Building a Computational Culture: A Pedagogical Study of a Computer Programming Requirement

Nick Senske

University of North Carolina at Charlotte, Charlotte, NC

ABSTRACT: As computational design becomes increasingly important to architectural practice, curricula must be updated to teach new outlooks and skills to the next generation of design students. Over the last two years, UNC Charlotte has tested a curriculum that emphasizes computational thinking and methods. The core of this curriculum is a required course that introduces over 70 students a year to the fundamental ideas of computation through exposure to programming in a design context. This paper describes our teaching methods and our findings from a study of the course, which includes attempts to measure student outcomes, attitudes about computing, and the application of computation after the course. The results of our study, which suggest an inclusive methodology and emphasize the cultural dimension of this pedagogical task, may help schools in the planning and implementation of their own courses that introduce computational design.

KEYWORDS: Pedagogy, Computational Design, Computational Thinking, Computer Programming

INTRODUCTION

Computational design is becoming increasingly important within the architectural discipline. In today's profession, CAD and 3D modeling alone are not enough to address the need for more economically and ecologically sustainable buildings. Computational methods such as parametric models, generative algorithms, simulations, and digital fabrication assist in the design and construction of buildings that are not only aesthetically innovative, but performative as well (Kalay, 2004). In order to stay competitive in a globalized market, architects will need to know not only how to use these tools, but how to integrate, modify, and write their own. Once an exceptional specialization, computational design will soon play an essential role within architectural practice and research.

This trend presents a challenge for architectural educators. How will schools teach their students these new ways of working? At the moment, the answers to this question are incomplete and unsatisfactory. First, what should be taught and where does it belong in the curriculum? How does computation fit within the subjects that architecture schools already must teach, especially when curricula keep growing in scope due to new requirements? Second, technology changes constantly. The software and techniques that students learn in their initial years could be insufficient or obsolete by the time they graduate from architecture school. What can schools teach about computation and design that will help students learn future tools?

Determining the pedagogy is another problem. Many architecture students are hesitant to learn computation because they think it will be difficult or too far removed from how they envision design (McCullough, 2006). Furthermore, there is evidence that computation is a difficult subject to teach and learn. Computer science, for example, has an attrition rate of almost 30% (Roumani, 2002) and a surprising number of graduates cannot design and write simple programs (McCracken, et al, 2001; Bennedsen and Casperson, 2007). Even accomplished computational designers admit there is a steep learning curve (Burry, 2012). Thinking in terms of explicit, procedural abstractions is hard, especially when, as Seymour Papert argues, there are no traditions of this within our own culture (Papert, 1980). In short, if the goal is to someday teach computation to all architecture students, more research is needed.

Towards this end, this paper describes the development of a required course in computational design and the results of a pilot study to assess its effectiveness.

1.0 Computational Thinking in the Curriculum

The School of Architecture at UNC Charlotte spent several years developing a curriculum that addresses computational design throughout the undergraduate and graduate sequences[1]. A core idea of the curriculum

is that our courses are not focused on developing fluency in particular kinds of software or tools, but rather *computational thinking*. We believe that if our students have the skills and mindset to use computation well, they will be able to adapt to changes in technology and possibly even participate in those changes through their own innovations.
Computational thinking is not a new idea. In 1961, Alan Perlis, an early pioneer in computer science, argued that computer programming should be a requirement of a liberal arts education (ibid.). His reasoning was that process plays a critical role in all fields, and the computer enables one to interactively create and run processes. Perlis believed that students should learn programming, not so they could run a computer, but because it is a means of learning about process.

Of course, learning to program does teach a great deal about how to operate computers. One of the most difficult parts of computation is translating one's thinking into the computer. To do this, a person not only has to know about their own field's processes, but also about the capabilities and limitations of computers, the commands, logic, and inputs/outputs of the software, and how to break down an idea into rules and steps for the computer to follow. Knowing about both human and computational processes endows a person with a powerful way of working and a means of understanding any computational device they may encounter (Sheil, 1983). This is the essence of computational thinking[2] – the ability to work well with computers.

