An Empirical Investigation 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, Stanford, CA 94025
{jbrandt, pg, jlewenstein, srk}@cs.stanford.edu
[2]Adobe Systems
601 Townsend, 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 study, conducted in the lab, found that programmers leverage the Web with three distinct intentions: They engage in *just-in-time learning* of new skills and approaches, they *extend their skills*, and they *strategically delegate their memory to online resources*. The results also suggest that queries for different purposes have different styles and durations. Do query styles robustly vary with intent, or is this result an artifact of the particular lab setting? To address this question, we analyzed a month-long set of Web queries to a commercial programming framework's online information sources. In this dataset, query style also corresponded to intent. These results contribute to a theory of online resource usage in programming, and suggest opportunities for tools to facilitate opportunistic programming.

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 leveraged the Web 27 times, accounting for 28% of the total time she spent building her application. This participant's behavior is indicative 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 an integral part of a broader, opportunistic approach to programming, where programmers emphasize speed and ease of development over code robustness and maintainability [4, 11, 6]. 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 their laboratory experiments, when entrepreneurs assemble complex spreadsheets to better understand how their business is operating, and when professionals adapt Agile development methods to build applications quickly [4, 6, 28, 24].

Opportunistic programming is often (though not exclusively) undertaken by non-professional programmers, many of whom are called "end-user" programmers [19, 20, 25, 8]. These users frequently tailor and "mash-up" existing systems to create applications that better meet their needs [22, 31]. 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" [28]. Programming by modification has long been part of the end-user programming world view [20]. It is well known that programmers create new systems by modifying existing ones, and recent papers provide indications that programmers are using the Web to find components [30, 13]. This paper contributes the first strong empirical evidence of how programmers use Web resources in practice.

This paper presents the results of two studies that empirically investigate how users leverage online resources while programming. The first study was conducted in the lab; we asked 20 programmers to rapidly prototype a Web application. The second study presents a quantitative analysis of a month-long sample of Web query data; we estimate that 25,000 unique programmers produced the 101,289 queries in the sample. We employed this mixed-methods approach to gather data that is both contextually rich and representative of authentic behavior [10, 5]. The results of these studies demonstrate that online resources provide value for both the learning and executing aspects of programming, and that users approach these goals differently.

We begin this paper by describing its relationship to prior work. Next, we cover the method and results of the two studies. We subsequently discuss the insights arising from these studies and their implications for tool design. We conclude by suggesting opportunities for future research.

RELATED WORK
Ko, Myers, and Aung performed an empirical study of learning barriers in end-user programming by observing 40 indi-

viduals new to programming as they learned to use Visual Basic .NET over the course of a semester [17]. They arrive at six classes of barriers—design, selection, coordination, use, understanding, and information—and suggest ways that tools could lower these barriers. This work is largely complimentary to ours—while they provide insight into the problems that programmers face, there is little discussion of how programmers currently go about overcoming these barriers. We feel that Ko *et al.*'s findings could combine with ours to help guide tool development for opportunistic programming.

The software engineering community has a long history of looking at code cloning *within* software projects, both via automated techniques [3, 9] and via an ethnographic account of this practice amongst professional software developers [16]. In general, this body of work is largely concerned with reducing code copying to reduce maintenance costs in large software systems [15], a concern that is far less relevant to opportunistic programming. Nonetheless, many of Kim *et al.*'s insights—most notably that it would be valuable for tools to explicitly record and visualize dependencies created when copying and pasting code—could prove valuable when designing tools for opportunistic programming.

There has been recent interest in building improved Web search for programmers, offering an indication that the Web is an important resource used in this practice [30, 27, 13, 2]. To inform the design of one of these tools, Stylos and Myers offer a description of how programmers may learn APIs, based on observations of what they describe as three "small programming projects" [30]. Specifically, they suggest that programmers begin with initial design ideas, gain a high-level understanding of potential APIs to use, and then move on to concretize their design by finding and integrating examples, which may cause them to return to earlier steps. The authors state that programmers use the Web in several of these steps: gaining high-level understanding of APIs, finding methods and examples, and integrating these examples—and further state that the Web is used in very different ways for each step. This paper's results largely support their intuitions, but in addition, we found that programmers leverage the Web in *all* of these steps, using tutorials and examples to inspire and guide design.

As part of designing a Web search tool for programmers, Hoffmann *et al.* manually classified 339 Web search sessions about Java programming into 11 search goals (*e.g.* beginner tutorials, APIs, and language syntax) [13]. We extend these findings to provide a clearer picture of *how* programmers go about performing these searches, and how they leverage foraged Web content.

