Learning on the Job: Characterizing the Programming Knowledge and Learning Strategies of Web Designers

Brian Dorn and Mark Guzdial
School of Interactive Computing
Georgia Institute of Technology
Atlanta, GA 30332-0760
{dorn, guzdial}@cc.gatech.edu

ABSTRACT
This paper reports on a study of professional web designers and developers. We provide a detailed characterization of their knowledge of fundamental programming concepts elicited through card sorting. Additionally, we present qualitative findings regarding their motivation to learn new concepts and the learning strategies they employ. We find a high level of recognition of basic concepts, but we identify a number of concepts that they do not fully understand, consider difficult to learn, and use infrequently. We also note that their learning process is motivated by work projects and often follows a pattern of trial and error. We conclude with implications for end-user programming researchers.

Author Keywords
Web development, informal learning

ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces—*training, help, and documentation*; K.3.2 Computers and Education: Computer and Information Science Education—*literacy, computer science education*

General Terms
Human Factors

INTRODUCTION
Computing for everyone, or universal access to and understanding of computational processes, has become a popular mission. In the computing education community, calls for contextualized learning and Wing's vision of *computational thinking* [35] have served as touchstones in efforts to broaden participation and rethink curricula at the elementary, secondary, and post-secondary levels. In HCI, end-user programming researchers are furthering our understanding of informal types of software development and building tools which lower the entry barriers to computation.

Whether designing new programming languages, tools, or educational interventions, a thorough understanding of the target users or learners is required. We have built considerable evidence about what novice programmers do and do not understand (see e.g., [6, 9, 17, 29, 32]), but these studies consider students in formal learning environments. End-user programmers often learn about scripting and programming without the aid of a classroom, and we know little about how they grasp conceptual computing knowledge. The increasing prevalence of software created by end users [30] motivates a need for the examination of their understandings. In so doing we will be able to more appropriately design tools and resources that scaffold their development processes.

The study presented in this paper begins to address this gap in the literature by providing a detailed characterization of what aspects of programming fundamentals one group of non-traditional software developers understand and how they go about learning. Our study is contextualized within the domain of professional web design and development, whose members make up a large and diverse group of end-user programmers. They regularly engage in programming activities, making use of textual markup and scripting languages like HTML, CSS, JavaScript, and PHP. Further, in studying practicing web developers we can explore notions of programming among a group of people who program in their careers but may lack a traditional educational background in computing. In this paper we investigate the following research questions:

1. What programming concepts do web developers recognize, and to what degree do they understand each?
2. How do web developers think about and associate foundational programming concepts with one another?
3. How do web developers go about learning new things as they go about their work?

The remainder of this paper begins with an overview of related work, and we then detail our study design. Results are divided into two sections. The first focuses on how participants categorized various computing concepts in a card sorting task. This is followed by a discussion of themes about learning derived from interview data. We end the paper with a discussion of the implications of our findings.

RELATED WORK
Recently the subject of end-user programming has garnered significant attention in the research community; see Lieberman et al. [20] for a thorough overview of this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CHI 2010, April 10 – 15, 2010, Atlanta, Georgia, USA
Copyright 2010 ACM 978-1-60558-929-9/10/04...$10.00.

Generally speaking, research in this area recognizes that programming and software development are difficult tasks for end users and therefore seeks to simplify the task of programming in some way. Often this results in the development of new tools or systems that target some facet of the programming task. Recent efforts to support end-user programmers include programming by example [16, 19], natural language programming [21, 23], and scaffolding users in testing and debugging activities [1, 34].

The work of Ko et al. [14] closely aligns with our research by framing the challenges of end-user programming as learning difficulties encountered while novices acquire expertise. In a study of non-programmers enrolled in a university course for GUI application development with Visual Basic.NET, they identified six types of learning barriers: design, selection, coordination, use, understanding, and information. These barriers were derived from breakdowns in learners' understandings that they could not resolve without assistance.

Ko et al.'s findings share many parallels with research on novice programmers. Computer science educators have long recognized that novices struggle to design algorithms [7], to select and assemble language components [32], to manage syntactic complexity [9], and to grasp the behavior of programs at runtime [6]. However, the approaches that educators use to support novices may not be appropriate for end users since their knowledge is likely highly situated in their context [10, 24]. For example, we know that mathematics as learned and enacted in the real world differs significantly from what is taught in classroom environments [15, 26].

Our work adopts the perspective that web developers and other end-user programmers are often informal learners of computing who develop their understandings in a piecemeal fashion. We recognize that knowledge developed in this way may differ from that of classroom learning. In this paper we explore conceptual understandings among a group of professional web designers and describe their strategies for learning new information.

An early study by McKeithen et al. used free-recall tasks with programming reserved words to explore how novices and experts taken from traditional educational settings relate concepts to one another [22]. They found that beginners often related concepts by natural language associations while experts used programming semantics to relate concepts, with intermediate learners using a mixture of the two. Our work shares similar research questions and a concept-based approach, but we employ card sorting to explore conceptual relationships held by participants who may or may not have had exposure to formal programming education.

In an online survey Rosson et al. explored learning behaviors of over 300 web developers [25]. They noted that nearly all respondents reported that at least some of their programming skills were self taught, and many indicated they relied on online resources. Participants were also more likely to refer to FAQs, books, code from similar websites, and colleagues than other resources like classes or tech support. We further characterize this class of users and provide additional evidence for their utilization of both online and offline resources.

