

EXAMPLE-CENTRIC PROGRAMMING:
INTEGRATING WEB SEARCH INTO THE DEVELOPMENT PROCESS


ADISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTERSCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Joel R. Brandt
December 2010

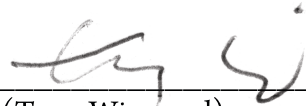
© Joel R. Brandt 2010
All Rights Reserved

Joel R. Brandt


I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.


(Scott R. Klemmer) Principal Advisor

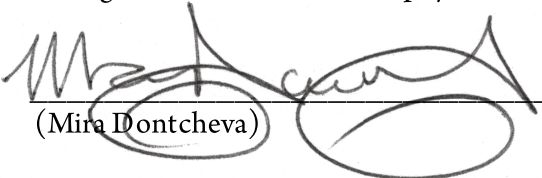
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.


(Terry Winograd)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.


(John K. Ousterhout)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.


(Mira Dontcheva)

Approved for the Stanford University Committee on Graduate Studies

ABSTRACT

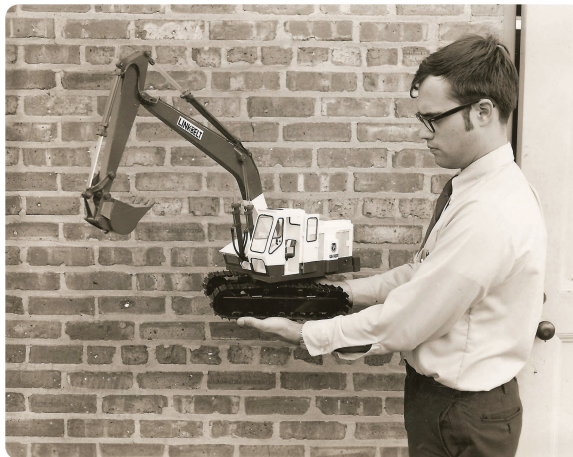
The Web is fundamentally changing programming. The increased prevalence of online source code—shared in code repositories, documentation, blogs and forums—enables programmers to build applications opportunistically by iteratively searching for, modifying, and combining examples. These Web resources are a ubiquitous and essential part of programming: in one of our studies, programmers spent 19% of their time consuming relevant online information. But our development tools haven't yet embraced these changes. How do we leverage the latent opportunity of Web-based example code in the next generation of programming tools?

This dissertation explores the roles that online resources play in creating software, making contributions in three areas. First, it presents a taxonomy of programmer Web usage. Programmers turn to the Web with a variety of goals: they learn new skills, transfer knowledge to new domains, and delegate their memory to the Web. Using our taxonomy, we suggest opportunities for tool support of programmer Web usage.

Second, this thesis contributes interaction techniques for lowering the cost of *locating* relevant example code on the Web. We created Blueprint, a task-specific search engine that embeds Web search inside the development environment. A laboratory study and large-scale deployment of Blueprint found that it enables participants to write significantly better code and find example code significantly faster than with a standard Web browser and search engine, and may cause a fundamental shift in how and when programmers search the Web.

Finally, this thesis contributes interaction techniques for helping programmers *understand* examples. Efficient understanding and effective adaptation 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. By augmenting the code authoring experience with execution visualization and linking of related lines, programmers can understand examples significantly faster.

*For my father, Richard Brandt
who enjoyed building things even more than I do*



ACKNOWLEDGMENTS

Thank you to all of my advisors, both formal and informal, who helped me through the Ph.D. process. Scott Klemmer provided amazing guidance, and most importantly, helped me develop good taste as a researcher. Mira Dontcheva both provided excellent guidance and put in a great deal of the grunt work necessary to make the systems presented in this dissertation a reality. I cannot overstate the amount of appreciation I feel for the help of these two individuals, and for the opportunities they gave me. Thank you as well to John Ousterhout and Terry Winograd. They were incredibly supportive throughout my Ph.D., and most importantly, were adept at asking the really hard questions that motivate great research. Early in my research career, Pat Hanrahan showed me by example what it meant to have intellectual curiosity.

Thank you, as well, to all of my collaborators: Björn Hartmann, Philip Guo, Joel Lewenstein, Marcos Weskamp, Iván Caveró Belaunde, William Choi, Ben Hsieh, Vignan Pattamatta, and Noah Weiss. It was a pleasure working with all of them. And, thank you to the friends who made the journey worthwhile: Jacob Leverich, Theresa Curtis, Jeff McMahan, Kat White, Wilmot Li, and Sharma Hendel.

Most of all, thank you to my family. My mother and father, Carol and Richard, and my brother Eric have provided encouragement through the hard times and shared in my joy during the good times. And, thank you to Lindsay. I hope I can be half as supportive of her in her future endeavors as she has been of me. Finally, thank you to my dog Jake — he liked the first draft of my thesis better than anyone else.

I was supported by the Andreas Von Bechtolsheim Fellowship Fund, part of the Stanford Graduate Fellowship program, during my final two years at Stanford. I was also partially funded by the Stanford MediaX organization and by NSF Grant IIS-0745320 while completing this research. The Blueprint project was further supported by Adobe Systems, Inc. Intel donated hardware for multiple projects.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION	1
1.1 Thesis Contributions	2
1.2 The Programmer as a Knowledge Worker	3
1.3 Solution Overview and Dissertation Roadmap	3
1.3.1 Understanding Web Use During Opportunistic Programming	4
1.3.2 Tool Support for Example-Centric Programming	5
1.4 Statement on Multiple Authorship and Prior Publications	6
CHAPTER 2 RELATED WORK.....	8
2.1 Barriers in Programming.....	9
2.2 Role of the Web in Overcoming Barriers	9
2.3 Example-Centric Development	10
2.3.1 Sources of Example Code	11
2.3.2 Task-Specific Search Interfaces	12
2.3.3 Tools for Understanding Examples	13
CHAPTER 3 OPPORTUNISTIC PROGRAMMING	15
3.1 Hacking in the Wild: Fieldwork with Museum Exhibit Designers.....	16
3.2 Opportunistic Programming in the Lab	17
3.3 Characteristics of Opportunistic Programming	17
3.3.1 Glue Together High-Level Components that Fit the Task	17
3.3.2 Leverage Examples to Add New functionality via Copy-and-Paste	21
3.3.3 Iterate Rapidly	22
3.3.4 Consider Code Impermanent	23

3.3.5	Face Unique Debugging Challenges	25
CHAPTER 4 UNDERSTANDING HOW PROGRAMMERS USE THE WEB		26
4.1	Study 1: Examining Web Use in the Lab	26
4.1.1	Method	26
4.1.2	Results	31
4.1.2.1	Goals driving Web use	32
4.1.2.2	Just-in-time learning of new skills	32
4.1.2.3	Clarification of existing knowledge	35
4.1.2.4	Reminders about forgotten details	37
4.2	Study 2: Web Search Log Analysis	39
4.2.1	Method	40
4.2.1.1	Determining query type	42
4.2.1.2	Determining query refinement method	42
4.2.1.3	Determining Web page type	43
4.2.2	Results	43
4.2.2.1	Programmers rarely refine queries, but are good at it	46
4.2.2.2	Query type predicts types of pages visited	47
4.3	Limitations of Our Findings	47
4.4	Five Key Insights and Implications for Tools	48
CHAPTER 5 BLUEPRINT: INTEGRATING WEB SEARCH INTO THE DEVELOPMENT ENVIRONMENT		51
5.1	Scenario: Developing with Blueprint	54
5.2	Implementation	55
5.2.1	Client-Side Plug-In	56

5.2.2	Blueprint Server	57
5.2.3	Extracting Example Code and Descriptions	58
5.2.3.1	Classifying example code	60
5.2.3.2	Extracting text and running examples	61
5.2.3.3	Keeping track of changes to examples	62
5.3	Evaluation: Studying Blueprint in the Lab	62
5.3.1	Method	62
5.3.2	Results	64
5.3.2.1	Directed task	64
5.3.2.2	Exploratory task	65
5.3.2.3	Areas for improvement	66
5.3.3	Discussion	66
5.3.3.1	Where Blueprint fails	67
5.4	Design Space of Web Tools for Programmers.....	68
CHAPTER 6 LONGITUDINAL STUDY OF BLUEPRINT: DEPLOYMENT TO 2,024 USERS		70
6.1	Insights from Interviewing Active Users	72
6.1.1	The Benefits of Consistent, Example-Centric Results Outweigh the Drawbacks of Missing Context.	72
6.1.2	Blueprint is Symbiotic with Existing IDE Features	73
6.1.3	Blueprint is Used Heavily for Clarifying Existing Knowledge and Reminding of Forgotten Details.	73
6.2	Method	74
6.3	Results	75
6.4	Exploratory Analysis.....	76

6.4.1	Using Blueprint as a Resource to Write Code by Hand is Common.....	77
6.4.2	People Search for Similar Things Using Blueprint and Community Help, but the Frequencies are Different.....	77
6.4.3	Both Interface Modalities are Important	78
6.5	User Retention	79
6.6	Conclusion	79
 CHAPTER 7 REHEARSE: HELPING PROGRAMMERS UNDERSTAND EXAMPLES		80
7.1	Observing Example Adaptation	82
7.1.1	Observations	83
7.2	Rehearse	84
7.2.1	Execution Highlighting	84
7.2.2	Related Lines	85
7.2.3	Implementation	85
7.3	Pilot Study of Rehearse	86
7.3.1	Method	86
7.3.2	Results	87
7.3.3	Discussion	88
7.4	Conclusion	89
 CHAPTER 8 FUTURE DIRECTIONS		90
8.1	Toward a Complete Picture of Knowledge Work on the Web.....	90
8.2	Further Tool Support for Opportunistic Programming.....	91
8.3	The Future of Programming.....	94
 REFERENCES		95

LIST OF FIGURES

Figure 3.1: The Exploratorium Museum in San Francisco, California. 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.....	16
Figure 3.2: 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.....	21
Figure 3.3: Histogram of per-subject edit-debug cycle times in our laboratory study. Each bar represents one subject. 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. Across all subjects, 80% of edit-debug cycles were less than 5 minutes in length.....	23
Figure 4.1: List of chat room features that lab study participants were asked to implement. The first four features are fairly typical; the fifth, retaining a limited chat history, is more unique.....	28
Figure 4.2: Screenshot of one participant's completed chat room. This participant met all of the specifications.	29
Figure 4.3: Overview of when participants referenced the Web during the laboratory study. Subjects are sorted by total amount of time spent using the Web. Web use sessions are shown in light blue, and instances of Web search are shown as dark bars.....	30
Figure 4.4: All 360 Web use sessions amongst the 20 participants in our lab study, sorted and plotted by decreasing length (in seconds). The left vertical bar represents the cutoff separating the 10% longest sessions, and the right bar the cutoff for 50% of sessions. The dotted line represents a hypothetical uniform distribution of session lengths.....	31
Figure 4.5: Example of how participants used Web search to remind themselves of forgotten low-level syntax. Here, the programmer forgot the exact name of the function used to access the result of a database query. Searching for “php	

fetch array” allowed him to quickly retrieve the exact name (highlighted) without visiting any additional Web pages.....	38
Figure 4.6: Web query result interface for Adobe’s <i>Community Help</i> search portal. This portal is implemented using a Google Custom Search Engine, and displays results in a format nearly identical to general-purpose search engines such as Google.	39
Figure 4.7: How query types changed as queries were refined. In both graphs, each bar sums all <i>i</i> th queries over all sessions that contained an <i>i</i> th query (e.g., a session with three queries contributed to the sums in the first three bars). The graph on the left is a standard histogram; the graph on the right presents the same data, but with each bar’s height normalized to 100 to show changes in proportions as query refinements occurred.....	46
Figure 5.1: The Blueprint plug-in for the Adobe Flex Builder development environment helps programmers locate example code. A hotkey places a search box (A) at the programmer’s cursor position. Search results (B) are example-centric; each result contains a brief textual description (C), the example code (D), and, when possible, a running example (E). The user’s search terms are highlighted (F), facilitating rapid scanning of the result set. Blueprint allows users to rate examples (G).....	52
Figure 5.2: Example-centric programming with Blueprint. The user presses a hotkey to initiate a search; a search box appears at the cursor location (1). Searches are performed interactively as the user types; example code and running examples (when present) are shown immediately (2). The user browses examples with the keyboard or mouse, and presses Enter to paste an example into her project (3). Blueprint automatically adds a comment containing metadata that links the example to its source (4).	53
Figure 5.3: Architecture of the Blueprint system. The process of servicing a user’s query is shown on the left; the background task of parsing Web pages to extract examples is shown on the right.....	56
Figure 5.4: Comparative laboratory study results. Each graph shows the relative rankings of participants. Participants who used Blueprint are shown as filled squares, those who used Community Help are shown as open squares.	65
Figure 5.5: Design space of tools to aid programmers’ Web use. Blueprint is designed to address the portion of the space shown with a shaded background.	69
Figure 6.1: Screenshot of the Blueprint Web page on Adobe Labs. Blueprint was made publicly available on May 27, 2009.	71

- Figure 6.2:** Comparison of Blueprint (left) and Community Help (right) search result interfaces for the query “Alert”. The desired information is immediately available in Blueprint; Community Help users must click the first result and scroll part way down the page to find the same information..... 78
- Figure 7.1:** 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)..... 81
- Figure 7.2:** 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..... 85

LIST OF TABLES

Table 4.1: Demographic information on the 20 participants in our lab study. Experience is given in number of years; self-rated proficiency uses a Likert scale from 1 to 7, with 1 representing “not at all proficient” and 7 representing “extremely proficient”.	27
Table 4.2: Summary of characteristics of three points on the spectrum of Web use goals.	33
Table 4.3: For hand-coded sessions of each type, proportion of first queries of each type (252 total sessions). Statistically significant differences between columns are shown in bold, * entry means only significant at $p < 0.05$.	44
Table 4.4: For queries in hand-coded sessions of each type, proportion of result clicks to Web sites of each type (401 total queries). Statistically significant differences between columns are shown in bold.	44
Table 4.5: For each refinement type, proportion of refinements of that type where programmers clicked on any links prior to the refinement (31,334 total refinements).	44
Table 4.6: For queries of each type, proportion of result clicks leading programmer to Web pages of each type (107,343 total queries). Statistically significant differences between columns are shown in bold.	44
Table 7.1: Task completion times for treatment (T) and control (C) participants. Participants using Rehearse completed the first task faster than those in the control condition ($p < 0.06$).	88

CHAPTER 1 INTRODUCTION

The Web is fundamentally changing programming [44, 79]. The increased prevalence of online source code—shared in code repositories, documentation, blogs and forums [1, 2, 24, 71]—enables programmers to opportunistically build applications by iteratively searching for, modifying, and combining examples [19, 40]. Consider Jenny, a programmer we observed in the lab. At one point during the programming process, she exclaimed “Good grief, I don’t even remember the syntax for forms!” Less than a minute after this outburst, she had found an example of an HTML form online, successfully integrated it into her own code, adapted it for her needs, and moved onto a new task. As she continued to work, she frequently interleaved foraging for information on the Web, learning from that information, and authoring code. Over the course of two hours, she turned to the Web for help 27 times, accounting for 28% of the total time she programming.

Reliance on the Web for instructive examples is a key part of what we call *opportunistic programming*. This approach emphasizes speed and ease of development over code robustness and maintainability [19, 40]. Programmers engage in opportunistic

programming to *prototype, ideate, and discover*—to address questions best answered by creating a piece of functional software. This type of programming is widespread, performed by novices and experts alike: it happens when designers build functional prototypes to explore ideas, when scientists write code to control laboratory experiments, when entrepreneurs assemble complex spreadsheets to better understand how their business is operating, and when professionals adopt agile development methods to build applications quickly [19, 61, 64, 76].

1.1 THESIS CONTRIBUTIONS

This dissertation provides an understanding of how programmers use online resources to support an opportunistic approach to development. Specifically, it offers contributions in three areas:

1. *A taxonomy of programmer Web usage* — Programmers turn to the Web with a variety of goals: they learn new skills, transfer knowledge to new domains, and delegate their memory to the Web. Using this taxonomy, we suggest opportunities for tool support of programmer Web usage.
2. *Interaction techniques for lowering the cost of locating relevant example code* — Embedding a task-specific search engine in the development environment can significantly reduce the cost of finding information and thus enable programmers to write better code more easily. Moreover, lowering the cost of search can fundamentally change how programmers approach routine tasks.
3. *Interaction techniques for helping programmers understand examples* — Efficient understanding and effective adaptation 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. By augmenting the code authoring experience with execution visualization and linking of related lines, programmers can understand examples significantly faster.

1.2 THE PROGRAMMER AS A KNOWLEDGE WORKER

The Web is changing many types of knowledge work [18]. Going forward, Web resources are likely to play an increasingly important role in a broad range of problem solving domains.

This dissertation studies programmers as an exemplar form of knowledge worker.

Moreover, the ability to think computationally is becoming an important skill in many types of knowledge work [83]. Scaffidi, Shaw, and Myers estimate that in 2012 there will be 13 million people in the USA that describe themselves as “programmers” [76]. It seems likely that several times that number will program to some extent. There is significant value in providing better tool support for this nascent population.

By using programming as a “petri dish,” we endeavor to contribute broadly applicable principles that further our understanding of knowledge work on the Web.

1.3 SOLUTION OVERVIEW AND DISSERTATION ROADMAP

We begin by reviewing related work (Chapter 2). The remainder of the thesis is divided into two parts: *empirical work* that aims to understand how and why programmers use the Web, and *tool building* that aims to support and amplify the role that Web-based example code plays in the development process.

1.3.1 UNDERSTANDING WEB USE DURING OPPORTUNISTIC PROGRAMMING

We conducted three studies to better understand opportunistic programming, and specifically, the role that the Web plays in this approach. We began by conducting fieldwork with exhibit designers at the Exploratorium Museum in San Francisco, California (Chapter 3). Designers routinely faced the “build or borrow” question [13]: build a piece of functionality from scratch, or locate and adapt existing systems? Designers showed a distinct preference for bricolage, often combining and tailoring many off-the-shelf components in Rube-Goldberg-esque systems rather than building from scratch [39, 55, 59, 82, 84].

After completing our fieldwork, we studied opportunistic development in the lab (§4.1). We observed 20 programmers as they each spent 2.5 hours building a Web-based chat room application. All participants used online resources extensively, accounting for 19% of the time they spent programming. More specifically, programmers leveraged online resources with a range of goals: They engaged in *just-in-time learning* of new skills and approaches, *clarified and extended* their existing knowledge, and *reminded* themselves of details deemed not worth remembering.

