Computer Science Construct Use, Learning, and Creative Credit in a Graphic Design Community

Brian Dorn, Allison Elliott Tew, and Mark Guzdial

Technical Report
#GT–IC–08–01
February 2008

Abstract: End-users, who are projected to outnumber professional programmers in the next decade, present a unique opportunity to understand how computer science knowledge is acquired in the real world. We conducted an analysis of projects created by end-user programmers to discern their adoption of introductory computing constructs. A variety of project sizes were represented in the data, ranging from fewer than 100 lines of source code to greater than 1500. Many introductory computing constructs were highly adopted, but some were relatively unused. As these variations in adoption could be indications of topic complexity, we compared our findings to previous work in the novice programming literature. Additionally, a data-driven analysis provided insight into user sharing and reuse practices. Many distinct approaches to copyright and code ownership concerns were found in the projects studied, and their potential impact on end-user learning was considered.

A short paper based on these results appears as: B. Dorn, A. E. Tew, and M. Guzdial. Introductory computing construct use in an end-user programming community. In *VL/HCC'07: Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 27–30, 2007

School of Interactive Computing
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0760

Computer Science Construct Use, Learning, and Creative Credit in a Graphic Design Community

Brian Dorn, Allison Elliott Tew, and Mark Guzdial
School of Interactive Computing
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{*dorn, allison, guzdial*}*@cc.gatech.edu*

Abstract

End-users, who are projected to outnumber professional programmers in the next decade, present a unique opportunity to understand how computer science knowledge is acquired in the real world. We conducted an analysis of projects created by end-user programmers to discern their adoption of introductory computing constructs. A variety of project sizes were represented in the data, ranging from fewer than 100 lines of source code to greater than 1500. Many introductory computing constructs were highly adopted, but some were relatively unused. As these variations in adoption could be indications of topic complexity, we compared our findings to previous work in the novice programming literature. Additionally, a data-driven analysis provided insight into user sharing and reuse practices. Many distinct approaches to copyright and code ownership concerns were found in the projects studied, and their potential impact on end-user learning was considered.

1. Introduction

End-users, who are projected to outnumber professional programmers by more than four to one in the next decade [17], present a unique opportunity to understand how computer science knowledge is acquired in the real world. Graphic designers and others involved in media editing make up a relatively new and growing group of end-user programmers (EUP). Through scripting, these users might build software to create custom effects or automate batch jobs to cut down on repetitive tasks. The Adobe® Photoshop® image-editing application is one widely available tool with affordances for scripting in the graphic design domain. These users often program with ExtendScript, a cross-platform extended implementation of the JavaScript™ language used in Adobe applications[1].

In a previous study of graphic design end-users, we gathered self-reported practices and knowledge through an online survey [4]. The respondents recognized and claimed to use many programming constructs like variables, subroutines, conditionals, loops, and data structures. They also indicated a propensity for code reuse by sharing and borrowing code. Following this analysis we were interested in exploring the ways end-users' reported behaviors correspond to actual scripted artifacts.

Additionally, graphic designers reported use of related example projects as a common source of support when learning something new—a result mirrored in other end-user contexts. We believe that online communities and forums provide a vital repository of example projects from which other end-users learn. As such, we were curious about the extent of computer science knowledge contained within projects found in online end-user communities.

To investigate these questions, we analyzed a corpus of scripting projects written by users of Photoshop, in particular looking for evidence of introductory computing construct use. The results of this code-centric exploration are presented here. The method used for this study is outlined in Section 2. An overview of the results is presented in Section 3, with a detailed discussion of patterns in construct adoption following in Section 4. Matters related to code as intellectual property are outlined in Section 5. We conclude with a brief commentary on future directions for this line of research.

[1] Adobe and Photoshop are registered trademarks of Adobe Systems, Inc. JavaScript is a trademark of Sun Microsystems, Inc.

2. Method

We conducted an artifact analysis of all scripts publicly available for download in the Photoshop section of the Adobe Exchange repository[2], an end-user programming community for graphic designers. To focus our analysis, we only considered scripts that were hosted in the Adobe forum directly and did not include forum contributions that referenced scripts hosted on other sites.

