Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code

Joel Brandt[1,2], Philip J. Guo[1], Joel Lewenstein[1], Mira Dontcheva[2], Scott R. Klemmer[1]
[1]Stanford University HCI Group
Computer Science Department
Stanford, CA 94305
{jbrandt, pg, jlewenstein, srk}@cs.stanford.edu
[2]Advanced Technology Labs
Adobe Systems
San Francisco, CA 94103
mirad@adobe.com

ABSTRACT
This paper investigates the role of online resources in problem solving. We look specifically at how programmers—an exemplar form of knowledge workers—opportunistically interleave Web foraging, learning, and writing code. We describe two studies of how programmers use online resources. The first, conducted in the lab, observed participants' Web use while building an online chat room. We found that programmers leverage online resources with a range of intentions: They engage in *just-in-time learning* of new skills and approaches, *clarify and extend* their existing knowledge, and *remind* themselves of details deemed not worth remembering. The results also suggest that queries for different purposes have different styles and durations. Do programmers' queries "in the wild" have the same range of intentions, or is this result an artifact of the particular lab setting? We analyzed a month of queries to an online programming portal, examining the lexical structure, refinements made, and result pages visited. Here we also saw traits that suggest the Web is being used for learning and reminding. These results contribute to a theory of online resource usage in programming, and suggest opportunities for tools to facilitate online knowledge work.

Author Keywords
opportunistic programming, prototyping, copy-and-paste

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces—*prototyping; user-centered design*

INTRODUCTION
"Good grief, I don't even remember the syntax for forms!" Less than a minute later, this participant in our Web programming lab study 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 used the Web 27 times, accounting for 28% of the total time she spent building her application. This participant's behavior is illustrative of programmers' increasing use of the Web as a problem-solving tool. How and why do people leverage online resources while programming?

Web use is integral to an *opportunistic* approach to programming that emphasizes speed and ease of development over code robustness and maintainability [4, 13, 8]. Programmers do this 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 [4, 8, 30, 25, 27]. Scaffidi, Shaw, and Myers estimate that in 2012 there will be 13 million people in the USA that describe themselves as "programmers", while the Bureau of Labor Statistics estimates that there will only be 3 million "professional programmers" [30]. We believe there is significant value in understanding and designing for this large population of amateur programmers.

To create software more quickly, programmers often take a bricolage approach by tailoring or mashing up existing systems [33, 21, 23, 34, 14]. As part of this process, they must often search for suitable components and learn new skills [4]. Recently, programmers began using the Web for this purpose [32, 15]. How do these individuals forage for online resources, and how is Web use integrated into the broader task of programming? This paper contributes the first strong empirical evidence of how programmers use online resources in practice.

We present the results of two studies that investigate how programmers leverage online resources. The first asked 20 programmers to rapidly prototype a Web application in the lab. The second quantitatively analyzed a month-long sample of Web query data. 24,293 programmers produced the 101,289 queries in the sample. We employed this mixed-methods approach to gather data that is both contextually rich and authentic [12, 5].

RELATED WORK

This paper builds on three bodies of related work: studies of how programmers reason and learn, investigations of code copying and reuse, and the design of systems that help programmers better leverage the Web.

There is a long history of research on cognitive aspects of programming, summarized well in Détienne's book [11] and Mayer's survey on how novices learn to program [26]. Most relevant to our work, Ko *et al.* observed novice programmers for a semester as they learned to use Visual Basic .NET [19]. 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—design, selection, coordination, use, understanding, and information—and suggested ways that tools could lower these barriers. This work is largely complementary to ours—while they provide insight into the problems that programmers face, there is little discussion of how programmers currently overcome these barriers.

Prior research in software engineering has studied code cloning *within* software projects through both automated [3, 10] and ethnographic [18] approaches. Many of Kim *et al.*'s insights—most notably that it would be valuable for tools to record and visualize dependencies created when copying and pasting code—could prove valuable when designing tools for opportunistic programming. However, because this software engineering research has been focused on minimizing intra-project duplicated code to reduce maintenance costs [17], it has generally ignored the potential value of copying code for learning and for between-project usage.