Lastly, Brandt et al. studied how programmers use the Web to solve problems that arise while developing code [3]. By studying information foraging behaviors, they found that programmers relied on Web resources for just-in-time learning of new concepts, to clarify existing knowledge, and to remind themselves of minor details which were forgotten. Our work is complementary to these findings in that it provides additional information about how web developers seek information online and assess its relevance to the task at hand.

STUDY PROTOCOL

Our study was conducted face to face and consisted of three separate parts. First, participants completed a survey that gathered basic demographic information and details about their professional background. Next, participants engaged in a card sorting activity about various introductory computer science concepts. Finally, we ended the session with a semi-structured interview. The sorting task and interview are discussed in more detail in the following subsections.

Card Sorting Task

Card sorting is a general purpose elicitation technique that can be applied in a wide range of settings [27]. At its most basic, it involves participants grouping items from a set of stimuli (e.g., pictures, words) into categories based on similarity along some dimension. Sorting tasks may be either *closed*, where participants are provided with the sort criteria and fixed categories in which to place the cards, or *open*, with participants developing their own criteria and categories. Through categorizing the physical cards in multiple ways, participants provide indications of their own mental representation of the concepts [8].

Card sorts are often employed in HCI as a usability tool for gaining an understanding about how users might naturally group certain aspects of a designed artifact (e.g., placement of content on web sites [13]). However, categorization tasks also allow us to investigate a person's existing knowledge about the stimuli. Fincher and Tenenberg argue that card sorting "can be effective in eliciting our individual, and often semi-tacit, understanding about objects in the world and their relationships to one another" [8, p. 90]. Accordingly, computer science education researchers have successfully used card sorting to elicit novice programmers' knowledge of fundamental computing concepts with cards containing terms about programming [17, 29].

Building on Sanders et al.'s work [29], we used card sorting to explore web developers' knowledge of introductory computing concepts. We developed a set of 26 cards containing terms from Sanders et al.'s study as well as terms from a study by Dorn et al. [5] that explored common introductory constructs found in an online repository of scripting code. After merging the two lists, we removed any duplicated terms and eliminated terms that lacked concreteness or relevance to the web programming domain (e.g., depen-

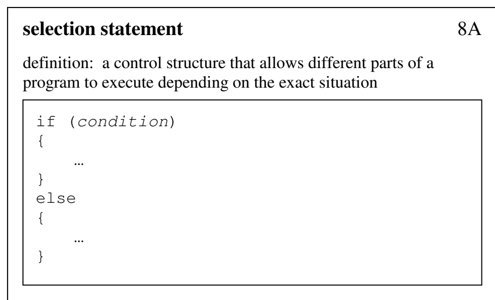

Figure 1. Example Card with Definitions

dency, thread). That is, we ensured the list of terms had clear syntactic representations in JavaScript. Our final cards contained the concepts listed in Table 1, with one term per card.

The task consisted of a repeated single-criterion card sort with both open and closed sorts. The first four sorts were prompted by researchers; these closed sorts explicitly explored participants' recognition and understanding of the 26 terms. The first sort is particularly notable in that it asked participants to separate the cards based on whether they recognized the term or not. Because we sought to explore participants' understanding of the underlying concepts and not simply vocabulary recognition, we had them repeat the sort for any cards originally placed in the "don't recognize" category. In this extra sort, we provided cards that contained the unfamiliar term, its definition,[1] and a JavaScript example of the concept in use (see Figure 1). Any concepts recognized with the aid of this additional information were added to the participant's "recognize" category, and any that remained unknown were eliminated from all subsequent sorts. Following the four closed sorts, participants were invited to openly sort the cards using one criterion at a time in as many ways as they could generate.

Interview

Once participants had exhausted their ideas for additional open sorts, we conducted a semi-structured interview that lasted approximately 30 minutes. The interview elicited information about participants' daily job responsibilities, use of programming or scripting languages, and use of software tools (e.g., Photoshop). We also inquired about typical strategies they employ while developing scripts and resources they rely on to learn new things about programming.

Audio recordings of the interviews were transcribed, and we used a multi-step thematic analysis [4] to analyze the qualitative data. Coding was done in both a top-down and bottom-up fashion. We coded transcripts based on particular questions asked of all participants but also allowed for emergent codes when other themes were mentioned by multiple participants. Additional passes were made through the transcripts to further refine the codes. Lastly, in preparing transcript excerpts for presentation in this paper, we have edited them as necessary for anonymity and brevity.

[1]Definitions were taken from the glossaries of introductory textbooks [12, 18, 36] and adapted where necessary to fit JavaScript.

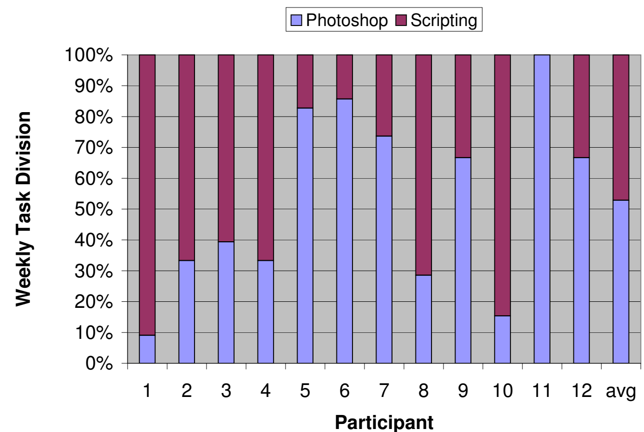

Figure 2. Average Weekly Division of Labor

Recruitment

Participants were recruited from a large metropolitan area via email. Solicitation messages for volunteers were sent primarily to mailing lists for three large Meetup[2] groups of local web designers, graphic designers, and users of Adobe Photoshop. Volunteers were then pre-screened using a short email survey to ensure that they were actively involved in the web design profession and had prior experience with writing scripts or programs in JavaScript. Face to face interviews were scheduled with participants meeting the study criterion, and participants were compensated $15 for their time.