Does programmers’ use of the Web “in the wild” have the same range of goals, or is this result an artifact of the particular lab setting? (Perhaps, for example, the performance pressure of being observed encouraged learning over trial-and-error experimentation.) To answer this, a second study (§4.2) analyzed one month of queries to an online programming portal, examining the lexical structure of queries, types of refinements made, and classes of result pages visited. These logs also exhibited traits that suggest the Web is being used for learning and reminding.

The results from these studies contribute to a theory of online resource usage in programming, and suggest opportunities for tools to facilitate online knowledge work.

1.3.2 TOOL SUPPORT FOR EXAMPLE-CENTRIC PROGRAMMING

Despite the demonstrated importance of Web resources, current search tools are not designed to assist with programming tasks and are wholly separate from editing tools. Chapter 5 explores the hypothesis that embedding a task-specific search engine in the development environment can significantly reduce the cost of finding information and thus enable programmers to write better code more easily.

We describe the design, implementation, and evaluation of Blueprint, a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code. Blueprint *automatically augments queries with code context*, presents a *code-centric view of search results*, *embeds the search experience into the editor*, and retains a *link between copied code and its source*. A comparative laboratory study (§5.3) found that Blueprint enables participants to write significantly better code and find example code significantly faster than with a standard Web browser.

Our laboratory study of Blueprint suggested that it helped programmers write code more quickly. But we also wondered: After programmers integrated an example-centric search tool into their daily practice, would they then approach the task of programming differently? A large-scale longitudinal deployment of Blueprint addressed this question. Chapter 6 details our one-and-a-half-year deployment, focusing specifically on a comparative analysis of the first three months of log data from approximately 2,024 users. With Blueprint, programmers use search more frequently, and for a broader range of tasks.

Improved Web search tools like Blueprint enable programmers to quickly *locate* relevant examples. However, existing code editors offer little support for helping users *understand* examples. Chapter 7 proposes that adapting an example quickly and accurately 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 Rehearse adapted example code significantly faster than those using an identical editor without these features.

This thesis concludes with directions for future research (Chapter 8). Specifically, it suggests a number of empirical questions aimed at gaining a more complete picture of knowledge work on the Web. It also examines the issue of tool support for opportunistic programming more broadly, suggesting a number of directions for research outside the domain of example-centric programming.

1.4 STATEMENT ON MULTIPLE AUTHORSHIP AND PRIOR PUBLICATIONS

The research presented in this dissertation was completed with the help of many talented individuals. While I initiated and led the projects described here, I want to acknowledge all of my collaborators. Without them, this research could not have been realized. Specifically, the fieldwork presented in Chapter 2 was completed with the help of Indrajit Khare, and was

supported by William Meyer from the Exploratorium. Philip Guo and Joel Lewenstein contributed heavily to the empirical work presented in Chapter 4 and Marcos Weskamp and Iván Cavero Belaunde were instrumental in designing and deploying Blueprint at large scale. Vignan Pattamatta, Ben Hsieh, and William Choi all contributed to Rehearse. Finally, my advisors Mira Dontcheva and Scott R. Klemmer played a very important role in all of the research presented in this dissertation.

This dissertation is partially based on papers published previously in ACM conference proceedings and IEEE publications. I am the primary author on all of these publications. Specifically, the studies presented in Chapter 4 were published at CHI 2009 [11], and the Blueprint system was published at CHI 2010 [12]. Background on Opportunistic Programming was published in IEEE Software [10]. A paper describing Rehearse (Chapter 7) is under submission at the time of publication of this dissertation.

CHAPTER 2 RELATED WORK

Assimilating knowledge is a key part of the programming process [25, 62]. For example, when adding functionality to an existing system, programmers must first understand how relevant parts of the existing system work, and how new libraries or frameworks will interact (Chapter 4). Similarly, when debugging, programmers must reconcile opaque information about execution state with an often incomplete understanding of how the code works [51]. Ever since Grace Hopper and her colleagues created the first compiler [46], programmers have been relying on tools to help manage this knowledge work.

This chapter first explores the barriers that programmers face when trying to assimilate knowledge. We then look at how examples can play a role in overcoming these barriers. Examples aid in analogical reasoning, and allow people to avoid “re-inventing the wheel” by copying prior successful actions. Finally, we survey existing tools for helping programmers leverage examples.

2.1 BARRIERS IN PROGRAMMING

Ko *et al.* observed novice programmers for a semester as they learned to use Visual Basic .NET [50]. The researchers classified all occurrences of *insurmountable barriers*, defined as problems that could only be overcome by turning to external resources. They identified six classes of barriers. Below are the six barriers, each with an example of how the barrier might be stated by a programmer:

Design — I don't know what I want the computer to do.

Selection — I think I know what I want the computer to do, but I don't know what to use.

Coordination — I think I know what things to use, but I don't know how to make them work together.

Use — I think I know what to use, but I don't know how to use it.

Understanding — I thought I knew how to do this, but it didn't do what I expected.

Information — I think I know why it didn't do what I expected, but I don't know how to check.

2.2 ROLE OF THE WEB IN OVERCOMING BARRIERS

Stylos and Myers offer evidence that the Web is being used to overcome many of these barriers [79]. Specifically, general-purpose Web search is frequently used to enumerate possible libraries when dealing with a "Selection" barrier. Once programmers have decided on a library, they often encounter "use" and "understanding" barriers. In these situations, programmers find examples useful. Unfortunately, official documentation rarely contains examples, and so programmers again turn to Web search to find third-party example code. Hoffmann *et al.* provide further support for the import of the Web in overcoming barriers [44]. They classified Web search sessions about Java programming into 11 search goals (*e.g.*,

beginner tutorials, APIs, and language syntax). They found that 34% were seeking API documentation (likely “Selection” or “Use” barriers) and 21% were seeking troubleshooting information (likely “Understanding” or “Information” barriers).

The Web is an important tool in overcoming barriers during programming. A goal of this dissertation is to inform the design of better tools by providing a richer picture of *how* and *why* programmers leverage Web content.

2.3 EXAMPLE-CENTRIC DEVELOPMENT

When programmers turn to online resources, they exhibit a strong preference for example code over written descriptions of how to do a task [44, 79]. Why might this be? Examples play an important role in analogical reasoning [32]: it is much more efficient to adapt someone else’s successful solution than it is to start from scratch [66].

Because examples support creation, many programmers make examples a central part of their development practice. In a recent study of programmers learning to use a new UI toolkit, over one-third of participants’ code consisted of modified versions of examples distributed with the toolkit [86]. In our studies (Chapter 4), we observed that programmers frequently compared multiple similar examples during a single task. The use of comparison has been shown to help people extract high-level principles [33]. As Gentner writes, “comparison processes can reveal common structure ... even early in learning when neither example is fully understood.” [31].

Examples may also help people avoid mistakes. 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) [72]. We suggest that making examples a central part of their programming practice allows developers to engage in rule-based performance more often 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.

Effecting support for example-centric development requires two things. First, there must be a source of relevant examples. Second, there must be a search interface that makes specifying queries and evaluating results efficient.

2.3.1 SOURCES OF EXAMPLE CODE

Several systems use data-mining techniques to locate or synthesize example code [75, 81].

XSnippet uses the current programming context of Java code (*e.g.*, types of methods and variables in scope) to automatically locate example code for instantiating objects [75].

Mandelin *et al.* show how to automatically synthesize a series of method calls in Java that will transform an object of one type into an object of another type, useful for navigating large, complex APIs[60]. A limitation of this approach is that the generated code lacks the comments, context, and explanatory prose found in tutorials.

An alternative is to use regular Web pages (*e.g.*, forums, blogs, and tutorials) as sources for example code. We believe using regular Web pages as sources for example code has two

major benefits: First, it may provide better examples. Code written for a tutorial is likely to contain better comments and be more general purpose than code extracted from an open source repository. Second, because these pages also contain text, programmers can use natural language queries and general-purpose search engines to find the code they are looking for.

2.3.2 TASK-SPECIFIC SEARCH INTERFACES

Increasing reliance on the Web for learning is happening in a wide variety of knowledge work domains. As a result, there has been recent interest in creating domain-specific search interfaces (*e.g.*, [8, 43, 63, 80, 85]). The Blueprint system presented in Chapter 5 follows the template-based approach introduced by Dontcheva *et al.* [26, 27]. Displaying diverse results in a consistent format through templates enables users to rapidly browse and evaluate search results.

There are research [9, 44, 79] and commercial [1-3] systems designed to improve search and code sharing for programmers. While these search engines are *domain-specific* (that is, they are for programming), they are designed to support a broad range of tasks (*e.g.*, finding libraries, looking up the order of method parameters, etc.). We suggest that there might be benefit in designing a *task-specific* search interface oriented specifically towards finding example code. This introduces a trade-off: a task-specific interface can be highly optimized for a specific task, but will lose generality.

Current domain-specific search engines for programmers are *completely independent* of the user's development environment. What benefits might be realized by bringing search into the development environment? CodeTrail explores the benefits of integrating the Web

into the development environment by linking the Firefox browser and Eclipse IDE [34]. We suggest there may be benefits from going one step further by placing search *directly inside* the development environment. Again, this introduces a trade-off: such an integration will give up the rich interactions available in a complete, stand-alone Web browser in favor of a more closely-coupled interaction for a specific task.

2.3.3 TOOLS FOR UNDERSTANDING EXAMPLES

After a programmer *locates* helpful a example, she must then work to *understand* both the example code and the existing code in her project. To help with this task, many programming-by-demonstration (PBD) tools provide a visual link between source code and execution at runtime [22, 23, 54]. For example, Koala [58] and Vegemite [56], 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 [38] and Looking Glass (the successor to Storytelling Alice) [37]. 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. Current visualization techniques do not adapt easily to this use case.

An alternative to realtime visualization of execution 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 [69]. 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 [51]. 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.

CHAPTER 3 OPPORTUNISTIC PROGRAMMING

Quickly hacking something together can provide both practical and learning benefits [40]. Novices and experts alike often work opportunistically [19]: professional programmers and designers prototype to explore and communicate ideas [40, 47], scientists program laboratory instruments, and entrepreneurs assemble complex spreadsheets to better understand their business [77]. Their diverse activities share an emphasis on speed and ease of development over robustness and maintainability. Often, the code is used for just hours, days, or weeks.

This chapter introduces opportunistic programming by detailing key characteristics of this approach. Programmers exhibit a strong preference for working from examples over building from scratch. To facilitate this, they leverage the Web for both the examples themselves, and for the instructional content that helps them put the examples to use.

3.1 HACKING IN THE WILD: FIELDWORK WITH MUSEUM EXHIBIT DESIGNERS

We first observed the practice of opportunistic programming while conducting fieldwork with exhibit designers at the Exploratorium Museum in San Francisco, California. The Exploratorium is a hands-on museum about science and art. All of the exhibits are developed in-house (see **Figure 3.1**), and the majority of these exhibits have interactive computational components.

Exhibit designers are responsible for conceiving and implementing interactive exhibits that will convey a particular scientific phenomenon. Many of these exhibits require



Figure 3.1: The Exploratorium Museum in San Francisco, California. 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.

custom software. For example, an exhibit on microscopy required exhibit designers to retrofit a research-grade microscope with a remote, kid-friendly interface. While designers must construct working exhibits, 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 focus on exploring many ideas as rapidly as possible over ensuring robustness and maintainability.

3.2 OPPORTUNISTIC PROGRAMMING IN THE LAB

To get a more fine-grained understanding of how people work opportunistically, we performed a lab study with 20 programmers. In this study, participants prototyped a Web-based chat room using HTML, PHP, and JavaScript. We provided participants with five broad specifications, like “the chat room must support multiple concurrent users and update without full page reloads.” We report on this study in depth in Chapter 4; we offer a few high-level results here to support our fieldwork findings and as motivation for the rest of the thesis.

3.3 CHARACTERISTICS OF OPPORTUNISTIC PROGRAMMING

This section presents characteristics of an opportunistic approach to programming.

3.3.1 GLUE TOGETHER HIGH-LEVEL COMPONENTS THAT FIT THE TASK

At the Exploratorium, we observed that designers selected self-contained building blocks in order to build systems largely by writing “glue” code. For example, a nature observation exhibit called “Out Quiet Yourself” teaches individuals 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. All of the sound processing necessary for this exhibit could have been completed in software using the same computer that displays the total. Instead, the exhibit designers chose to use a series of hardware audio compressors and mixers to do the majority of the processing. They 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.

We observed that participants were most successful at bricolage development when components were fully functioning systems by themselves. For example, we asked the designer of the “Out Quiet Yourself” exhibit why he chose to use specialized audio hardware instead of a software library. He explained that the hardware could be experimented with independently from the rest of the system, which made understanding and tweaking the system much easier.

In contrast, a digital microscope exhibit contained an off-the-shelf controller that made it possible to adjust microscope settings (*e.g.*, slide position and focus) programmatically. This controller was driven by a custom piece of C++ code written by a contractor several years ago, and could not be used without this software. When the remote control of the microscope malfunctioned, it was very difficult to debug. Was the problem with the controller itself (*e.g.*, a dead motor) or was there a bug in the code? Answering such questions was difficult because components could not be used independently and no current employees understood the custom software.

In general, gluing together fully-functioning systems helps reduce several of the barriers that less-experienced programmers face [50]: First, because whole systems are easy

to experiment with, programmers are able to more easily understand how the pieces work, and can immediately intuit how to use them. Second, because there is a clearly defined boundary between each piece, programmers avoid coordination barriers. There is exactly one way to connect the pieces, and it is easy to see what is happening at the connection point.

Two important considerations when selecting components are the designer's *familiarity* with the component and the component's *fitness* to the designer's task. We call this the familiarity/flexibility trade-off. What factors affect the relative weight of these considerations? 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 chose to learn 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 example, when one participant in our lab study was asked if he knew of libraries to make AJAX calls (a particular programming paradigm used in the study) 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 libraries for AJAX, and in all cases these individuals already had significant experience with those libraries.

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 lab study, three individuals did this, and two of those failed to meet some of the specifications. Leveraging an existing system allowed them to make quick initial progress, but made the last mile difficult. For example, one participant built upon an existing content-management system with a chat module that met all but two of the specifications. He spent 20 minutes finding and 10 minutes installing the system, meeting the first three specifications faster than all other participants. However, it took him an additional 58 minutes to meet one more specification (adding timestamps 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, though not as dramatic, frustrations.

The distinction between *reusing* and *tailoring* an existing system is subtle but important. To *reuse* a system, the programmer only needs to understand how to use its interface; to *tailor* a system, the programmer needs to understand how it is built. The main challenge of tailoring an existing system is building a mental model [30] 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 authors of the original code are available for consultation, mental model formation can still take a considerable amount of time [52]. 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 grains of sand one at a time.

3.3.2 LEVERAGE EXAMPLES TO ADD NEW FUNCTIONALITY VIA COPY-AND-PASTE

Even when programmers build software from existing components, some glue code must be written to hook these pieces together. Participants would frequently write this glue code by iteratively searching for, copying, and modifying short blocks of example code (< 30 lines) with desired functionality. We call this approach *example-centric programming*.

Example-centric programming is particularly beneficial when working in an unfamiliar domain: modifying examples is easier than writing the code by oneself. For example, the majority of participants in our lab study who were unfamiliar with the AJAX programming paradigm chose to copy-and-paste snippets of AJAX setup code rather than try to learn to write it from scratch.

