LEVERAGING HISTORICAL TIES BETWEEN COGNITIVE SCIENCE AND COMPUTER SCIENCE TO GUIDE PROGRAMMING EDUCATION

Darren K. Maczka, Jacob R. Grohs
Engineering Education
Virginia Tech

Coding for Everyone

In the past few years, there has been increasing interest in encouraging more people, regardless of background, to learn to program. In fact, President Obama recently made a statement calling on all children to have the opportunity to learn about computer science [1]. Sites such as code.org promote CS education opportunities for all, citing statistics about STEM jobs and arguing that in the 21st century, knowledge about computer science is foundational [2]. Many institutions of higher education either have programming requirements for non-computer-science majors, or have been expanding programming to non-majors [3,4,5]. At our institution, all first year general engineering students are required to complete an introductory programming module as part of their first-year engineering course, and in general "basic programming skills" is a common desired outcome across engineering curricula [6].

This coding-for-all paradigm juxtaposed with the observation that technology is changing rapidly, and the half-life of technical skills is decreasing [7], might seem to indicate that teaching programming is a questionable use of class time: the language and paradigm students are taught will likely not be the same they are asked to use in their first job post-graduation. However, if we acknowledge that learning to program is more than simply learning to write code, and in particular, if we draw on work from the early era of computing which closely linked programming to problem solving and cognitive skills, we can argue that regardless of language or paradigm, learning to program may actually be a way to learn to problem solve, organize knowledge, and reason about processes.

At the dawn of the computing revolution, visionaries predicted that computers had the potential to "augment the human intellect" [8], a much more expansive view than simply becoming tools to automate tedious calculations. In other words, computing allows for new ways of dealing with complexity, and provides new metaphors for thinking about systems [9]. Early work demonstrated that with the aid of computers, children could engage with tasks and concepts that had previously been considered "advanced", and through interacting with computers, children could become more aware of their own cognitive processes [10]. Thus, the fact that specific technologies may be obsolete by the time students graduate may be beside the point, there are lasting cognitive benefits that learning to interact with a computer can provide.

While there are some recent studies which investigate programming education as a means of teaching problem-solving, there is also a large body of research that primarily focuses on teaching students the details of how to code. We argue that in most cases these details are less important than centering programming education around problem-solving skills. If we make learning problem-solving skills a goal of introductory programming courses, then all aspects of the course, from the course objectives, to the curriculum used, to the assessments administered, must be chosen intentionally with this goal in mind.

In this paper we will present a brief summary of current published work on introductory programming education. In particular, we will explore the alignment of course objectives, pedagogical strategies, and assessments, and will offer a potential framework to help think about both the design and evaluation of introductory programming courses. We will then provide a hypothetical case study of an introductory programming course organized around this framework and discuss implications of different instructional strategies through the lens of a theory of cognitive processes called cognitive load theory.

Methods

Assuming that there would be more comprehensive research of introductory programming in the computer science community, we began with targeted searches of the ACM digital library for the key phrase ‘introductory programming’. Once we found key papers that seemed to indicate a call to action [11,12] or a survey of practices [13,14] we used the ‘cited by’ list to find additional relevant papers.

Because the literature search of introductory programming resulted in a variety of examples of how and why programming was taught, we needed a documented framework to consistently evaluate each one and guide our hypothetical course design. Starting from Jamieson and Lohmann’s (2009) call to ground educational research activities in how people learn, we identified the concept of *constructive alignment* as a promising framework to evaluate and design learning environments. Following the references in Jamieson and Lohmann’s report as well as conducting targeted literature searches for “constructive alignment” we were able to find descriptions of this framework [15,16,17,18].