2.1. Development of Coding Scheme

We developed a coding scheme that considered both general introductory computing constructs as well as EUP domain specific constructs. The computing constructs were informed by the computing education literature, while the domain specific constructs were suggested by end-user programming studies and derived in a data-driven manner by the scripts themselves.

To avoid bias from a particular language or pedagogical approach, we wanted to identify computing constructs that were common across introductory courses. We began by conducting an analysis of the table of contents of the top two CS1 textbooks as identified by each of the major publishers—12 books in total. This list of concepts was revised using the framework of the Computer Science volume of Computing Curricula 2001 [1] as an organizing principle. It was further refined by analyzing the content of canonical texts representing each of the common introductory approaches (objects-first [3, 13], functional-first [5], and imperative-first [22]). A construct was included in the coding scheme if it was covered by all of the texts or excluded by only one of the texts. The coding scheme was then further modified considering the EUP domain. Some concepts, such as scope, were too abstract to operationalize; others were not relevant or practical in the domain (e.g., Big-O notation and class-based inheritance). The resulting computing constructs included in the coding scheme are listed in Figure 1.

Most of these constructs in JavaScript are similar to their counterparts in general-purpose computing languages. However, a few warrant additional explanation. The "number" coding element included use of any kind of numeric literal as JavaScript does not distinguish between types of numerics (e.g., integer, floating point). In this community of graphic designers, user input and output is inherently graphical in nature. As such, the I/O constructs in the coding scheme included input dialogs and message boxes. Lastly, since the nature of

[2]All files retrieved November 30, 2006 from `http://share.studio.adobe.com`

variable	selection (`if`)
mathematical operators	definite loop (`for`)
relational operators	indefinite loop (`while`)
logical operators	nested loops
assignment	recursion
number	user defined functions
boolean	user defined objects
string	user input
array	output to user
type conversion	

Figure 1. Textbook-Based Coding Elements

Photoshop scripting considered here almost always requires calling of functions and using objects from the API, we limited our scope to instances of user-defined functions and objects. User-defined functions had to be explicitly defined and named, and user-defined objects had to include a constructor and be instantiable.

To properly analyze EUP scripts, it was important to supplement the general introductory computing concepts with domain specific ones. Previous studies of end-user programming practices [4, 15, 16] suggested that intellectual property and code modularity could be important considerations in this domain. We added three items to the coding scheme (copyright notice, end-user license agreement, and credits external sources) to address the issue of intellectual property. ExtendScript allows for importing and exporting of code to aid in modularity and code reuse, so these items were also added to the coding scheme.

A few data-driven constructs were included as well. We noted that some users had attempted to make their scripts unreadable by humans; others employed built-in functionality in Photoshop to record their script via the user interface rather than typing code; and others still incorporated rather sophisticated exception handling mechanisms. We wondered how common these practices were and added these to the coding scheme. The resulting EUP constructs are listed in Figure 2.

copyright notice	exception paths (`try`/`catch`)
end user license agreement	use of recorded code
credits external sources	includes external code
code obfuscation	externalizes code to client

Figure 2. EUP Coding Elements

2.2. Coding Process Details

We began the coding process by establishing the reliability of the coding scheme. The first two authors coded a random sample consisting of 13 scripts ($\approx 20\%$ of the total data set) according to the coding scheme.

We computed the kappa statistic [2] as a measure of inter-rater reliability, and while most of our coding elements exceeded the κ=0.80 threshold expected in the social sciences [11], some revisions were necessary. After updating the criteria and recoding another sample, we achieved a κ=1.00 on all remaining coding elements. Our high κ values may be partially attributed to binary coding categories and the lack of rater judgment required on some constructs. Once inter-rater reliability was confirmed, the first two authors each coded half of the scripts according to the revised coding scheme.

3. Results

The initial set of Photoshop scripts was collected from the Adobe Exchange community and then cleaned of any entries that were corrupt or incorrectly categorized. After removing the improper entries, the final data set contained a total of 62 individual scripts making up 48 distinct projects contributed by 27 unique users. We use the term *project* to refer to one downloadable entry in the online community. For example, a project could consist of a single script posted as a text file, or it could be an archive file containing multiple, related scripts and associated data files. Figure 3 illustrates the distribution of project submissions. Most users posted only one project, though one-third of users made multiple contributions to the community.