However, example-centric programming 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 3.2**. 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 looking at an example, even though he fully

```
<?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 3.2: 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.

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 opens up interesting questions on how programmers locate “promising” code. In opportunistic programming, we believe the primary source is through Web search. Indeed, in our laboratory study, participants used the Web a great deal: On average, each participant spent 19% of their programming time on the Web, spread out over 18 distinct sessions.

3.3.3 ITERATE RAPIDLY

Successful opportunistic programmers in our lab study favored a short edit-debug cycle.

Figure 3.3 presents an overview of the length of each participant’s edit-debug cycles. The graph shows that for the vast majority of subjects, 50% of the cycles were less than 30 seconds in length, and for all subjects, 80% of the 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 during traditional software engineering; in a 2006 O’Reilly technical blog article, a Java developer estimates that an average cycle takes 31 minutes and a short cycle takes 6.5 minutes [67].

We believe that frequent iteration is a necessary part of learning unfamiliar tools and understanding found code. Therefore, successful opportunistic programmers select tools that make iteration fast. For example, interpreted languages are preferred over compiled languages because they emphasize human productivity over code execution speed [70]. Recently, the commercial software development has begun to embrace this observation. As

just one datapoint, 63% of the code in Adobe's flagship photo management application, Lightroom 2.0, is written in a scripting language [29].

3.3.4 CONSIDER CODE IMPERMANENT

Code written opportunistically 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. Because much of the code they write opportunistically is thrown away, developers often consider code to be impermanent. This perception affects the way code is written in two important ways.

First, programmers spend little time documenting and organizing code that is written opportunistically. Interestingly, this is typically the right decision. An exhibit designer at the Exploratorium remarked that it simply wasn't worth his time to document

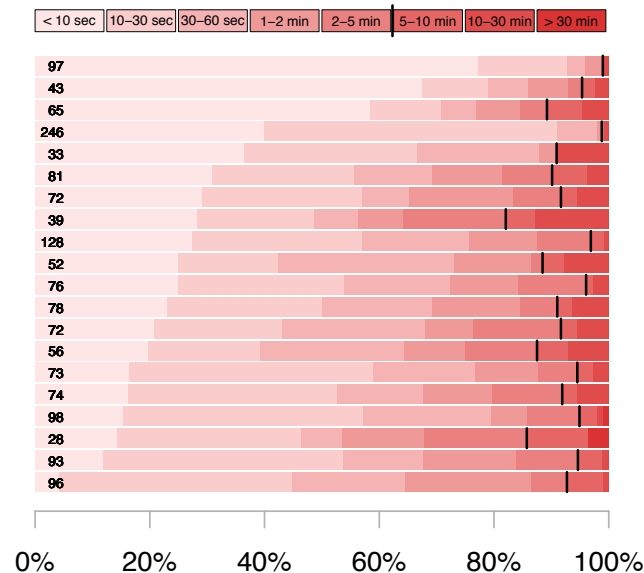


Figure 3.3: Histogram of per-subject edit-debug cycle times in our laboratory study. Each bar represents one subject. Total number of edit-debug cycles for each subject are given by the black number on each bar, and bar length is normalized across subjects. A black line separates cycles of less than and greater than 5 minutes. Across all subjects, 80% of edit-debug cycles were less than 5 minutes in length.

code because “[he] ended up throwing so much away”. Instead, 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 the strengths and weaknesses of a particular tool, or why a user interface was not successful. Reuse of code written opportunistically is rare. Another exhibit designer reported that the only time he reuses code is when “[he] wrote 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 both common and invaluable.

Second, the perceived impermanence of code written opportunistically 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 [21]. For example, a participant in our lab was attempting to implement a fixed-length queue using an array in order to store chat history. 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”. Initially, it appeared she had made the right decision, as she was able to test the history functionality only seconds later. 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. As this example illustrates, code satisficing can be both good and bad. Successful opportunistic programmers are good at weighing the trade-offs between implementing something “right” and implementing something “quickly”.

3.3.5 FACE UNIQUE DEBUGGING CHALLENGES

Opportunistic programming leads to unique debugging challenges. As mentioned above, programmers often glue together many disparate components. One consequence of this 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!) When projects employ a federation of languages, programmers often cannot make effective use of sophisticated debugging tools intended for a single language. Instead, they are 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!

CHAPTER 4 UNDERSTANDING HOW PROGRAMMERS USE THE WEB

This chapter presents the results of two studies that investigate how programmers leverage online resources. In the first study, we asked 20 programmers to rapidly prototype a Web application in the lab. In the second study, we conducted a quantitative analysis of a month-long sample of Web query data to better understand if our results generalized to the real-world. We employed this mixed-methods approach to gather data that is both contextually rich and authentic [16, 36].

4.1 STUDY 1: EXAMINING WEB USE IN THE LAB

We conducted an exploratory study in our lab to understand how programmers leverage online resources, especially during opportunistic programming.

4.1.1 METHOD

20 Stanford University students (3 female), all proficient programmers, participated in a 2.5-hour session. The participants (5 Ph.D., 4 Masters, 11 undergraduate) had an average of 8.3

Subject #	Experience (years)	Self-Rated Proficiency				Tasks Completed				
		HTML	JavaScript	PHP	AJAX	Username	Post	AJAX Update	Timestamp	History
1	11	7	4	6	5	●	●	●	●	●
2	17	5	4	2	1	●	●	●		●
3	13	7	5	5	2	●	●	●	●	
4	4	6	4	5	2	●	●	●	●	●
5	15	6	7	6	5	●	●	●	●	●
6	2	6	5	3	4	●	●	●	●	●
7	7	5	4	4	4	●	●	●	●	●
8	8	5	2	4	2	●	●	●		
9	5	7	2	5	6	●	●	●	●	
10	6	5	3	4	2	●	●	●	●	●
11	13	4	5	5	5	●	●	●	●	●
12	2	6	3	5	2	●	●	●	●	●
13	6	7	4	5	2	●	●	●	●	●
14	1	5	3	3	2	●	●	●	●	●
15	8	5	2	3	2	●	●	●	●	●
16	8	7	7	6	7	●	●	●	●	●
17	15	7	2	7	2	●	●	●	●	●
18	7	5	4	5	4	●	●	●	●	●
19	13	5	5	4	5	●	●	●	●	●
20	5	6	3	6	2	●	●	●	●	

Table 4.1: Demographic information on the 20 participants in our lab study. Experience is given in number of years; self-rated proficiency uses a Likert scale from 1 to 7, with 1 representing "not at all proficient" and 7 representing "extremely proficient".

years of programming experience; all except three had at least 4 years of experience.

However, the participants had little *professional* experience: only one spent more than 1 year as a professional developer.

When recruiting, we specified that participants should have basic knowledge of PHP, JavaScript, and the AJAX paradigm. However, 13 participants rated themselves as novices in at least one of the technologies involved. (Further demographic information is presented in **Table 4.1**) Participants were compensated with their choice of class research credit (where applicable) or a \$99 Amazon.com gift certificate.

The participants' task was to prototype a Web chat room application using HTML, PHP, and JavaScript. They were asked to implement five specific features (listed in **Figure 4.1**). Four of the features were fairly typical but the fifth (retaining a limited chat history) was more unusual. We introduced this feature so that participants would have to do some

- Chat Room Features:**
 1. Users should be able to set their username on the chat room page (application does not need to support account management). [Username]
 2. Users should be able to post messages. [Post]
 3. The message list should update automatically without a complete page reload. [AJAX update]
 4. Each message should be shown with the username of the poster and a timestamp. [Timestamp]
 5. When users first open a page, they should see the last 10 messages sent in the chat room, and when the chat room updates, only the last 10 messages should be seen. [History]

Figure 4.1: List of chat room features that lab study participants were asked to implement. The first four features are fairly typical; the fifth, retaining a limited chat history, is more unique.

programming, even if they implemented other features by downloading an existing chat room application (3 participants did this). We instructed participants to think of the task as a hobby project, not as a school or work assignment. Participants were not given any additional guidance or constraints.

We provided each participant with a working execution environment within Windows XP (Apache, MySQL, and a PHP interpreter) with a “Hello World” PHP application already running. They were also provided with several standard code authoring environments (Emacs, VIM, and Aptana, a full-featured IDE that provides syntax highlighting and code assistance for PHP, JavaScript and HTML) and allowed to install their own. Participants were allowed to bring any printed resources they typically used while

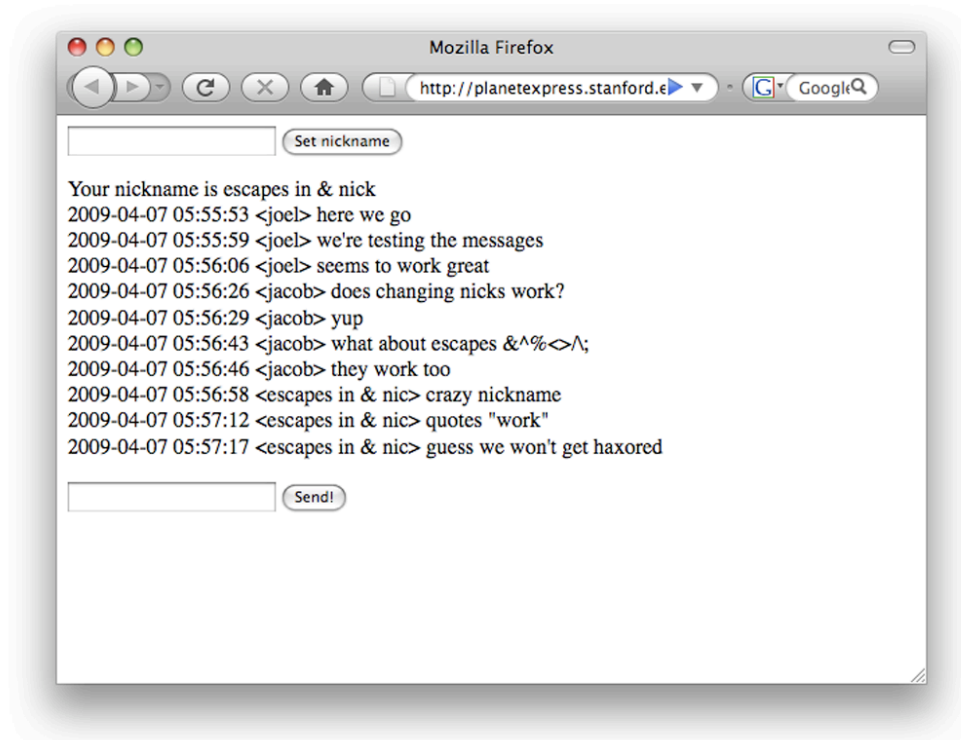


Figure 4.2: Screenshot of one participant's completed chat room. This participant met all of the specifications.

programming and were told that they were allowed to use *any* resources, including any code on the Internet and any code they had written in the past that they could access.

Three researchers observed each participant; all took notes. During each session, one researcher asked open-ended questions such as “why did you choose to visit that Web site?” or “how are you going to go about tracking down the source of that error?” that encouraged think-aloud reflection at relevant points (in particular, whenever participants used the Web as a resource). Researchers compared notes after each session and at the end of the study to arrive at the qualitative conclusions. Audio and video screen capture were recorded for all participants and were later coded for the amount of time participants used the Web.

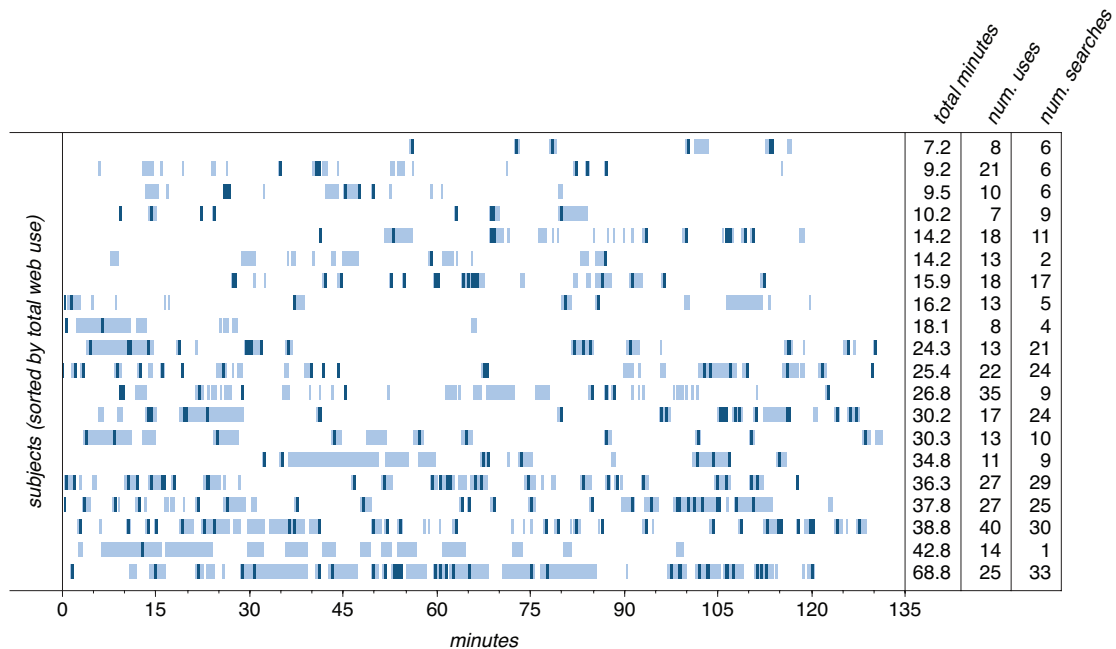


Figure 4.3: Overview of when participants referenced the Web during the laboratory study. Subjects are sorted by total amount of time spent using the Web. Web use sessions are shown in light blue, and instances of Web search are shown as dark bars.

4.1.2 RESULTS

The majority of participants met most or all of the chat room specifications: All but one met at least four of the five, and 75% met them all. (**Figure 4.2** shows one participant's finished chat room, which met all of the specifications.) All participants used the Web extensively (see **Figure 4.3**). On average, participants spent 19% of their programming time on the Web (25.5 of 135 minutes; $\sigma = 15.1$ minutes) in 18 distinct sessions ($\sigma = 9.1$).

The lengths of Web use sessions resembles a power-law distribution (see **Figure 4.4**). The shortest half (those less than 47 seconds) compose only 14% of the total time; the longest 10% compose 41% of the total time. This suggests that *individuals are leveraging the Web to accomplish several different kinds of activities*. Web usage also varied considerably between participants: The most-active Web user spent an order of magnitude more time online than the least active user.

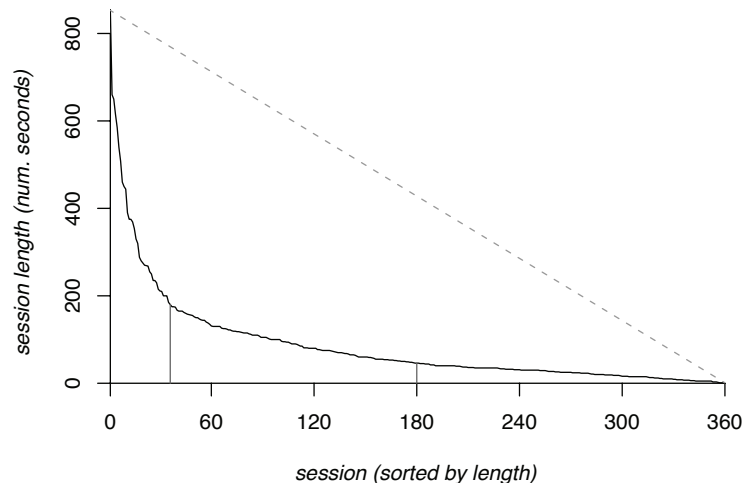


Figure 4.4: All 360 Web use sessions amongst the 20 participants in our lab study, sorted and plotted by decreasing length (in seconds). The left vertical bar represents the cutoff separating the 10% longest sessions, and the right bar the cutoff for 50% of sessions. The dotted line represents a hypothetical uniform distribution of session lengths.

4.1.2.1 Goals driving Web use

Why do programmers go to the Web? At the long end of the spectrum, participants spent tens of minutes *learning* a new concept (e.g., by reading a tutorial on AJAX-style programming). On the short end, participants delegated their memory to the Web, spending tens of seconds to *remind* themselves of syntactic details of a concept they knew well (e.g., by looking up the structure of a *foreach* loop). In between these two extremes, participants used the Web to *clarify* their existing knowledge (e.g., by viewing the source of an HTML form to understand the underlying structure). This section presents typical behaviors, anecdotes, and theoretical explanations for these three styles of online resource usage (see **Table 4.2** for a summary).

4.1.2.2 Just-in-time learning of new skills

Participants routinely stated that they were using the Web to *learn* about unfamiliar technologies. These Web sessions typically started with searches used to locate tutorial Web sites. After selecting a tutorial, participants frequently used its source code as a scaffold for learning-by-doing.

Searching for tutorials: Participants' queries usually contained a natural-language description of a problem they were facing, often augmented with several keywords specifying technology they planned to use (e.g., "php" or "javascript"). For example, one participant unfamiliar with the AJAX paradigm performed the query "update web page without reloading php". Query refinements were common for this type of Web use, often before the user clicked on any results. These refinements were usually driven by familiar, or newly learned, terms seen on the query result page: In the above example, the participant refined the query to "ajax update php" before clicking on any links.

Selecting a tutorial: Participants typically clicked several query result links, opening each in a new Web browser tab before evaluating the quality of any of them. After several pages were opened, participants would judge their quality by rapidly skimming. In particular, several participants reported using cosmetic features—*e.g.*, prevalence of advertising on the Web page or whether code on the page was syntax-highlighted—to evaluate the quality of potential Web sites. When we asked one participant how she decided what Web pages are trustworthy, she explained, “I don’t want [the Web page] to say ‘free scripts!’ or ‘get your chat room now!’ or stuff like that. I don’t want that because I think it’s gonna be bad, and most developers don’t write like that ... they don’t use that kind of language.” This assessing behavior is consistent with information scent theory, in that users decide which Web pages to explore by evaluating their surface-level features [71].

Using the tutorial: Once a participant found a tutorial that he believed would be useful, he would often immediately begin experimenting with its code samples (even before reading the prose). We believe this is because tutorials typically contain a great deal of prose,

Web session goal	Learning	Clarification	Reminder
Reason for using Web	Just-in-time learning of unfamiliar concepts	Connect high-level knowledge to implementation details	Substitute for memorization (<i>e.g.</i> , language, syntax, or function usage lookup)
Web session length	Tens of minutes	About 1 minute	< 1 minute
Starts with web search?	Almost always	Often	Sometimes
Search terms	Natural language related to high-level task	Mix of natural language and code, cross-language analogies	Mostly code (<i>e.g.</i> , function names, language keywords)
Example search	"ajax tutorial"	"javascript thread"	"mysql_fetch_array"
Num. result clicks	Usually several	Fewer	Usually zero or one
Num. query refinements	Usually several	Fewer	Usually zero
Types of Web pages visited	Tutorials, how-to articles	API documentation, blog posts, articles	API documentation, result snippets on search page
Amount of code copied from Web	Dozens of lines (<i>e.g.</i> , from tutorial snippets)	Several lines	Varies
Immediately test copied code?	Yes	Not usually, often trust snippets	Varies

Table 4.2: Summary of characteristics of three points on the spectrum of Web use goals.

which is time-consuming to read and understand. Subject 10 said, “I think it’s less expensive for me to just take the first [code I find] and see how helpful it is at ... a very high level ... as opposed to just reading all these descriptions and text.”

Participants often began adapting code before completely understanding how it worked. One participant explained, “there’s some stuff in [this code] that I don’t really know what it’s doing, but I’ll just try it and see what happens.” He copied four lines into his project, immediately removed two of the four, changed variable names and values, and tested. The entire interaction took 90 seconds. This learning-by-doing approach has one of two outcomes: It either leads to deeper understanding, mitigating the need to read the tutorial’s prose, or it isolates challenging areas of the code, guiding a more focused reading of the tutorial’s prose.

For programmers, what is the cognitive benefit of experimentation over reading? Results from cognitive modeling may shed light on this. Cox and Young developed two ACT-R models to simulate a human learning the interface for a central heating unit [20]. The first model was given “how-to-do-the-task’ instructions” and was able to carry out only those specific tasks from start to finish. The second model was given “how-the-device-works’ instructions,” (essentially a better mapping of desired states of the device to actions performed) and afterwards could thus complete a task from any starting point. When tutorials are used as an aid in the middle of the development process, the programmer is typically only interested in a small portion of the tutorial. Programmers therefore commonly pick up the tutorial’s task “in the middle”. We suggest that when participants experiment with code, it is precisely to learn these action/state mappings.

Approximately 1/3 of the code in participants' projects was physically copied and pasted from the Web. This code came from many sources: while a participant may have copied a hundred lines of code altogether, he did so ten lines at a time. This approach of programming by example modification is consistent with Yeh *et al.*'s study of students learning to use a Java toolkit [86].

4.1.2.3 Clarification of existing knowledge

There were many cases where participants had a high-level understanding of how to implement functionality, but did not know how to implement it in the specific programming language. They needed a piece of *clarifying* information to help map their schema to the particular situation. The scenario presented at the beginning of Chapter 1 is an example of this behavior: The participant had a general understanding of HTML forms, but did not know all of the required syntax. These *clarifying* activities are distinct from *learning* activities because participants can easily recognize and adapt the necessary code once they find it. Because of this, *clarifying* uses of the Web are significantly shorter than *learning* uses.