Current State of Research

The bulk of literature about teaching introductory programming (CS1) revolves around what paradigms are “best” (e.g. procedural vs. object oriented [19,20,21]), which language is “best”, often times with regard to a particular paradigm (e.g. object oriented programming (OOP) with Python vs. Java [22,23]), and best teaching practices (e.g. use of “pair programming” environments [24,25,26,27,28]). However, the *reasons* for picking one paradigm over another tend to be about industry demand, rather than intentionally choosing a paradigm to support the learning objectives of the course. This behavior, adopting a popular practice into a learning environment without critically analyzing whether or not it is a good fit, is by no means exclusive to programming education. For example, flipped classrooms [29], in which students are asked to watch recorded lectures at home and use class time for working on activities, has recently been a hot topic in higher education and there have been a continuous number of examples of well-intentioned practitioners flipping their classroom without adjusting course objectives, content, and activities to work under the flipped-classroom model [30]. Likewise, anyone can teach elements of object oriented programming in their introductory programming courses, but unless critical aspects of the course are adjusted: background reading, rethinking what the “fundamental concepts” are, the types of problems assigned, and methods of assessment, the incorporation of OOP will likely create confusion for students and frustration for instructors [12].

Trends in Programming Education Research

A 2003 study by Robins, Rountree, and Rountree identifies four trends in research of programming education: 1) distinguishing between novice and expert programmers with emphasis on deficiencies of novices, 2) the distinction between knowledge and strategies, 3) the distinction between program comprehension and generation, and 4) a comparison of OOP to procedural styles of programming [14]. They found that the biggest challenge for novice programmers was not in understanding the basic concepts, but knowing how to apply them. This finding is consistent with current understanding of how we learn [31,32], and with studies of expert/novice behavior in other domains, such as statics [33]. In both cases the suggestion is that helping students learn strategies that work is crucial to helping them become *effective* novices, i.e. devote instructional time to problem-solving strategies and how to apply them.

A 2001 study by McCracken et al. [11], which concluded that regardless of nation or institution, most students do not know how to program by the end of their introductory programming courses, sparked much conversation and a push to reform introductory programming in the next decade. Like the previous example of adopting novel pedagogical practices without adjusting other aspects of the course, this phenomenon is also not exclusive to introductory programming: disconnect between what we think we are teaching and what students learn has been observed across engineering [34,35]. Possible ways of addressing this disconnect include recognizing differences in students' prior knowledge, helping students see connections between abstract concepts, and relating difficult concepts to ones the students already know [34,36].

While the McCracken et al. study validated the experiences of many introductory programming instructors, it did not result in critical change to address the problems. Rather, what we have seen since the 1990s has been an increasing interest in where object oriented programming should be placed within the curriculum with debates between objects-first vs procedural-first, but little question of why object oriented design should be included at all other than that it continues to be a popular paradigm in industry.

There remains a desire that introductory programming courses should teach problem-solving skills, though only a few authors present evidence that the implementation of this goal is informed by cognitive science [37,38,39]. One thread in particular has received regular attention: East et al. propose a pattern-based approach that places emphasis on recognizing and learning to work with algorithmic patterns over syntax and design paradigms [40]. This approach closely aligns with understanding of differences in expert/novice behavior by helping students identify patterns of solutions and learning how and when to apply them to particular problems [38,40,41,42].

Pears et al. conclude their 2007 literature survey with the observation that despite the volume of work exploring the implementation of introductory programming courses, "there is little systematic evidence to support any particular approach" [13]. We think this could be due to two reasons:

1. There has been a lack of consensus on what the goals of an introductory course should be: the ability to write a complete, working program, the ability to answer conceptual questions about computing and computer science, or the ability to analyze problems and design solutions (problem-solving). The "appropriate" approach to teaching would depend on the goal.

2. Even if goals were clearly stated, it is likely that many different approaches could be used successfully if chosen intentionally to align with the goals and measurements of success (assessment). In many of the studies, it is not immediately clear if the measures of success used are appropriate for the stated objectives. That is to say, the assessments used might not always be measuring all aspects of the desired learning objectives.

Finally, in 2014, Koulouri, Lauria, and Macredie conducted a quantitative analysis of different approaches to teaching introductory programming by varying three aspects: the language used, providing formative feedback, and introducing explicit problem solving training [23]. They found that choosing a language with a simpler syntax (Python instead of Java), and introducing explicit problem-solving training, improved student learning. They also found, however, that providing more formative assessment did not seem to help. They speculate that the reason for the latter finding could be that students do not spend time looking at formative feedback, or do not understand the feedback that is given.