PARTICIPANT DEMOGRAPHICS AND BACKGROUND

In total, we interviewed 12 people—seven men and five women. Ten participants indicated on the survey that they actively work in the web design field, and the remaining two were students currently enrolled in web design degree programs at local institutions. Most participants (58%) selected "Web Developer" as their job title with only two people choosing the title "Programmer."

Our participants were well educated. All but two participants (one of whom was a current student) held a bachelor's degree, and four participants either had earned or were pursuing a master's degree. However only one person held a degree in computer science. About a third of them held undergraduate degrees in areas related to web or graphic design (e.g., visual communications) with the rest holding degrees in the humanities or other fields (e.g., English, psychology, ministry).

We were successful in recruiting broadly from the web design/development community with respect to reaching those with a wide range of professional experience. The number of years of experience with Photoshop ranged from 2 to 13 years, and participants reported between 2 months and 15 years of scripting or programming language experience. On a scale of 1 (novice) to 5 (expert), the average self reported level of expertise in scripting was 3.21 (σ = 1.16). Every participant listed exposure to more than one script-

[2]`http://www.meetup.com/`

ing or programming language with JavaScript, ActionScript, and PHP being those most frequently mentioned. Figure 2 illustrates participants' estimated weekly division of labor between scripting and graphics manipulation in Photoshop. On average, our participants' time was split roughly evenly between these two tasks; however most people tended to concentrate more heavily on one or the other.

When elaborating on the nature of their work, most participants noted being involved in front-end web development or design. This job often requires them to build functional web sites from prototypes that have been mocked up with tools like Photoshop that either they or someone else designed. They make decisions about how to slice up visual components in the layout so that they render properly across different browsers, and they write scripts using languages like JavaScript or PHP to enable the intended interactivity features in the design.

CLOSED SORT RESULTS

As mentioned earlier, we asked participants to first sort the 26 programming concepts into two piles based on whether or not they recognized the term. The results of this initial sort are presented in Table 1 with concepts ordered by their level of recognition. Despite the lack of what might be considered a traditional computing education, a majority of participants recognized nearly all concepts, and ten of the concepts were universally recognized based on the term alone. The least frequently recognized terms were selection statement, nesting control structures, and functional decomposition, with the last being markedly less familiar than all others.

Once provided with cards containing definitions and examples of the concepts, the rate of recognition increased significantly (paired t-test; $t(25) = -3.696$, $p = 0.001$). Over half of the concepts were then familiar to everyone, and the minimum recognition rate increased to 83.3% with this additional information. Many participants commented on the fact that they did use these concepts often, but they had not initially recognized the terms simply because they had never learned the names. For example:

> P6: Mathematical operator, goodness gracious. [laughter] I've just never heard it called that before. Plus, minus, sure.
>
> P9: Um, some of them I picked up by seeing the code. I just didn't know the name of it, like nesting control structures. You know, putting if statements and when statements inside each other is common practice in code, but I just had never given it a name.

Additional Closed Sorts

After participants identified the subset of concepts they recognized, we prompted them with three additional sorts. Participants sorted the concepts based on their own level of understanding of the concept, based on how often they use the concepts in scripts, and based on how difficult they perceive the concepts are to learn. These closed sorts were intended to provide additional information about conceptual understanding beyond concept recognition. Results of these sorts are

Table 1. Percentage of Participants Recognizing Card Concepts

CS Concept	Term Only	With Def'n
assignment	100.0%	
input	100.0%	
object	100.0%	
function	100.0%	
parameters	100.0%	
array	100.0%	
string	100.0%	
output	100.0%	
number	100.0%	
variable	100.0%	
mathematical operator	91.7%	100.0%
definite loop	83.3%	100.0%
importing code	83.3%	100.0%
indefinite loop	83.3%	100.0%
boolean	83.3%	91.7%
constant	83.3%	91.7%
exception handling	83.3%	83.3%
type conversion	75.0%	91.7%
exporting code	75.0%	83.3%
logical operator	75.0%	83.3%
relational operator	66.7%	91.7%
variable scope	66.7%	91.7%
recursion	66.7%	83.3%
selection statement	58.3%	91.7%
nesting control structures	58.3%	83.3%
functional decomposition	8.3%	83.3%

summarized in Tables 2, 3, and 4, respectively. In each table, concepts are sorted by a "rating" value which is computed as a weighted average of the response frequency across the ordered categories. For example, in Table 2 categories are assigned values between one and four (similar to a Likert-type scale) and the rating corresponds to the average value that participants assigned to this concept. Further, each of these tables divide the upper, middle, and bottom thirds of the rating values with a double line.

