Introductory Computing Construct Use in an End-User Programming Community

Brian Dorn, Allison Elliott Tew, and Mark Guzdial
Georgia Institute of Technology
{brian.dorn, allison.tew, mark.guzdial}@cc.gatech.edu

Abstract

Previous studies of end-user programmers have indicated a reliance on related examples for learning. Accordingly, we analyzed the projects contained in an online community with respect to their use of introductory computing constructs. In general, the projects resemble those of novice programmers, implying the opportunity for supporting additional learning. Project authors' attention to matters of intellectual property may also directly impact other end-users' willingness to learn.

1. Introduction

End-user programmers 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.

In a previous study of graphic design end-users, we gathered self-reported practices and knowledge through an online survey [2]. 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. 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.

Thus, we believe that online communities provide vital example projects from which other end-users learn. The goal of this paper is to closely examine a repository of example code with respect to the type of introductory computing constructs embodied therein. Such a description is an important step in understanding the computing constructs end-user programmers are likely to encounter when seeking help, as well as an indication of those concepts that may need additional attention if we wish users to engage with them.

We present results of this code-centric exploration of Photoshop scripts here. In the sections that follow we outline our methodology, examine the construct adoption results, and discuss the importance of intellectual property (IP) matters in end-user communitites.

2. Method

We conducted an artifact analysis of all publicly-available scripts hosted in the Photoshop section of the Adobe Exchange repository, an end-user programming community for graphic designers.[1]

2.1. Development of Coding Scheme

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

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]. It was further refined by analyzing the content of canonical texts representing each of the common introductory paradigms. A construct was included in the coding scheme if it was covered by all

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

0-7695-2987-9/07 $25.00 © 2007 IEEE
DOI 10.1109/VLHCC.2007.35

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

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

Figure 2. EUP Coding Elements

of the texts or excluded by only one of the texts. These coding elements were 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). The resulting computing constructs included in the coding scheme are listed in Figure 1.

To properly analyze end-user scripts, it was important to supplement the general introductory computing concepts with a few domain-specific ones. Many scripts made use of exception handling, so we augmented the scheme with this construct. Previous studies of EUP practices [2, 7] 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 IP. 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. The resulting EUP constructs are listed in Figure 2.

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 of 20% of the scripts. We then computed Cohen's kappa statistic as a measure of interrater reliability, and while most of our coding elements exceeded the κ=0.80 threshold expected in the social sciences [6], some revisions were necessary. After updating the criteria and recoding another random sample, the kappa statistic was acceptable for all remaining constructs. Each of the first two authors then coded half of the scripts according to the revised coding scheme.

3. Results

After removing corrupt and non-script entries, our 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 scripts and data files. Most users posted only one project, though one-third of users made multiple contributions.

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. What we present here is a characterization of the computing content embodied in examples from which another end-user could learn.

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. Each of the line types computed has a large degree of variation, as noted by the standard deviations.

The distribution of these lengths provided a more detailed picture of project size. A large number of projects (37.5%) were 200 or fewer lines of code, 27.1% contained over 1000 lines, and the remainder of projects were spread relatively evenly between 200 and 1000 lines. The preponderance of short projects might be predicted if users are expected to implement short, simple programs, but the number of examples in the highest category suggests code with significant complexity.

Table 1. Project Line Length Breakdown

	Mean	StDev	Median	Min	Max
Code	555.56	674.89	246.5	9	3224
Comment	63.54	65.18	26.5	0	237
Whitespace	65.46	158.47	20.5	0	1057

Table 2. Construct Use by Project

Construct		Use %
Variable		100.00%
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%
IP	Copyright Notice	62.50%
	End User License Agreement	47.92%
	Credit Given to External Sources	22.92%

3.2. Project Content

A more detailed analysis was needed to understand the nature of the computing knowledge evidenced in the code base. We applied the coding scheme to each individual script and aggregated the results 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 frequent programming constructs were: variable, assignment, relational operators, selection, number, and string. The least common were: indefinite loop, nested loops, recursion, type conversion, user-defined objects, and exporting/importing code. Some of these observations could be tied to language influences (e.g., creation of instantiable objects in JavaScript is awkward). Others, like the decline from definite loops to nested loops to recursion, seem indicative of conceptual difficulties noted elsewhere (e.g., [8, 9]).