We wish to build upon the past reviews of the field by suggesting an approach to designing learning systems that may help to consistently address some of the shortcomings observed, and bring understanding to the successes. A large number of papers surveyed contain discussions of course objectives, pedagogical tools used, and assessments given, but rarely are these three components discussed together. We know from the engineering education literature that an effective learning system depends on the alignment of these three components [15,16].

Components of a learning system. A popular way to visualize an effective learning system is as a triad of objectives, instruction, and assessment [17], shown in Figure 1. The design of these three components is not sequential: each informs the other.

Effective learning systems. Assessment results can help inform changes to course objectives and instructional methods [17]. Similarly, as instructional methods improve, the objectives may be modified to aim for higher levels of understanding. Being intentional about these interactions is crucial to an effective learning system; if the assessment addresses lower levels of learning than the objectives specify, "the system will be driven by backwash from testing, not by the curriculum." [18]. For example, if an objective of a programming course is that learners will develop problem solving skills, but assessment only consists of tests of programming concepts and syntax, teaching/learning activities would tend to center around those that promote understanding of these concepts and exclude activities that might have a better chance at promoting general problem solving [18].

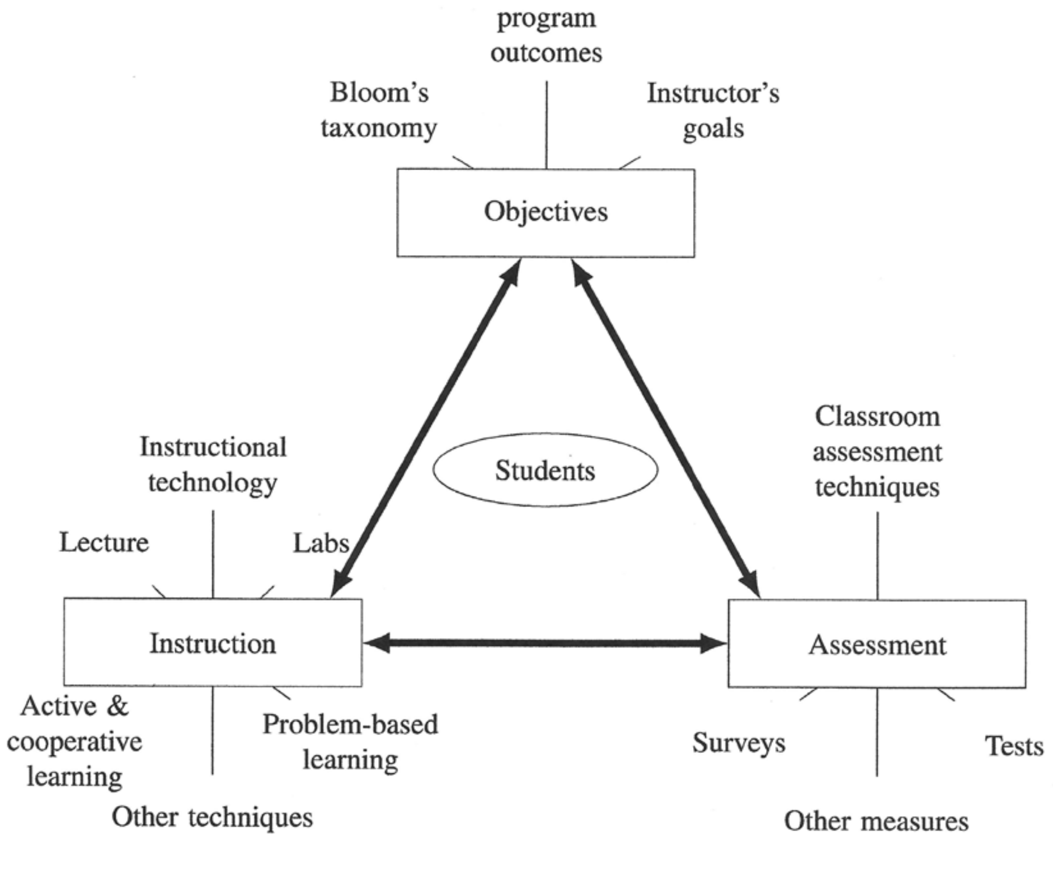

Figure 1: Learning system as derived from Felder and Brent [17].