There has been recent interest in building improved Web search and data mining tools for programmers [32, 29, 15, 2]. Stylos and Myers describe how programmers may learn APIs, based on observations of three "small programming projects" [32]. They suggest that programmers begin with initial design ideas, gain a high-level understanding of potential APIs to use, and then finalize the details by finding and integrating examples, which may cause them to return to earlier steps. The authors suggest that programmers use the Web at all three stages, but in very different ways at each stage. As part of designing a Web search tool for programmers, Hoffmann *et al.* classified Web search sessions about Java programming into 11 search goals (*e.g.* beginner tutorials, APIs, and language syntax) [15]. We extend this literature by providing richer data, a clearer picture of *how* programmers go about performing these searches, and how they leverage foraged Web content.

Several systems use data-mining techniques to locate or synthesize example code. 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 [29]. 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 [24]. A limitation of this approach is that the generated code lacks the comments, context, and explanatory prose found in tutorials.

		Self-Rated Proficiency				Tasks Completed				
Subject #	Experience	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 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".

STUDY 1: OPPORTUNISTIC PROGRAMMING IN THE LAB

We conducted an exploratory study in our lab to understand how programmers leverage online resources, especially for rapid prototyping.

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 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 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 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 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.

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 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.

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 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.

Results

All participants used the Web extensively (see Figure 3). On average, each 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 2). 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.

Intentions behind 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 new 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 2 for a summary).