Based on their sorting results, participants reported considerable understanding of and comfort with these concepts. Table 2 shows the rating value for each concept as well as a breakdown of participants' self-assessed level of understanding using the categories: I have heard the term but am not comfortable using it in my scripts (1); I understand the meaning of the term but have problems using it correctly in my scripts (2); I understand the meaning of the term and am comfortable using it in my scripts (3); and I have a strong understanding of the term and feel I could explain it to someone else (4). With the exception of recursion, every concept had a rating of 3 or higher, meaning that participants understood the term's meaning and were comfortable using it. The top two-thirds of topics rated 3.58 or higher, indicating a high degree of knowledge and an ability to explain the concepts to others. Among the lowest ranked terms were variable scope, type conversion, indefinite looping, exception handling, functional decomposition, and recursion. Interestingly, concepts where individual participants indicated

Table 2. Personal Level of Understanding; Sorted by Decreasing Understanding

CS Concept	Rating (1–4)	1	2	3	4
number	3.92			8.3%	91.7%
boolean	3.91			9.1%	90.9%
variable	3.83			16.7%	83.3%
mathematical operator	3.75		8.3%	8.3%	83.3%
function	3.75		8.3%	8.3%	83.3%
array	3.75		8.3%	8.3%	83.3%
object	3.75			25.0%	75.0%
selection statement	3.73	9.1%			90.9%
nesting control structures	3.70			30.0%	70.0%
string	3.67		8.3%	16.7%	75.0%
parameters	3.67		8.3%	16.7%	75.0%
input	3.67		8.3%	16.7%	75.0%
definite loop	3.67			33.3%	66.7%
relational operator	3.64	9.1%		9.1%	81.8%
constant	3.64		18.2%		81.8%
output	3.58		16.7%	8.3%	75.0%
importing code	3.58		16.7%	8.3%	75.0%
logical operator	3.50	10.0%		20.0%	70.0%
assignment	3.50	8.3%		25.0%	66.7%
exporting code	3.50		10.0%	30.0%	60.0%
variable scope	3.45	9.1%	9.1%	9.1%	72.7%
type conversion	3.45	9.1%		27.3%	63.6%
indefinite loop	3.42		16.7%	25.0%	58.3%
exception handling	3.40	10.0%	10.0%	10.0%	70.0%
functional decomposition	3.40			60.0%	40.0%
recursion	2.90	20.0%	20.0%	10.0%	50.0%

Table 3. Frequency of Use; Sorted by Decreasing Frequency

CS Concept	Rating (1–4)
number	3.92
string	3.92
relational operator	3.91
selection statement	3.91
boolean	3.91
logical operator	3.90
variable	3.83
mathematical operator	3.83
array	3.75
object	3.75
definite loop	3.75
parameters	3.73
nesting control structures	3.70
assignment	3.67
input	3.67
output	3.67
function	3.67
importing code	3.45
functional decomposition	3.40
variable scope	3.36
constant	3.18
exporting code	3.10
exception handling	3.10
type conversion	3.09
indefinite loop	3.08
recursion	2.70

Table 4. Difficulty to Learn; Sorted by Increasing Difficulty

CS Concept	Rating (1–3)
number	1.08
boolean	1.09
relational operator	1.09
variable	1.17
constant	1.18
logical operator	1.20
string	1.25
mathematical operator	1.33
input	1.33
assignment	1.42
parameters	1.50
selection statement	1.55
output	1.58
importing code	1.67
function	1.75
type conversion	1.82
nesting control structures	1.90
array	1.92
exporting code	2.00
object	2.00
definite loop	2.08
indefinite loop	2.08
variable scope	2.09
recursion	2.20
functional decomposition	2.30
exception handling	2.50

trouble were spread throughout the table and were not localized to the bottom third, where one might expect.

We also asked participants to sort the cards into four categories depending on how frequently the concepts were used in code that they wrote. The four categories provided were Never (1), Rarely (2), Occasionally (3), and Frequently (4). Table 3 presents the results of this sort, ordered by decreasing frequency of use. Unlike the sort on understanding, the distribution of responses was fairly uniform across the three tiers—no concept in the top two thirds was placed in the "never" category, and the six lowest ranking terms were all categorized as frequently used by 50% or fewer of the participants. In other words, these results indicate a reasonably strong consensus about how frequently these programming concepts arise in typical web development work. The topics listed in the top third (number to mathematical operator) are frequently used, and those in the bottom third (importing code to recursion) are used sporadically.

The final closed sort requested of participants was to categorize their perception of how difficult the concepts are to learn to use correctly. They were prompted with three categories for this sort: Easy (1), Intermediate (2), and Advanced (3). Responses on this sort are outlined in Table 4 and are sorted by increasing level of difficulty. This sort exhibited the lowest agreement with 73% of the individual concepts being ranked in all three categories (easy, intermediate, and advanced) by different people. Despite this variation, the final ordering of concepts maps relatively closely to what one might find in an introductory textbook table of contents: basic data types and operators; followed by selection statements and functions; followed by looping, recursion, and exceptions.

Comparison of Closed Sorts

Comparing the results from Tables 2–4, we observed that many concepts appeared to be similarly rated in each of the three sorts. Indeed, a Pearson correlation analysis revealed a statistically significant positive correlation between the ratings for level of understanding and frequency of use ($r = 0.808$, $N = 26$, $p < 0.001$). We also noted statistically significant negative correlations between ratings for frequency of use and learning difficulty ($r = -0.641$,

$N = 26$, $p < 0.001$) and between difficulty and understanding ($r = -0.586$, $N = 26$, $p = 0.002$).