Some may argue that students do not need to study programming to learn computational thinking, and, instead, there might be other ways to teach students how to think about process and approach computing. This may be true. However, two things must be made clear. First, programming does not necessarily mean writing code, so this is no reason to avoid it. Whenever a person authors instructions for a computer to follow, whether it is with visual programming (e.g. Grasshopper, Scratch, etc.), scripting, or by setting the controls in a simulation, etc., this is arguably a form of programming (Blackwell, 2002). What matters is the thinking involved, not the interface. Second, it makes sense to learn about process in terms of the dominant representational medium of the day, which is computing. Transfer of abstract ideas is difficult to achieve, so it is best to teach lessons that are as close as possible to the concrete objective (Perkins and Salomon, 1989). In a single semester, transfer of specific contextual ideas is much more likely (Palumbo, 1990). Therefore, if we want architecture students to learn computational design, why would we start to teach logic and process in anything other than the computer? While it may be possible to use other metaphors to learn about process, programming is most directly relevant.

The goal of the curriculum is to create what we call a "computational culture" in our school. By teaching computational thinking to all of our students, we hope to affect the values, attitudes, and beliefs they bring to their studies. We are not looking to create a school full of software programmers, but rather we want students to learn about computation early, so we can build upon it in other courses and have more critical conversations about its use in design. When everyone in our school learns computation, the students will see it as something they can participate in – not something than only computer people or other schools are able to do. In short, our objective is to make computational design as commonplace and as useful as 3D modeling has become. Not everyone has to make it his or her focus, but an architect needs to be able to understand its potential role in design and recognize when it applies. We believe one of the best ways to prepare our students for the future of design is to immerse them in a culture where computation is the norm and not the exception.

2.0 Teaching Methodology

The foundation course in our computational sequence is called *Computational Methods*. It is attended by our third year undergraduates as well as graduate students (in either their first or second year, depending upon whether they are in a post-professional or pre-professional program, respectively). The total class size is over 70 students. To the best of our knowledge, it is the only required course in computation that is taught this early in a professional architectural curriculum. It was taught for the first time in fall 2011. A second iteration followed in the fall of 2012.

The primary objective of the course is to teach students the fundamental concepts of computation. The secondary objective is for students to learn specific methods for applying these ideas in a design context. Our vehicle for this is computer programming. To be clear, we do not propose that learning programming necessarily makes one a better designer. Rather, it can assist designers if used properly. The aim of the course is not to train great programmers but to instill a greater awareness of computation in architecture as well as other fields, so that students can approach it critically – whether they choose to use it or not.

2.1 Active labs

The class meets twice a week, first with a lab session and then with a follow-up lecture. This might seem backwards, as it would be more common to use the lecture as background to set up the concepts for the lab,

but the arrangement is deliberate. Educational research suggests that students learn best when they have the opportunity to work with ideas in a concrete form first, and in the abstract later (Bransford et al, 2000). Before the labs, students learn commands and procedures through online videos we produce. This kind of information is explicit and does not require much student / teacher interaction. Once the students learn the basic software technique, we use our labs to introduce the underlying computational concepts in a hands-on way, by giving our students situations where their existing knowledge must be extended to solve a problem[3]. This is sometimes referred to as inquiry- or problem-based learning (Gallagar, 1997). For example, from a previous lesson, students know how to loft a single list of parametrically generated curves, but in the following lab, they would need to learn about data structures in order to create several lofts from multiple lists of curves – i.e. lofting a list of lists. Through a guided series of questions, students work together to discover and teach each other new concepts. This pedagogy is different from most computing tutorials, where the instructor merely tells students the concepts.

Our lab pedagogy is a marked contrast from the typical computing lab, where students follow along with passive tutorials that are dependent upon the instructor and the speed of the lesson must accommodate the slowest learner. In our "active" labs, the instructor is there to answer questions and coach, but the students work at their own pace. Later in the week, the lecture abstracts the lessons, reinforcing and clarifying the concepts and placing them within a historical and architectural context through the use of examples and precedents. It is not yet clear whether this format results in better learning, but it seems to improve engagement. In their course evaluations, many students cited the role of the labs in overcoming their apprehension about programming.

2.2 Assignments

In general, there is not much design in the course – or rather, the design exercises are highly controlled. The first version of the course taught us that it was difficult for our students to both learn about computation and to produce meaningful design artifacts at the same time. The cognitive load was much too great. The active labs are part of our effort to be clearer and more focused about what we ask students to do, while at the same time challenging them to solve problems creatively. To practice their skills, the students produce weekly lab reports, where they write their solutions to the lab problems and describe their thought processes. The report prompts are designed this way so students have experience externalizing their thoughts and communicating with others about computational processes.