Searching with synonyms: Participants often used Web search when they were unsure of exact terms. We observed that search works well for this task because synonyms of the correct programming terms often appear in online forums and blogs. For example, one participant used a JavaScript library that he had used in the past but “not very often,” to implement the AJAX portion of the task. He knew that AJAX worked by making requests to other pages, but he forgot the exact mechanism for accomplishing this in his chosen library (named *Prototype*). He searched for “prototype request”. The researchers asked, “Is ‘request’

the thing that you know you're looking for, the actual method call?" He replied, "I don't know. I just know that it's probably similar to that."

Clarification queries contained more programming-language-specific terms than *learning* ones. Often, however, these terms were not from the correct programming language! Participants often made what we call *language analogies*: For example, one participant said "Perl has [a function to format dates as strings], so PHP must as well". Similarly, several participants searched for "javascript thread". While JavaScript does not explicitly contain threads, it supports similar functionality through interval timers and callbacks. All participants who performed this search quickly arrived at an online forum or blog posting that pointed them to the correct function for setting periodic timers: *setInterval*.

Testing copied code (or not): When participants copied code from the Web during *clarification* uses, it was often not immediately tested. Participants typically trusted code found on the Web, and indeed, it was typically correct. However, they would often make minor mistakes when adapting the code to their needs (*e.g.*, forgetting to change all instances of a local variable name). Because they believed the code correct, they would then work on other functionality before testing. When they finally tested and encountered bugs, they would often erroneously assume that the error was in recently-written code, making such bugs more difficult to track down.

Using the Web to debug: Participants also used the Web for clarification *during* debugging. Often, when a participant encountered a cryptic error message, he would immediately search for that exact error on the Web. For example, one participant received an error that read, "XML Filtering Predicate Operator Called on Incompatible Functions." He mumbled, "What does that mean?" then followed the error alert to a line that contained

code previously copied from the Web. The code did not help him understand the meaning of the error, so he searched for the full text of the error. The first site he visited was a message board with a line saying “This is what you have:” followed by the *exact* code in question and another line saying “This is what you should have:” followed by a corrected line of code. With this information, the participant returned to his code and successfully fixed the bug without ever fully understanding the cause.

4.1.2.4 Reminders about forgotten details

Even when participants were familiar with a concept, they often did not remember low-level syntactic details. For example, one participant was adept at writing SQL queries, but unsure of the correct placement of a *limit* clause. Immediately after typing “ORDER BY respTime”, he went online and searched for “mysql order by”. He clicked on the second link, scrolled halfway down the page, and read a few lines. Within ten seconds he had switched back to his code and added “LIMIT 10” to the end of his query. In short, when participants used the Web for *reminding* about details, they knew *exactly* what information they were looking for, and often knew *exactly* on which page they intended to find it (*e.g.*, official API documentation).

Searching or browsing for reminders: When participants used the Web for learning and clarification, they almost always began by performing a Web search and then proceeded to view one or more results. In the case of reminders, sometimes participants would perform a search and view only the search result snippets without viewing any of the results pages. For example, when one participant forgot the exact name of the PHP function used to access the result of a database query. A Web search for “php fetch array” allowed him

to quickly retrieve this information (“mysql_fetch_array”) the exact name of the function simply by browsing the snippets in the results page (See **Figure 4.5**). Other times, participants would view a page without searching at all. (The Gantt chart in **Figure 4.3** contains many Web sessions that do not begin with a dark bar indicating a Web search.) This is because participants often kept select Web sites (such as official language documentation) open in browser tabs to use for reminders when necessary.

The Web as an external memory aid: Several participants reported using the Web as an alternative to memorizing routinely-used snippets of code. One participant browsed to a page within PHP’s official documentation that contained six lines of code necessary to

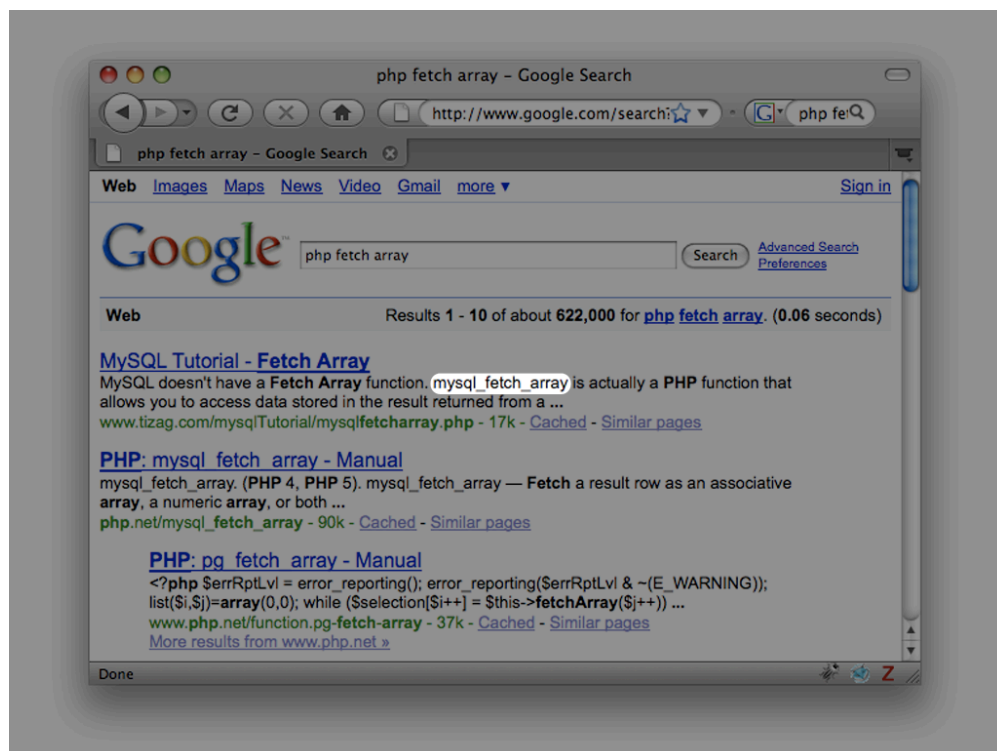


Figure 4.5: Example of how participants used Web search to remind themselves of forgotten low-level syntax. Here, the programmer forgot the exact name of the function used to access the result of a database query. Searching for “php fetch array” allowed him to quickly retrieve the exact name (highlighted) without visiting any additional Web pages.

connect and disconnect from a MySQL database. After he copied this code, a researcher asked him if he had copied it before. He responded, “[yes,] hundreds of times”, and went on to say that he never bothered to learn it because he “knew it would always be there.” We believe that in this way, programmers can effectively distribute their cognition [45], allowing them to devote more mental energy to higher-level tasks.

4.2 STUDY 2: WEB SEARCH LOG ANALYSIS

Do programmer’s Web query styles in the real world robustly vary with their goal, or are the results presented above an artifact of the particular lab setting? To investigate this, we

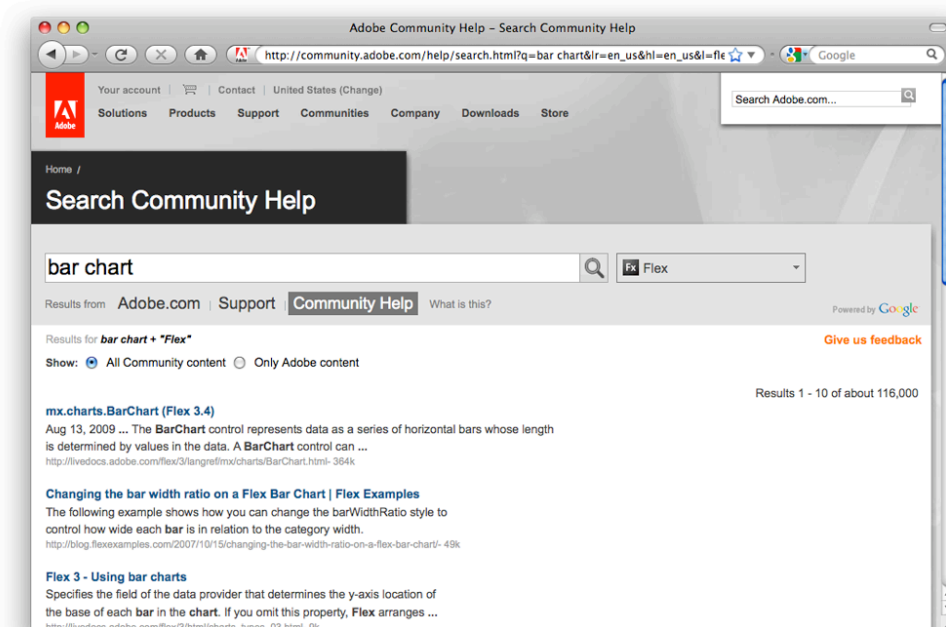


Figure 4.6: Web query result interface for Adobe’s *Community Help* search portal. This portal is implemented using a Google Custom Search Engine, and displays results in a format nearly identical to general-purpose search engines such as Google.

analyzed Web query logs from 24,293 programmers making 101,289 queries about the Adobe Flex Web application development framework in July 2008. These queries came from the *Community Search* portal [5] on Adobe’s Developer Network Web site [6]. This portal is implemented using a Google Custom Search Engine [7], and indexes documentation, articles, blogs, and forums by Adobe and vetted third-party sources. It displays search results in a format nearly identical to general-purpose search engines such as Google (See **Figure 4.6**).

To cross-check the lab study against this real-world data set, we began our analysis by evaluating four hypotheses derived from those findings:

H1: *Learning* sessions begin with natural language queries more often than *reminding* sessions.

H2: Users more frequently refine queries without first viewing results when *learning* than when *reminding*.

H3: Programmers are more likely to visit official API documentation in *reminding* sessions.

H4: The majority of *reminding* sessions start with code-only queries. Additionally, code-only queries are least likely to be refined, and contain the fewest number of result clicks.

4.2.1 METHOD

We analyzed the data in three steps. First, we used IP addresses (24,293 total unique IPs) and timestamps to group queries (101,289 total) into sessions (69,955 total). A session was defined as a sequence of query and result-click events from the same IP address with no gaps longer than six minutes. (This definition is common in query log analysis, *e.g.*, [78].)

Second, we selected 300 of these sessions and analyzed them manually. We found it valuable to examine all of a user’s queries because doing so provided more contextual

information. We used unique IP addresses as a proxy for users, and randomly selected from among users with at least 10 sessions. 996 met this criteria; we selected 19. This IP-user mapping is close but not exact: a user may have searched from multiple IP addresses, and some IP addresses may map to multiple users. It seems unlikely, though, that conflating IPs and users would affect our analysis.

These sessions were coded as one of *learning*, *reminding*, *unsure*, or *misgrouped*.

(Because the query log data is voluminous but lacks contextual information, we did not use the *clarifying* midpoint in this analysis.) We coded a session as *learning* or *reminding* based on the amount of knowledge we believed the user had on the topic he was searching for, and as *unsure* if we could not tell. To judge the user's knowledge, we used several heuristics: whether the query terms were specific or general (*e.g.*, “radio button selection change” is a specific search indicative of *reminding*), contents of pages visited (*e.g.*, a tutorial indicates *learning*), and whether the user appeared to be an expert (determined by looking at the user's entire search history—someone who occasionally searches for advanced features is likely to be an expert.) We coded a session as *misgrouped* if it appeared to have multiple unrelated queries (potentially caused by a user performing unrelated searches in rapid succession, or by pollution from multiple users with the same IP address).

Finally, we computed three properties about each search session:

1. *Query type*—whether the query contained only code (terms specific to the Flex framework, such as class and function names), only natural language, or both.
2. *Query refinement method*—between consecutive queries, whether search terms were generalized, specialized, otherwise reformulated, or changed completely.

3. *Types of Web pages visited*—each result click was classified as one of four page types: *Adobe APIs*, *Adobe tutorials*, *tutorials/articles* (by third-party authors), and *forums*.

4.2.1.1 Determining query type

We first split each query string into individual tokens using whitespace. Then we ran each token through three classifiers to determine if it was *code* (i.e., Flex-specific keywords and class/function names from the standard library). The first classifier checked if the token was a (case-insensitive) match for any classes in the Flex framework. The second checked if the token contained camelCase (a capital letter in the middle of the word), which was valuable because all member functions and variables in the Flex framework use camelCase. The third checked if the token contained a dot, colon, or ended with an open and closed parenthesis, all indicative of code. If none of these classifiers matched, we classified the token as a *natural-language word*.

4.2.1.2 Determining query refinement method

We classified refinements into five types, roughly following the taxonomy of Lau and Horvitz [53]. A *generalize* refinement had a new search string with one of the following properties: it was a substring of the original, it contained a proper subset of the tokens in the original, or it split a single token into multiple tokens and left the rest unchanged. A *specialize* refinement had a new search string with one of the following properties: it was a superstring of the original, it added tokens to the original, or it combined several tokens from the original together into one and left the rest unchanged. A *reformulate* refinement had a new search string that contained some tokens in common with the original but was neither a generalization nor specialization. A *new* query had no tokens in common with the original.

Spelling refinements were any queries where spelling errors were corrected, as defined by Levenshtein distances between corresponding tokens all being less than 3.

4.2.1.3 Determining Web page type

We built regular expressions that matched sets of URLs that were all the same type. A few Web sites, such as the official Adobe Flex documentation and official tutorial pages, contain the majority of all visits (and can be described using just a few regular expressions). We sorted all 19,155 result click URLs by number of visits and classified the most frequently-visited URLs first. With only 38 regular expressions, we were able to classify pages that accounted for 80% of all visits (10,909 pages). We did not hand-classify the rest of the pages because the cost of additional manual effort outweighed the potential marginal benefits. Result clicks for the remaining 8246 pages (20% of visits) were labeled as *unclassified*.

4.2.2 RESULTS

Out of 300 sessions, 20 appeared misgrouped, and we were unsure of the goal for 28. Of the remaining 252 sessions, 56 (22%) had *learning* traits and 196 (78%) *reminding* traits. An example of a session with *reminding* traits had a single query for “function as parameter” and a single result click on the first result, a language specification page. An example of a session with *learning* traits began with the query “preloader”, which was refined to “preloader in flex” and then “creating preloader in flex”, followed by a result click on a tutorial.

Type of first query	Session type		All hand-coded
	Learning	Reminding	
code only	0.21	0.56	0.48
nat. lang. & code	0.29	0.10	0.14
nat. lang. only	0.50*	0.34	0.38
Total	1.00	1.00	1.00

Table 4.3: For hand-coded sessions of each type, proportion of first queries of each type (252 total sessions). Statistically significant differences between columns are shown in bold, * entry means only significant at $p < 0.05$.

Result click Web page type	Session type		All hand-coded
	Learning	Reminding	
Adobe APIS	0.10	0.31	0.23
Adobe tutorials	0.35	0.42	0.40
tutorials/articles	0.31	0.10	0.17
forums	0.06	0.04	0.05
unclassified	0.18	0.13	0.15
Total	1.00	1.00	1.00

Table 4.4: For queries in hand-coded sessions of each type, proportion of result clicks to Web sites of each type (401 total queries). Statistically significant differences between columns are shown in bold.

Refinement type					all
generalize	new	reformulate	specialize	spelling	
0.44	0.61	0.51	0.39	0.14	0.48

Table 4.5: For each refinement type, proportion of refinements of that type where programmers clicked on any links prior to the refinement (31,334 total refinements).

Result click Web page type	query type			All clicks
	code	nat. lang. & code	nat. lang.	
Adobe APIS	0.38	0.16	0.10	0.23
Adobe tutorials	0.31	0.33	0.39	0.34
tutorials/articles	0.15	0.22	0.19	0.18
forums	0.03	0.07	0.06	0.05
unclassified	0.13	0.22	0.27	0.20
Total	1.00	1.00	1.00	1.00

Table 4.6: For queries of each type, proportion of result clicks leading programmer to Web pages of each type (107,343 total queries). Statistically significant differences between columns are shown in bold.

We used the Mann-Whitney U test for determining statistical significance of differences in means and the chi-square test for determining differences in frequencies (proportions). Unless otherwise noted, all differences are statistically significant at $p < 0.001$.

H1: The first query was exclusively natural language in half of *learning* sessions, versus one third in *reminding* sessions (see **Table 4.3**).

H2: *Learning* and *reminding* sessions do not have a significant difference in the proportion of queries with refinements before first viewing results.

H3: Programmers were more likely to visit official API documentation in *reminding* sessions than in *learning* sessions (31% versus 10%, see **Table 4.4**).

H4: Code-only queries accounted for 56% of all *reminding* queries (**Table 4.3**). Among all (including those not hand-coded) sessions, those beginning with code-only queries were refined less ($\mu = 0.34$) than those starting with natural language and code ($\mu = 0.60$) and natural language only ($\mu = 0.51$). It appears that when programmers perform code-only queries, they know what they are looking for, and typically find it on the first search.

After evaluating these hypotheses, we performed further quantitative analysis of the query logs. In this analysis, we focused on how queries were refined and the factors that correlated with types of pages visited.

4.2.2.1 Programmers rarely refine queries, but are good at it

In this data set, users performed an average of 1.45 queries per session (the distribution of session lengths is shown in **Figure 4.7**). This is notably less than other reports, *e.g.*, 2.02 [78].

There are a number of possible explanations for this: improvements in search algorithms, that programming as a domain is well-suited to search, that participants were skilled, or that users abandon specialized search tools like Community Help more quickly than they do general purpose search engines.

Across all sessions and refinement types, 66% of queries *after refinements* have result clicks, which is significantly higher than the percentage of queries before refinements (48%) that have clicks. This contrast suggests that refining queries generally produces better results.

When programmers refined a query to make it more *specialized*, they generally did so without first clicking through to a result (see **Table 4.5**). Presumably, this is because they

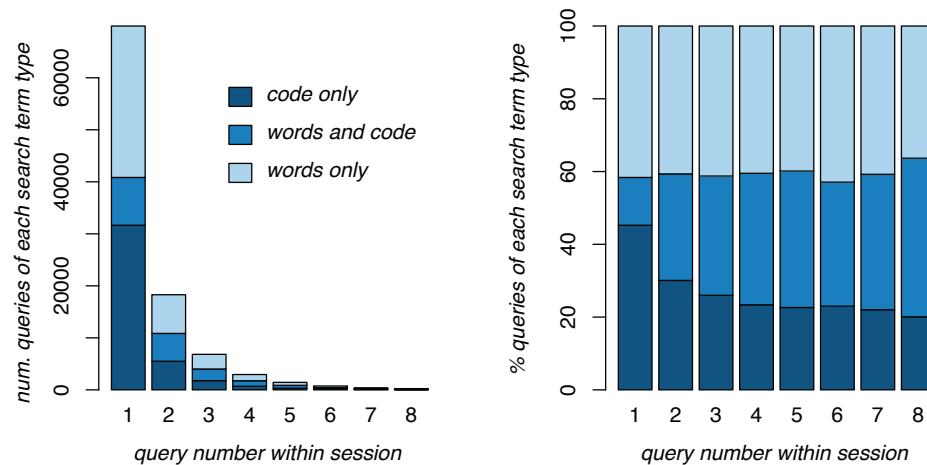


Figure 4.7: How query types changed as queries were refined. In both graphs, each bar sums all *i*th queries over all sessions that contained an *i*th query (*e.g.*, a session with three queries contributed to the sums in the first three bars). The graph on the left is a standard histogram; the graph on the right presents the same data, but with each bar's height normalized to 100 to show changes in proportions as query refinements occurred.

assessed the result snippets and found them unsatisfactory. Programmers may also see little risk in “losing” a good result when specializing—if it was a good result for the initial query, it ought to be a good result for the more specialized one. This hypothesis is reinforced by the relatively high click rate before performing a completely new query (presumably on the same topic)—good results may be lost by completely changing the query, so programmers click any potentially valuable links first. Finally, almost no one clicks before making a spelling refinement, which makes sense because people mostly catch typos right away.

Users began with code-only searches 48% of the time and natural language searches 38% of the time (see **Figure 4.7**). Only 14% of the time was the first query mixed. The percent of mixed queries steadily increased to 42% by the eighth refinement, but the percent of queries containing only natural language stayed roughly constant throughout.

4.2.2.2 Query type predicts types of pages visited