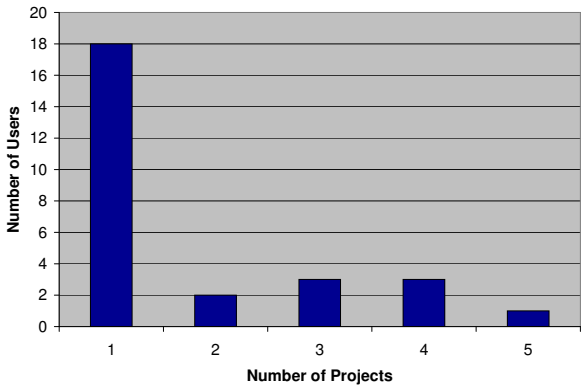

Figure 3. Distribution of Project Submissions

The bulk of the results presented here uses a per-project unit of analysis, rather than a per-user or per-script approach. Focusing on individual projects mitigates skewing effects that might be introduced by single projects that contain multiple scripts (as in a per-script analysis). We also avoid a per-user analysis as it would be somewhat precarious to infer knowledge of programming based solely on constructs used in a minimal set of examples, particularly given that most users only contributed one project. Although previous studies indicate that many end-users lack formal training in computer science (e.g., [4, 15]), we do not know the particular training background of the users who posted to this forum, nor can we infer whether these scripts are the result of personal or work projects. In a sense, what we present here is an analysis of the computing content embodied in projects that might serve as case examples from which another end-user could learn [10].

3.1. Project Size

While there were as many as eight scripts in a single project, most (87.5%) contained only one script. In order to gain insight into the size and complexity of the projects being created, we calculated the total number of lines used for code statements, whitespace, and comments for each. We report based on the sum of the individual script line lengths for projects containing multiple scripts. Table 1 summarizes basic statistics for project size. There was a large amount of variation in each of the line types computed, as noted by the standard deviations. However, the median lengths indicated the creation of moderately sized scripts that included a fair amount of commenting, though the nature of the comments was not closely examined.

Table 1. Project Line Length Breakdown

	Mean	StDev	Median	Min	Max
Code	555.56	674.89	246.5	9	3224
Comment[3]	63.54	65.18	26.5	0	237
Whitespace	65.46	158.47	20.5	0	1057
Total	676.96	760.61	403.5	11	3300

Looking at the distribution of these lengths provided a more detailed picture of project size. Figure 4 depicts the range of project sizes in terms of the number of source code lines. There were two noticeable peaks in this distribution, the first of which occurred at 200 or fewer lines of code. This might be predicted if users are expected to implement short programs that accomplish relatively simple tasks. More surprisingly, there was a clear second peak occurring in projects with greater than 1000 lines of code.

Project sizes provided an initial feel for the size and complexity of code, but a more detailed analysis of each project's content was needed to understand the types of computing knowledge evidenced in the code base.

[3]Lines counted under "Comment" include both comment-only lines and code lines which have terminal comments.

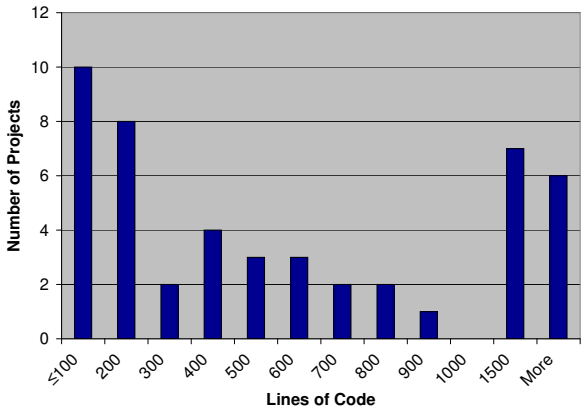

Figure 4. Distribution of Project Lengths

3.2. Project Content

Application of the coding scheme to all of the individual scripts resulted in an overview of construct use. These results were then aggregated to form a per-project summary. For those projects containing multiple scripts, a construct was indicated as being used if one or more of the constituent scripts used the construct. The aggregate use amounts for each construct, grouped by higher-order concern, are presented in Table 2.