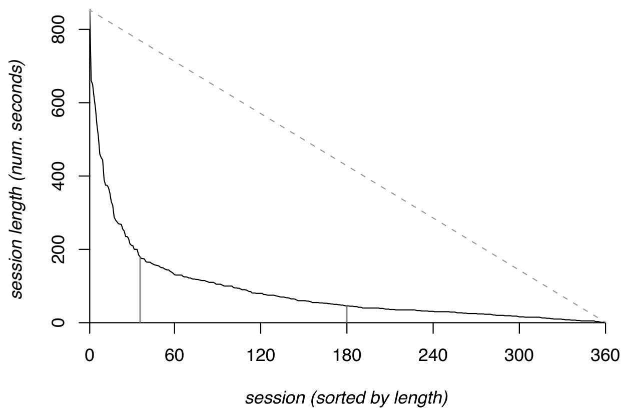


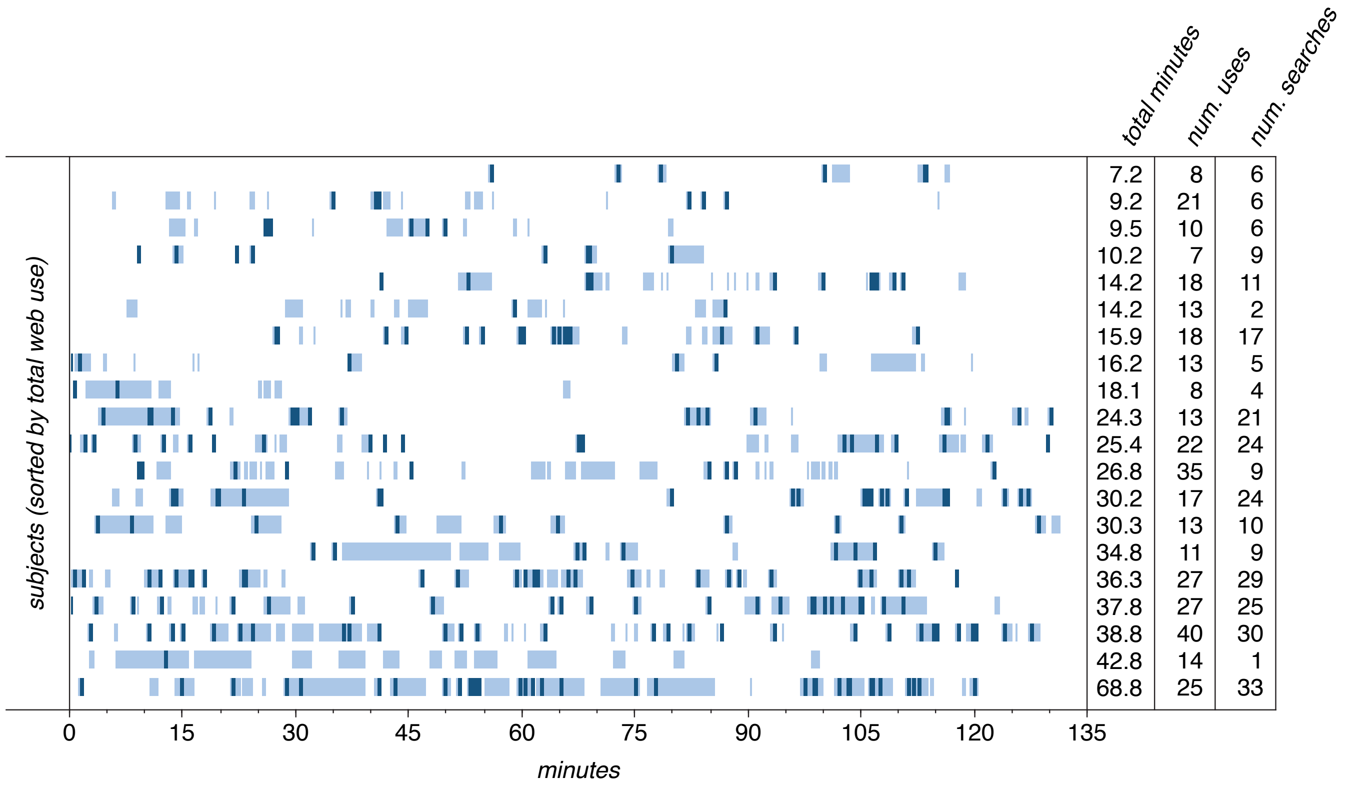
Figure 2. All 360 Web use sessions amongst the 20 participants in our lab study, sorted and plotted by decreasing length (in seconds). The left vertical bar represents the cutoff separating the 10% longest sessions, and the right bar the cutoff for 50% of sessions. The dotted line represents a hypothetical uniform distribution of session lengths.

Scaffolds for learning-by-doing

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 terms seen on the query result page: In the above example, the participant refined the query to "ajax update php".

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



total minutes	num. uses	num. searches
7.2	8	6
9.2	21	6
9.5	10	6
10.2	7	9
14.2	18	11
14.2	13	2
15.9	18	17
16.2	13	5
18.1	8	4
24.3	13	21
25.4	22	24
26.8	35	9
30.2	17	24
30.3	13	10
34.8	11	9
36.3	27	29
37.8	27	25
38.8	40	30
42.8	14	1
68.8	25	33

Figure 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.

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 [28].

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, 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 [9]. 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. Placing example code into one's project amounts to picking up a 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 [35].

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 introduc-

WEB SESSION INTENTION:	LEARNING	CLARIFICATION	REMINDER
Reason for using Web	Just-in-time learning of unfamiliar concepts	Connect high-level knowledge to implementation details	Substitute for memorization (*e.g.*, language syntax or function usage lookup)
Web session length	Tens of minutes	$\sim$ 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 (*e.g.*, function names, language keywords)
Example search	"ajax tutorial"	"javascript timer"	"mysql_fetch_array"
Num. result clicks	Usually several	Fewer	Usually zero or one
Num. query refinements	Usually several	Fewer	Usually zero
Types of webpages 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 (*e.g.*, from tutorial snippets)	Several lines	Varies
Immediately test copied code?	Yes	Not usually, often trust snippets	Varies

Table 2. Summary of characteristics of three points on the spectrum of Web use intention.

tion presented 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 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, "No. 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 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 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.