Additionally, we have two major projects that introduce limited design into the course. The first is a precedent study, where students extract a parametric assembly from an existing project and apply it in a new context. This assignment works well because students do not have to invent the assembly; they only need to decipher and implement it. The second project is their final project, where they must take a design of theirs (either current or previous) and use computation to iteratively study some aspect of it in a data-driven way. Using a design of their own allows them to jump into the project with a program, site, and a building that are familiar. This way, they can focus on using computation to explore or revisit problems. As a final exercise, it is motivating and relevant, and a good way to review the lessons from the course.

2.3 Technology

One of the pedagogical principles of *Computational Methods* is that learning how to think is more important than learning tools. This being said, the first tool that students are exposed to is important in providing motivation and forming their earliest opinions about computation. For this reason, the most recent version (fall 2012) of the course is taught exclusively in Grasshopper, which is a visual scripting language for Rhinoceros (McNeel, 2012). Earlier, we taught the course using both Processing (a scripting language for Java) (Reas and Fry, 2011) and Grasshopper, but the students did not find Processing as relevant. The benefit of Grasshopper is that it builds upon the Rhinoceros skills they learn in their second year and the output is expressed as Rhinoceros models, which they can directly apply to studio projects. The disadvantage is that, with Grasshopper, it is not possible to talk explicitly about concepts such as looping constructs, subroutines, and object oriented programming, which are important computational ideas. The reception for the Grasshopper lessons has been overwhelmingly positive, however. In fact, after taking the course, several of our students explicitly asked to learn Processing. *They want to learn to code.* This is a positive outcome for an introductory course and further evidence that the right tools can be motivating.

2.4 Course Topics

The following is a rough outline of our syllabus, to provide some idea of the topics we cover and the manner in which we approach them:

- Week 1 - Parametrics: Introduction to the strengths of computation and how computation is applied in architecture. Introduction to basic parametric constructs in Grasshopper.

- Week 2 - Variables: Covers more of the Grasshopper interface, simple geometric components, constants, controlling variables, setting up relationships (dependencies), and basic transforms (i.e. move, rotate, and scale). The first part of the lab covers object positioning as it relates to parametric dimensions. The second part examines proportional transformations (related, dependent variables) using a parametric column form.

- Week 3 - Repetition and Loops: Covers the repetition of data in Grasshopper (i.e. creating lists) with the Series and Range components. Also, one- and two-dimensional systems for positioning and transformations, and the use of the Graph Mapper for visual generation of numerical data patterns. Data structures for looping (lists of lists) are briefly introduced. In the lab, we create: a parametric "tower" with repeated transformations; numerical patterns (e.g. sine waves) from equations with Graph Editor; "Spirograph" demonstration with stacked repeated transformations.

- Week 4 - Distributions, References, and Cull: The idea of "distribution" combines the notion of variables (e.g. coordinate locations) with repetition to create dependent parametric constructs. We implement distributions of parametric objects (e.g. points along a curve) and referenced (locally-oriented) geometry over curves and surfaces. We also study the basic Cull (remove / delete) components to create rhythmic patterns in repetitive constructs.

- Week 5 - Parametric Design Process: Begins with a thorough review of 3D modeling topology - points, curves, surfaces - and then describes the thought processes and methods used in the creation of a parametric masonry wall. Topics include: script design, task decomposition, and iterative script refinement.

- Week 6 - Procedural Diagrams: Our midterm lessons describe a method for creating 2D diagrams to visually explain how a script generates a form. The underlying premise of this exercise is that showing someone code does not provide a useful explanation of how a parametric system works. By learning to create informative process diagrams, we can better communicate our design intent.

- Week 7 - Debugging: We explicitly discuss and practice strategies for problem solving and debugging programs. Debugging is an essential part of the programming / computational design process, but is seldom covered in much detail.

- Week 8-9 - Adaptive Parametric Patterns: This two week review unit covers some general and advanced algorithmic methods for adapting parametric patterns to different contexts. Specifically, we explore: referencing surfaces and matching curves to surface forms; review of surface aligned frames; using frames to create local coordinate systems for shifting and transforming points; the Box Morph object for adapting arbitrary geometry.

- Week 10 – Algorithms: This week focuses on algorithm development: the design of reusable logical constructs for architectural projects. As a demonstration of these principles, the lesson provides an overview of attractor concepts (an algorithm that references distances within a parametric system) and different applications of attractor systems in design and architecture.

- Week 11 - Data Structures and Manipulation: This series of exercises covers basic list operations: list length, extracting indices, accessing list items, removing list items, shifting lists, combining lists, etc. Building upon these lessons, we generate structural framing patterns that deal with points, lines, and surfaces derived from subdivided surfaces. These patterns make use of several kinds of data access and filtering, and review earlier lessons in topology.