The most commonly used programming constructs were: variable, assignment, relational operators, selection, number, and string. These results were largely expected given that tasks like assigning a numeric value to a variable are fundamental to most coding activities. Excluding those related to intellectual property and recorded code (as these are not programmatic constructs per se), the least frequent constructs were: indefinite loop, nested loops, recursion, type conversion, user-defined objects, and exporting/importing code. Some of these observations could be tied to tool and language influences (e.g., creation of instantiable objects in JavaScript is awkward). Others, like the decrease in use from definite loops to nested loops to recursion, seem indicative of conceptual difficulties noted in previous research (e.g., [18, 21]). We present a detailed discussion of these issues in the section that follows.

4. Discussion of Construct Adoption

We observed that within the higher-order concerns Expressions, Control Structures, and Modularity some constructs are heavily adopted while others are used at much lower levels (see Table 2). End-user programmers can be viewed in many ways as novices because they traditionally lack formal training in computer science and learn content just-in-time as it relates to their specific tasks [4, 15]. Therefore, previous studies of novice programmer behavior provide useful information for interpreting our results.

Table 2. Construct Use by Project

Construct		Use %
Variable		100.00%
Use of Recorded Code		33.33%
Expressions	Assignment	100.00%
	Relational Operators	97.92%
	Mathematical Operators	83.33%
	Logical Operators	54.17%
Control Structures	Selection (`if`)	97.92%
	Definite Loop (`for`)	60.42%
	Exception Paths (`try/catch`)	60.42%
	Indefinite Loop (`while`)	37.50%
	Nested Loops	29.17%
	Recursion	2.08%
Data Types & Structures	Number	100.00%
	String	95.83%
	Array	83.33%
	Boolean	79.17%
	Type Conversion	29.17%
I/O	Output to User	83.33%
	User Input	60.42%
Modularity	User-defined Functions	70.83%
	User-defined Objects	18.75%
	Import or Include External Code	0.00%
	Export Code to External Client	0.00%
Intellectual Property	Copyright Notice	62.50%
	End User License Agreement	47.92%
	Credit Given to External Sources	22.92%
	Human Unreadable Code Obfuscation	8.33%

4.1. Learning Complexity

4.1.1. Operators. Almost all (97.92%) of the projects used a relational operator (e.g., `<`, `>=`, `!=`), most often inside the condition of a selection (`if`) statement. A clear majority (83.33%) also used mathematical operators. While some projects did include numeric calculations to resize images or parts of images, many of the mathematical operators noted were uses of the unary increment operator as part of a definite loop (`for`) construct. However, markedly fewer projects (54.14%) used a logical operator (`&&`, `||`, `!`).

Our previous work with introductory students [19]

indicated that beginners tend to struggle with boolean logic in conditional statements. Pane, et al. [14] found that boolean operators are particularly difficult for beginners because they require statements to be expressed in ways that are unfamiliar. Since many end-user programmers are self-taught and have learned to program to support their own goals, it is perhaps not surprising that they seem to have adopted the operators in their projects which are most familiar.

4.1.2. Control Structures. Control structures of some kind were included in most of the projects we analyzed. Almost all (97.92%) of them included a selection (`if`) statement, and most (60.42%) used a definite loop (`for`) construct. However, only a third of the projects used the indefinite loop (`while`) or nested loop constructs.

Soloway, et al. [18] identified the inherent complexity of the `while` loop because it conflicts with the preferred cognitive strategy that students employ when solving iterative problems. The definite loop (`for`) construct more closely matches the *read, then process* strategy, thus possibly explaining its higher adoption rate by the end-user programmers we studied. Additionally, the infrequent use of nested loops could be another sign of cognitive complexity. Boundary condition errors are a frequent mistake when beginning students write loops, and loop nesting only exacerbates these boundary concerns [7].

