May, 2014

Confronting the Challenges of Computational Design Instruction

Nicholas Shane Senske, *University of North Carolina at Charlotte*

N. Gu, S. Watanabe, H. Erhan, M. Hank Haeusler, W. Huang, R. Sosa (eds.), *Rethinking Comprehensive Design: Speculative Counterculture, Proceedings of the 19th International Conference on Computer-Aided Architectural Design Research in Asia CAADRIA 2014,* 831–840.

CONFRONTING THE CHALLENGES OF COMPUTATIONAL DESIGN INSTRUCTION

NICHOLAS SENSKE
University of North Carolina at Charlotte, Charlotte, NC, USA
nsenske@uncc.edu

Abstract. Many architects understand that learning to program can be a challenge, but assume that time and practice are the only barriers to performing well enough at it. However, research from computer science education does not support this assumption. Multinational studies of undergraduate computer science programs reveal that a significant number of students in their first and second year of full-time instruction still have serious misconceptions about how computer programs work and an inability to design programs of their own. If computer science students have trouble learning to think and express themselves computationally, what does this say about architects' chances of learning to program well? Moreover, if common problems have been identified, can architectural educators learn anything from findings in computer science education research? In order to determine if this research is relevant to architecture, the author conducted a pilot study of architecture students consisting of program analysis and conceptual knowledge tests. The study found that student performance was poor in ways similar to those revealed in the computer science education research. Because architects face similar challenges as computer science majors, this suggests that the discipline could benefit from more investment in educational collaborations. In addition, empirical research – from architecture as well as other fields – must play a more substantial role in helping architects learn computational thinking and expression.

Keywords. Computational design education; programming; computer science education research; empirical research

1. Introduction

In recent years, computational methods, such as those found in BIM, generative scripting, energy simulation, and environmental analysis, have moved

from the fringes to the mainstream of architectural practice. As a consequence, many educators and researchers have identified the need for architecture students to learn what is often referred to as computational (Menges and Ahlquist, 2011; Burry, 2011), algorithmic (Coates, 2010), or parametric (Karle and Kelly, 2011) thinking. Implicitly, learning to think this way within the medium of computing involves learning to program. However, to be clear, programming does not necessarily involve writing code. Specifying procedures for a computer to follow is sufficient (Mateas, 2005; Perlis, 1961), such as when one defines parametric relationships, manipulates visual scripting blocks, designs a simulation, etc. In all of these cases, the same concepts and techniques for structuring, communicating, and coordinating information and logic still apply. Thus, programming, as a general means of expressing oneself computationally, is fast becoming an essential skill for architects.

While there are many books and primers to teach architects programming (e.g. Mitchel, 1987; Terzidis, 2009; Jabi, 2013), neither the text nor structure of the lessons explicitly addresses the difficulty of the task. Unfortunately, not much is known about how to teach programming well (Papert, 1980; Sheil, 1983; Kay, 1996; Mateas, 2005). Indeed, within architecture, this is not acknowledged as a significant problem. There seems to be a widely held belief among architects that, given the right tools and enough time to experiment with them, anyone can develop a sufficient amount of programming skill without much difficulty. However, there is no empirical data from architecture to support this idea. In fact, the data from other fields suggests that learning programming in a classroom, let alone by oneself, is a significant challenge for many people.

Two multi-intuitional international studies found that a majority of computer science students could not program well even after the completion of their first and second years of full-time study. On a test that measured program-writing ability, the average student score was only 22.89 out of 110 points (McCracken et al., 2001). Another study that measured code-reading skills also found that a majority of students did not perform well. Later analysis determined that close to 25% of the students appeared to be guessing their answers to the test (Lister et al., 2004). These studies are widely cited as evidence that programming is difficult skill to learn and to communicate the pressing need for improvements in programming education.

The goal of this paper is to acknowledge some of the challenges of computational design instruction by drawing links between data from architecture students learning programming and studies performed in the field of Computer Science Education (CSEd) research[1]. Two pilot studies are presented, which show that architecture students learning programming in com-

putational design courses appear to have similarly poor performance as computer science students taking introductory courses within their major. An additional goal of this paper is to argue for more empirical data collection and the establishment of a research agenda towards improving programming instruction in architecture.