Several other systems use completely automated 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 [27]. Somewhat similarly, 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, which is useful for navigating large, complex APIs [23]. These techniques could be leveraged to create semi-automated tools that keep the user in the loop to support opportunistic programming.

Chatroom 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 the user first opens 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 chatroom 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.

STUDY 1: OPPORTUNISTIC PROGRAMMING IN THE LAB

To understand how programmers leverage online resources, especially for rapid prototyping, we conducted an exploratory laboratory study.

Method

We recruited 20 students (5 Ph.D., 4 Masters, 11 undergraduate; 3 female, 17 male) from our university who were proficient programmers. Each session lasted 2.5 hours; we spent the first 15 minutes presenting the task to the participant. We asked participants to prototype an AJAX-style Web chatroom application using HTML, PHP, and JavaScript. Participants were given a list of five features that the chatroom should have (listed in Figure 1). The first four features are fairly typical; the fifth, retaining a limited chat history, is more unique. This feature was introduced so that participants would need to do some amount of programming even if they implemented other features by downloading and installing an existing open-source chatroom application (3 participants did this). To encourage participants to work opportunistically, they were instructed to think of this as a hobby project, not as a school or work assignment where they would be evaluated on programming style.

We provided the participants with a working execution environment within Windows XP (Apache, MySQL, and a PHP interpreter) with a "Hello World" PHP application already running. 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.) Participants were notified ahead of time that they could bring any printed resources they typically used while programming, and were told at the beginning of the study that they were allowed to use *any* resources they wanted while prototyping, including any code on the Internet and any code they had written in the past that they could access.

Participants reported an average of 8.3 years of programming experience; all except three had at least 4 years experience. However, participants had little *professional* experience, with only one participant having spent more than 1 year as a professional software developer.

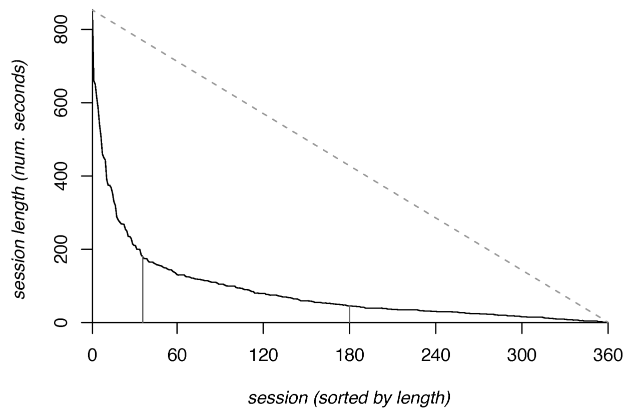


Figure 2. All 360 Web use sessions amongst the 20 participants in our lab study, sorted and plotted by decreasing length (in seconds). The left vertical bar represents the cutoff separating the 10% longest sessions, and the right bar the cutoff for 50% of sessions. Dotted line represents a hypothetical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

When recruiting, we specified that participants should have basic knowledge of PHP, JavaScript, and the AJAX paradigm. However, almost all participants rated themselves as novices in at least one of the technologies involved. Participants were compensated with their choice of class research credit (where applicable) or a $99 Amazon.com gift certificate. Three researchers observed each participant. 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-style reflection at relevant points in the programming process (in particular, whenever participants used the Web as a resource). All researchers took notes during these sessions. These notes were compared after each session and at the end of the study to arrive at our qualitative conclusions. Audio and video screen capture was recorded for all participants; we subsequently hand-coded these videos to obtain data on the amount of time participants used the Web.

Results

All participants made extensive use of the Web while programming: on average, each participant used the Web 18 times ($max = 40$, $min = 7$, $\sigma = 9.1$), for a total of 25.5 minutes, or 19% of the 135 minutes spent programming ($max = 68.8$ minutes, 51%; $min = 7.2$ minutes, 5.3%; $\sigma = 15.1$ minutes).

Figure 3 presents an overview of participants' Web usage patterns throughout the study. This graph shows that participants used the Web a great deal, and used it throughout their programming process. Figure 2 shows that the distribution of lengths of Web use sessions is highly non-uniform, with the shortest half of Web use sessions (those less than 47 seconds) comprising only 14% of the total time spent on the Web, and the longest 10% comprising 41% of the total time. This indicates that individuals are leveraging the Web to help solve several different kinds of problems. The majority of these problems can be solved very quickly, but some take a very long time to solve. Indeed, we found that understanding a programmer's *intention* was paramount when analyzing how they used the Web.