Only one project included recursion, a topic with which many novices struggle [21]. A common pedagogical technique to address this difficulty is to introduce recursion by way of analogy, but Photoshop lacks readily apparent concrete examples. However, there are some tasks in this context that lend themselves to recursive solutions. For example, the one use of recursion in our analysis, in effect, traversed a tree made of image layers (leaf nodes) and layer sets (internal nodes) removing all non-visible layers along the way.

4.1.3. Abstraction and Modular Coding. A large portion (70.83%) of the projects contained user-defined functions, while significantly fewer (18.75%) implemented objects. Despite the fact that ExtendScript documentation highlights the ability to create reusable code modules using external files, we noted that no project incorporated either the import or export construct—though ExtendScript is a relatively recent addition to the Adobe product line, and users may still be discovering and learning its extended functionality.

Fleury [6] observed that students consistently preferred programs containing duplicated code rather than programs that used abstracted functions, claiming that it was more easily read and debugged. While we note that functions were highly used in this domain, more advanced abstractions for modularity were largely ignored. Hoadley, et al. suggest an explanation that "abstract understanding of a function and belief in the benefits of reusing code" [8, p. 109] impact whether or not a user is likely to invest time in programming for abstraction.

4.2. Pathways for Learning

The novice programming literature sheds light on why some constructs are being adopted more heavily than others. Being optimists, we do not believe that even an end-user programmer will be satisfied using only the most common constructs. Being realists, we do not believe that end-user programmers will use more sophisticated constructs having never experienced the more common ones. Notably, the projects in this corpus provide a unique illustration of points along the continuum towards expert programming practices.

4.2.1. Functional Decomposition. One such set of points can be seen in the transition from programs containing neither functions nor notions of scope to near textbook examples of procedural coding. We saw several examples of monolithic code blocks, lacking any indication of functional separation. Other projects had clearly demarcated sections with specific roles, but lacked formal declaration of functions—though creating them would have been relatively easy given the end-user's forethought. Still other scripts contained what might best be called "ad hoc functions" where function declarations were intermixed with mainline code and were immediately called following the definition (as well as later in the main program). Here we imagine a programming process where an end-user realizes the need for code written previously and decides to mark the earlier code block as a function, thus reusing, rather than copying. Lastly, we did have one example of model procedural code with a main program, distinct functions, logical organization of program components, and detailed comments including a discussion of identifier scope.

4.2.2. Recorded Code. Uses of recorded code in this data set also hint at possible first, critical steps from "non-programmer" to "programmer." Photoshop provides the ability to record a user's activity in the interface (e.g., mouse clicks, menu selections) as a textual script, which can later be inspected and modified. Figure 5 shows the code produced when selecting the "rotate clockwise 90 degrees" option from the menu. The generated code is rudimentary—a replay of windowing events, with little connection to the underlying

```
var id905 = charIDToTypeID( "Rtte" );
    var desc44 = new ActionDescriptor();
    var id906 = charIDToTypeID( "null" );
        var ref6 = new ActionReference();
        var id907 = charIDToTypeID( "Dcmn" );
        var id908 = charIDToTypeID( "Ordn" );
        var id909 = charIDToTypeID( "Frst" );
        ref6.putEnumerated( id907, id908, id909 );
    desc44.putReference( id906, ref6 );
    var id910 = charIDToTypeID( "Angl" );
    var id911 = charIDToTypeID( "#Ang" );
    desc44.putUnitDouble( id910, id911, 90.000000 );
executeAction( id905, desc44, DialogModes.NO );
```

Figure 5. Recorded Code for Rotating Image

semantics of the operations performed. It also uses indentation in a seemingly mysterious, nonstandard way. Matt Kloskowski, education and curriculum developer for the National Association of Photoshop Professionals, notes:

> I have to warn you, though—this is not for those with weak hearts or those who frighten easily. This [recorded] log file is really scary looking, but you should be able to see some familiar text in it. [9, p. 165]

The familiar text to which he refers often takes the form of input values, like the parameter 90.0 in the rotate example. Despite its awkwardness, the recording feature can be useful to generate code for actions which one would like to incorporate but are not easily achieved using the API [9].