- Week 12 - Metric-Based Design Components: This week integrates with lessons from our daylighting and building systems courses. It covers several basic algorithms that use sun angle as a driving parameter: variable louvers, warped louvers, apertures / openings, light canons, light shelves.

- Week 13 - Conditional Logic: Introduces the basic concepts and components involved in conditional (rule-based) logic. We cover the use of the Dispatch component with the Larger / Smaller, Equals, and Modulo components. Examples include: 2D containment patterns, filtering geometry based upon coordinates and height, and 3D intersections as a basis for subtractive and additive modeling.

- Week 14 - Randomness and Generative Design: Our final week surveys some basic algorithms for randomness in architectural contexts: erosion, "Pick and Choose", "jitter", etc. We follow this with a brief discussion on the concept of randomness and order in art and design. Finally, we talk about Processing and carrying the ideas of the course beyond Grasshopper and into generative design.

The order of topics – especially our choice of final topics – is not what you might find in a typical introductory computer science course or textbook. We did this deliberately to make sure that students were exposed to ideas in their most explicit form before we added more layers of abstraction and automation.

The course is not intended to be comprehensive, in terms of either computation or Grasshopper. Rather, it is meant to give students strong background they can immediately apply to their work in other classes. As the lessons increase in complexity, we revisit ideas in several different contexts to create depth. We believe this is the best way to encourage students to retain the material in the long term.

3.0 Assessment

To study the effectiveness of the course, we collected data through an online post-class survey. The survey was voluntary and included multiple choice questions and the opportunity to provide short written answers. 53 out of 72 students participated in the survey. University course evaluations provided additional written feedback for the study.

Our first concern was whether students felt the material applied to them, as programming is not a skill many architects feel they can or need to master. Measuring this is important because educational research suggests that if students, particularly older ones, do not see the application of what they are learning right away, it can be de-motivating (Pugh and Bergin, 2006; Lepper, 1985). We seem to have succeeded at capturing their interest. In response to the question: “How relevant is computation to your future career?” 92.4% of students said it was in some way relevant, 33% said it was extremely relevant, and 81% of students agreed that the course should remain a requirement.

The course was also well received. 90% of students agreed or strongly agreed with the statement: “Are you satisfied with your experience in Computational Methods?” and 77% said they learned more than they expected. Reading the feedback from students, we have reason to believe that we have taken an intimidating subject and made it approachable:

> "I feel that I have learned more than I could have anticipated. I was very lost and confused at first, but slowly things started to make since and I truly felt accomplished about my work. I think that the course gives a great foundation understanding about how computer designing works."
>
> "This course was not quite what I expected, but I was pleasantly surprised. Learning about the logic of computation is incredibly important and relevant today."

A good indicator of student engagement is whether students are motivated to keep learning about a subject after completing the course. In the post-class survey, 70% percent said they would consider taking an advanced version of Computational Methods. This is a positive outcome for an introductory course.

Another goal of the study is determining how much students understand about computation after a semester in *Computational Methods*. Are students really learning computational thinking and does this affect how they approach design? Unfortunately, we found this difficult to determine from the assignments and projects we collected. Our students can write programs, but we do not have the assessment tools yet to examine *how* they write programs. They learn to document and present their process in the course, but what they produce does not tell the whole story (or even the whole truth). The student evaluations offer more insight:

> I was skeptical about digital design when I started this course (other than using architecture-minded software like Revit, for example). However, now that I've seen how parametrics actually works [sic], learned a particular scripting software, and have seen examples in class of some inspired, digitally designed architecture, I do believe that digital design can result not just in 'blobs' but in some really beautiful, metric-based designs. I still don't quite know how and when to apply this technique to my own studio designs, but I think it's a good thing.
>
> I leave the class definitely thinking about new ways to approach a design. I understand that the process is not exactly as linear as it once was, as rules and decisions can be made before hand as you get to build a system before a project. Even within a design project, I already find little moments where Grasshopper would programmatically accelerate the design.

> I think that I learned a lot in this class, mainly because I can begin to apply it outside of just comp methods. This has taught me a new way of thinking about things and the design process and now I can begin to see how this can be incorporated in studio.