A Hypothetical Case Study

As noted, teaching OOP is a common area of interest in the literature. While Robins, Rountree, and Rountree note there is a lack of evidence supporting any claim that OOP makes modeling problems any easier [14], we will incorporate it into a hypothetical introductory programming course to explore how its inclusion would affect the constructive alignment between course objectives, instructional methods, and assessment.

It should be strongly noted that we are not advocating for OOP as a "best" teaching paradigm. Rather, we want to point out that from what has been reported, teaching OOP effectively can be challenging. What many implementations seem to miss is that OOP defines an entirely different paradigm for problem solving than is used in procedural programming [12]. As should not be surprising, there are some problems for which OOP provides a good set of tools to solve in an efficient, elegant manner, namely those with solutions that naturally map to real-world objects [43]. Different problems may be better solved using a procedural, functional, or data-flow paradigm [9,44,45].

Ultimately, our goal should be to help students analyze problem structures and make informed decisions regarding what paradigm would be best for a particular problem. What is not clear from the literature is whether or not this consideration is made when designing problems for a particular course. Indeed, in some cases there is evidence that OOP concepts are added to a course "on top of" older content, without necessarily changing the problem sets [12,46].

Cognitive load theory. Cognitive load theory (CLT) is based on the premise that the capacity of our working memory, where all conscious cognitive processing takes place, is finite and relatively small: we are able to manage 2-3 interacting elements at a time. Examples of "interacting elements" in the current context would be recalling and using the syntax and grammar of a particular programming language, reasoning about the behavior of a single object in an OOP environment, or reasoning about a solution to a particular high-level problem. CLT posits that in light of this finite capacity of working memory, for learning to occur, all aspects of a task must fit into this finite space. While some contributors to cognitive load are necessary for the task at hand, others are not [47]. Cognitive load can be divided into three different categories: *intrinsic* cognitive load is due to the complexity of the task itself and generally cannot be reduced without simplifying the task. *Extraneous* cognitive load is associated with work that is due to the instructional method, but not part of the task at hand. For instance, if the educational goal is to complete a programming task, but learners must search and interpret reference material in order to do so, the search and interpretation of this material would be considered extraneous cognitive load. Finally, *germane* cognitive load is also a result of the instructional design, but unlike extraneous load, germane load enhances the learning process. For instance, if a learning goal is that students are able to navigate and use reference material, then the previous example would be considered germane cognitive load. What this example highlights is that whether complexity due to the instruction design is extraneous or germane depends on the specific learning goals. The role of the instructional designer then is to be aware of intrinsic load, reduce extraneous load and intentionally managing germane load.

Object oriented programming. Alan Kay originally conceived of object oriented programming as *one* way to think about complex systems [9]. The underlying metaphor of Kay's conception for OOP was the biological cell: each object is designed to accomplish a specific set of tasks such that the complexity of these tasks is encapsulated within the object. Objects interact with one another by sending and receiving messages, and systems of these objects are assembled to solve complex problems [9].

In several reports of attempts to incorporate OOP into introductory programming courses, practitioners worried that time devoted to OOP concepts would necessarily reduce time spent on the “basics of programming”, usually defined as conditional statements, loops, and in some cases pointers. This concern is misplaced; under an OOP paradigm the “basics of programming” are not the same as when working from a procedural paradigm. OOP defines a completely different set of tools for problem solving, and it is these tools and concepts that become the “basics”: objects, messages, inheritance, and polymorphism [9,12].