Further we compared the terms with respect to their relative grouping in the tiers of the three sorts. This provided an indication for the concepts that were uniformly ranked in terms of the participants' level of understanding, the frequency with which they are used, and the perceived conceptual difficulty. We noted four terms that consistently appeared in the first tier, two in the second, and six in the bottom tier. The concepts that were rated the most highly understood, most frequently used, and easiest to learn were number, boolean, variable, and mathematical operator. Inversely, those which ranked least understood, least used, and most difficult were exporting code, indefinite loop, variable scope, recursion, functional decomposition, and exception handling. The concepts parameters and output were consistently in the middle tier.

OPEN SORTING

Once participants had completed the final closed sort, we provided them with the opportunity to sort the cards into groups using criteria of their own choosing. Through open sorting, we aimed to gather additional insight about web developers' knowledge of these 26 concepts and their associations between concepts. Participants were encouraged to generate as many sorts as they could and researchers recorded the participant's sort criterion, category names, and placement of the cards within the groups. Altogether, our participants generated 28 sorts. With an average of 2.3 sorts per participant, they generated noticeably fewer sorts than introductory computing students or educators engaged in a similar task (4.5 and 5.2, respectively) [29]. However, given that our participants completed a number of closed sorts prior to open sorting, this value may be artificially low. We also noted that our participants used fewer categories per sort on average (2.6) than the students (4.0) or educators (3.7).

To further explore the data gathered from the open sort activity we employed superordinate analysis to classify similar sorts into thematic groups [28]. These groups bring together sorts that relate to a common theme, regardless of differences in the wording that participants used to describe them. The purpose of such an analysis is to determine commonalities in sorts across the participants, indicating the typical ways people think about this particular set of stimuli.

Two independent raters grouped the 28 sorts into mutually exclusive categories based on the similarity of their criterion. To aid in making decisions about whether two sorts were similar, raters had access to the criteria and category names given for a sort by the participant as well as the excerpt of the interview transcript relevant to each open sort. Transcripts enabled raters to make an informed decision about a sort's meaning, particularly in the case where participants had difficulty in succinctly naming their sort criterion but were able to talk generally about what they were trying to accomplish with the sort. Raters achieved 79% agreement on the thematic grouping of the 28 sorts on their first pass. They then collaboratively negotiated the group definitions relevant to the six sorts where there was initial disagreement. In the end, seven thematic groups which each contained more than one sort were derived. These themes were (the number of sorts related to each theme appear in parenthesis):

Conceptual Ordering (4) Sorts which classify concepts by the order in which they should be learned or the order in which concepts build on one another.

Quality Metrics (4) Sorts which separate concepts by various software quality metrics like readability, maintainability, and efficiency.

Terminology (3) Sorts which classify cards by terminology considerations. For example, a sort whose categories are labeled "terms you need to know to communicate with others" and "terms that are academic."

Language Decomposition (3) Sorts which attempt to separate concepts into functional groups based on their semantics (e.g., "related to functions" or "related to numbers").

Expertise of Others (3) Sorts expressing beliefs about the expertise or understanding of others.

Relevance to Scripting (2) Sorts that distinguish concepts based on whether they are generally applicable to the typical code or scripts that web developers write.

Desire to Know More (2) Sorts that prioritize concepts by an interest in learning more about them.

The results of the card sorting task provide a detailed picture about what computing concepts web developers understand and how they relate the concepts to one another, but they provide little information about how professional web developers learn as they go about their work. For this, we must turn to the qualitative data presented in the next section.

LEARNING AND RESOURCES

The primary focus of our semi-structured interview with participants was to elicit their strategies for learning new information. Our analysis of interview transcripts resulted in four themes related to learning: motivation to learn new things, learning processes, resources used for learning, and heuristics for judging information quality. Each of these themes is discussed in the following subsections.

Impetus for Learning

While some participants indicated that they enjoyed learning new languages or details about scripting for curiosity's sake, most expressed that their decision to learn something was a matter of necessity. The computing concepts that they chose to learn needed to contribute in some way to the completion of their current project (in a similar fashion to Blackwell's attention investment model [2]). Incorporation of new web features like login-based access or embedded streaming video (and learning the necessary underlying programming skills) were driven by project needs. Learning of new features was also motivated by a need to remain up-to-date in order to write standards-compliant code. Participants two and five discuss their reasons for learning new things below.

> P2: I don't care where technology is going. It's like, does my check get cashed on Friday? Ok. And if they have a new something that comes out that will impede my check being cashed on Friday, then I will learn it.

> P5: Like when CSS was officially considered a standard, and I went, oh crap, now I have to learn it.

Even among those who discussed learning new languages or language features for fun, they often did so by choosing to use the unfamiliar concepts in an upcoming project. In these cases the participants were willing to tolerate some inefficiency in completing the project because they recognized they were learning something new.

