create tools which improve the experience of developers engaging in opportunistic programming.

4. REFERENCES

[1] S. Clarke. What is an end-user software engineer? In *End-User Software Engineering Dagstuhl Seminar*, Dagstuhl, Germany, 2007.

[2] M. Csíkszentmihályi. *Flow: The Psychology of Optimal Experience*. Harper Collins, New York, NY, USA, 1990.

[3] B. Hartmann, S. Doorley, and S. R. Klemmer. Hacking, mashing, and gluing: A study of opportunistic design and development. Technical Report 2006-14, Stanford HCI Group, 2006. `http://hci.stanford.edu/cstr/reports/2006-14.pdf`.

[4] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[5] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the International Conference on Software Engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[6] D. A. Norman. *Things that Make Us Smart*, chapter 3, pages 43–76. Perseus Books, New York, NY, USA, 1993.

[7] T. M. O'Brien. Dead time (...code, compile, wait, wait, wait, test, repeat). `http://www.oreillynet.com/onjava/blog/2006/03/dead_time_code_compile_wait_wa.html`.

[8] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, pages 23–30, 1998.

[9] P. L. T. Pirolli. *Information Foraging Theory*. Oxford University Press, Oxford, England, 2007.

[10] J. Reason. *Human Error*. Cambridge University Press, Cambridge, England, 1990.

[11] J. Schachter. Guest lecture on creating del.icio.us. *Stanford CS343 course: What Do Great Software Developers Know?* `http://cs343-spr0607.stanford.edu/index.php/Writeups:Joshua_Schachter`.