Three intentions behind Web use

Why do programmers go to the Web? It was immediately apparent that every time a participant accessed the Web, he had a clear goal. However, participants' behavior varied dramatically between sessions. As we analyzed our results, we found that understanding a programmer's *intention* was key to explaining his behavior. Based on our observations, we propose a taxonomy of Web use intention, presented in Table 1. This taxonomy breaks Web use into three categories: ***learning*** new concepts, ***clarifying*** existing knowledge, and ***reminding*** about specific implementation details.

In the remainder of this section, we examine each of these intentions in turn. For each, we state typical behaviors observed, present anecdotes from the study that support these claims, and offer theoretical explanations for these actions.

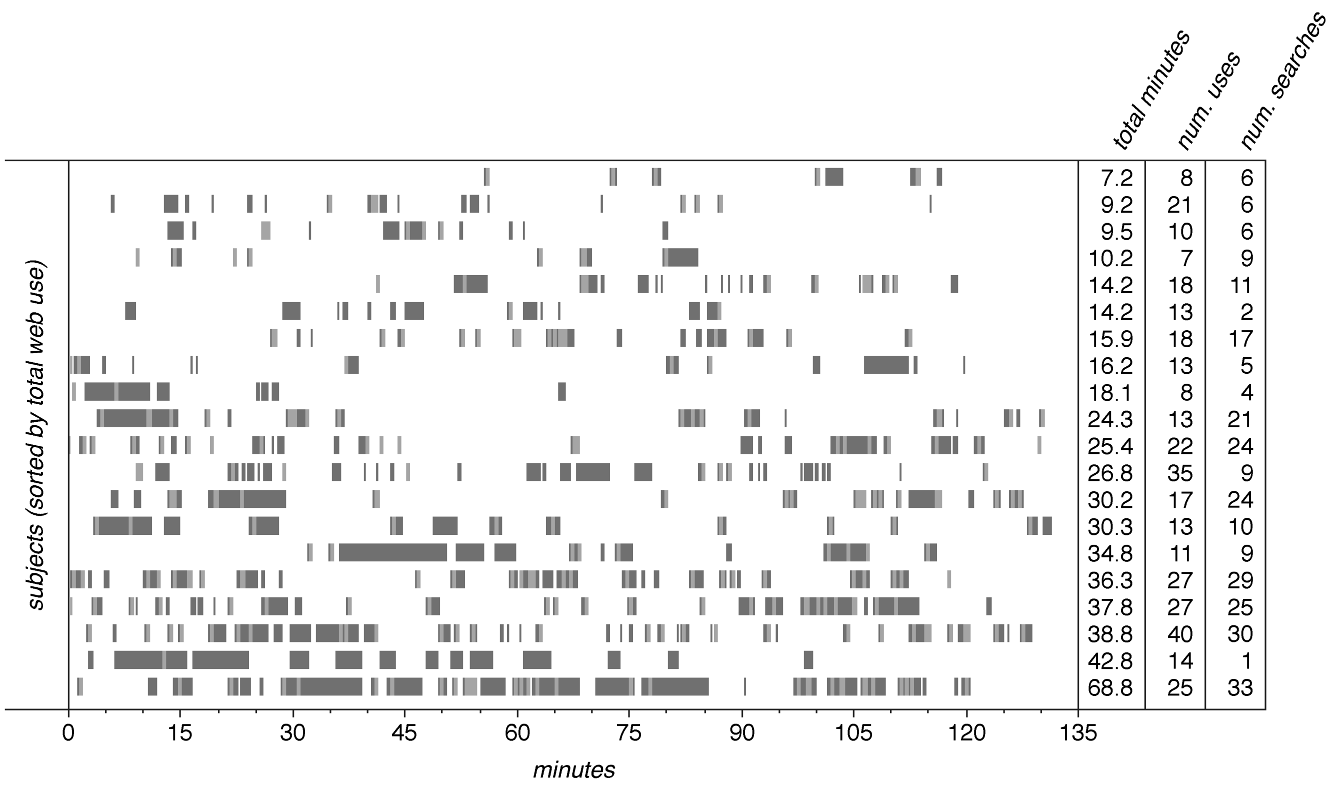
Scaffolds for learning-by-doing

Opportunistic programming often involves learning unfamiliar technologies or paradigms [4]; in our study, participants routinely leveraged the Web when doing so. Web sessions typically started with searches used to locate tutorial Web sites. These tutorials served as scaffolds for the programmer to learn by doing.