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 for reminders (or not): 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 a word in a long function name, a Web search allowed him to quickly confirm the exact name of the function simply by browsing the snippets in the results page. Other times, participants would view a page without searching at all. 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 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 [16], allowing them to devote more mental energy to higher-level tasks.

STUDY 2: WEB SEARCH LOG ANALYSIS

Do query styles in the real world robustly vary with intent, or is this result an artifact of the particular lab setting? To investigate this, we 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 on Adobe's Developer Network Web site. This portal indexes documentation, articles, blogs, and forums by Adobe and vetted third-party sources [1].

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

1. *Learning* sessions begin with natural language queries more often than *reminding* sessions.
2. Users more frequently refine queries without first viewing results when *learning* than when *reminding*.
3. Programmers are more likely to visit official API documentation in *reminding* sessions.
4. 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.

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.* [31].)

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 volumenous 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. The appendix gives a description of how we computed each property.

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*.

For the final property, 10,909 of the most frequently visited pages were hand-classified (out of 19,155 total), accounting for 80% of all visits. Result clicks for the remaining 8246 pages (20% of visits) were labeled as *unclassified*.

Type of first query	Session type: learning	Session type: reminding	All hand-coded
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 3. For hand-coded sessions of each type, proportion of first queries of each type (252 total sessions). Significant majorities across each row in bold, * entry means only significant at $p < 0.05$.

Result click	Session type		All
Web page type	learning	reminding	hand-coded
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. For queries in hand-coded sessions of each type, proportion of result clicks to Web sites of each type (401 total queries). Significant majorities across each row in bold.

Results

Out of 300 sessions, 20 appeared misgrouped, and we were unsure of the intent of 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.

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 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). Notably, in *reminding* sessions, 42% of results viewed were Adobe tutorials.

H4: Code-only queries accounted for 51% of all *reminding* queries. 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.

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). This is notably less than other reports, *e.g.*, 2.02 [31]. This may be a function of improving search engines, that programming as a domain is well-suited to search, or that the participants were skilled.

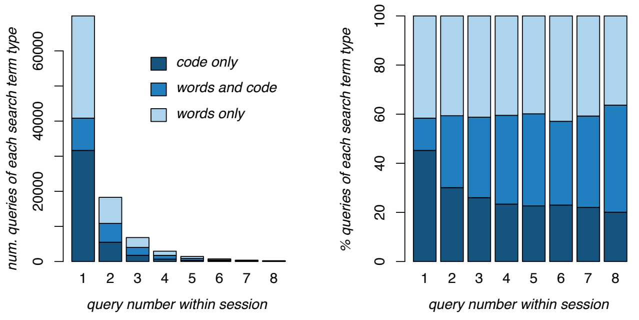


Figure 4. How query types changed as queries were refined. In both graphs, each bar sums all ith queries over all sessions that contained an ith query (*e.g.* a session with three queries contributed to the sums in the first three bars). The graph on the left is a standard histogram; the graph on the right presents the same data, but with each bar's height normalized to 100 to show changes in proportions as query refinements occurred.

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 5). Presumably, this is because they 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). 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.

Query type predicts types of pages visited

There is some quantitative support for the intuition that query type is indicative of query intent (see Table 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

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

Table 5. For each refinement type, proportion of refinements of that type where programmers clicked on on any links *prior* to the refinement (31,334 total refinements).

Result click Web page type	query type code	nat. lang. & code	nat. lang.	All clicks
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 6. For queries of each type, proportion of result clicks leading programmer to Web pages of each type (107,343 total queries). Significant majorities and near-ties across each row in bold.

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).

DISCUSSION

This section presents five insights from these studies and suggests the import of each for programming tools. It then discusses broader implications of the work and limitations of the findings.