2. Two Studies of Student Programmers

In order to test whether computer science education research is relevant to architectural education, the author conducted two small studies using materials obtained from three semesters teaching a required computational methods course at the University of North Carolina at Charlotte. Each session of the course lasted 13 weeks with the stated objective of teaching students the concepts and application of computation within design (see Senske, 2013b). The typical class size varied from 70 to 74 students. Three-quarters of the class were undergraduates (third year) with the remaining students from two and three year graduate programs. An early version of the course used the Processing programming language (Fry and Reas, 2001) and the Grasshopper scripting plugin for Rhinoceros (Rutten, 2010). Later versions used only Grasshopper. The author performed the analysis of the student materials after the end of the courses and expressly for the purposes of this paper.

3. Program analysis

3.1. METHODOLOGY

The objective of the first study was to identify and catalogue programming errors in a sampling of student code. The students in the study were introduced to Processing as part of a unit which lasted 8 weeks. During this time, they learned the language and concepts in labs of 15-20 students. The labs featured hands-on tutorials delivered by the author and support provided by a TA. Once a week, all of the students met for a summary lecture. The students completed weekly homework assignments that involved writing programs from a specification (e.g. "write a program that inscribes a circle inside of a square") and answering questions about their programs.

The code for the study came from an assignment to design and program a simple rule-based "drawing machine" using Processing. The objective of the project was to experiment with the aesthetic potential of variables, loops, and conditionals – basic computational building-blocks – introduced in previous weeks. Besides learning about program implementation, creating a drawing machine introduces students to ideas about automation, random vs. logical processes, and creativity with generative systems. These are essential con-

cepts of computational design which extend well beyond the assignment. Because the students were given high-quality sample code and completed earlier homework assignments to practice these skills, they were assumed to have knowledge of the aforementioned programming structures and their proper use. Furthermore, the prompt instructed the students to write programs that demonstrated computation expressively using clean and efficient code.

The author collected a random sample of 30 programs (out of 72 submissions) and carried out the following protocol: 1) first, read the student explanation of the program, 2.) run each program several times to test the output, and 3.) review the code line-by-line and record any errors.

For the purposes of this study, errors were defined as incorrect sections of code. This is to say, code which exhibited misconceptions or misapplications of programming practices. It is important to note that none of the programs in the study contained any bugs or crashes that prohibited the code from running. This can be attributed to the fact that the program had to work in order to be accepted for credit. Instead, the errors made the code less capable of variation, less efficient, more difficult to maintain, and more error-prone. In other words, the programs with errors might be considered evidence of designs that are procedural but do not take full advantage of the powers of computation.

3.2. DATA AND FINDINGS

Out of the 30 sampled programs, 17 contained one or more compositional errors – about 57%. Four consistent error types were identified: *1. Explicit looping* - a condition where the programmer writes a series of explicit statements with minor variations instead of utilizing a looping construct; *2. Repeat conditionals:* conditional statements which could be combined or nested to better control program flow, but are instead executed in sequence; *3. Redundant looping:* sections of code that contain the same loop yet are repeated several times, instead of being constructed as a reusable function (or, in later lessons, as an object method); *4. State confusion:* failure to properly keep track of system state. For instance, reinitializing variables, repeatedly changing program flags that only need to be set one time, and resetting object attributes (e.g. fill colour) when they were already properly established within the code.

In the sample, there was little difference in the frequency of the error types. The most common error was the use of repeat conditionals, which accounted for seven of the programs with errors (41.1%). Three of the programs with errors had two or more errors.

Table 1. Program Composition – students w/ errors

	E. LOOP	R. COND'L	R. LOOP	STATE
Student 1	●	●		
Student 2	●	●		
Student 3			●	
Student 4		●		
Student 5				●
Student 6	●			
Student 7				●
Student 8	●			
Student 9				●
Student 10		●		●
Student 11			●	
Student 12		●		
Student 13		●		
Student 14			●	
Student 15			●	
Student 16				●
Student 17		●		

Errors, left to right columns:
Explicit looping, Repeat conditionals, Redundant looping, State confusion

Despite their training and the instructor's prompt, nearly 3 in 5 students from the study submitted programs with errors[1]. While their programs worked, the students with errors seemed to be missing the point of designing computationally, producing code that lacked efficiency, flexibility, and clarity. The students either did not understand that their programs were incorrect or could not do anything about it (impossible to tell from the code alone). Both possibilities suggest challenges for teaching programming to architects.