Searching for tutorials: Participants' queries usually contained an English description of a problem they were facing, often augmented with several keywords indicating the technology they hoped to use to solve the problem (*e.g. php* or *javascript*). For example, study participants unfamiliar with the AJAX paradigm would often perform a query like *update web page without reloading php*. Query refinements were common in 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.

Selecting a tutorial: Participants typically clicked several query result links, opening each in new Web browser tabs, before evaluating the quality of any of them. After several pages were opened, participants would judge the quality of each by rapidly skimming the page. In particular, several participants reported using superficial 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 asked about what types of Web pages she found to be trustworthy, subject 3 stated that "I don't want [the Web page] to say 'free scripts!', '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 Foraging Theory [26]—surface level Web page features are used as "information scent" when evaluating multiple options.

Using the tutorial: Once programmers find a tutorial that they believe will be useful, they often immediately begin experimenting with code it contains (even before reading the prose). We believe this is because tutorials typically contain



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 Web usage. Web use sessions are shown in blue, and instances of Web search are shown in orange.

a great deal of prose and programmers have trouble deciding what is most worth reading. Said Subject 10: "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 read all these descriptions and text."

Additionally participants would often begin adapting this code before completely understanding how it worked. Subject 15 described this, saying "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 then copied four lines into his project, immediately removed two of the four, changed variable names and values, and tested 1.5 minutes later by reloading his chatroom application Web page. This learning-by-doing approach had one of two outcomes: it either resulted in deeper understanding, mitigating the need to read the tutorial's prose, or it isolated challenging areas of the code, guiding a more focused reading of the tutorial's prose.

Previous research has examined learning-by-doing through the lens of ACT-R. Cox and Young developed two ACT-R models to simulate a human learning the interface for a central heating unit [7]. 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.

Anecdotally, we observed that approximately 1/3 of the code in participants' projects was physically copied and pasted from the Web. This trend of *programming by example modification* has been observed elsewhere. In one study, Yeh and colleagues analyzed code written by students learning to use a Java library that was bundled with example code. Among 17 students' projects, they located 159 occurrences of modified examples [32].

Clarification of existing knowledge

During our study, participants often faced problems where they knew how to implement something at a high level (*i.e.* in pseudo-code), but did not have enough knowledge to implement it in the current programming languages. They did not know, for instance, the names of relevant library functions or the appropriate syntax to use—some piece of *clarifying* information is needed to enable them to use their existing knowledge. The example given in the introduction is representative of how programmers use the Web for clarification: the participant completely understood how HTML forms worked, but could not write one from scratch.

Uses of the Web for clarifying differ from uses for learning in two important ways. First, users are searching for an implementation, not a guide that tells them *how* to implement something. They know exactly how code works when they find it, and have no trouble adapting it to their needs. Sec-

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	English words related to high-level task	Mix of English 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	None or one-liners
Immediately test copied code?	Yes	Not usually, often trust snippets	Yes

Table 1. Summary of characteristics of the three intentions we identified for Web usage during opportunistic programming.

ond, these uses occur much more rapidly. While learning to program using the AJAX paradigm can take tens of minutes, clarifying one's understanding of HTML form syntax takes less than 60 seconds.

Searching the Web to clarify: Clarification uses are typically driven by Web search because programmers are often unsure of the exact name of what they are searching for. We observed that Web search works well for this task because "synonyms" of the correct programming terms often appear in online forums and tutorials. Subject 18, for example, used a third-party JavaScript library named PROTOTYPE, which 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 through which PROTOTYPE allowed this to happen. He searched Google for *prototype request*. When the experimenters 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."

When compared with queries issued for learning uses, clarification queries contain more programming-language-specific terms. Often, however, these terms are not from the correct programming language! We found that programmers often make *language analogies*, saying things like "Perl has this functionality, so PHP must as well". For instance, we saw several subjects search 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 pointing them to the correct function for setting periodic timers: *setInterval*.

Testing copied code (or not): When code is copied from the Web during clarification uses, it is 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). They would then work on other functionality before testing, so 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: The Web is also often used for clarification *during* debugging. Often, when a programmer encountered a cryptic error message, he would immediately search for that exact error on the Web. For example, Subject 11 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 number which contained functions previously copied from the Web. The code did not help him understand the meaning of the error, so he entered the full text of the error into Google. 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 subject returned to his code and successfully fixed the bug.