Learning Processes

Participant nine succinctly conveys his learning process, and that expressed by most participants, by stating, "generally, the best way I learn is to just jump in headfirst." Several participants used the phrase "trial-and-error" to characterize their script development. When asked to elaborate they described a process akin to bricolage programming [33], iteratively writing code, examining the results, and seeking out information as necessary. In this way, our participants exemplified the opportunistic approach to programming described by Brandt et al. [3]. One participant explains:

> P1: I start off actually trying to do something that I need to complete as my first step even not knowing anything about it. And I guess the first thing that I'll do is I'll Google the subject and see what I can pull out on the Web. What information I can get out of it. And then I just hit the floor running, or at least I try. And then of course I come to points where I stumble, and I can't go forward cause it's too complex there's just some stuff that I don't know. So at that point I have a couple of choices.

He goes on to describe his decision making process for what to do when web resources aren't enough—whether to consult with a colleague for help or search for a book.

However, while going to the Web to look for an answer was almost universally the first line of defense, it may not always be the most fruitful activity. One participant realized that this strategy was suboptimal while reflecting on the sources of information she used and which were the most useful in answering her questions.

> P2: The Internet, of course you can Google anything, that's my number one place. And it's fairly useful. Wow, that's a good question. The order in which I tap my resources are from least useful to most useful. So my colleagues are my second level, cause you know, different companies you work at have different systems. So something might work good in practice or I might find it on Google, but it just doesn't work well with servers and the software we use. So it would be Internet, colleagues, books as far as the order that I tap my resources. The most useful would be books, colleagues, Internet.

Table 5. Resources for Learning

Online	Offline
• code samples or example demos • walkthroughs and tutorials (e.g., `www.w3schools.com`, `www.smashingmagazine.com`) • language or library references (e.g., `www.ruby-doc.org`) • subscription-based online training sites (e.g., `www.lynda.com`) • forums or user groups • blogs, both as authors and as readers • podcasts	• books • code samples • tutorials or other help files provided with software • manuals • colleagues, friends, or instructors • strangers with similar job descriptions (e.g., other webmasters) • classes

Resources Used

Over the course of the interviews, participants mentioned relying on over a dozen different resources for learning something new. In addition to generic occurrences of "the Internet" or "Google", seven different online resources were discussed by different participants. We also noted seven offline resources. Table 5 summarizes these 14 unique sources.

To provide greater detail about these web developers' use of resources, we included an extra question at the end of the demographic survey. Participants rated how likely they would be to consult various resources when attempting something new on a scale of 1 (very unlikely) to 5 (very likely). Based on Rosson et al.'s [25] prompt, we inquired about interactive wizards, example code, classes/seminars, books, FAQs/tutorials/online documentation, friend/coworker, and technical support. Figure 3 depicts the percentage of participants rating each resource as likely or very likely to consult. Similar to Rosson et al.'s findings, our participants indicated a strong preference for online documentation, books, examples, and personal communication.

Interestingly, participants offered differing opinions with respect to the use of books. Some participants utilized books as a means to build a solid foundational understanding prior to using less formal references found on the Web. Others portrayed buying a book as an indication of a desire for deeper knowledge or a long term commitment to a particular concept. For example, P6 says he purchases a book when, "like I really wanna master it."

Judging Relevance

The final theme related to learning deals with how web developers judge the quality and relevance of content they find online. The breadth of web content can be a double-edged sword; on the one hand, chances are good that an answer to one's question exists online, but on the other hand, locating that information can be time consuming. One participant reflects on her ability to find relevant information:

> P11: I'm starting to learn where those online resources are, but early on here it's been kinda daunting to figure

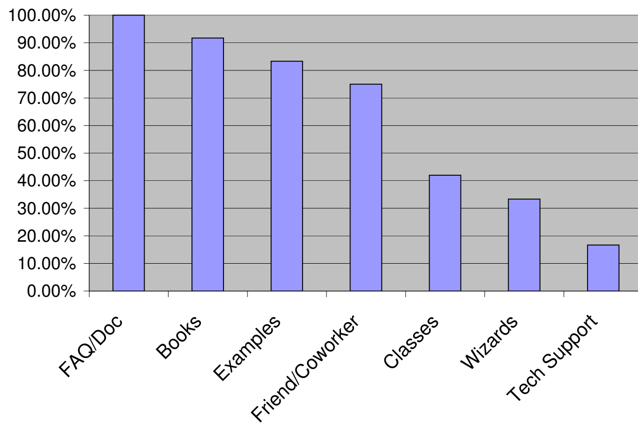

Figure 3. Percent of Participants Rating Resource as Likely or Very Likely to Use

> out. I've done a lot of online that just takes me nowhere. It's, you know, spend an hour just clicking around trying to figure out where to find the answer at.

Though some participants were unable to articulate specific strategies they use to evalute information, relying on their "gut reaction", many outlined informal heuristics that they employ while searching the Web. We noted 10 such rules of thumb in the interview corpus, ranging from some that are rather specific to others which could be highly subjective:

1. Legitimacy of sources, with a preference for content hosted by established or official publishers
2. Author credentials, preferring people recognized within the web development community
3. Google's page rank algorithm as a predictor of utility
4. Conformity of provided code to W3C standards
5. Availability of working code demos
6. Similarity of language features used in code examples to those used in the participant's code
7. Positive and negative comments of others posted in reply to tutorial, blog, or forum entries
8. Opinion of peers about a particular source of information
9. "Digestibility" of the information, with a preference for more easily consumable things
10. Overall aesthetic feel of the hosting web site

If not always sophisticated, these heuristics provide evidence that web developers do develop meta-cognitive strategies for evaluating information. They also have ramifications for how educational content might be delivered to end-user programmers via the Web. In the next section we discuss this and other implications of our findings.