4. Concept test

4.1. METHODOLOGY

A second study measured student comprehension and knowledge of programming. This group of 72 students (with a similar proportion of graduates to undergraduates as the previous study) took a different version of the course from the group in the first study. In response to student feedback, the author made revisions to cover similar material using the Grasshopper visual scripting language. Additionally, the course was changed to use a "flipped classroom" format utilizing video tutorials to train skills before labs and peer programming to complete homework in the labs (see Senske, 2013a). The class lecture format remained the same as earlier.

The course midterm tested computational concepts using examples and methods the students learned from Grasshopper. For instance, a question might ask a student to name the three primary geometric transformations (answer: move, rotate, and scale). The material on the test was derived from units in the course and covered parametric variables (constraints, independent and dependent variables), geometric distribution, iterative patterns, topology, and debugging. Questions were formatted in one of three ways: visual identifications, multiple choice, and short answers. The test was administered to each lab section and was open-note and open-book. Students also had access to the course website and Grasshopper during the test. A research assistant compiled the results and the author verified them.

In 2013, a second group of students took the concepts test, but with a change in pre-test preparation. A colleague suggested the format of the first quiz might have affected the results because it was different from the way students were used to thinking about the subject. To account for this difference, the author showed the students examples of the question format a week before the test. In the second test, the format of the questions remained the same as the first version, but the answers changed.

4.2. DATA AND FINDINGS

The collected data is listed in the Table 2 below. The average score on the comprehension test was 72.9% correct. Roughly a third (34%) of the students in both classes did not receive a satisfactory (passing) grade on the test.

Table 2. Comprehension test – 2012 class vs. 2013 class

	V. VAR	C. VAR	DIST	PAT	S. TOP	M. TOP	S. DEB	V. DEB	TOT
2012	74.1%	94.0%	48.6%	85.4%	48.3%	74.9%	83.2%	49.1%	72.7%
2013	94.0%	96.3%	48.2%	69.4%	40.5%	67.2%	68.4%	68.6%	73.0%
AVE.	84.1%	95.2%	48.4%	77.4%	44.4%	71.1%	75.8%	58.9%	72.9%

Concepts, from left to right columns:
Variables (visual ID), Variable concepts, distribution patterns, Topology (mult. choice), Topology (short answer), debugging (short answer), debugging (visual ID), total score

Another interesting finding is that an increase in preparation did not seem to have a significant effect on the overall score. In fact, the difference in average scores between the two groups was less than 1%. However, there were greater differences in individual categories. Of the second group's scores, performance on questions about variables – already fairly high at an average of 82.6% correct – improved, as did visual debugging. In other categories, performance actually decreased an average of 11.5%. Whether this was due

to overconfidence (because the students were told what kinds of questions to expect) or some other factor is unknown. More notable are the low average scores for geometric distribution patterns (48.4% correct) and short answer topology (44.4%) questions. Furthermore, while performance on visual debugging questions improved with preparation, it remained another low-scoring category (58.9%). Although students had access to both their notes and the software during the test, they were unable to correctly answer the prompts. Overall, their performance suggests that they did not have a firm grasp of basic computational concepts.

5. Discussion

The results of the two pilot studies imply that, similar to the McCracken et al. and Lister et al. studies, architecture students in an introductory course perform poorly with program-writing (composition) and conceptual / code-reading (comprehension) tasks. Regardless of one's professional orientation, a majority of novices appear to be challenged by learning to program.

Many of the specific problems found in the architecture student studies are familiar to CSEd researchers. For instance, in Papert's (1981) study of children writing LOGO programs, he observed many students writing explicit statements instead of iterative loops. du Boulay (1986) and Pea (1984) recorded this naïve programming behaviour in groups of adults as well as children. Other studies noted how novices struggle to understand program flow with loops and conditionals (Soloway et al., 1982; Brooks, 1990; Shackelford et al., 1993). Sleeman et al. (1986) investigated the many ways students have trouble determining the state of variables. Descriptions of the four types of errors from the program analysis can all be found in the above studies.