Reminders about forgotten details

Even familiar functionality is not always known completely. Participants often did not remember low-level syntactic details, like whether a method name contained underscores or in written in camelCase, or the correct order of clauses in a complicated SQL statement. For example, Subject 10 was in the middle of writing one such SQL statement. Immediately after typing *ORDER BY respTime*, he switched to Google and searched for *mysql order by*. He said that he "want[ed] to see the syntax of the 'order by' [clause]." 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 programmers use the Web for *reminding* about details, they know *exactly* what information they are looking for, and often know *exactly* what page they intend to find it on (*e.g.* official API documentation).

Searching (or not) for reminders: With learning and clarification uses, subjects almost always begin by performing a Web search—this is not always the case with reminder uses. Participants often kept select Web sites (such as official language documentation) open in browser tabs to use for reminders when necessary, This is why many of the brief Web use sessions in Figure 3 do not begin with a Web search.

Web search, however, is occasionally instrumental in making a *reminder* use efficient. For example, a programmer may forget a word in a function name. A Web search will quickly confirm the exact name of the function simply by browsing the snippets in the results page. This is why many brief Web sessions in Figure 3 *only* contain a Web search.

The Web as an external memory aid: Several participants reported using the Web as an alternative to memorizing routinely-used snippets of code. One subject 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 [14], allowing them to devote more mental energy to the higher-level tasks of programming.

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 and result click logs from approximately 25,000 programmers making 101,289 queries about the Adobe Flex Web application development framework during the month of July 2008. These queries came from the *Community Search* portal on Adobe's Developer Network Web site, which indexes official Adobe documentation and 3rd-party articles, blogs, forums, and other Web pages that have been vetted by technical writers at Adobe [1].

Method

We began our log analysis by generating a set of hypotheses about programmer search behavior based on our observations from our lab study. (These hypotheses are stated in the Results section below.) Evaluating several of these hypotheses required data about search intention, which was clearly not present in the raw logs. To address this, we used an approach common in query log analysis (*e.g.* [18]): We began by hand-coding 300 search sessions according to user intention. When evaluating a hypothesis, we then used the hand-coded data to understand what sessions relevant to the hypothesis looked like structurally (*e.g.*, lexical structure of queries, session length, and types of Web sites visited) and leveraged this information to examine the entire log corpus.

In the remainder of this section, we detail the machinery used to analyze the logs.

Pre-processing

We pre-processed the logs to group queries into search sessions. The raw logs contain only *query* and *result click* events. All events consist of a timestamp and an anonymized version of the user's IP address. Each query event is succeeded by zero or more result click events which contain the URLs that users clicked on after performing the query.

We combined these events into *sessions* by grouping all events by IP address, sorting by time, and then partitioning the events for each IP address. A session is defined as the longest sequence of consecutive query and result click events such that no adjacent events had a time gap greater than six minutes (similar to the technique used in previous query log analyses, *e.g.* [29]). Pre-processing resulted in 69,955 total search sessions. When hand-coding a subset of the data (described below), we verified that this method of computing sessions worked well: only 7% of sessions appeared to have multiple unrelated queries. This also confirmed that log pollution from HTTP proxies was not a problem. (Users accessing the Web through an HTTP proxy all appear to have the same IP address, which creates problems if two users using the same proxy perform queries at the same time.)

Hand-coding sessions

After pre-processing, one researcher hand-coded 300 sessions using the intention taxonomy developed in our lab study. Initially, we had hoped to classify the intention of each session into one of three categories: *learning*, *clarification*, or *reminder* (identical to those summarized in Table 1). However, due to the lack of contextual information, it was very difficult to differentiate between *clarification* and *reminder* sessions.

We took two additional steps to improve coding accuracy: First, we randomly selected active users of the search portal (users with between 10 and 40 sessions during the month), and coded all sessions for each of the selected users. By inspecting all of a user's sessions, we were able to estimate his overall proficiency with the Flex framework, thus giving us a clearer picture of the intention behind his individual sessions. Second, we chose to place each session into one of four categories: *learning*, *reminder/clarification*, *unsure* (researcher could not accurately determine intention), and *multiple unrelated queries* (when it was clear that two distinct sessions about different topics were incorrectly grouped because their events were less than six minutes apart).

We were able to confidently determine intention for 84% of sessions; among these, we found 22% were *learning* sessions and 78% were *reminder/clarification* sessions. We were unsure of intention for 9% of the sessions, and 7% appeared to have multiple unrelated queries.

Analysis

To evaluate our hypotheses, we found it useful to compute several properties about the search sessions. These properties are listed below; the Appendix gives a description of how we computed each property.

1. *Types of queries performed* — did search terms contain only code (terms specific to the Flex framework, such as class and function names), only words, or a mix of both?