4. Discussion of Construct Adoption

We observed that within expressions, control structures, and modularity some constructs are heavily adopted while others are not. 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 [2, 7]. Therefore, previous studies of novice programmers provide an interesting lens for interpreting our results. We consider two such comparisons below.

4.1. Control Structures

Control structures were included in most of the projects we analyzed. Almost all (97.92%) of them included a selection statement, and most (60.42%) used a definite loop. However, only a third of the projects used the indefinite loop or nested loop constructs. Soloway, et al. [8] identified the inherent complexity of the **`while`** loop because it conflicts with the preferred cognitive strategy that students employ when solving iterative problems. The **`for`** loop more closely matches the *read, then process* strategy, thus possibly explaining its higher adoption rate in the code base we studied. The infrequent use of nested loops could also be due to cognitive complexity. Boundary condition errors are frequent when beginning students write loops, and loop nesting only exacerbates these issues [4].

Only one project included recursion, a topic with which many novices struggle [9]. Often instructors 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 traversed a tree made of image layers (leaf nodes) and layer sets (internal nodes) removing all non-visible layers along the way.

4.2. Abstraction and Modular Coding

A large portion (70.83%) of the projects contained user-defined functions, while significantly fewer (18.75%) implemented objects. Despite ExtendScript documentation highlighting the ability to create reusable code modules, no project incorporated either the import or export construct. Fleury [3] observed that students preferred programs containing duplicated code rather than programs that used abstracted functions, claiming that it was more easily read and debugged. While functions were highly used in this domain, more advanced abstractions for modularity were largely ignored. Hoadley, et al. suggest that “abstract understanding of a function and belief in the benefits of reusing code" [5, p. 109] impact whether or not one is likely to invest time in programming for abstraction.

5. Intellectual Property Concerns

The data collected to explore issues of code sharing and reuse hint at aspects of end-user culture related to intellectual property. A significant number (62.50%) of projects contained explicit declaration of their author's copyright. There was also evidence that collaboration and reuse occurs between users 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), and the range of agreement types was surprising. Seven distinct EULA types were in the corpus: Public Domain, Freeware (modifiable), Freeware (non-modifiable), Charity-ware, Donation-ware, Demo-ware, and GNU GPL v2.

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 could lead to many different outcomes. Further, several licenses were incomplete and only specified part of the acceptable use terms. For example, the donation-ware case states that users should make some remuneration if they find the script useful. However, the license does not indicate whether code reuse is acceptable following payment.

This analysis also raises questions for those who seek to support the end-user practice of learning from pre-existing code. Restrictive license agreements could discourage others from using a project as a source of knowledge. With the attention paid to patent lawsuits and other IP disputes in the media today, vague or missing EULAs could confuse many users about what actions are or are not acceptable. Exploring new types of licensing that balance sharability and credit seem important for fostering learning in EUP communities.

6. Conclusion & Future Work

The purpose of this study was to characterize the nature of computing knowledge contained in examples of interest to end-user programmers. In many respects, the projects were typical of what one might expect from a novice programmer. Though we found some use of more sophisticated constructs, others which are potentially relevant—and beneficial—to the community and its practices were missing. Enabling this resource to reach its full potential as an EUP learning community demands filling these gaps. Additionally, we conjecture that sharing and reuse in these communities is connected with ownership of and recognition for creative artifacts. Yet, the real effect of licensing agreements on learning in these communities remains unclear.

The claims here about computer science knowledge are limited to the constructs that were present in the projects examined. However, it would also be of interest to determine the true extent of CS knowledge among code contributors. A natural next step would be to conduct interviews to discern construct understanding and perceptions of construct relevance.

Through further study of CS knowledge acquisition "in the wild," we hope to better understand existing end-user practices and their educational implications in both formal and informal settings.

7. Acknowledgments

This work is supported in part by NSF 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] 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.

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

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.