DISCUSSION

The discussion here first considers our results relative to the three research questions posed. We then draw out four implications for the research community. The first two are tied closely to our plans for future work, while the final two have broad applicability to end-user programming research.

What programming concepts do web developers recognize, and to what degree do they understand each? On the whole, the participants recognized nearly all of the concepts with the aid of a definition and example. The terms used in our study were standard terminology from introductory materials, and several participants lacked knowledge of the formal names for these concepts. Thus, our results suggest the importance of multiple indexes in reference materials which target informal learners. We also found that participants expressed remarkably normative judgements about concept difficulty; in many ways the average ratings matched what we might expect from a computer scientist.

How do web developers think about and associate foundational programming concepts with one another? The open sorting data provides some insight into how web developers' associations may differ from other populations. When compared to Sanders et al.'s card sort study with novice computer science students [29], our web developers generated fewer sorts per person with fewer categories per sort. This suggests that introductory CS students may have a more sophisticated understanding of these concepts than our web developers. The common open sort criteria noted in this study also support this interpretation. While not necessarily organized by natural language groupings (as previously used by novices [22]), only one of the seven sort themes we identified involved grouping cards based on programming language semantics. Contrastingly, the most frequently occurring category groupings generated by introductory CS students all appear to make use of programming syntax or semantic concerns [29]. At the very least, these results indicate that practicing web developers think about these concepts in different ways than do traditional students and teachers.

How do web developers go about learning new things as they go about their work? We found that learning in this context is often motivated by project demands, whether that be a need to learn a specific new technique or to update one's skill set to continue to write standards-compliant code. Participants expressed a trial and error approach to programming where writing code is interleaved with information foraging. We found that participants learned from a wide variety of online and offline resources, with a preference for FAQ-style documentation, books, and related code examples. An ongoing challenge suggested by our results lies in helping web developers acquire useful strategies for assessing the relevance and quality of material found online.

Highlight Relevance of Uncommon Concepts

Our data exhibited a strong correlation between frequency of use and concept difficulty. Further we noted that our web developers choose to learn concepts they perceive to be directly related to their tasks. Taken together they seem to suggest that web developers are learning those concepts that are either the easiest or the most useful. While perhaps not surprising, web developers and other end-user programmers may be missing out on more advanced concepts that could be quite useful but are not entirely obvious to them. For example, concepts like indefinite loop, exception handling, and program decomposition were uniformly ranked at the bot-

tom of the sorts, but use of these constructs could easily aid these programmers in developing more robust, reusable software. Additionally, as the Web continues to permeate daily life, web designers may naturally seek to develop content for new platforms (e.g., iPhone applications), and doing so requires considerable knowledge about computer science content well beyond the scope of the concepts considered here. As a research community we need to continue identifying relevant conceptual content for non-traditional programmer populations to inform the design of new resources and tools.

Design Resources to Support Informal Learning

After identifying the uncommon, difficult conceptual information, we should explore innovative ways to reach informally trained programmers with instructional content while respecting their current work practices. Our results further confirm a reliance on resources like tutorials and example code, often found through web searches. These practices closely match the affordances of case-based learning aids which use collections of example projects as a vehicle to provide conceptual instruction [11]. Our future work will explore the use of contextualized case-based resources for supporting informal learning among web designers.

Study Practicing Developers

We must also pay careful attention to the learning and information consumption strategies of practicing developers as we design resources. Often, studies of non-traditional developers or end-user programmers utilize university students as proxies for practitioners for pragmatic reasons (e.g., accounting students in the place of professional accountants, non-majors enrolled in elective programming classes). Though such studies provide valuable insights about those who lack traditional backgrounds in computing, they should also be complemented by research conducted in the field. The heuristics for evaluating information outlined here stem from years of learning embedded in the work environment, and that experience also plays a role in the construction of knowledge. For example, while participants' combined ratings of concepts resembled normative views, these web developers seemed to relate these concepts to one another along dimensions different from computing students.

Use Learning as a Research Lens

Lastly, taking a learner-centered view of the challenges informally trained programmers face, as we have taken in our work, opens new research avenues [31]. It challenges us to consider pedagogical approaches to addressing these challenges, rather than solely technological ones. As with the study of practitioners, using learning as an additional lens for our research permits a more holistic depiction of the rich domain of end-user programming. There are many unique opportunities in this space for collaboration between HCI, computing education, and software engineering researchers.

CONCLUSION

In this paper we have presented the results of a study of 12 web developers. Through our analysis of card sorting data we contributed the first detailed depiction of this group of non-traditional programmers' understanding of foundational programming concepts. Our qualitative results provided additional evidence in support of models of opportunistic programming, and we further elaborated on the common resources web developers seek out in order to learn something new. Lastly, we have distilled four implications for future research in end-user programming.

Millions of users are developing informal notions of computation through their daily experiences with the Web and computers more generally. By focusing on these users' behaviors as acts of learning, we can better understand what they know, how they learn, and how we can help them achieve a more thorough understanding. Doing so also allows us to build towards a future of true universal computational literacy.