There is quantitative support for the intuition that query type is indicative of goal (see **Table 4.6**). Code-only searches, which one would expect to be largely *reminding* queries, are most likely to bring programmers to official Adobe API pages (38% versus 23% overall) and least likely to bring programmers to all other types of pages. Natural-language-only queries, which one would expect to be largely *learning* queries, are most likely to bring programmers to official Adobe tutorials (39% versus 34% overall).

4.3 LIMITATIONS OF OUR FINDINGS

One limitation of studying student programmers in the lab is that their behavior and experience may differ from the broader population of programmers. Our query log analysis, prior work (e.g., [44, 79]), and informal observation of online forums suggest that

programmers of all skill levels are indeed turning to the Web for help. An important area for future work will be to better understand how the behaviors of these populations differ.

A limitation of the query log study is that it does not distinguish queries that were “opportunistic” from those that were not. It remains an open question whether there is a causal relationship between programming style and Web usage style.

Finally, our studies do not consider any resources other than the Web, such as printed media, or one’s colleagues. (While we notified the lab participants that they could bring printed materials, none did.) This dissertation looks exclusively at Web usage; other researchers have similarly examined other information resources individually (*e.g.*, Chong *et al.* examined collaboration between programmers during solo and pair programming [17]). Future work is needed to compare the trade-offs of these different information resources.

4.4 FIVE KEY INSIGHTS AND IMPLICATIONS FOR TOOLS

In this section, we present five insights distilled from our findings. For each insight, we suggest implications for the design of tools that better support programmers’ use of the Web. In the chapters that follow, we explore and evaluate these implications.

Programmers deliberately choose not to remember complicated syntax. Instead, they use the Web as external memory that can be accessed as needed. This suggests that Web search should be integrated into the code editor in much the same way as identifier completion (*e.g.*, Microsoft’s IntelliSense and Eclipse’s Code Assist). Another possible approach is to build upon ideas like keyword programming [57] to create authoring environments that allow the programmer to type “sloppy” commands which are automatically transformed into syntactically correct code using Web search.

Web search often serves as a “translator” when programmers don’t know the exact terminology or syntax. Using the Web, programmers can adapt existing knowledge by making analogies with programming languages, libraries and frameworks that they know well. The Web further allows programmers to make sense of cryptic errors and debugging messages. Future tools could proactively search the Web for the errors that occur during execution, compare code from search results to the user’s own code, and automatically locate possible sources of errors.

Programmers are good at refining their queries, but need to do it rarely. Query refinement is most necessary when users are trying to adapt their existing knowledge to new programming languages, frameworks, or situations. This underscores the value of keeping users in the loop when building tools that search the Web automatically or semi-automatically. In other cases, however, query refinements could be avoided by building tools that automatically augment programmers’ queries with contextual information, such as the programming language, frameworks or libraries in the project, or the types of variables in scope.

Programmers use Web tutorials for just-in-time learning, gaining high-level conceptual knowledge when they need it. Tools may valuably encourage this practice by tightly coupling tutorial browsing and code authoring. One system that explores this direction is d.mix, which allows users to “sample” a Web site’s interface elements, yielding the API calls necessary to create them [39]. This code can then be modified inside a hosted sandbox.

Programmers often delay testing code copied from the Web, especially when copying routine functionality. As a result, bugs introduced when adapting copied code are

often difficult to find. Tools could assist in the code adaptation process by, for example, highlighting all variable names and literals in any pasted code. Tools could also clearly demarcate regions of code that were copied from the Web and provide links back to the original source.

CHAPTER 5 BLUEPRINT: INTEGRATING WEB SEARCH INTO THE DEVELOPMENT ENVIRONMENT

This chapter investigates whether a task-specific search engine integrated into existing programming environments can significantly reduce the cost of searching for relevant information, and thus change how programmers use search. Small performance improvements can cause categorical behavior changes that far exceed the benefits of decreased task completion time [35]. For example, slight changes in the efficiency of an interface for an 8-tile puzzle game (*e.g.*, direct manipulation versus keyboard commands for moving a tile) have been shown to influence how individuals plan higher-level tasks [68]. Similarly, individuals are 1.5 times more likely to prefer a search engine with a 250 millisecond delay in returning results over one with a 2 second delay (when they are unaware that the difference between the two search engines is this delay) [14]. This effect exists even at seemingly imperceptible levels: introducing a 100 millisecond delay in presenting Web

search results causes a 0.2% decrease in daily searches per user [15], which represents a significant loss of revenue for major commercial search engines.

We believe that reducing search cost through tool integration may increase and change how programmers find and use examples. These ideas are manifest in *Blueprint*, a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code.

Two insights drove Blueprint's design (see **Figure 5.1** and **Figure 5.2**). First, *embedding search into the development environment* allows the search engine to leverage the users' context (e.g., programming languages and framework versions in use). This lowers the

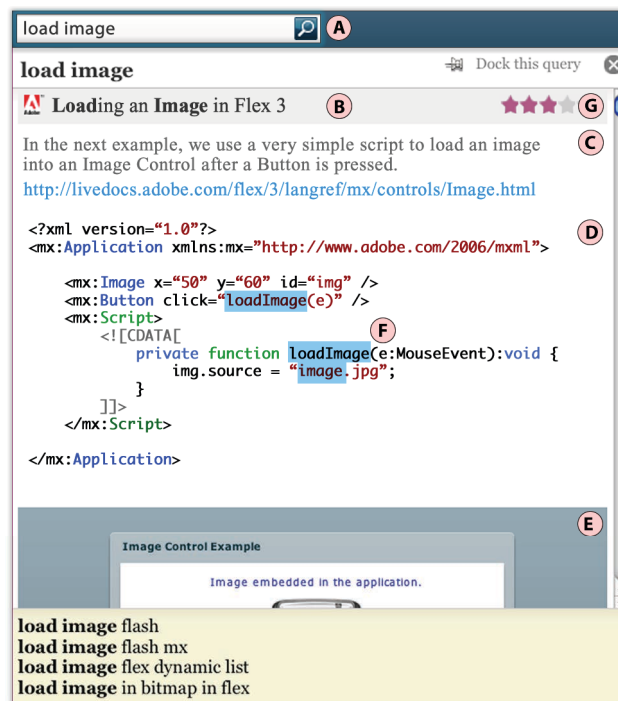


Figure 5.1: The Blueprint plug-in for the Adobe Flex Builder development environment helps programmers locate example code. A hotkey places a search box (A) at the programmer's cursor position. Search results (B) are example-centric; each result contains a brief textual description (C), the example code (D), and, when possible, a running example (E). The user's search terms are highlighted (F), facilitating rapid scanning of the result set. Blueprint allows users to rate examples (G).

cost of constructing a good query, which improves result quality. Second, *extracting code examples from Web pages and composing them in a consistent, code-centric search results view* reduces the need to click through to Web pages to find example code. This allows users to evaluate results much more rapidly than with traditional Web search interfaces, reducing the cost of selecting a good result.

We first evaluated Blueprint through a comparative laboratory study with 20 participants. In the lab, participants in the Blueprint condition found and adapted example code significantly faster than those in the traditional Web search condition. Blueprint participants also wrote significantly better code, perhaps because they could look at many more examples and choose a better starting point. To better understand how Blueprint would affect the workflow of real-world programmers, we deployed Blueprint on the Adobe Labs Web site, and studied how it was used by thousands of developers over one year. We report on this deployment in Chapter 6.

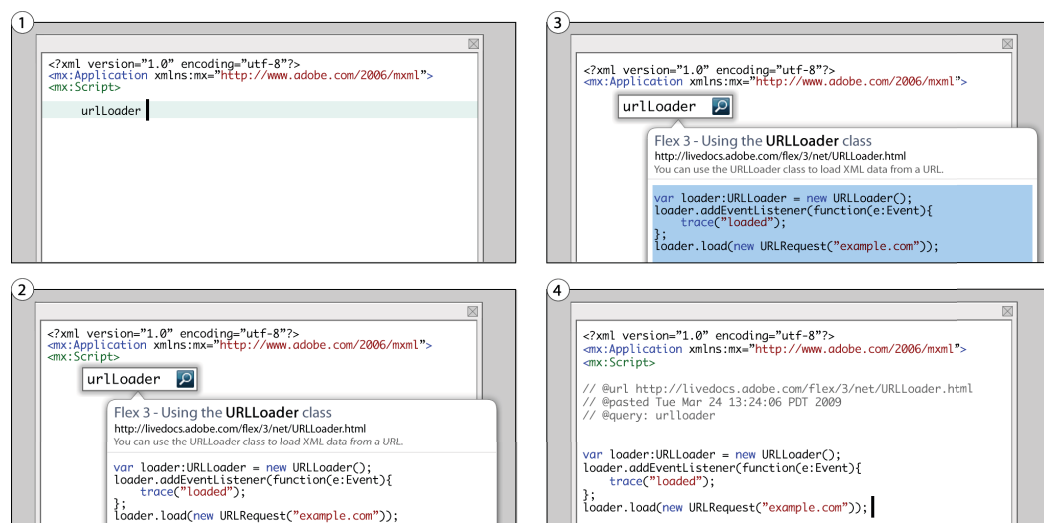


Figure 5.2: Example-centric programming with Blueprint. The user presses a hotkey to initiate a search; a search box appears at the cursor location (1). Searches are performed interactively as the user types; example code and running examples (when present) are shown immediately (2). The user browses examples with the keyboard or mouse, and presses Enter to paste an example into her project (3). Blueprint automatically adds a comment containing metadata that links the example to its source (4).

The remainder of this chapter proceeds as follows. We first present a scenario that describes the use of Blueprint and presents its interface. We then describe the implementation of Blueprint. Next, we detail the laboratory evaluation of Blueprint. We conclude by positioning Blueprint in a design space of tools that support example-centric development.

5.1 SCENARIO: DEVELOPING WITH BLUEPRINT

Blueprint is designed to help programmers with directed search tasks and allow them to easily remind themselves of forgotten details, and clarify existing knowledge. Let's follow Jenny as she creates a Web application for visualizing power consumption.

First, Jenny needs to retrieve power-usage data from a Web service. Although Jenny has written similar code previously, she can't remember the exact code she needs. She *does* remember that one of the main classes involved began with "URL". So, she types "URL" into her code and uses auto-complete to remember the "URLLoader" class. Although, she now knows the class name, Jenny still doesn't know how to use it (**Figure 5.2**, step 1). With another hotkey, Jenny brings up the Blueprint search interface, which automatically starts searching for URLLoader (step 2). Blueprint augments Jenny's query with the language and framework version she is using, and returns appropriate examples that show how to use a URLLoader. She scans through the first few examples and sees one that has all the pieces she needs (step 3). She selects the lines she wants to copy, presses *Enter*, and the code is pasted in her project. Blueprint augments the code with a machine- and human-readable comment that records the URL of the source and the date of copy (step 4). When Jenny opens this source file in the future, Blueprint will check this URL for changes to the source example (e.g., with a bug fix),

and will notify her if an update is available. Jenny runs the code in Flex’s debugger to inspect the XML data.

Next, Jenny wants to explore different charting components to display power usage. She invokes Blueprint a second time and searches for “charting”. Jenny docks the Blueprint result window as a panel in her development environment so she can browse the results in a large, persistent view. When source pages provide a running example, Blueprint presents this example next to the source code. Eventually Jenny picks a line chart, copies the example code from the Blueprint panel into her project, and modifies it to bind the chart to the XML data.

Finally, Jenny wants to change the color of the lines on the chart. She’s fairly confident that she knows how to do this, and types the necessary code by hand. To make sure she didn’t miss any necessary steps, she presses a hotkey to initiate a Blueprint search from one of the lines of code she just wrote. Blueprint automatically uses the contents of the current line as the initial query. Because terms in this line of code are common to many examples that customize charts, she quickly finds an example that matches what she is trying to do. She confirms her code is correct, and begins testing the application. After only a few minutes her prototype is complete.

5.2 IMPLEMENTATION

Blueprint comprises a *client plug-in*, which provides the user interface for searching and browsing results, and the *Blueprint server*, which executes searches for example code. **Figure**

5.3 provides a visual system description.

5.2.1 CLIENT-SIDE PLUG-IN

The Blueprint client is a plug-in for Adobe Flex Builder. Flex Builder, in turn, is a plug-in for the Eclipse Development Environment. The Blueprint client provides three main pieces of functionality. First, it provides a user interface for initiating queries and displaying results. Second, it sends contextual information (e.g., programming language and framework version) with each user query to the server. Third, it notifies the user when the Web origin of examples they adapted has updated (e.g., when a bug is fixed). All communication between the client and server occurs over HTTP using the JSON data format.

Blueprint's query and search results interface is implemented using HTML and JavaScript that are rendered by the browser widget provided in Eclipse's UI toolkit. Search results are rendered sequentially in a list below the query box (**Figure 5.1**, part a). Each search result includes the source Web page title (b), a hyperlink to the source Web page, English description of the example (c), the code example (d), and, if available, a running example (e) showing the functionality of the code. All examples include syntax highlighting (produced by the Pygments [4] library), and users' search terms are also highlighted throughout the code (f). Users can navigate through examples using the Tab key and

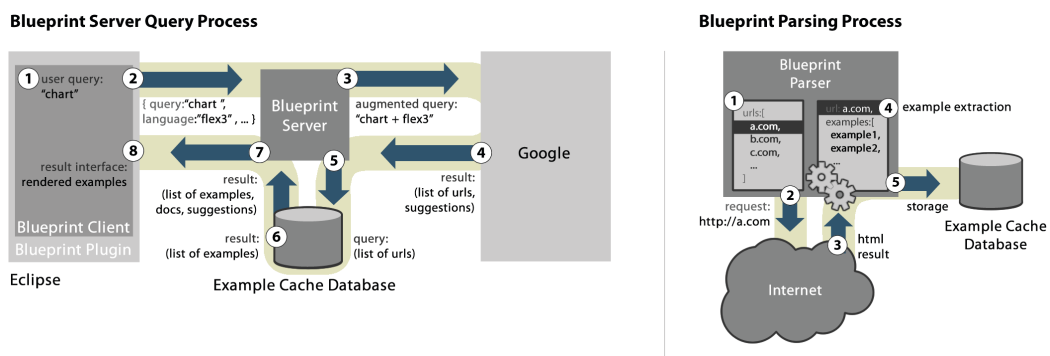


Figure 5.3: Architecture of the Blueprint system. The process of servicing a user's query is shown on the left; the background task of parsing Web pages to extract examples is shown on the right.

copy/paste selections by pressing enter. Users can rate examples (g) and dock the Blueprint floating window as an Eclipse panel. Blueprint also allows users to follow hyperlinks to view search results in context, and maintains a browsing and search history.

When users paste example code into a project, Blueprint inserts a Javadoc-like comment at the beginning. This comment tags the example code with its *URL source*, the insertion *date and time*, and the *search terms* used in the initial query. This comment aids the user in revisiting the source example at a later time if necessary.

5.2.2 BLUEPRINT SERVER

The Blueprint server executes queries for example code and returns examples to the client. To maximize speed, breadth, and ranking quality, the server leverages the Adobe Community Help search APIs, a Google Custom Search engine. This search engine indexes Adobe product-specific content from across the Web. When the Blueprint server receives a query, it first augments the query with the user's context (*e.g.*, programming language and framework version), which is sent along with the query by the client. Then the server sends the new augmented query to the search engine, which returns a set of URLs. Since Blueprint users are interested in code *examples* and *not Web pages*, the server retrieves the Web pages returned by the search engine and processes them to extract source code examples.

Since processing each page requires on average 10 seconds (8 seconds to retrieve the page, 2 seconds to extract examples), we preprocess pages and cache extracted examples. When the search engine returns URLs that are not in the Blueprint cache, the URLs are added to the cache by a background process. (Note that the parsing process is easily parallelized at a per-Web-page level. We have found that retrieving and parsing 100 pages in

parallel works well on our infrastructure.) Code examples from those URLs are returned in future queries.

Before deploying Blueprint, we pre-populated the cache with approximately 50,000 URLs obtained from search engine query logs. To keep the cache current, Blueprint crawls the URLs in the cache as a background process. Since pages containing examples are relatively static, the Blueprint prototype re-crawls them weekly. As of November 2010, after 1.5 years of use, the Blueprint cache includes 208,916 examples from 84,403 unique Web pages.

Leveraging an existing commercial search engine to produce a candidate result set has a number of advantages over building a new search engine (*e.g.*, [44, 79]). First, it is substantially more resource-efficient to implement, as keeping a document collection up to date is expensive. Second, generating high-quality search results from natural-language queries is a hard problem and a highly-optimized commercial search engine is likely to produce better results than a prototype search engine with a restricted domain. Finally, a general-purpose search engine surfaces examples from tutorials, blogs, and API pages. Examples found on such pages are more likely to be instructive than examples extracted from large source code repositories.

5.2.3 EXTRACTING EXAMPLE CODE AND DESCRIPTIONS

To extract source code from Web pages, Blueprint segments the page and classifies each segment as source code or other type of content. First, Blueprint uses the BeautifulSoup library [73] to transform HTML into proper XHTML, and then it divides the resulting hierarchical XHTML document into independent segments by examining block-level

elements. Blueprint uses 31 tags to define blocks; the most common are: P, DIV, PRE, and heading tags. It also extracts SCRIPT and OBJECT blocks as block-level elements, because running examples that show executing example code are usually contained within these tags. To find block-level elements, Blueprint performs a depth-first traversal of the document. When it reaches a leaf element, it backtracks to the nearest block-level ancestor and creates a segment. If the root of the tree is reached before finding a block-level element, the element immediately below the root is extracted as a segment. This algorithm keeps segments ordered exactly as they were in the original file. Finally, to more easily and reliably determine whether a segment contains code, Blueprint converts each segment to formatted plain text using w3m, a text-based Web browser. This conversion allows for classification of code based on its textual appearance to a user on a Web page and not based on its HTML structure.

Blueprint stores the HTML and plain text versions of all segments in a database. On average, a Web page in our dataset contains 161 segments. However, 69% of these are less than 50 characters long (these are primarily created by navigational elements). Although this algorithm leads to a large number of non-source code segments, it correctly parses blocks of example code into single segments, which enables our classifiers to prune non-source code segments.

One limitation of this extraction algorithm is that it assumes code examples on Web pages are independent and so it does not handle Web pages that provide several related code examples that should be considered in concert, such as tutorials that list several steps or offer several complementary alternatives. This limitation is not a large concern for *reminder* tasks (as described in §4.1.2.4), as programmers typically know exactly what code they are looking