2. *Ways that queries were refined* — between consecutive queries, were search terms generalized, specialized, otherwise reformulated, or changed completely?

Type of first query	Session type		All
	learning	reminder/clarification	hand-coded
code-only	0.21	**0.56**	0.48
words+code	**0.29**	0.10	0.14
words-only	**0.50***	0.34	0.38
Total	1.00	1.00	1.00

Table 2. For hand-coded sessions of each type, proportion of first queries (252 total) of each type (significant majorities across each row in bold, * entry means only significant at $p < 0.05$).

Result click Web page type	Session type		All
	learning	reminder/clarification	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
other	0.18	0.13	0.15
Total	1.00	1.00	1.00

Table 3. For queries in hand-coded sessions of each type, proportion of result clicks (401 total) to Web sites of each type (significant majorities across each row in bold).

3. *Types of Web pages visited* — we sorted all 19,155 result click URLs by number of visits and classified each into one of these categories (starting with most frequently-visited sites) until we accounted for 80% of all visits:
   - **Adobe APIs**: official Adobe API documentation
   - **Adobe tutorials**: tutorials on Adobe Web site, many containing example code snippets
   - **tutorials/articles**: reputable, high-traffic tutorials and how-to articles that are not on Adobe's Web site
   - **forums**: forums, mailing lists, and bulletin boards

   (A final category, **other**, contains the 8246 least frequently-visited pages, collectively accounting for 20% of visits.)

Results

We begin this section by listing our hypotheses and detailing our findings. Based on these results, we found it interesting to take a broader look at how programmers make and refine Web search queries, and what types of Web sites they choose to visit; we report on this at the end of this section.

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 we present as "significant" are statistically significant at $p < 0.001$.

H1: *Learning* sessions have more query refinements and more total result clicks than other types of sessions.

Hand-coded *learning* sessions have significantly more refinements ($\mu = 0.64$) than *reminder/clarification* sessions ($\mu = 0.30$). Similarly, *learning* sessions have significantly more result clicks ($\mu = 1.52$) than *reminder/clarification* sessions ($\mu = 1.0$). These findings correlate well with two behaviors we observed during the lab study: When participants went to the Web to learn, they first clicked on some results to determine how to refine their query. Then once their query was sufficiently refined, they clicked on several pages and selected the best candidate from that refined set.

H2: *Learning* sessions often begin with queries containing only words (*i.e.* not code).

Hand-coded *learning* sessions begin with significantly more words-only and words+code queries than code-only queries. See Table 2 for specific values.

H3: Refinements of *learning* queries often occur without prior result clicks (suggesting that people learn from snippets on the search results page) and transition from words-only to queries with a mix of words and code.

We found no statistical support in the hand-coded sessions for this hypothesis. However, out of *all* sessions, those with words-only first queries (a characteristic of *learning* queries) were most likely to have no result clicks in first query (39% of sessions versus 30% for words+code, 26% for code-only).

H4: Programmers are more likely to visit official Adobe API documentation in *reminder* and *clarification* sessions than in *learning* sessions.

Programmers are significantly more likely to visit official API documentation during *reminder/clarification* sessions than during *learning sessions* (Table 3).

More interestingly, among *reminder/clarification* session, 42% of the result clicks are to official Adobe tutorials. There are two possible explanations: First, the search results page may not contain enough information for programmers to determine that a page is a tutorial, and thus they might mistakenly visit a tutorial page. Second, programmers may realize that the page is a tutorial, but assume that it contains exactly the code they are looking for. From the query logs alone, it is impossible to determine which is the case. We did not observe either behavior during the lab study, but perhaps it was because the official PHP documentation (used extensively in the lab study) contains lots of example code, whereas Adobe's API documentation contains far less, thus forcing programmers to turn to tutorials to find common snippets of code.

H5: *Reminder* sessions typically start with code-only queries and are rarely refined. These sessions also contain the fewest number of result clicks.

We found it difficult to test this hypothesis given our dataset, as we could not easily differentiate between *reminder* and *clarification* sessions. However, among *all* sessions, those beginning with code-only queries are refined significantly fewer times ($\mu = 0.34$) than those starting with words+code ($\mu = 0.60$) and words-only ($\mu = 0.51$); in other words, when programmers perform code-only queries, they know *exactly* what they are looking for, and typically find it on the first search.

Interestingly, when manually looking through sessions, we observed several occurrences of programmers refining what appeared to be *reminder* queries. Typically, these refinements added contextual details, such as the language they were programming in or the version of the framework they were using. Better tool support could have mitigated the need for these refinements by automatically augmenting queries with contextual information from the development environment.