Some of the poor performance on the comprehension test can be attributed to students learning the commands of the language, which is often a problem (though not a lasting one) for novices (Soloway, 1986). A reason why many students continue to have problems understanding programs is that they learn commands and syntax in rote patterns, but do not comprehend the conditions under which they apply (Brooks, 1990). This could be a reason why, when asked questions about topological inputs/outputs and distribution patterns, most of the students answered incorrectly. The students likely memorized the implementation pattern – which gave them results in Rhino – without knowing how the pattern worked. Pea (1984) famously referred to this as behaviour as "production without comprehension." Still, if this was the case, it is interesting that so few of them could not manifest the patterns

during the test using Grasshopper and work backwards to observe the conditions. This would seem to be a skill that could be taught.

The visual debugging activity on the test also gave many students trouble, with only 58.9% of students answering correctly on average. For this particular question, students were shown screenshots of a malfunctioning program along with images of the script that created it. The students were asked to define the error and explain how to correct it. They did not have to write any code; only explain the solution. The bugs should have been familiar as they were selected out of common occurrences in student programs and explicitly discussed in class. While a majority of the students (85%) correctly identified the bug, only half of those students (on average) could explain the cause. Debugging is known to be another challenging aspect of programming (Spohrer and Soloway, 1986; Winslow, 1996), but this particular instance did not ask students to fix the program. If the students were told to make the program work in Grasshopper, they might have succeeded through novice "hacking" behaviours (Leron, 1985; Pea, 1984) – essentially guessing with combinations – and might not have needed to know why the program worked. However, when asked to explain their solution, they could not. Throughout the test, the students presented evidence that many of them did not seem to comprehend what was actually happening within their code.

The findings of the architecture study seem to agree with the general findings of the computer science studies, but more than this, the CSEd research seemed to shed light on the particular problems of the architecture students. In this sense, it would appear to be relevant.

This discussion introduced only a small sampling of the CSEd research already available to those who might apply it to computational design instruction. Without reviewing it all, the point should be clear: the problems architecture students have learning to program are not new (the same problems existed nearly 30 years ago!), nor are they specific to the particular software or language architects may use. Rather, they are something that many professions have in common. If more architectural educators were familiar with these problems, and addressed them while teaching how to design with computational tools, student performance might improve. In the future, perhaps our field can learn from and contribute to solutions from other computational professions, as well.

6. Conclusion

The pilot study found that architecture student performance was poor in ways similar to those revealed in the computer science education research. A majority of architecture students in the study demonstrated problems with

program comprehension and composition. The implication of the study is that developing computing aptitude appears to be a universal issue; computational expression is difficult for many people. This research seems to indicate that whether architecture students are self-taught or taught explicitly, there is much work to be done in helping them learn how to master computation. For instance, care must be taken to ensure that students are properly assessed, not just on their design outcomes but on how well they understand concepts like data structures, program composition, and logic. It may take more time to develop this understanding, so starting earlier in the curriculum and continuing to review computation in later courses may help. In addition, if traditional tutorial methods do not result in computational thinking, then instructors may need to develop alternative ways of teaching programming. And so, while some architects may have a talent or drive that enables them to learn computational design thinking on their own, architectural educators should not mistake the performance of an exceptional few as the norm. The evidence suggests that teaching and learning computation is a challenging educational problem worthy of further research. If all architects are to learn computational design, researchers and educators must move beyond folk pedagogy and anecdotes. Empirical research – from architecture as well as other fields – must play a more substantial role in helping architects learn computational design thinking.

Endnotes

1. See (Guzdial, 2007) for a compact overview of the field.
2. A few details about the program analysis study may have influenced the findings. First, the students were not all asked to write the same program. Thus, the task was one of composition and design, and not a straightforward test of programming ability. Second, the students self-reported what they intended to create and decided what to submit for the project. It may be that other students had similar misconceptions, but removed parts of their code that did not work or they did not understand. In addition, school culture and peer expectations may have affected the choices students made with their assignments. These details do not disqualify the findings, but they must be taken into account. Most architecture students learn and use programming in a design context, so examining their work this way, although it lacks many experimental controls, may have more authenticity.