for. However, it presents challenges when using Blueprint for *learning* tasks. This is discussed further in §5.3.3.1 below.

5.2.3.1 Classifying example code

Given a set of clean, separate segments, the most straightforward approach to classifying them as source code is to use a programming language parser and label segments that parse correctly as code. For Blueprint, this would require ActionScript and MXML parsers, because they are the two languages used by Adobe Flex. In practice, this approach yields many false negatives: segments that contain code but are not labeled as such. For example, code with line numbers or a typo will cause parsing to fail.

An alternate approach is to specify heuristics based on features unique to code, such as curly braces, frequent use of language keywords, and lines that end with semi-colons [44]. This approach produces many fewer false negatives, but introduces false positives, such as paragraphs of text that discuss code. Such paragraphs usually describe other source code found on the page and are not useful on their own.

To remove buggy code that appears in forums and blog post comments, we ignore all segments that follow a comment block (where a comment block is a block that includes the word “comment”) and all Web pages that include “group” or “forum” in the URL.

We computed precision (MXML: 84%, AS: 91%) and recall (MXML: 90%, AS: 86%) on 40 randomly sampled Web pages from a corpus of the 2000 most frequently visited Web pages from the Adobe Community Help Search Web site. We compared the examples extracted by Blueprint to the examples manually extracted by two researchers. (“Precision” measures the percentage of extracted blocks that are actually examples; “Recall” measures the percentage of

actual examples that are correctly extracted.) Precision was mainly affected by misclassifying source examples in other languages (e.g., HTML, JavaScript, and ColdFusion) as MXML or ActionScript. Recall differed among types of Web pages. API reference Web pages, which are often produced automatically, were much easier to parse than tutorial Web pages, which vary greatly in the types of examples they show.

5.2.3.2 Extracting text and running examples

In addition to extracting source code, Blueprint extracts English descriptions and, where possible, running examples for each code segment. Informal inspection of pages containing example code found that the text immediately preceding an example almost always described the example, and running examples almost always occurred after the example code.

To build descriptions, Blueprint iteratively joins the segments immediately preceding the code until any of three conditions is met: 1.) we encounter another code segment, 2.) we encounter a segment indicative of a break in content (those generated by DIV, HR, or heading tags), or 3.) we reach a length threshold (currently 250 words). Using this strategy the English we extract is the correct example description roughly 83% of the time as compared to the descriptions manually extracted by two researchers.

To find running examples, Blueprint analyzes the k segments following a code example. Because we are concerned with Flex, all examples occur as Flash SWF files. We search for references to SWF files in OBJECT and SCRIPT tags. In practice, we have found $k = 3$ works best; larger values resulted in erroneous content, such as Flash-based advertisements.

5.2.3.3 Keeping track of changes to examples

Each time a page is crawled, Blueprint checks for updates to the examples (*e.g.*, bug fixes). It performs an exhaustive, pairwise comparison of examples on the new and old pages using the *diff* tool. Pages typically contain fewer than ten examples. If an example on the new and old pages matches exactly, they are deemed the same. If a new example has more than two-thirds of its lines in common with an old example, it is recorded as changed. Otherwise, the new example is added to the repository. When an example is no longer available on the Web, we keep the cached versions but do not display it as part of search results. The database stores each example with a timestamp, and keeps all previous versions. These timestamps allow Blueprint to notify users when an example changes.

5.3 EVALUATION: STUDYING BLUEPRINT IN THE LAB

We conducted a comparative laboratory study with 20 participants to better understand how Blueprint affects the example-centric development process. The laboratory study evaluated three hypotheses:

H1: Programmers using Blueprint will complete directed tasks more quickly than those who do not because they will find example code faster and bring it into their project sooner.

H2: Code produced by programmers using Blueprint will have the same or higher quality as code written by example modification using traditional means.

H3: Programmers who use Blueprint produce better designs on an exploratory design task than those using a Web browser for code search.

5.3.1 METHOD

We recruited twenty professional programmers through an internal company mailing list and compensated them with a \$15 gift card. The participants had an average of 11.3 years of

professional experience. Fourteen reported at least one year of programming experience with Flex; twelve reported spending at least 25 hours a week programming in Flex.

The participants were given an off-the-shelf installation of Flex Builder, pre-loaded with three project files. The participants in the control condition were provided with the Firefox Web browser; they were asked to use the Adobe Community Help Search engine to look for example code. Participants in the treatment condition were provided with Blueprint to search for code samples; they were not allowed to use a Web browser.

Participants were first asked to complete a *tutorial*, which taught them about the search interface they had been assigned. After completing the tutorial, participants were given a *directed task*, and an *exploratory task*. Participants were told that they would be timed and that they should approach both tasks as though they are prototyping and not writing production-level code. Participants began each task with a project file that included a running application, and they were asked to add additional functionality.

For the *tutorial*, participants were given a sample application that contained an HTML browsing component and three buttons that navigated the browser to three different Web sites. Participants received a written tutorial that guided them through adding fade effects to the buttons and adding a busy cursor. In the control condition, the participants were asked to use the Web browser to find sample code for both modifications. The tutorial described which search result would be best to follow and which lines of code to add to the sample application. In the treatment condition, the participants were asked to use Blueprint to find code samples.

For the *directed programming* task, the participants were instructed to use the *URLLoader* class to retrieve text from a URL and place it in a text box. They were told that

they should complete the task as quickly as possible. In addition, the participants were told that the person to complete the task fastest would receive an additional gift card as a prize. Participants were given 10 minutes to complete this task.

For the *exploratory programming* task, participants were instructed to use Flex Charting Components to visualize an array of provided data. The participants were instructed to make the best possible visualization. They were told that the results would be judged by an external designer and the best visualization would win an extra gift card. Participants were given 15 minutes to complete this task.

To conclude the study, we asked the participants a few questions about their experience with the browsing and searching interface.

5.3.2 RESULTS

This section reports on the results of the comparative study, broken down by task.

5.3.2.1 Directed task

Nine out of ten Blueprint participants and eight out of ten control participants completed the directed task. Because not all participants completed the task and completion time may not be normally distributed, we report all significance tests using rank-based non-parametric statistical methods (Wilcoxon-Mann-Whitney test for rank sum difference and Spearman rank correlation).

We ranked the participants by the time until they pasted the first example (See **Figure 5.4**). Participants using Blueprint pasted code for the first time after an average of 57 seconds, versus 121 seconds for the control group. The rank-order difference in time to first paste was significant ($p < 0.01$). Among finishers, those using Blueprint finished after an

average of 346 seconds, compared to 479 seconds for the control. The rank-order difference for all participants in task completion time was not significant ($p = 0.14$). Participants' first paste time correlates strongly with task completion time ($r_s = 0.52, p = 0.01$). This suggests that lowering the time required to search for, selecting and copying examples will speed development.

A professional software engineer external to the project rank-ordered the participants' code. He judged quality by whether the code met the specifications, whether it included error handling, whether it contained extraneous statements, and overall style. Participants using Blueprint produced significantly *higher-rated* code ($p = 0.02$). We hypothesize this is because the example-centric result view in Blueprint makes it more likely that users will choose a good starting example. When searching for "URLLoader" using the Adobe Community Help search engine, the first result contains the best code. However, this result's *snippet* did not convey that the page was likely to contain sample code. For this reason, we speculate that some control participants overlooked it.

5.3.2.2 Exploratory task

A professional designer rank-ordered the participants' charts. To judge chart quality, he considered the appropriateness of chart type, whether or not all data was visualized, and



Figure 5.4: Comparative laboratory study results. Each graph shows the relative rankings of participants. Participants who used Blueprint are shown as filled squares, those who used Community Help are shown as open squares.

aesthetics of the chart. The sum of ranks was smaller for participants using Blueprint (94 vs. 116), but this result was not significant ($p = 0.21$). While a larger study may have found significance with the current implementation of Blueprint, we believe improvements to Blueprint's interface (described below) would make Blueprint much more useful in exploratory tasks.

5.3.2.3 Areas for improvement

When asked "How likely would you be to install and use Blueprint in its current form?" participants responses averaged 5.1 on a 7-point Likert scale (1 = "not at all likely", 7 = "extremely likely"). Participants also provided several suggestions for improvement.

The most common requests were for greater control over result ranking. Two users suggested that they should be able to rate (and thus affect the ranking of) examples. Three users expressed interest in being able to filter results on certain properties such as whether result has a running example, the type of page that the result was taken from (blog, tutorial, API documentation, etc.), and the presence of comments in the example. Three participants requested greater integration between Blueprint and other sources of data. For example, one participant suggested that all class names appearing in examples be linked to their API page. Finally, three participants requested maintaining a search history; one also suggested a browseable and searchable history of examples used. We implemented the first two suggestions before the field deployment. The third remains future work.

5.3.3 DISCUSSION

In addition to the participants' explicit suggestions, we identified a number of shortcomings as we observed participants working. It is currently difficult to compare multiple examples

using Blueprint. Typically, only one example fits on the screen at a time. To show more examples simultaneously, one could use code-collapsing techniques to reduce each example's length. Additionally, Blueprint could show all running examples from a result set in parallel. Finally, visual differencing tools might help users compare two examples.

We assumed that users would only invoke Blueprint once per task. Thus, each time Blueprint is invoked, the search box and result area would be empty. Instead, we observed that users invoked Blueprint multiple times for a single task (*e.g.*, when a task required several blocks of code to be copied to disparate locations). Results should be persistent, but it should be easier to clear the search box: when re-invoking Blueprint, the terms should be pre-selected so that typing replaces them.

5.3.3.1 Where Blueprint fails

Blueprint does not work equally well for all tasks. In particular, Blueprint is not appropriate for many *learning* tasks (described in Chapter 4). For example, if a programmer wants to learn how to implement drag-and-drop in Flex, Blueprint would be cumbersome to use. Implementing drag-and-drop requires adding about three distinct blocks of code to different parts of one's codebase, and all of these blocks must interact with each other. Code examples for these blocks don't completely convey why each is necessary; explanatory prose makes this much more clear. Blueprint's interface makes it difficult to see multiple examples at the same time, and eliminates much of the related explanatory prose.

5.4 DESIGN SPACE OF WEB TOOLS FOR PROGRAMMERS

Blueprint represents one point in the design space of tools that support programmers as they use the Web (see **Figure 5.5**). We discuss Blueprint’s limitations in the context of this design space and suggest directions for future work.

Task: At a high level, programming comprises: planning and design; implementation; and testing and debugging. Blueprint helps programmers find code that implements desired functionality. Other tasks could (and do) benefit from Web search [79], but are not easily completed with Blueprint’s interface. For example, to decipher a cryptic error message, one may want to use program output as the search query [42].

Expertise: Programmers vary in expertise with the tools they use (*e.g.*, languages and libraries), and their tasks (*e.g.*, implementing a piece of functionality). Because Blueprint presents code-centric results, programmers must have the expertise required to evaluate whether a result is appropriate.

Time scale: We designed Blueprint to make small tasks faster by directly integrating search into the code editor. This removes the activation barrier of invoking a separate tool. While Blueprint can be docked to be persistent, for longer information tasks, the advantages of a richer browser will dominate the time savings of direct integration.

Approach: Programmer Web use can include very directed search tasks as well as exploratory browsing tasks. Given its emphasis on search, the Blueprint prototype is best suited to directed tasks: a well-specified query can efficiently retrieve a desired result. It is possible to use Blueprint for exploratory tasks, such as browsing different types of charts,

however support for such tasks can be improved by incorporating traditional Web browser features such as tabbed browsing and search results sorting and filtering.

Integration Required: Some examples can be directly copied. Others require significant modification to fit the current context. Because Blueprint inserts example code directly into the user's project, it provides the most benefit when example code requires little modification. When a piece of code is part of a larger project, the programmer may need to read more of the context surrounding the code in order to understand how to adapt it.

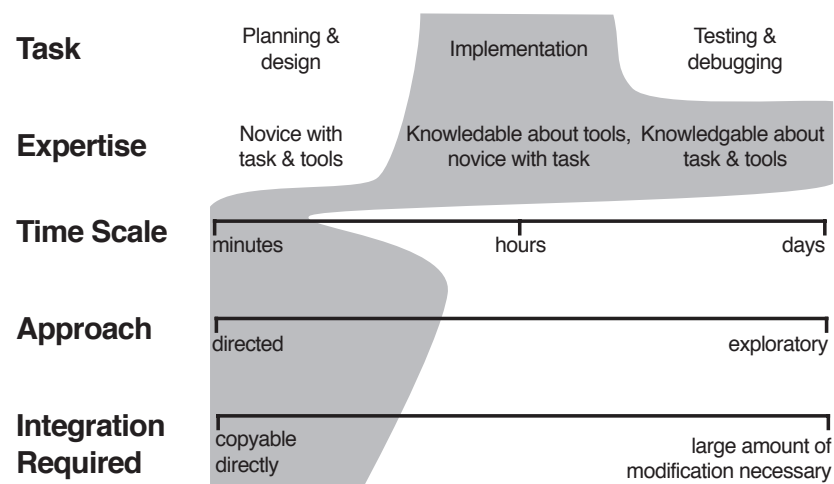


Figure 5.5: Design space of tools to aid programmers' Web use. Blueprint is designed to address the portion of the space shown with a shaded background.

CHAPTER 6 LONGITUDINAL STUDY OF BLUEPRINT: DEPLOYMENT TO 2,024 USERS

To better understand how Blueprint would affect the workflow of real-world programmers, we made Blueprint publicly available on the Adobe Labs Web site (see **Figure 6.1**), and logged its use. After three months, we conducted open-ended interviews with four frequent users. Three themes emerged. First, the interviewees felt that the benefits of consistent, example-centric results outweigh the drawbacks of missing context. Second, they claimed that Blueprint is symbiotic with existing IDE features. Third, they reported using Blueprint primarily to clarify existing knowledge and remind themselves of forgotten details.

To understand whether these three themes applied broadly, we compared Blueprint's query logs to logs from a traditional search interface. We tested three hypotheses: First, if additional context is not necessary, Blueprint queries should have a significantly lower click-through rate. Second, if users are using Blueprint in concert with other IDE features, they are likely querying with code and more Blueprint search terms should contain correctly formatted code. Third, if Blueprint is used for reminders, Blueprint users should repeat queries more

frequently across sessions. Evidence for all three of these hypotheses was found in the logs, indicating that users are searching differently with Blueprint than with traditional tools. These findings suggest that task-specific search interfaces may cause a fundamental shift in how and when individuals search the Web.

Over the course of the deployment, we performed bug fixes and minor design improvements (often based on user feedback through the Adobe Labs Web forum); the main interaction model remained constant throughout the study.

At the completion of the study, we conducted 30-minute interviews with four active Blueprint users to understand how they integrated Blueprint in their workflows. Based on the interviews, we formed three hypotheses, which we tested with the Blueprint usage logs. After evaluating these hypotheses, we performed further exploratory analysis of the logs. This additional analysis provided high-level insight about current use that we believe will help guide future work in creating task-specific search interfaces.

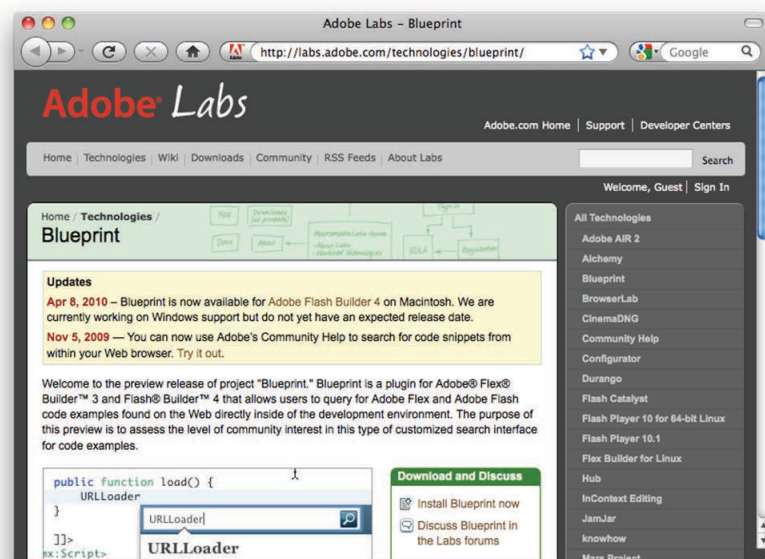


Figure 6.1: Screenshot of the Blueprint Web page on Adobe Labs. Blueprint was made publicly available on May 27, 2009.

6.1 INSIGHTS FROM INTERVIEWING ACTIVE USERS

Our interviews with active users uncovered three broad insights about the Blueprint interface. To understand if these insights generalize, we distilled each insight into a testable hypothesis. The insights and hypotheses are presented here; the results of testing them are presented in the following section.