Five Key Insights and Implications for Tools

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 [14]. This code can then be modified inside a hosted sandbox.

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 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 [22] to create authoring environments that allow the programmer to type "sloppy" commands which are automatically transformed into syntactically correct code using Web search.

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.

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.

Knowledge Work on the Web

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 [7]. This paper 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 all knowledge workers.

Limitations

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.* [32, 15]), 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 paper 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 [6]). Future work is needed to compare the trade-offs of these different information resources.

CONCLUSIONS AND FUTURE WORK

We have presented empirical data on how programmers, as an exemplar form of knowledge workers, leverage the Web to solve problems while programming. Web resources will likely play an increasingly important role in problem solving; throughout the paper, we have suggested several directions for tools research.

This research also suggests several directions for future empirical work. 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 [4].

Third, understanding 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? 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.

Finally, how does the increasing prevalence and accessibility of Web resources change the way we teach people to program? The skill set required of programmers is changing rapidly—they may no longer need any training in the language, framework, or library *du jour*, but instead may need ever-increasing skill in formulating and breaking apart complex problems. It may be that programming is becoming less about knowing how to do something and more about knowing how to ask the right questions.

ACKNOWLEDGEMENTS

We thank Rob Liebscher and Diana Joseph at Adobe Systems for their help in acquiring the Web query logs; Beyang Liu for his help in coding video data from our lab study; Intel for donating PCs for this research; and all of the study participants for sharing their insights. This research was supported in part by NSF Grant IIS-0745320.

APPENDIX: QUERY LOG ANALYSIS METHOD DETAILS

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*.

Determining Query Refinement Method

We classified refinements into five types, roughly following the taxonomy of Lau and Horvitz [20]. 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.

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. We did not hand-classify the rest of the pages because the cost of additional manual effort outweighed the potential marginal benefits.

REFERENCES

1. Adobe Flex Developer Center, 2008. http://www.adobe.com/devnet/flex/.
2. S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: A Search Engine for Open Source Code Supporting Structure-Based Search. In *Companion to OOPSLA 2006: ACM Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 681–682, Portland, Oregon, 2006.
3. I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of ICSM 1998: IEEE International Conference on Software Maintenance*, page 368, Washington, D.C., USA, 1998.
4. J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer. Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice. In *WEUSE 2008: International Workshop on End-User Software Engineering*, pages 1–5, Leipzig, Germany, 2008.
5. S. Carter, J. Mankoff, S. R. Klemmer, and T. Matthews. Exiting the Cleanroom: On Ecological Validity and Ubiquitous Computing. *Human-Computer Interaction*, 23(1):47–99, 2008.
6. J. Chong and R. Siino. Interruptions on Software Teams: A Comparison of Paired and Solo Programmers. In *Proceedings of CSCW 2006: ACM Conference on Computer Supported Cooperative Work*, 2006.
7. C. W. Choo, B. Detlor, and D. Turnbull. *Web Work: Information Seeking and Knowledge Work on the World Wide Web*. Kluwer Academic Publishers, 2000.
8. S. Clarke. What is an End-User Software Engineer? In *End-User Software Engineering Dagstuhl Seminar*, Dagstuhl, Germany, 2007.

9. A. L. Cox and R. M. Young. Device-Oriented and Task-Oriented Exploratory Learning of Interactive Devices. *Proceedings of ICCM 2000: International Conference on Cognitive Modeling*, pages 70–77, 2000.

10. S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of ICSM 1999: IEEE International Conference on Software Maintenance*, page 109, Oxford, England, 1999.

11. Françoise Détienne. *Software Design: Cognitive Aspects*. Springer, 2001.

12. C. Grimes, D. Tang, and D. M. Russell. Query Logs Alone are Not Enough. In *Workshop on Query Log Analysis at WWW 2007: International World Wide Web Conference*, Banff, Alberta, Canada, 2007.