For instance, if taking a patterns-based approach to teaching programming, one could use either a procedural paradigm or an OOP paradigm, but the fundamental patterns would be quite different depending on the paradigm. [42] Useful procedural patterns might be:

read-evaluate-print reading a piece of data, evaluating it, and output the result.
guarded-action if a guard-condition is satisfied, take action.

While useful OOP patterns might be:

state provide controlled access to a body of data.
view decouple an object’s state from its representation.
decorator add functionally to an object without modifying its internal structure.

Ultimately, if our goal is to teach problem solving in general, we would be doing a grave disservice to our students to leave them with the impression that there is only one paradigm, whether it be OOP, procedural, functional, or something not yet invented. As previously mentioned, OOP and procedural paradigms are comprised of fundamentally different concepts for problem solving, and so again is the functional paradigm. Unsurprisingly, there is no paradigm that is the “best” fit for every problem. Leaving students with the belief that only one paradigm exists makes it very difficult for them to learn how to intentionally select and apply the “best” paradigm to solve a particular problem.

Kölling argues that performing a paradigm shift from procedural style to OOP is difficult, and so concludes that if OOP is a desired outcome then it should be taught *first* since the reverse shift, from OOP to procedural, is conceptually easier. What this recommendation neglects to account for however is that both paradigms depend upon authoring programs in some language, and this authoring process takes a certain amount of cognitive capacity. To justify an objects-first approach we would have to verify that the cognitive load associated with OOP concepts themselves combined with the cognitive load of learning to author a program in a particular environment, does not exceed the learner’s working memory capacity.

In a superficial way, CLT theory would suggest that OOP is a helpful paradigm for managing cognitive complexity: objects encapsulate complexity by hiding the details of implementation behind a simple interface. For example, an object that represents a list of items may have operations to add and remove items from the list. Conceptually we can use this object with our common understanding of what it means to “add” an “item” to a “list”. Critically, we do *not* need to be aware of how the adding and removal of the item is actually implemented in code, or even how the storage is structured in memory (e.g. an array or a linked-list). We refer to this use of CLT to justify the reduction of complexity of OOP as superficial because it focuses on only the conceptual paradigm itself without regard to implementing it in some programming environment. In practice, we usually ask our students to implement OOP concepts in a particular programming language. That task requires not only understanding OOP as a paradigm, but also working with the programming environment, authoring a program, understanding the syntax and grammar of the language, and most likely debugging it as well.

Cognitive challenges teaching OOP. A behavior that characterizes expert programmers is the ability to easily shift between different abstraction layers of a problem space [48]. This ability may be especially crucial to becoming fluent in OOP as the concepts are somewhat further removed from the actual syntax than in other paradigms. For example, in the procedural paradigm the central concept is that of the procedure, or subroutine. Subroutines are simply collections of imperative commands grouped together under a label. Thus, if someone can write and understand imperative code, the conceptual leap to understanding the procedural paradigm is relatively small. In contrast, to understand objects in OOP one must understand the concept of state and methods operating on the state. Both state and method are implemented with code (generally imperative), but to understand how an object functions one has to imagine its initial state, and then how each method call would change this state. Cognitive load theory would suggest that understanding the dynamic interaction of state and method calls is more challenging than grouping lines of imperative codes into re-usable procedures. Thus to be fluent in OOP design, a programmer *must* be able to switch between thinking at the object layer and the program layer. Teaching abstraction is best done by doing it by example, while being explicit about the abstractions used [48]. However, we must keep in mind the limited capacity of working memory: if the cognitive load associated with writing code and thinking about state and methods fills up a learner's working memory, the learner may have difficulty grasping concepts of abstraction, even when the instructor explicitly points them out. To reduce the cognitive load associated with writing code and working with object state and methods, these tasks must be practiced until somewhat automated: to a point at which they can be done without conscious thought [32].