6.1.1 THE BENEFITS OF CONSISTENT, EXAMPLE-CENTRIC RESULTS OUTWEIGH THE DRAWBACKS OF MISSING CONTEXT.

A consistent view of results makes scanning the result set more efficient. However, in general, removing content from its context may make understanding the content more difficult. None of our interviewees found lack of context to be a problem when using Blueprint. One interviewee walked us through his strategy for finding the right result: “Highlighting [of the search term in the code] is the key. I scroll through the results quickly, looking for my search term. When I find code that has it, I can understand the code much faster than I could English.” We hypothesize that examining code to determine if a result is relevant has a smaller gulf of evaluation [48] than examining English. Presenting results in a consistent manner makes this process efficient.

When users desire additional context for a Blueprint result, they can click through to the original source Web page. This Web page opens in the same window where Blueprint results are displayed. If additional context is rarely necessary, we expect a low click-through rate.

H1: Blueprint will have a significantly lower click-through rate than seen in a standard search engine.

6.1.2 BLUEPRINT IS SYMBIOTIC WITH EXISTING IDE FEATURES

Three interviewees reported using Blueprint as an “extension” to auto-complete. They use auto-complete as an index into a particular object’s functionality, and then use Blueprint to quickly understand how that functionality works. For example, auto-complete would help them select a particular “write” method to call on a File object, and Blueprint would then provide an example for how to call that method as well as related error handling code necessary to use the “write” method robustly. This suggests that embedding search into the development environment creates a symbiotic relationship with other features. Here, auto-complete becomes more useful because further explanation of the auto-complete results is one keystroke away. We believe that this symbiotic relationship is another example of how integrating task-specific search into a user’s existing tools can lower search costs.

Programmers in our lab study routinely search with code terms when using standard search engines (see Chapter 4). However, when these search terms are typed by hand, they frequently contain formatting inconsistencies (*e.g.*, method names used as search terms are typed in all lowercase instead of camelCase). By contrast, when search terms come *directly from* a user’s code (*e.g.*, generated by output from auto-complete), the search terms will be correctly formatted. If Blueprint is being used in a symbiotic manner with other code editing tools, we expect to see a large number of correctly formatted queries.

***H2:** Blueprint search terms will contain correctly formatted code more often than search terms used with a standard search engine.*

6.1.3 BLUEPRINT IS USED HEAVILY FOR CLARIFYING EXISTING KNOWLEDGE AND REMINDING OF FORGOTTEN DETAILS.

One interviewee stated that, using Blueprint, he could find what he needed “60 to 80 percent of the time without having to go to API docs.” He felt that Blueprint fell in the “mid-space

between needing to jump down into API docs when you don't know what you're doing at all and not needing help because you know exactly what you are doing." Other interviewees echoed this sentiment. In general, they felt that Blueprint was most useful when they had some knowledge about how to complete the task at hand, but needed a piece of clarifying information. That is, Blueprint was most useful for the *reminding* and *clarifying* tasks described in the taxonomy from Chapter 4.

In general, understanding a user's search goal from query logs alone is not feasible—there is simply not enough contextual information available [36]. However, if uses of Blueprint tend more toward reminding and clarifying existing knowledge than learning new skills, we expect that users will more commonly repeat queries they have performed in the past.

H3: *Users of Blueprint are more likely to repeat queries across sessions than users of a standard search engine.*

6.2 METHOD

To evaluate these hypotheses, one needs a comparison point. Adobe's Community Help search engine presents a standard Web search interface that is used by thousands of Flex programmers. Furthermore, Community Help uses the same Google Custom Search Engine that is part of Blueprint. In short, Blueprint and Community Help differ in their interaction model, but are similar in search algorithm, result domain, and user base.

We randomly selected 5% of users who used the Community Help search engine over the same period as the Blueprint deployment. We analyzed all logs for these users. In both datasets, queries for individual users were grouped into *sessions*. A session was defined as a sequence of events from the same user with no gaps longer than six minutes (identical to the

definition used in Chapter 4.) Common “accidental” searches were removed (*e.g.*, empty or single-character searches, and identical searches occurring in rapid succession) in both datasets.

We used the z-test for determining statistical significance of differences in means and the chi-square test for determining differences in rates. Unless otherwise noted, all differences are statistically significant at $p < 0.01$.

6.3 RESULTS

Blueprint was used by 2024 individuals during the 82 day deployment, with an average of 25 new installations per day. Users made a total of 17012 queries, or an average of 8.4 queries per user. The 100 most active users made 1888 of these queries, or 18.8 queries per user.

The Community Help query logs used for comparison comprised 13283 users performing 26036 queries, an average of 2.0 queries per user.

H1: Blueprint will have a significantly lower click-through rate than seen in a standard search engine

Blueprint users clicked through to source pages significantly less than Community Help users ($\mu = 0.38$ versus 1.32). To be conservative: the mean of 0.38 for Blueprint is an over-estimate. For technical reasons owing to the many permutations of platform, browser, and IDE versions, click-throughs were not logged for some users. For this reason, this analysis discarded all users with zero click-throughs.

H2: *Blueprint search terms will contain correctly formatted code more often than search terms used with a standard search engine.*

To test this hypothesis, we used the occurrence of camelCase words as a proxy for code terms. The Flex framework’s coding conventions use camelCase words for both class and method names, and camelCase rarely occurs in English words.

Significantly more Blueprint searches contained camelCase than Community Help: 49.6% (8438 of 17012) versus 16.2% (4218 of 26036). The large number of camelCase words in Blueprint searches indicates that many searches are being generated directly from users’ code. This suggests that, as hypothesized, Blueprint is being used in a symbiotic way with other IDE features. The large number of camelCase queries in Blueprint searches also indicates that the majority of searches use precise code terms. This suggests that Blueprint is being used heavily for clarification and reminding, where the user has the knowledge necessary to select precise search terms.

H3: *Users of Blueprint are more likely to repeat queries across sessions than users of a standard search engine.*

Significantly more Blueprint search sessions contained queries that had been issued by the same user in an earlier session than for Community Help: 12.2% (962 of 7888 sessions) versus 7.8% (1601 of 20522 sessions).

6.4 EXPLORATORY ANALYSIS

To better understand how Blueprint was used, we performed additional exploratory analysis of the usage logs. We present our most interesting findings below.

6.4.1 USING BLUEPRINT AS A RESOURCE TO WRITE CODE BY HAND IS COMMON.

A large percentage of sessions (76%) did not contain a copy-and-paste event. There are two possible reasons for this high number: First, as our interviewees reported, we believe Blueprint is commonly used to confirm that the user is on the right path – if they are, they have nothing to copy. Second, sometimes Blueprint’s results aren’t useful. (For technical reasons, copy-and-paste events were not logged on some platforms. The statistic presented here is only calculated amongst users where we could log this event. In this data set, there were 858 sessions that contained copy-and-paste events out of a total of 3572 sessions.)

6.4.2 PEOPLE SEARCH FOR SIMILAR THINGS USING BLUEPRINT AND COMMUNITY HELP, BUT THE FREQUENCIES ARE DIFFERENT.

We examined the most common queries for Blueprint and Community Help and found that there was a large amount of overlap between the two sets: 10 common terms appeared in the top 20 queries of both sets. The relative frequencies, however, differed between sets. As one example, the query “Alert” was significantly more frequent in Blueprint than Community Help. It was 2.2 times more frequent, ranking 8th versus 34th.

The initial result views for search “Alert” for both Blueprint and Community Help are shown in **Figure 6.2**. In the case of this particular search, we believe the difference in frequency is explained by the granularity of the task the user is completing. Namely, this task is small. When a user searches for “Alert,” he is likely seeking the one line of code necessary to display a pop-up alert window. In Blueprint, the desired line is immediately visible and highlighted; in Community Help, the user must click on the first result and scroll part way down the resulting page to find the code. Alerts are often used for debugging, where there are reasonable—but less optimal—alternative approaches (*e.g.*, “trace” statements). It may be

the case that Blueprint's lowered search cost changes user behavior. Users who do not have Blueprint more frequently settle for sub-optimal approaches because of the relatively higher cost of taking the optimal approach.

6.4.3 BOTH INTERFACE MODALITIES ARE IMPORTANT

Users can interact with blueprint either as a pop-up window or inside a docked panel. The default modality is the pop-up window. Users must explicitly dock the pop-up if they wish to use this interface, but may start subsequent search sessions from the docked interface if it is already open. Among all users, 59% of sessions used only the pop-up interface, 9% used only the docked interface, and 32% used both. This suggests that providing both interfaces is important. Furthermore the fact that users frequently switched between interfaces mid-session suggests that some tasks are more appropriate for a particular interface.

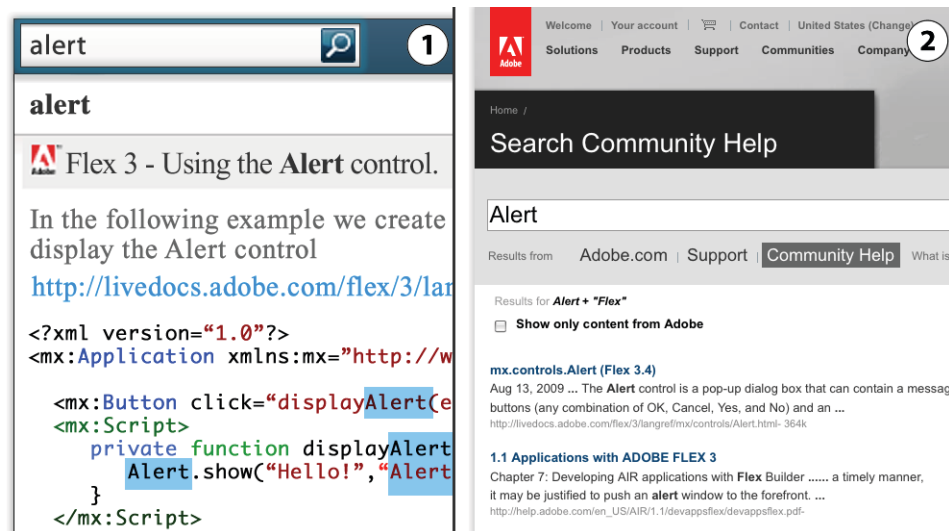


Figure 6.2: Comparison of Blueprint (left) and Community Help (right) search result interfaces for the query "Alert". The desired information is immediately available in Blueprint; Community Help users must click the first result and scroll part way down the page to find the same information.

6.5 USER RETENTION

Are early adopters of Blueprint still using it, or is Blueprint simply an interesting curiosity that users pick up, try a few times, and set aside? The controlled study presented above reports on usage over 3 months. After Blueprint had been available for 200 days, its user base had grown to 3253, with an average of 16.3 new users per day. During this time, the most active third of users (1084) searched with Blueprint over at least a 10-day span. The top 10% of users (325) queried Blueprint over at least a 59-day span, and the top 1% of users (33) queried Blueprint over at least a 151-day span. For legal reasons, we were unable to track users across re-installations of Flex Builder. So, if a user re-installed or upgraded Flex Builder, they were counted as a new user. As such, these numbers under-report actual retention.

6.6 CONCLUSION

To support programming by example modification, this chapter introduced a user interface for accessing online example code from *within the development environment*. It discussed the Blueprint client interface, which displays search results in an *example-centric manner*. The Blueprint server introduced a lightweight architecture for using a general-purpose search engine to create code-specific search results that include written descriptions and running examples. In evaluating Blueprint, we found that it enabled users to search for and select example code significantly faster than with traditional Web search tools. Log analysis from a large-scale deployment with 2,024 users suggested that task-specific search interfaces may cause a fundamental shift in how and when individuals search the Web.

CHAPTER 7 REHEARSE: HELPING PROGRAMMERS UNDERSTAND EXAMPLES

Instructive examples have long played a central role in programming practice [74], and Web search tools like Blueprint (Chapter 5) help programmers to *locate* high-quality examples. However, locating quality examples is just the first step in the example-use pipeline: with a potentially useful example in hand, a programmer must *understand* the example, and then *adapt* the example to her particular use case.

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? While answering this question completely is beyond the scope of this thesis, this chapter presents initial work on developing interaction techniques that help programmers *understand* examples.

Previous research suggests that programmers prefer examples that are complete, executable applications [74]. 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 chapter 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.*

To explore this insight, we built *Rehearse*, which is an extension of the open source Processing development environment [28]. 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 7.1**). 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 7.2**).

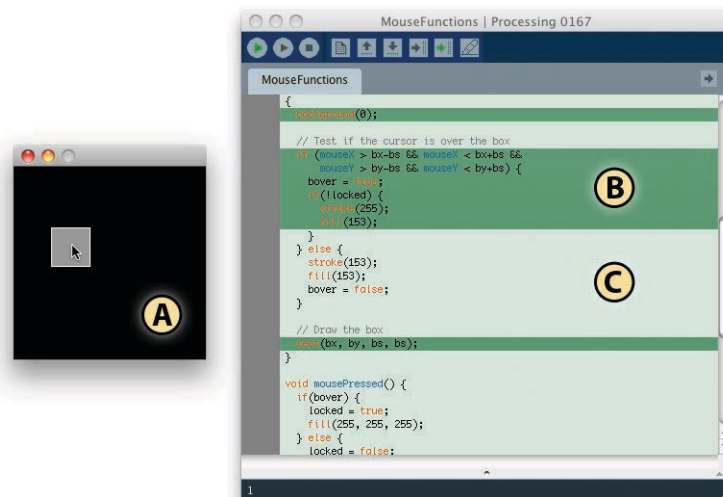


Figure 7.1: The Rehearse development environment, visualizing the execution of an example application. The user interacts with the running application (A). Lines that have recently executed are highlighted in dark green (B). As execution progresses, lines executed less recently fade to light green (C).

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.

7.1 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 observation. 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.

7.1.1 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 identify 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 a small modification to that line and then re-execute the application. This modification was largely epistemic [49]: 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.

7.2 REHEARSE

Rehearse extends the Processing development environment [28] with two interactions designed to support understanding and adapting example code.

7.2.1 EXECUTION HIGHLIGHTING

During execution of the user's program, Rehearse highlights each line of code as it is executed (**Figure 7.1**). 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 realtime, 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.

7.2.2 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 7.2**). 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 described below.

7.2.3 IMPLEMENTATION

The execution highlighting feature of Rehearse was implemented by adding a custom Java interpreter to Processing. Our interpreter is based heavily on BeanShell [65], which was

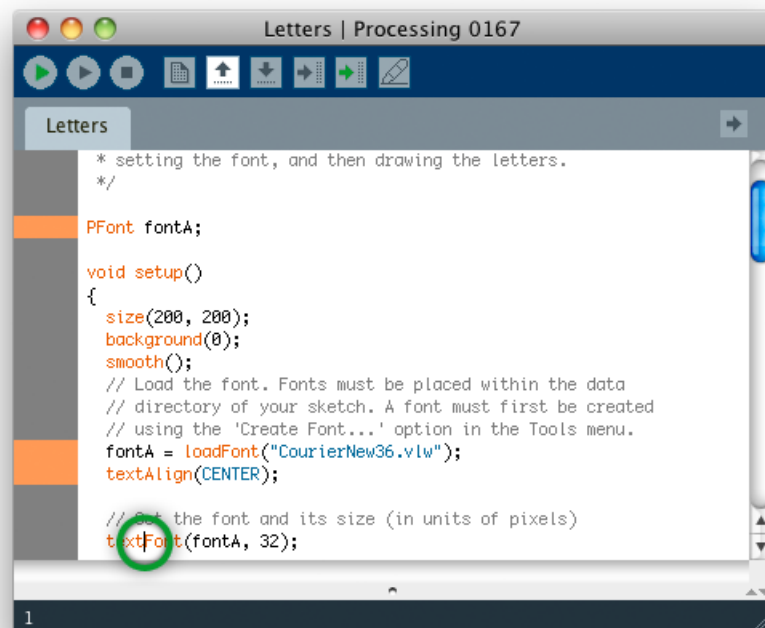


Figure 7.2: Rehearse indicating lines related to the line currently being edited. The user's cursor is circled in green; related lines are identified by orange highlights in the left margin. Display of related lines is triggered by a

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.

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.

7.3 PILOT STUDY OF REHEARSE

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 explore this hypothesis, we ran a pilot lab study.

7.3.1 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.

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.

7.3.2 RESULTS

In task 1, Rehearse users completed the task faster than the control group ($p < 0.06$, Mann-Whitney U test). Control participants completed the task in 18.3 minutes on average; Rehearse users spent 12.6 minutes on average, a 31% speed-up (see **Table 7.1**). One

participant in the treatment group chose not to attempt the first 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 ($p \approx 0.18$, Mann-Whitney U test). One participant in the treatment group chose not to attempt the task, and is not included in these statistics.

7.3.3 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

	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
<i>Average</i>	<i>12.6</i>	<i>18.3</i>	<i>17.4</i>	<i>22.2</i>

Table 7.1: Task completion times for treatment (T) and control (C) participants. Participants using Rehearse completed the first task faster than those in the control condition ($p < 0.06$).

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.

7.4 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.

CHAPTER 8 FUTURE DIRECTIONS

The Web has a substantially different cost structure than other information resources: It is cheaper to search for information, but its diverse nature may make it more difficult to understand and evaluate what is found. Understanding the Web’s role in knowledge work is a broad area of research [18]. This dissertation illustrates an emerging problem solving style that uses Web search to enumerate possible solutions. However, programmers—and likely, other knowledge workers—currently lack tools for rapidly understanding and evaluating these possible solutions. Experimenting with new tools in the “petri dish” of programming may offer further insights about how to better support other knowledge workers.

8.1 TOWARD A COMPLETE PICTURE OF KNOWLEDGE WORK ON THE WEB

This dissertation presented empirical data on how programmers leverage the Web to solve problems while programming. In many respects, programmers are an exemplar form of knowledge worker: their work centers on identifying, breaking down, and solving problems. Web resources will likely play an increasingly important role in problem solving in a broad

range of domains. In order to build a more complete picture of knowledge work on the Web, we must address several related issues.

First, the work presented here looks expressly at the Web. Many additional resources exist, such as colleagues and books. It is clear that different resources have very different cost structures: The cost of performing a Web query is substantially lower than interrupting a colleague, but the latter may provide much better information. More work is needed to fully understand these trade-offs.