ACKNOWLEDGMENTS

We sincerely thank our study participants for volunteering their time. This material is based upon work supported by the National Science Foundation under Grant Nos. 0613738 and 0618674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

1. L. Beckwith, C. Kissinger, M. Burnett, S. Wiedenbeck, J. Lawrance, A. Blackwell, and C. Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of CHI '06*, pages 231–240, 2006.
2. A. F. Blackwell. First steps in programming: a rationale for attention investment models. In *Proceedings of HCC '02*, pages 2–10, 2002.
3. J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of CHI '09*, pages 1589–1598, 2009.
4. V. Braun and V. Clarke. Using thematic analysis in psychology. *Qualitative Research in Pyschology*, 3(2):77–101, 2006.
5. B. Dorn, A. E. Tew, and M. Guzdial. Introductory computing construct use in an end-user programming community. In *Proceedings of VL/HCC '07*, pages 27–30, 2007.
6. B. Du Boulay. Some difficulties of learning to program. In E. Soloway and J. Spohrer, editors, *Studying the Novice Programmer*, pages 283–299. LEA, Hillsdale, NJ, 1989.
7. M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.
8. S. Fincher and J. Tenenberg. Making sense of card sorting data. *Expert Systems*, 22(3):89–93, 2005.
9. T. R. G. Green and S. J. Payne. Organization and learnability in computer languages. *International Journal of Man-Machine Studies*, 21:7–18, 1984.

10. J. G. Greeno, A. M. Collins, and L. B. Resnick. Cognition and learning. In D. C. Berliner and R. C. Calfee, editors, *Handbook of educational psychology*, pages 15–46. Simon and Schuster Macmillan, New York, NY, 1996.

11. M. Guzdial and C. Kehoe. Apprenticeship-based learning environments: A principled approach to providing software-realized scaffolding through hypermedia. *Journal of Interactive Learning Research*, 9(3/4):289–336, 1998.

12. C. Horstmann. *Big Java*. John Wiley and Sons, Hoboken, NJ, 2nd edition, 2006.

13. C. Katsanos, N. Tselios, and N. Avouris. Autocardsorter: designing the information architecture of a web site using latent semantic analysis. In *Proceedings of CHI '08*, pages 875–878, 2008.

14. A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proceedings of VL/HCC '04*, pages 199–206, 2004.

15. J. Lave, M. Murtaugh, and O. de la Rocha. The dialectic of arithmetic in grocery shopping. In B. Rogoff and J. Lave, editors, *Everyday Cognition*, pages 67–94. Harvard University Press, Cambridge, MA, 1984.

16. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of CHI '08*, pages 1719–1728, 2008.

17. G. Lewandowski, A. Gutschow, R. McCartney, K. Sanders, and D. Shinners-Kennedy. What novice programmers don't know. In *Proceedings of ICER '05*, pages 1–12, 2005.

18. J. Lewis and W. Loftus. *Java Software Solutions (Java 5.0 version): Foundations of Program Design*. Addison Wesley, Boston, MA, 4th edition, 2005.

19. H. Lieberman, editor. *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, CA, 2001.

20. H. Lieberman, F. Paternó, and V. Wulf, editors. *End User Development*. Springer, 2006.

21. G. Little and R. C. Miller. Translating keyword commands into executable code. In *Proceedings of UIST '06*, pages 135–144, 2006.

22. K. B. McKeithen, J. S. Reitman, H. H. Rueter, and S. C. Hirtle. Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13:307–325, 1981.

23. J. F. Pane, C. Ratanamahatana, and B. A. Myers. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54:237–264, 2001.

24. B. Rogoff and J. Lave, editors. *Everyday Cognition*. toExcel, New York, NY, 1999.

25. M. B. Rosson, J. Ballin, and J. Rode. Who, what, and how: A survey of informal and professional web developers. In *Proceedings of VL/HCC '05*, pages 199–206, 2005.

26. W.-M. Roth. Mathematical inscriptions and the reflexive elaboration of understanding: An ethnography of graphing and numeracy in a fish hatchery. *Mathematical Thinking and Learning*, 7(2):75–110, 2005.

27. G. Rugg and P. McGeorge. The sorting techniques: a tutorial paper on card sorts, picture sorts and item sorts. *Expert Systems*, 14(2):80–93, 1997.

28. G. Rugg and M. Petre. *A gentle guide to research methods*. Open University Press, Berkshire, UK, 2007.

29. K. Sanders, S. Fincher, D. Bouvier, G. Lewandowski, B. Morrison, L. Murphy, M. Petre, B. Richards, J. Tenenberg, L. Thomas, R. Anderson, R. Anderson, S. Fitzgerald, A. Gutschow, S. Haller, R. Lister, R. McCauley, J. McTaggart, C. Prasad, T. Scott, D. Shinners-Kennedy, S. Westbrook, and C. Zander. A multi-institutional, multinational study of programming concepts using card sort data. *Expert Systems*, 22(3):121–128, 2005.

30. C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of VL/HCC '05*, pages 207–214, 2005.

31. E. Soloway, M. Guzdial, and K. E. Hay. Learner-centered design: The challenge for HCI in the 21st century. *Interactions*, 1(2):36–48, April 1994.

32. J. Spohrer and E. Soloway. Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, November 1985.

33. S. Turkle and S. Papert. Epistemological pluralism and the revaluation of the concrete. In I. Harel and S. Papert, editors, *Constructionism: Research reports and essays, 1985-1990*, pages 161–192. Ablex, Norwood, N.J., 1991.

34. A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of CHI '03*, pages 305–312, 2003.

35. J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

36. J. M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin Beedle, Wilsonville, OR, 2004.