Judging from our evaluations, the course appears to be successful at teaching many of our students to see computation as another way of thinking about design. We are particularly encouraged by the statements about process in the comments that many of them wrote. Still, we do not know with certainty how many students think this way and how sophisticated their thinking is. For example, our students can use computation to solve the small problems we give them in class, but when they want to design something for studio, many of them have trouble breaking down their idea into steps they can turn into a program. Our students also report that it is difficult for them to come up with their original ideas for using computation or that they do not always recognize when they could use it in a project. In future versions of the course, we hope to address these problems.

While we seem to have raised their awareness of computation and made them inclined to apply it, it is also unclear whether the knowledge and skills we teach students will transfer to other subjects they study. Transfer to new contexts is one of the most important measures of learning (Thorndike and Woodworth, 1901; Singley and Anderson, 1989). We are only beginning to measure this rigorously. However, we can report some initial findings. At the end of the course, when asked whether they thought their experience would help them learn Revit or other computational tools, 98% percent of our students said they believed it would. This may turn out to be true. In our advanced computing courses such as digital fabrication and our BIM seminar, the faculty report that students who have taken Computational Methods tend to learn new material faster and perform better on assignments, because they have a firm grasp of computational fundamentals. Eventually, the instructors believe that they will be able to teach more in their courses, because they do not need to introduce so many computing concepts themselves. Granted, this is only anecdotal support. At the end of this semester, we plan to track the outcomes of our Computational Methods students in later courses to determine if there is any measurable difference.

4.0 Reflection and future plans

Our two year experiment with Computational Methods has taught us several lessons. First, it is essential to consider student attitudes and expectations of the subject. In order to change the design culture, we must respect the existing culture. It is not enough to make the course required and expose everyone to programming. Students come to the course without much exposure to computation and apprehensive about their prospects of learning it[4]. With this in mind, we designed the lessons to make the material relevant and to connect to ideas, software, and precedents that students already know. In addition, labs were made into active social experiences where students were encouraged to work and solve problems together. As a result, our surveys showed that students were motivated to learn the subject and finished the course with a different mindset about computation.

Second, we learned that architecture students can learn to write programs, but this does not necessarily mean they can apply them in a design context. While they recognize commands and can exploit programming patterns we teach them, students still have difficulty with program flow and design. This should be expected, as these are common problems even for computer science students (Soloway et al., 1982; Eckerdal et al., 2006; Linn, 1985). Still, it makes our students less effective at using computation in their studio projects. We also want them to be able to intelligently choose where to apply computation (instilling a sense of “computational ethics”, as it were), and it seems that they are still unclear about this. They do not learn these ideas implicitly, even though we show them examples of “good” uses of computation and critique their work on its merits. It might be necessary to devote more course time to explicitly teaching these ideas, or to design assignments that scaffold these kinds of choices for students.

Third, we may need to adjust our expectations regarding our students’ learning outcomes. Computation is a challenging subject. As Alan Kay reflected in his research with children learning programming, developing thinking that goes beyond the superficial characteristics of the language often takes time (Kay, 1993). Better tools and pedagogy can help, but even with the best of both, learning to design computationally will probably take more than a semester. We can introduce students to a different way of thinking and teach them the language of computation – which we believe is worthwhile, pedagogically – but their ability to express themselves in this language is low, at least initially. This makes sense, if we can analogize learning programming to the acquisition of a foreign language. Fluency is not expected in a single semester. This understanding has possible repercussions for our teaching. To help achieve the depth of thinking we want our students to possess, we may need a second required computation class that focuses more on design and process, or perhaps a studio where all of the students apply computation.

Our evaluation of the course is iterative and ongoing. We recognize that, at this time, we cannot quantifiably measure whether the course is meeting all of its goals. This is because we have not performed the some measurements at the time of this writing and also because, in hindsight, some of our assessment tools were not capturing the right data. Our plan is to perform a follow-up assessment of the course, to track how well students have retained what they studied, to study the differences in later classes between students who have taken Computational Methods and who have not, and to record how often computational design is applied in later studio work. To help answer the question of whether students are learning computational thinking, we are working with a cognitive scientist to develop a procedural assessment tool for future versions of the course.

5.0 Conclusion

The School of Architecture at UNC Charlotte believes that computational design skills will someday become a standard part of an architectural education. Computational Methods is our attempt to begin this transformation at our own institution. By learning about programming early in their education, students are exposed to computational thinking, which may help them learn software and other tools in the future. As an introduction to the subject that motivates students to continue learning, the course appears to be fulfilling its purpose.