Second, it would be valuable to better understand how a programmer's own code is reused between projects. In earlier fieldwork we observed that programmers had a *desire* to reuse code, but found it difficult to do so because of lack of organization and changes in libraries (Chapter 2).

Third, a complete understanding of knowledge work and the Web requires a richer theory of what motivates individuals to *contribute* information, such as tutorials and code snippets. How might we lower the threshold to contribution so that more developers share sample code? Is it possible to “crowdsource” finding and fixing bugs in online code? Can we improve the experience of reading a tutorial by knowing how the previous 1,000 readers used that tutorial? These are just some of the many open questions in this space.

8.2 FURTHER TOOL SUPPORT FOR OPPORTUNISTIC PROGRAMMING

We have identified four broad areas that we believe would benefit from better tool support:

Code Reuse and Adaptation — The Web and tools like Blueprint have made a wealth of example code available and easier to *locate*. Similarly, tools like Rehearse help programmers *understand* examples more easily. The next step in this chain is to help

programmers *adapt* those examples. Our group’s recent work on d.mix explores one potential solution to this problem [39]. d.mix makes it easier for programmers to experiment with Web APIs by allowing them to “sample” existing user interfaces, and then experiment with the resulting code inside a wiki-like sandbox.

Additionally, we may 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 ten programmers to use an example all changed a particular portion of the code, it’s likely that the eleventh programmer should as well.

Debugging — In opportunistic programming, debugging is 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 there is significant value in building debugging tools that embrace the way opportunistic programmers already work. For example, perhaps print statements should 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, perhaps development environments could take advantage of the rapid iteration inherent in opportunistic programming — code that was written 30 seconds ago is likely the code that the programmer wants to test and debug. Perhaps the “execution highlighting” interaction introduced in Chapter 7, could be adapted to provide something like real-time coverage checking. Simply indicating which lines of code were executed during the last run of the program would help programmers avoid time consuming debugging mistakes.

Alternatively, tools may be able to eliminate the need for rapid iteration in specialized cases, such as parameter tuning. Through a tool called Juxtapose, our research group introduced techniques for programmers to easily tune parameter values at runtime [41]. Interactive tuning is particularly valuable for exploring user interface variations, as alternatives can be considered without having to stop execution, edit, compile, execute, and navigate to the previous state.

Version Control — Current version control systems have a large up-front setup and learning cost, and are targeted at supporting 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, *e.g.*, two tests ago, or quickly branch and explore two ideas in parallel. Perhaps single-user version control could be brought inside the editor, eliminating the setup burden of current tools. In such a system, code committal could be performed 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 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 went back to look at code from prior projects, he often reviewed his process. Right now, however, the tools for documenting

process (*e.g.*, a notebook) are independent from the tools actually being used (*e.g.*, Adobe Flash). We believe that bridging this divide is a valuable path for future research.

8.3 THE FUTURE OF PROGRAMMING

Ultimately, opportunistic programming is as much about having the right skills as it is about having the right tools. As tools become better, the skill set required of programmers changes. In the future, programmers may no longer need any training in the language, framework, or library *du jour*. Instead they will likely need ever-increasing skill in formulating and breaking apart complex problems. It may be that programming will become less about knowing how to do something and more about knowing how to ask the right questions.

REFERENCES

1. *Google Code Search*. [cited 2010 November 17] <http://code.google.com>
2. *Krugle*. [cited 2010 November 17] <http://www.krugle.com>
3. *pastebin*. [cited 2010 November 17] <http://pastebin.com/>
4. *Pygments*. [cited 2010 November 17] <http://pygments.org/>
5. *Flex & Flash Builder Help and Support*. [cited 2010 November 9] <http://www.adobe.com/support/flex/>
6. *Adobe Developer Connection*. [cited 2010 November 9] <http://www.adobe.com/devnet.html>
7. *Google Custom Search*. [cited 2010 November 9] <http://www.google.com/cse/>
8. Adar, Eytan, Mira Dontcheva, James Fogarty, and Daniel S. Weld, *Zoetrope: Interacting with the Ephemeral Web*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2008, Monterey, California. p. 239-248.
9. Bajracharya, Sushil, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes, *Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search*, in *Companion to OOPSLA: ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 2006, Portland, Oregon. p. 681-682.
10. Brandt, Joel, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer, *Opportunistic Programming: Writing Code to Prototype, Ideate, and Discover*, in *IEEE Software*. 2009. p. 18-24.
11. Brandt, Joel, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott 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*. 2009, Boston, Massachusetts. p. 1589-1598.
12. Brandt, Joel, Mira Dontcheva, Marcos Weskamp, and Scott 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, Atlanta, Georgia. p. 513-522.
13. Brooks, Frederick P., *The Mythical Man-Month: Essays on Software Engineering*. 1995, Reading, Massachusetts: Addison-Wesley.
 14. Brutlag, Jake D., Hilary Hutchinson, and Maria Stone, *User Preference and Search Engine Latency*, in *Proceedings of QPRC: Quality & Productivity Research Conference*. 2008, Madison, Wisconsin.
 15. Brutlag, Jake D., *Speed Matters for Google Web Search*. 2009.
 16. Carter, Scott, Jennifer Mankoff, Scott R. Klemmer, and Tara Matthews, *Exiting the Cleanroom: On Ecological Validity and Ubiquitous Computing*. Human-Computer Interaction, 2008. **23**(1): p. 47-99.
 17. Chong, Jan and Rosanne Siino, *Interruptions on Software Teams: A Comparison of Paired and Solo Programmers*, in *Proceedings of CSCW: ACM Conference on Computer Supported Cooperative Work*. 2006.
 18. Choo, Chun Wei, Brian Detlor, and Don Turnbull, *Web Work: Information Seeking and Knowledge Work on the World Wide Web*. 2000, Dordrecht: Kluwer Academic Publishers. 219 pp.
 19. Clarke, Steven, *What is an End-User Software Engineer?*, in *End-User Software Engineering Dagstuhl Seminar*. 2007, Dagstuhl, Germany.
 20. Cox, Anna L. and Richard M. Young, *Device-Oriented and Task-Oriented Exploratory Learning of Interactive Devices*. Proceedings of ICCM 2000: International Conference on Cognitive Modeling, 2000: p. 70-77.
 21. Csíkszentmihályi, Mihály, *Flow: The Psychology of Optimal Experience*. 1990, New York: Harper Collins.
 22. Cypher, Allen, *Watch What I Do: Programming by Demonstration*. 1993, Cambridge, Massachusetts: The MIT Press. 652 pp.
 23. Cypher, Allen, Mira Dontcheva, Tessa Lau, and Jeffrey Nichols, *No Code Required: Giving Users Tools to Transform the Web*. 2010, Burlington, Massachusetts: Morgan Kaufmann. 512 pp.

24. deHaan, Peter. *Flex Examples*. [cited 2010 November 17]
<http://blog.flexexamples.com/>
25. Detienne, Françoise, *Software Design: Cognitive Aspects*. 2001, New York: Springer. 146 pp.
26. Dontcheva, Mira, Steven M. Drucker, Geraldine Wade, David Salesin, and Michael F. Cohen, *Summarizing Personal Web Browsing Sessions*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2006, Montreux, Switzerland. p. 115-124.
27. Dontcheva, Mira, Steven M. Drucker, David Salesin, and Michael F. Cohen, *Relations, Cards, and Search Templates: User-Guided Web Data Integration and Layout*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2007, Newport, Rhode Island. p. 61-70.
28. Fry, Ben and Casey Reas. *Processing*. <http://processing.org>
29. Gaul, Troy. *Lightroom Exposed, Presentation at C4 Macintosh Development Conference*. [cited 2010 November 9] <http://www.viddler.com/explore/rentzsch/videos/37>
30. Gentner, D., *Mental Models, Psychology of*. International Encyclopedia of the Social and Behavioral Sciences, 2002: p. 9683-9687.
31. Gentner, D., J. Loewenstein, and L. Thompson, *Learning and Transfer: A General Role for Analogical Encoding*. Journal of Educational Psychology, 2003. **95**(2): p. 393-408.
32. Gentner, Dedre, Keith J. Holyoak, and Boicho N. Kokinov, *The Analogical Mind: Perspectives from Cognitive Science*. 2001, Cambridge: MIT Press.
33. Gick, M. L. and Keith J. Holyoak, *Schema Introduction and Analogical Transfer*. Cognitive Psychology, 1983.
34. Goldman, Max and Robert C. Miller, *Codetrail: Connecting Source Code and Web Resources*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2008, Herrsching am Ammersee, Germany. p. 65-72.
35. Gray, Wayne D. and Deborah A. Boehm-Davis, *Milliseconds Matter: An Introduction to Microstrategies and to Their Use in Describing and Predicting Interactive Behavior*. Journal of Experimental Psychology: Applied, 2000. **6**(4): p. 322-335.

36. Grimes, Carrie, Diane Tang, and Daniel M. Russell, *Query Logs Alone are Not Enough*, in *Workshop on Query Log Analysis at WWW 2007: International World Wide Web Conference*. 2007, Banff, Alberta.
37. Gross, Paul A., Micah S. Herstand, Jordana W. Hodges, and Caitlin L. Kelleher, *A Code Reuse Interface for Non-Programmer Middle School Students*, in *Proceedings of IUI: International Conference on Intelligent User Interfaces*. 2010, Hong Kong, China. p. 219-228.
38. Hartmann, Björn, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher, and Jennifer Gee, *Reflective Physical Prototyping through Integrated Design, Test, and Analysis*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2006, Montreux, Switzerland. p. 299-308.
39. Hartmann, Björn, Leslie Wu, Kevin Collins, and Scott R. Klemmer, *Programming by a Sample: Rapidly Creating Web Applications with d.mix*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2007, Newport, Rhode Island. p. 241-250.
40. Hartmann, Björn, Scott Doorley, and Scott R. Klemmer, *Hacking, Mashing, Gluing: Understanding Opportunistic Design*, in *IEEE Pervasive Computing*. 2008. p. 46-54.
41. Hartmann, Björn, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R. Klemmer, *Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2008, Monterey, California.
42. Hartmann, Björn, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer, *What Would Other Programmers Do? Suggesting Solutions to Error Messages*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2010, Atlanta, Georgia.
43. Hearst, Marti A., *Search User Interfaces*. 2009, Cambridge: Cambridge University Press.
44. Hoffmann, Raphael, James Fogarty, and Daniel S. Weld, *Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2007, Newport, Rhode Island. p. 13-22.
45. Hollan, James, Edwin Hutchins, and David Kirsh, *Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research*. *ACM Transactions on Computer-Human Interaction*, 2000. 7(2): p. 174-196.

46. Hopper, Grace, *Keynote Address*, in *History of Programming Languages*, Richard L. Wexelblat, Editor. 1981, New York: ACM Press. p. 7-20.
47. Houde, Stephanie and Charles Hill, *What do Prototypes Prototype?*, in *Handbook of Human-Computer Interaction*, Martin G. Helander, Thomas K. Landauer, and Prasad V. Prabhu, Editors. 1997, Amsterdam: Elsevier Science. p. 367-381.
48. Hutchins, Edwin L., James D. Hollan, and Donald A. Norman, *Direct Manipulation Interfaces*. *Human-Computer Interaction*, 1985. **1**(4): p. 311-338.
49. Kirsh, David and Paul Maglio, *On Distinguishing Epistemic from Pragmatic Action*. *Cognitive Science*, 1994. **18**(4): p. 513-549.
50. Ko, Andrew J., Brad A. Myers, and Htet Htet Aung, *Six Learning Barriers in End-User Programming Systems*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2004, Rome, Italy. p. 199-206.
51. Ko, Andrew J. and Brad A. Myers, *Finding Causes of Program Output with the Java Whyline*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2009, Boston, Massachusetts. p. 1569-1578.
52. LaToza, Thomas D., Gina Venolia, and Robert DeLine, *Maintaining Mental Models: A Study of Developer Work Habits*, in *Proceedings of ICSE: International Conference on Software Engineering*. 2006, Shanghai, China. p. 492-501.
53. Lau, Tessa and Eric Horvitz, *Patterns of Search: Analyzing and Modeling Web Query Refinement*, in *Proceedings of UM: International Conference on User Modeling*. 1999, Banff, Alberta, Canada. p. 119-128.
54. Lieberman, Henry, *Your Wish Is My Command: Programming by Example*. 2001, San Francisco: Morgan Kaufmann. 448 pp.
55. Lieberman, Henry, Fabio Paternò, and Volker Wulf, *End-User Development*. 2006, New York: Springer. 492 pp.
56. Lin, James, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau, *End-User Programming of Mashups with Vegemite*, in *Proceedings of IUI: International Conference on Intelligent User Interfaces*. 2009, Sanibel Island, Florida. p. 97-106.
57. Little, Greg and Robert C. Miller, *Translating Keyword Commands into Executable Code*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2006, Montreux, Switzerland. p. 135-144.

58. Little, Greg, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan, *Koala: Capture, Share, Automate, Personalize Business Processes on the Web*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2007, San Jose, California. p. 943-946.
59. MacLean, Allan, Kathleen Carter, Lennart Löfvstrand, and Thomas Moran, *User-Tailorable Systems: Pressing the Issues with Buttons*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 1990, Seattle, Washington. p. 175-182.
60. Mandelin, David, Lin Xu, Rastislav Bodík, and Doug Kimelman, *Jungloid Mining: Helping to Navigate the API Jungle*, in *Proceedings of PLDI: ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2005, Chicago, Illinois. p. 48-61.
61. Martin, Robert C., *Agile Software Development, Principles, Patterns, and Practices*. 2002, Upper Saddle River, New Jersey: Prentice-Hall. 529 pp.
62. Mayer, Richard E., *The Psychology of How Novices Learn Computer Programming*. ACM Computing Surveys, 1981. **13**(1): p. 121-141.
63. Medynskiy, Yevgeniy, Mira Dontcheva, and Steven M. Drucker, *Exploring Websites through Contextual Facets*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2009, Boston, Massachusetts. p. 2013-2022.
64. Myers, Brad, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko, *How Designers Design and Program Interactive Behaviors*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2008. p. 177-184.
65. Niemeyer, Pat. *BeanShell*. <http://www.beanshell.org>
66. Novick, L., *Analogical Transfer, Problem Similarity, and Expertise*. Journal of Experimental Psychology, Learning, Memory, and Cognition, 1988. **13**(3): p. 510-520.
67. O'Brien, Timothy M. *Dead Time (...code, compile, wait, wait, wait, test, repeat)*. 2006 [cited 2010 November 17] http://www.oreillynet.com/onjava/blog/2006/03/dead_time_code_compile_wait_wa.html
68. O'Hara, Kenton P. and Stephen J. Payne, *The Effects of Operator Implementation Cost on Planfulness of Problem Solving and Learning*. Cognitive Psychology, 1998. **35**(1): p. 34-70.

69. Oney, Stephen and Brad Myers, *FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2009, Corvallis, Oregon. p. 105-108.
70. Ousterhout, John K., *Scripting: Higher-Level Programming for the 21st Century*. IEEE Computer, 1998: p. 23-30.
71. Pirolli, Peter L. T., *Information Foraging Theory*. 2007, Oxford: Oxford University Press.
72. Reason, James, *Human Error*. 1990, Cambridge: Cambridge University Press.
73. Richardson, Leonard. *Beautiful Soup*.
<http://www.crummy.com/software/BeautifulSoup>
74. Rosson, Mary Beth and John M. Carroll, *The Reuse of Uses in Smalltalk Programming*. TOCHI: ACM Transactions on Human-Computer Interaction, 1996. **3**(3): p. 219-253.
75. Sahavechaphan, Naiyana and Kajal Claypool, *XSnippet: Mining for Sample Code*, in *Proceedings of OOPSLA: ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 2006. p. 413-430.
76. Scaffidi, Christopher, Mary Shaw, and Brad A. Myers, *Estimating the Numbers of End Users and End User Programmers*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2005, Dallas, Texas. p. 207-214.
77. Schrage, Michael, *Serious Play: How the World's Best Companies Simulate to Innovate*. 1999, Boston: Harvard Business School Press. 244 pp.
78. Silverstein, Craig, Hannes Marais, Monika Henzinger, and Michael Moricz, *Analysis of a Very Large Web Search Engine Query Log*. ACM SIGIR Forum, 1999. **33**(1): p. 6-12.
79. Stylos, Jeffrey and Brad A. Myers, *Mica: A Web-Search Tool for Finding API Components and Examples*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2006, Brighton, United Kingdom. p. 195-202.
80. Teevan, Jaime, Edward Cutrell, Danyel Fisher, Steven M. Drucker, Gonzalo Ramos, Paul André, and Chang Hu, *Visual Snippets: Summarizing Web Pages for Search and Revisitation*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2009, Boston, Massachusetts. p. 2023-2032.

81. Thummalapenta, Suresh and Tao Xie, *PARSEweb: A Programmer Assistant for Reusing Open Source Code on the Web*, in *Proceedings of ASE: IEEE/ACM International Conference on Automated Software Engineering*. 2007, Atlanta, Georgia. p. 204-213.
82. Turkle, Sherry and Seymour Papert, *Epistemological Pluralism: Styles and Voices within the Computer Culture*. *Signs: Journal of Women in Culture and Society*, 1990. **16**(1).
83. Wing, Jeannette M, *Computational Thinking*, in *Communications of the ACM*. 2006. p. 33-35.
84. Wong, Jeffrey and Jason I. Hong, *Marmite: Towards End-User Programming for the Web*, in *Proceedings of VL/HCC: IEEE Symposium on Visual Languages and Human-Centric Computing*. 2007. p. 270-271.
85. Woodruff, Allison, Andrew Faulring, Ruth Rosenholtz, Julie Morrisson, and Peter Pirolli, *Using Thumbnails to Search the Web*, in *Proceedings of CHI: ACM Conference on Human Factors in Computing Systems*. 2001, Seattle, Washington. p. 198-205.
86. Yeh, Ron B., Andreas Paepcke, and Scott R. Klemmer, *Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces*, in *Proceedings of UIST: ACM Symposium on User Interface Software and Technology*. 2008, Monterey, California.