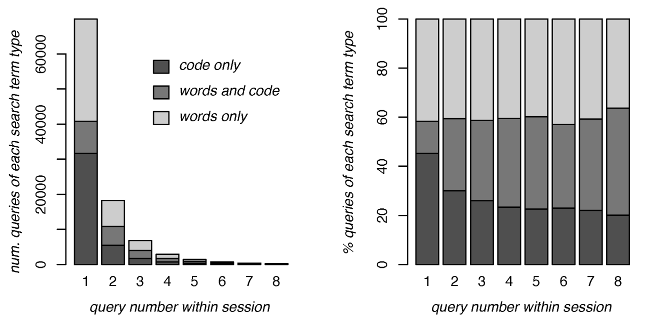


Figure 4. How query types change as queries are refined. Each bar sums all ith queries over all sessions containing an ith query.

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

Table 4. For each refinement type, proportion of refinements of that type (31,334 total) where programmers clicked on on any links *prior* to the refinement.

Programmers rarely refine queries, but are good at it

While evaluating the above hypotheses, we became interested in how programmers refined their queries, so we looked at both how *query types* changed as queries were refined, as well as the *ways in which queries were refined*.

Figure 4 gives an overview of how query type changes as queries are refined. The graph on the left is a histogram showing query types for the ith query in a session. That is, the ith bar shows the total number of each type of query when summing over all ith queries in all sessions that have an ith query (*e.g.*, the first bar is the highest since all 69,955 sessions have a 1st query). The graph on the right presents the same data, but with each bar's height normalized to 100, so that it is easier to see the change in proportions as query refinements occur.

There are two main conclusions we can draw from these graphs. First, as one might expect, the distribution of refinements per session appears to roughly match a power law distribution. Interestingly, sessions have, on average, 1.45 queries per session. This is significantly lower than for the general population of searches — the lowest reported mean we were able to find is 2.02 [29]. This indicates that programmers are relatively good at constructing precise queries.

Second, initial queries are most likely to contain *only* code or *only* words, and the proportion of queries containing only words stays roughly constant as refinements happen. One possible explanation is that programmers search with words only when they don't know how to perform the search with code. This explanation is reinforced by the fact that words-only queries are most likely to stay words-only when refined (65%), as opposed to 59% of words+code refining to words+code, and 48% code-only to code-only.

Table 4 shows whether programmers clicked on results prior to making refinements of each type. Relatively few people click on results before making a *specialize* refinement, likely because they look at the search result snippets, immediately deem them unsatisfactory, and specialize their queries to add more detail. 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.

Result click Web page type	query type code-only	 words+code	 words-only	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
other	0.13	0.22	**0.27**	0.20
Total	1.00	1.00	1.00	1.00

Table 5. For queries of each type, proportion of result clicks (107,343 total) leading programmer to Web pages of each type (significant majorities and near-ties across each row in bold).

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; in other words, when programmers need to refine their queries, they are good at it.

Query type predicts types of pages visited

While evaluating the above hypotheses, it became clear that query type was indicative of intention. As such, we wondered if it would also be indicative of types of pages visited.

Table 5 shows how query type influences the types of pages programmers visit. This data provides some quantitative support for the intuition that query type is somewhat indicative of query intent. Namely, code-only searches, which one would expect to be largely *reminder* queries, are most likely to bring programmers to official Adobe API pages (38% vs. 23% overall) and least likely to bring programmers to all other types of pages. In contrast, word-only queries, which one would expect to be largely *learning* queries, are most likely to bring programmers to official Adobe tutorials (39% vs. 34% overall).

DISCUSSION

In this section, we first list the five key insights that come out of our studies, and suggest how each of these insights could be used to create more effective tool support for opportunistic programming.

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. They often use a learn-by-doing approach when leveraging tutorials, copying code and experimenting before reading the tutorial's prose. A tighter coupling of tutorials and code authoring tools may better facilitate this code experimentation process—one system that offers insight into what this might look like is d.mix [12]. In this system, a Web site's interface elements can be "sampled" to yield the API calls necessary to create them. 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 the error.

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

Code copied from the Web is often not immediately tested, especially when copying routine functionality (*i.e.*, in *clarification* or *reminder* sessions). As a result, bugs introduced when adapting copied code are often difficult to find. To help prevent the introduction of bugs, tools could assist in the code adaptation process by, for example, highlighting all variable names and literals in the pasted code to make sure that they are changed thoroughly and consistently. To help in locating and fixing bugs that still get through, tools could clearly demarcate regions of code that are copied from the Web and provide links back to the original source of the code for later reference.