Language choice. Choice of language does matter: If the learning objectives of an introductory programming course are to learn problem solving concepts then a language with a simpler syntax is preferable to one with more complex syntax features [23]. The explanation for this can be found in CLT. Given the current example of a goal to teach problem-solving techniques, there will be cognitive load associated with thinking about different strategies one could use. Learning the syntax rules of a programming language adds to the cognitive load, but it does not aid in the process of learning problem-solving techniques. In other words, learning the syntax of a programming language is an extraneous cognitive load in this context. Thus, it is in the educator's best interest to minimize cognitive load associated with features that do not directly help with the learning objectives of the course and select a teaching language with easy-to-learn syntax.

In an attempt to address the concern that the two primary languages, C++ and Java, used to teach OOP in introductory courses may be too complex for novice students, Goldwasser and Letscher select Python as a good choice for introductory programming. They claim that its relatively simple syntax allow new students to better engage with the OOP concepts rather than getting bogged down by details of the language [22].

Iconic versus textual languages. In theory, an iconic programming environment such as Scratch [49] or Alice [50] should aid in the learning of higher level concepts, such as those associated with OOP, since graphical languages eliminates the extraneous cognitive load of dealing with syntax and grammar of a language. In fact, there has been success in using iconic languages to teach first year programming [51], and research indicates that teaching graphical programming first does not inhibit a student from later transferring learned high level concepts to textual programming languages [52].

Snook et al. describe efforts to use the Alice programming environment to teach introductory programming concepts [53]. Interestingly a later report assessing the efficacy of the curriculum using Alice indicated that while pre-post testing indicated learning gains with the environment, focus group data indicated a dissatisfaction with Alice as an introductory language, resulting in a switch to LabVIEW [54]. A possible explanation given for the dissatisfaction of Alice was that students did not perceive it as being a “real” programming environment they might use in industry, while LabVIEW was. Certainly, helping students connect the utility of skills they learn in the classroom to those that will be important in their later career is important. In some cases, switching to a tool that is known to be used in industry may be useful, though this approach seems shortsighted: it is unlikely that all graduates will end up in an industry that uses a particular tool (e.g. LabVIEW), so if that is the motivation for a particular tool we must still address the utility of this tool to students who may never have reason to use it again. A different strategy than swapping tools is to better articulate *why* a particular tool was chosen for the learning environment. Students might have less resistance to an educational environment such as Alice if they know that it can help them learn high level concepts that *will* be directly applicable to their future career *and* that the use of Alice as an introductory environment will not hinder them from later transferring those concepts to other environments.

Problem solving. Developing problem solving skills was widely regarded as a common objective for introductory programming courses. Achieving this objective is more likely if the course begins with a focus on problem solving strategies, before any programming is introduced [23]. Separating problem-solving instruction in this way has a number of benefits grounded in cognitive science research: first, it serves to activate prior knowledge [55]. With problem-solving strategies fresh in their memory, learners will more easily apply them when they begin programming. Second, it helps manage cognitive load. Whereas in the previous example of language choice, we reduced cognitive load by choosing a language with a simpler syntax, here we separate two tasks that we know individually demand a high level of cognitive load for novices: learning problem-solving strategies, and learning programming techniques.

Assessment

Often times assessment becomes the weak point of a learning system. Designing good assessment is hard [56,57], and often good assessment can require significant time on the part of the instructor. Because of this, assessment is often simplified, lowering the effectiveness of the entire system.

If our course objective is to teach learners to solve problems with programming, then an ideal assessment would be to give students a problem to solve, ask them to write a program to solve it as well as reflect upon the patterns they used in their solution. Asking for reflection prompts students to engage in meta-cognition, becoming aware of their own thinking process, which helps them become more effective learners [55,58]. Of course evaluating this type of assessment is much more time consuming than a multiple-choice exam that can be automatically scored, or even than evaluating that the finished program runs correctly, which can be done automatically as well [59].