The impact of Computational Methods extends beyond the two cohorts of students who have taken the course. The other students and faculty are seeing and hearing more about computational design, and this is strengthening the impact of our new curriculum. The school is more aware of computation, its potential, and its accessibility. They see that programming is not only for certain professions or the math-savvy, but something that anyone can do. What was once an esoteric subject is on the way to becoming normative – another way for students to think about and pursue design. It will take time and further study, but we are making progress towards creating a computational culture.

REFERENCES

Bennedsen, Jens, and Michael E. Caspersen. "Failure rates in introductory programming." ACM SIGCSE Bulletin 39, no. 2 (2007): 32-36.

Blackwell, Alan F. 2002. What is Programming? In 14th Workshop of the Psychology of Programming Interest Group. Brunel University.

Bransford, J.D., A.L. Brown, and R.R. Cocking. 2000. *How people learn*: National Academy Press Washington, DC.

Burry, Mark. 2011. Scripting cultures: Architectural design and programming: John Wiley & Sons.

Eckerdal, Anna, Robert McCartney, Jan Erik Mostr, Mark Ratcliffe, and Carol Zander. 2006. Can graduating students design software systems? In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*. Houston, Texas, USA: ACM.

Gallagher, S. A. 1997. Problem-based learning. *Journal for the Education of the Gifted*, 20(4), 332-362.

Guzdial, Mark. 2008. Education: Paving the way for computational thinking. Communications of the ACM 51 (8):25-27.

Kalay, Yehuda E. 2004. Architecture's New Media. Cambridge: MIT Press.

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

Lepper, Mark R. 1985. Microcomputers in education: Motivational and social issues. *American psychologist* 40 (1):1-18.

Linn, M.C. 1985. The cognitive consequences of programming instruction in classrooms. *Educational Researcher* 14 (5):14.

McCracken, Michael, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin* 33 (4):125-180.

McCullough, Malcolm. 2006. 20 Years of Scripted Space. *Architectural Design* 76 (4):12-15.

McNeel, R. 2012. Rhinoceros (version 4.0 R9) [software].

Palumbo, David B. 1990. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research* 60 (1): 65-89.

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

Perlis, A.J. 1961. The Computer in the University. Paper read at Management and the Computer of the Future.

Perkins, D. N., and Gavriel Salomon. 1989. Are Cognitive Skills Context-Bound? *Educational Researcher* 18 (1):16-25.
Pugh, K.J., and D.A. Bergin. 2006. Motivational influences on transfer. *Educational Psychologist* 41 (3):147-160.
Reas, C. and Fry, B. 2011. Processing (version 1.5) [software].
Roumani, Hamzeh. "Design guidelines for the lab component of objects-first CS1." In *ACM SIGCSE Bulletin*, vol. 34, no. 1, pp. 222-226. ACM, 2002.
Senske, Nicholas. 2011. A Curriculum for Integrating Computational Thinking. In *Parametricism: ACADIA Regional Conference Proceedings*. Lincoln, NE.
Senske, Nicholas. 2013. Rethinking the Computer Lab: an inquiry-based methodology for teaching computational skills. In *Proceedings of the 29th National Conference on the Beginning Design Student.* Philadelpha, PA: Temple University. (forthcoming)
Sheil, B.A. 1983. Coping With Complexity. *Information Technology & People* 1 (4):295 - 320.
Singley, M., and J.R. Anderson. 1989. Transfer of Cognitive Skill. Cambridge, MA: Harvard University Press.
Soloway, Elliot, Kate Ehrlich, and Jeffrey Bonar. 1982. Tapping into tacit programming knowledge. In *Proceedings of the 1982 conference on Human factors in computing systems*. Gaithersburg, Maryland, United States: ACM.
Thorndike, E.L., and R.S. Woodworth. 1901. The influence of improvement in one mental function upon the efficiency of other functions. *Psychological Review* 8 (4):384.
Wing, Jeannette M. 2006. Computational Thinking. *Communications of the ACM* 49 (3):33-36.

ENDNOTES

[1] For those interested, a more detailed description this curriculum can be found in (Senske, 2011).
[2] Perlis did not coin the term computational thinking. It is not clear who did. However, in contemporary usage this is the phrase used to describe the educational ideas he proposed. See (Matteas, 2005), (Wing, 2006), and (Guzdial, 2008) for more recent discussions of the need for computational thinking in schools.
[3] For more information on our lab pedagogy, refer to (Senske, 2013).
[4] In our pre-class survey, 87.7% of students reported no previous introduction to computation. Of these, 47% expressed concern with their possible performance in the course.