13. B. Hartmann, S. Doorley, and S. R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing*, September 2008.

14. B. Hartmann, L. Wu, K. Collins, and S. R. Klemmer. Programming by a Sample: Rapidly Creating Web Applications with d.mix. In *Proceedings of UIST 2007: ACM Symposium on User Interface Software and Technology*, pages 241–250, Newport, Rhode Island, 2007.

15. R. Hoffmann, J. Fogarty, and D. S. Weld. Assieme: Finding and Leveraging Implicit References in a Web Search Interface for Programmers. In *Proceedings of UIST 2007: ACM Symposium on User Interface Software and Technology*, pages 13–22, Newport, Rhode Island, 2007.

16. J. Hollan, E. Hutchins, and D. Kirsh. Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research. *ACM Transactions on Computer-Human Interaction*, 7(2):174–196, 2000.

17. A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, October 1999.

18. M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of ISESE 2004: IEEE International Symposium on Empirical Software Engineering*, pages 83–92, Redondo Beach, California, 2004.

19. A. J. Ko, B. A. Myers, and H. H. Aung. Six Learning Barriers in End-User Programming Systems. In *Proceedings of VL/HCC 2004: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, Rome, Italy, 2004.

20. T. Lau and E. Horvitz. Patterns of Search: Analyzing and Modeling Web Query Refinement. In *Proceedings of UM 1999: International Conference on User Modeling*, pages 119–128, Banff, Alberta, Canada, 1999.

21. H. Lieberman, F. Paternò, and V. Wulf. *End-User Development*. Springer, October 2006.

22. G. Little and R. C. Miller. Translating Keyword Commands into Executable Code. In *Proceedings of UIST 2006: ACM Symposium on User Interface Software and Technology*, pages 135–144, Montreux, Switzerland, 2006.

23. A. MacLean, K. Carter, L. Lövstrand, and T. Moran. User-Tailorable Systems: Pressing the Issues with Buttons. In *Proceedings of CHI 1990: ACM Conference on Human Factors in Computing Systems*, pages 175–182, Seattle, Washington, 1990. ACM.

24. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid Mining: Helping to Navigate the API jungle. In *Proceedings of PLDI 2005: ACM Conference on Programming Language Design and Implementation*, pages 48–61, Chicago, Illinois, 2005.

25. R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, 2002.

26. R. E. Mayer. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13(1):121–141, 1981.

27. B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko. How Designers Design and Program Interactive Behaviors. In *Proceedings of VL/HCC 2008: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 177–184, 2008.

28. P. L. T. Pirolli. *Information Foraging Theory*. Oxford University Press, Oxford, England, 2007.

29. N. Sahavechaphan and K. Claypool. XSnippet: Mining for Sample Code. *ACM SIGPLAN Notices*, 41(10):413–430, 2006.

30. C. Scaffidi, M. Shaw, and B. A. Myers. Estimating the Numbers of End Users and End User Programmers. pages 207–214, Dallas, Texas, 2005.

31. C. Silverstein, H. Marais, M. Henzinger, and M. Moricz. Analysis of a Very Large Web Search Engine Query Log. *ACM SIGIR Forum*, 33(1):6–12, 1999.

32. J. Stylos and B. A. Myers. Mica: A Web-Search Tool for Finding API Components and Examples. In *Proceedings of VL/HCC 2006: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, Brighton, United Kingdom, 2006.

33. S. Turkle and S. Papert. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs: Journal of Women in Culture and Society*, 16(1), 1990.

34. J. Wong and J. I. Hong. Marmite: Towards End-User Programming for the Web. In *Proceedings of VL/HCC 2007: IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 270–271, 2007.

35. R. B. Yeh, A. Paepcke, and S. R. Klemmer. Iterative Design and Evaluation of an Event Architecture for Pen-and-Paper Interfaces. In *Proceedings of UIST 2008: ACM Symposium on User Interface Software and Technology*, Monterey, California, 2008.