Considerations and Limitations

In the above hypothetical example, we had the luxury of complete freedom in designing all aspects of an introductory programming curriculum. We recognize this is almost always never the case in practice; however, we argue that following a framework, such as constructive alignment, as close as possible will help. Thinking about the framework will also encourage critical engagement with the constraints imposed by the institution. For example, there has been much interest in assessment tools used for introductory programming courses [56,60,61], and in particular, automated assessment [57,59,62]. In many cases the push towards automated assessment techniques is a response to increasing class sizes that instructors typically have little control over. In the event that a large class size dictates that some form of automated assessment be used, following the constructive alignment framework would constrain the types of course objectives and instruction that would be feasible. For instance, if the primary course objective was to teach conceptual understanding of computer science, then a well-designed multiple-choice assessment could be used effectively [56,60]. Using the framework of constructive alignment to critically evaluate the alignment of assessment, objectives, and instruction, may help make explicit some of the compromises that must be made for larger class sizes. This positions the course designers to be more intentional about what compromises to learning are made.

Discussion and Future Work

This literature review was by no means comprehensive, but we hope that it touched upon some of the notable areas in which educators have been experiencing challenges and successes in teaching introductory programming. We have pointed out a number of possible connections between computer science and cognitive science that may be worth deeper investigation. While we have seen how cognitive load theory can help reason about course design, there are also potentially interesting connects between cognitive load theory and object oriented design: objects are designed to encapsulate complexity to better manage it. Additionally, further exploring concepts of knowledge organization and differences between novices and exports may be a fruitful area to investigate the use of teaching programming to aid in analogical reasoning and abstract thinking [63].

An area that has not been addressed in this paper but has strong connections is that of attention in cognition [64]. There are strong parallels between our limitations to focus our conscious attention at one or two tasks at a time, and the serial processing inherent in most computing and language design. Of particular interest is the move in hardware to increasingly parallel designs to keep pace with Moore's law. Our cognitive constraints of attention may suggest an inherent difficulty in thinking and reasoning about parallel processing that must be addressed if we want to prepare students for an environment that demands parallel-processing solutions to programming problems. These demands may shift focus to languages such as Erlang that are designed for high levels of concurrency, result in more interest in functional paradigms over OOP, as well as demand we adopt different metaphors for computing altogether [65].

Conclusion

Recognizing that introductory programming is a common component of an engineering curriculum, and that there is a national push to introduce it to an even larger population, it is important to think carefully about how we design programming education to impart the cognitive skills that will generally benefit learners even after the specific technologies they learn are no longer relevant. We strongly advocate for the use of a framework such as constructive alignment, described in this paper, to inform the choices for course objectives, instructional techniques, and assessments. Using such a framework well help identify components of a course that may be contributing unnecessarily to cognitive load, or reasons why student outcomes might not match expectations assumed from course objectives.

Analyzing a proposed curriculum through the lens of cognitive load theory can aid in instructional decisions that will increase the chance of achieving learning goals. Educators must critically review the goals of their introductory programming courses: if goals are to understand high level concepts such as OOP, but not necessarily implement these concepts in textual languages, it may be prudent to consider iconic languages for these courses. If, however, goals include learning OOP concepts *and* implementing these concepts in textual language, educators must be aware of the increased cognitive load associated with each of these tasks and provide scaffolding to help students automate aspects of each task individually before expecting them to combine both in successful programs.

Taken together, using an established framework, such as constructive alignment, to design and analyze a curriculum, and utilizing cognitive load theory to help reduce extraneous cognitive load while being intentional about inclusion of germane cognitive load, should help instructional designers create learning environments in which students are able to achieve learning goals. In some cases, intended goals may have to be adjusted to account for the finite limitations of working memory capacity: it is likely not feasible, in the span of a single semester, to expect students with no programming experience to learn the syntax and grammar of a new language *and* higher level concepts associated with a paradigm like OOP, *and* be able to switch between abstraction layers in the fluid manner necessary to solving complex problems with programs. What this suggests is that closer coordination is needed across courses in a curriculum so that goals of one become background knowledge of subsequent courses, resulting in successful achievement of program learning goals by the time of graduation.

References

1. U.S. White House. *Computer Science For All*. whitehouse.gov. Jan. 30–2016. URL: https://www.whitehouse.gov/blog/2016/01/30/computer-science-all (visited on 02/01/2016).