Programmers are good at refining their queries, but need to do it rarely. Query refinement is most necessary when 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 many cases, however, query refinements could be avoided by building tools that automatically and in parallel augment programmers' queries with contextual information, such as the programming language and frameworks in use and the types of variables in scope.

CONCLUSIONS AND FUTURE WORK

We have presented empirical data on how programmers leverage and learn from the Web while programming. We believe that Web resources will continue to play an increasingly important role in how programming gets done, and hope to see increased work in the area of tool support. We have suggested several directions for tools research in prior sections; here we focus on several directions for further empirical investigation of opportunistic programming.

First, it would be interesting to better understand how a programmer's own code is reused between projects. In other fieldwork (not discussed in this paper), programmers reported a *desire* to reuse code, but stated that it was difficult to do so because of lack of organization and changes in libraries and execution environments.

Second, we know very little about what motivates individuals to *contribute* information, such as tutorials and code snippets, to the Web. How might we lower the threshold to contribution? Is it possible to "crowdsource" finding and fixing bugs in code found online? 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. Programming is becoming less and less about knowing how to do something and more and more about knowing how to ask the right questions.

ACKNOWLEDGEMENTS

We thank Rob Liebscher and Diana Joseph at Adobe 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 #0745320.

APPENDIX: SEARCH LOG ANALYSIS METHOD DETAILS

Determining Query Type

We first break the query search term into individual tokens (splitting on whitespace). Then we run each token through a series of classifiers to determine if it is *code* (*i.e.*, Flex-specific keywords and class/function names). The first classifier checks if the token is a (case-insensitive) match for any classes in the Flex framework. The second classifier checks if the token contains camelCase (capital letter in the middle of the word), which is valuable because all member functions and variables in the Flex framework use camelCase. Third, we check if the token contains a dot, colon, or ends with an open and closed parenthesis, all indicative of code. If none of these classifiers match, we determine that the token is a (non-code) *word*.

Determining Query Refinement Type

We classified refinements into five types, roughly following the taxonomy of Lau and Horvitz [18]: *generalize* refinements indicate that the new search modified the original search by being its substring, by deleted words, or by splitting a single word into multiple words; *specialize* refinements indicate that the new search was a superstring of the original, added words, or combined several words together into one; *reformulate* refinements indicate that the new search contains some words in common with the original but is neither a generalize nor a specialize refinement;

new queries indicate that a completely lexically new query is formed with no words in common to the original; *spelling* refinements indicate that spelling errors are corrected, as indicated by Levenshtein distance between adjacent search terms being less than 3.

Determining Web Page Type

We build regular expressions that match URLs of particular types (*e.g.*, official API docs, official tutorials, *etc.*). A few Web sites, such as the official Adobe Flex documentation, contain the majority of all visits (and can be described using just a few regular expressions), so we sorted all 19,155 result click URLs by number of visits and started classifying most frequently-visited URLs first. With only 38 regular expressions, we were able to classify highest-traffic 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. S. Clarke. What is an End-User Software Engineer? In *End-User Software Engineering Dagstuhl Seminar*, Dagstuhl, Germany, 2007.
7. 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.
8. A. Cypher. *Watch What I Do: Programming by Demonstration*. The MIT Press, 1993.
9. 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.
10. 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.
11. B. Hartmann, S. Doorley, and S. R. Klemmer. Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing*, September 2008.
12. 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.
13. 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.
14. 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.
15. A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, October 1999.
16. 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.
17. 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.
18. 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.
19. H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 1st edition, February 2001.
20. H. Lieberman, F. Paternò, and V. Wulf. *End User Development*. Springer, October 2006.
21. 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.
22. 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.
23. 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.
24. R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, 2002.
25. B. A. Myers. Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In *Proceeding of CHI 1986: ACM Conference on Human Factors in Computing Systems*, pages 59–66, Boston, Massachusetts, 1986.
26. P. L. T. Pirolli. *Information Foraging Theory*. Oxford University Press, Oxford, England, 2007.
27. N. Sahavechaphan and K. Claypool. XSnippet: Mining for Sample Code. *ACM SIGPLAN Notices*, 41(10):413–430, 2006.
28. C. Scaffidi, M. Shaw, and B. A. Myers. Estimating the Numbers of End Users and End User Programmers. pages 207–214, Dallas, Texas, 2005.
29. 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.
30. 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.
31. 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.
32. 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.