There were projects in our corpus, including the largest project of over 3000 lines, that were entirely recorded, but we also saw scripts that consisted of recorded sections intermixed with user-written comments and code. At the far end, the distinction between recorded and written code was blurred when some end-users appeared to use code that looked recorded but lacked the characteristic auto-generated variable names, like those seen in Figure 5. We also had examples of recorded code that had been encapsulated in user-defined functions with descriptive comments about their purpose, a strategy suggested by Photoshop reference materials (e.g., [9, 20]).

5. Intellectual Property Concerns

In addition to our analysis of introductory construct use, we added a number of items to our coding scheme to explore issues of code sharing and reuse. The data collected for these items hint at aspects of end-user culture related to intellectual property and raise questions for those who seek to support the end-user practice of learning from pre-existing example code. A significant number (62.50%) of projects contained explicit declaration of their author's copyright in the project source code or an accompanying documentation file. There was also evidence that collaboration and code reuse between users does take place in this community; 22.92% of the projects contained acknolwedgements of code borrowed or adapted from other people. Notably, 47.92% of projects contained some form of end-user license agreement (EULA). Though these projects were produced by only 8 of the 27 total users, the range of agreement types represented was surprising. Seven distinct EULA types were in the corpus:

- Public Domain
- Freeware, modifiable & redistributable
- Freeware, non-modifiable & non-redistributable
- Charity-ware
- Donation-ware
- Demo-ware
- GNU General Public License v2

The least restrictive EULA was an explicit "public domain" declaration in the source code comments. There were variations on the notion of freeware with some definitions permitting reuse of source code, while others did not. Interestingly, one project implored the user to make a donation to any charity if he or she found the script to be useful. Others politely requested a small contribution to the author, and some even tempted users with premium versions of the script with additional functionality upon payment. The most sophisticated agreement, legally speaking, was one author's use of the GNU GPL open source software license in four separate projects.

A number of potential issues were unaddressed by the EULAs in this set of projects. Many of the licenses were implied by simple use of a single term (e.g., freeware). Individual interpretations of what such terms mean could lead to many different outcomes. Further, several licenses were incomplete and only specified part of the acceptable use terms. For example, the charity-ware and donation-ware cases state that users should make some remuneration if they find the script useful. However, the license does not specify in which activities the user may engage following payment—would it be acceptable to borrow some of the code in a new script of the user's design?

The difficulty with EULAs in this context is the confusion of source code with its executable artifact.

Many of these license paradigms make sense if they are applied to a compiled, executable program; however, the fact that these scripts are human-readable and later interpreted by the Photoshop host application blurs the boundaries. One author appeared to have this in mind and obfuscated projects by using escape characters to replace all of the ASCII text from the source files.

Our analysis of intellectual property concerns related to sharing has implications for the potential of this community to serve as a knowledge base for others. Restrictive license agreements could discourage others from trying to use a project as a source of knowledge, and code obfuscation makes any learning impossible. With the attention paid to patent lawsuits and other intellectual property disputes in the mass media today, vague EULAs (or the lack of any terms of use) could confuse many users about what actions are and are not acceptable.

There have been recent efforts to make licensing more accessible to the general public. The Creative Commons[4] offers a simple way for content creators to mark their works and outline acceptable uses through license standards [12]. Unfortunately Creative Commons agreements are not intended to cover source code, and authors are merely referred to GNU's open source license agreements. The range of license types in this small corpus suggests that a one-size-fits-all approach will not be sufficient, but that some standardization of terms is necessary. Finding new ways to balance sharability and credit seem particularly important for fostering learning in end-user programming communities.

6. Conclusion & Future Work

Our analysis of the Adobe Photoshop Exchange scripting community indicates that graphic design end-users are producing moderately sized projects. Artifacts from this community exhibit varying degrees of introductory computing construct use. The high rate of use of some constructs matches our intuitions, but some results are suprising. For example, every project contained variable and assignment constructs, but unexpectedly more than half incorporated exception handling mechanisms. Research findings on construct complexity from the study of novice programmers appear to explain much of the variation in construct adoption in this end-user sample.