References

Brooks, R., 1990: Categories of programming knowledge and their application, *International Journal of Man-Machine Studies* **33**(3): 241-246.

Burry, M., 2011: *Scripting Cultures: Architectural design and programming*, Wiley.

Coates, P., 2010: *Programming.Architecture*, Routledge.

du Boulay, J. B. H., 1986: Some difficulties of learning to program, *Journal of Educational Computing Research* **2**(1): 57-73.

Guzdial, M., 2007: CS Education -- Guzdial's Blog, October 5 – November 7, 2007., retrieved August 2, 2013, from *http://home.cc.gatech.edu/csl/uploads/6/Guzdial-blog-pieces-on-what-is-CSEd.pdf*.

Jabi, W., 2013. *Parametric Design for Architecture*. London, Laurence King Publishing.

Kay, A., 1993: The Early History of Smalltalk, *ACM SIGPLAN Notices*, **28**(3): 69-95.

Karle, D. and Kelly, B., 2011: Parametric Thinking, in *Parametricism (SPC) ACADIA Regional 2011 Conference Proceedings*.

Leron, U., 1985: Logo Today: Vision and Reality, *Computing Teacher* **12**(5): 26-32.

Lister, R., E. S. Adams, et al., 2004: A multi-national study of reading and tracing skills in novice programmers. *Working group reports from ITiCSE on Innovation and technology in computer science education*. Leeds, United Kingdom, ACM**:** 119-150.

Mateas, M., 2005: Procedural Literacy: Educating the New Media Practitioner, *On The Horizon. Special Issue. Future of Games, Simulations and Interactive Media in Learning Contexts* **13**(1).

McCracken, M., V. Almstrum, et al., 2001: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *Working group reports from ITiCSE on Innovation and technology in computer science education*. Canterbury, UK, ACM**:** 125-180.

Menges, A. and Ahlquist, S. (Eds.), 2011: *Computational Design Thinking*, Wiley.

Mitchell, W. J., R. S. Liggett, et al., 1987. *The art of computer graphics programming*, Van Nostrand Reinhold co. New York.

Pea, R. D. and D. M. Kurkland, 1984: On the cognitive effects of learning computer programming, *New Ideas in Psychology* **2**: 137-168.

Reas, C. and B. Fry, 2006: *Processing* [computer software].

Rutten, D., 2010: *Grasshopper* [computer software], Seattle: McNeel.

Papert, S., 1980. *Mindstorms: children, computers, and powerful ideas.* Cambridge, Perseus Publishing.

Perlis, A. J., 1961: The Computer in the University. *Management and the Computer of the Future*, MIT Press.

Shackelford, R. L. and A. N. Badre, 1993: Why can't smart students solve simple programming problems?, *International Journal of Man-Machine Studies* **38**(6): 985-997.

Sheil, B. A., 1983: Coping With Complexity, *Information Technology & People* **1**(4), 295 - 320.

Senske, N. 2013a: Building a Computational Culture. *The Visibility of Research: Proceedings of the 2013 ARCC Spring Research Conference*. Charlotte, NC: 343-350.

Senske, Nicholas. 2013b. Rethinking the Computer Lab. *Proceedings of the 29th National Conference on the Beginning Design Student.* Philadelpha, PA: (in press)

Sleeman, D., R. T. Putnam, et al., 1986: Pascal and high school students: A study of errors, *Journal of Educational Computing Research* **2**(1): 5-23.

Soloway, E., 1986: Learning to program = learning to construct mechanisms and explanations, ACM. **29:** 850-858.

Soloway, E., K. Ehrlich, et al., 1982: What Do Novices Know About Programming? *Directions in human-computer interaction*. B. Shneiderman and A. Badre. Norwood, NJ, Ablex Publishing**:** 27–53.

Spohrer, J. C. and E. Soloway, 1986: Novice mistakes: are the folk wisdoms correct?, *Communications of the ACM* **29**(7): 624-632.

Terzidis, K., 2009: *Algorithms for visual design using the processing language*, Wiley.

Winslow, L. E., 1996: Programming Pedagogy -- A Psychological Overview, *SIGCSE Bulletin* **28**(3): 17-25.