The transitional processes described in Section 4.2 raise questions beyond the scope of this study. Are these truly points on a theoretical developmental curve along which end-users progress, or do individuals retain their personal strategies over time? Do these points represent a natural progression that we might leverage in our formal classrooms to help better develop student understanding?

We speculated on end-user practices in Section 5 that are deeply connected with ownership of and recognition for creative artifacts. Yet, the real effect of licensing agreements on learning in these communities is unclear. Pursuing more rich understandings of intellectual property concerns and developing new paradigms for sharing source code could be fruitful.

The claims here about computer science knowledge are limited to the constructs that were present in the online community—the projects were treated as case studies. However, it would also be of interest to determine the true extent of computer science knowledge among these end-users. Thus, a natural next step would be to conduct comprehensive interviews to tease out the depth of construct understandings. Such an analysis would also allow us to investigate which constructs users perceive as most relevant to their practices.

On a larger scale, end-user programming provides a unique outlet for the examination of informal computer science learning. Not only could further research here allow tool builders to better support end-users, but insights gained about how and why computer science knowledge is acquired "in the wild" could also impact traditional classroom environments. We hope to better understand existing end-user practices in order to explore their educational implications in both formal and informal settings.

7. Acknowledgments

This material is based upon work supported in part by the National Science Foundation under grants ITR-SoD 0613738 and CCLI-ASA 0512213.

References

[1] Computing curricula 2001. *Journal on Educational Resources in Computing*, 1(3es):1–240, 2001.

[2] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Pyschological Measurement*, 20(1):37–46, 1960.

[3] H. Deitel and P. Deitel. *C++: How to Program*. Prentice Hall, Upper Saddle River, NJ, 5th edition, 2005.

[4] B. Dorn and M. Guzdial. Graphic designers who program as informal computer science learners. In *ICER '06: Proceedings of the Second International Computing Education Research Workshop*, pages 127–134, 2006.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to*

[4] http://www.creativecommons.org

Programming and Computing. MIT Press, Cambridge, MA, 2001.

[6] A. Fleury. Code reuse through the eyes of students and professionals. In *Proceedings of the National Educational Computing Conference*, pages 136–142, 1997.

[7] D. Ginat. On novice loop boundaries and range conceptions. *Computer Science Education*, 14(3):165–181, 2004.

[8] C. M. Hoadley, M. C. Linn, L. M. Mann, and M. J. Clancy. When, why, and how do novice programmers reuse code? In W. D. Gray and D. A. Boehm-Davis, editors, *Empirical Studies of Programmers: 6th Workshop*, pages 109–129. Ablex, Norwood, NJ, 1996.

[9] M. Kloskowski. *The Photoshop CS2 Speed Clinic: Automating Photoshop to Get Twice the Work Done in Half the Time*. Peachpit Press, Berkeley, CA, 2006.

[10] J. L. Kolodner. Educational implications of analogy. *American Psychologist*, 52(1):57–66, 1997.

[11] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

[12] L. Lessig. *Code version 2.0*. Basic Books, New York, NY, 2006.

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

[14] 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.

[15] M. B. Rosson, J. Ballin, and J. Rode. Who, what, and how: A survey of informal and professional web developers. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 199–206, 2005.

[16] C. Scaffidi, A. Ko, B. Myers, and M. Shaw. Dimensions characterizing programming feature usage by information workers. In *VL/HCC '06: 2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 59–62, 2006.

[17] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 207–214, 2005.

[18] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. In E. Soloway and J. C. Spohrer, editors, *Studying the novice programmer*, pages 191–207. Lawrence Erlbaum Associates, Hillsdale, NJ, 1989.

[19] A. E. Tew, W. M. McCracken, and M. Guzdial. Impact of alternative introductory courses on programming concept understanding. In *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research*, pages 25–35, 2005.

[20] J. Tranberry. Installing and using the ScriptingListener plug-in. Retrieved March 13, 2007, from `http://www.tranberry.com/photoshop/photoshop_scripting/tips/listener.html`.

[21] S. Wiedenbeck. Learning recursion as a concept and as a programming technique. In *SIGCSE '88: Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, pages 275–278, 1988.

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