2. code.org. *Every child deserves opportunity*. Code.org. 2016. URL: https://code.org/(visited on 02/01/2016).

3. Lauren Rich, Heather Perry, and Mark Guzdial. “A CS1 Course Designed to Address Interests of Women”. In: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’04. New York, NY, USA: ACM, 2004, pp. 190–194. DOI: 10.1145/971300. 971370.

4. Andrea Forte and Mark Guzdial. “Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses”. In: *Education, IEEE Transactions on* 48.2 (2005), pp. 248–253.

5. Mark Guzdial and Andrea Forte. “Design process for a non-majors computing course”. In: *ACM SIGCSE Bulletin*. Vol. 37. ACM, 2005, pp. 361–365.

6. Kenneth Reid and David Reeping. “A classification scheme for “introduction to engineering” courses: defining first-year courses based on descriptions, outcomes, and assessment”. In: *American Society for Engineering Education Annual Conference & Exposition. Indianapolis, IN (1-11). Washington DC: American Society for Engineering Education*. 2014.

7. National Academy of Engineering NAE. *Educating the Engineer of 2020: visions of engineering in the new century*. Washington, DC: National Academies Press, 2004.

8. Douglas C. Engelbart. “Augmenting human intellect: a conceptual framework”. In: *PACKER, Randall and JORDAN, Ken. Multimedia. From Wagner to Virtual Reality. New York: WW Norton & Company* (1962), pp. 64–90.

9. Alan C. Kay. “The Early History of Smalltalk”. In: ed. by Thomas J. Bergin Jr. and Richard G. Gibson Jr. New York, NY, USA: ACM, 1996, pp. 511–598.

10. Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

11. Michael McCracken et al. “A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students”. In: *SIGCSE Bull.* 33.4 (Dec. 2001), pp. 125–180. DOI: 10.1145/572139. 572181.

12. Michael Kölling “The problem of teaching object-oriented programming”. In: *Journal of Object Oriented Programming* 11.8 (1999), pp. 8–15.

13. Arnold Pears et al. “A survey of literature on the teaching of introductory programming”. In: *ACM SIGCSE Bulletin* 39.4 (2007), pp. 204–223.

14. Anthony Robins, Janet Rountree, and Nathan Rountree. “Learning and teaching programming: A review and discussion”. In: *Computer Science Education* 13.2 (2003), pp. 137–172.

15. Leah H Jamieson and Jack R Lohmann. “Creating a Culture for Scholarly and Systematic Innovation in Engineering Education: Ensuring US engineering has the right people with the right talent for a global society”. In: *American Society of Engineering Educators (ASEE)* (2009).

16. James W Pellegrino. “Rethinking and redesigning curriculum, instruction and assessment: What contemporary research and theory suggests”. In: *commissioned by the National Center on Education and the Economy for the New Commission on the Skills of the American Workforce* (2006).

17. Richard M. Felder and Rebecca Brent. “Designing and teaching courses to satisfy the ABET engineering criteria”. In: *JOURNAL OF ENGINEERING EDUCATION-WASHINGTON-* 92.1 (2003), pp. 7–26.

18. John Biggs. “Enhancing teaching through constructive alignment”. In: *Higher Education* 32.3 (Oct. 1996), pp. 347–364. DOI: 10.1007/BF00138871.

19. Frances Bailie et al. “Objects First - Does It Work?” In: *J. Comput. Sci. Coll.* 19.2 (Dec. 2003), pp. 303–305.

20. Sally H. Moritz et al. “From Objects-first to Design-first with Multimedia and Intelligent Tutoring”. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’05. New York, NY, USA: ACM, 2005, pp. 99–103. DOI: 10.1145/1067445.1067475.

21. Stuart Reges. “Back to Basics in CS1 and CS2”. In: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’06. New York, NY, USA: ACM, 2006, pp. 293–297. DOI:10.1145/1121341. 1121432.

22. Michael H. Goldwasser and David Letscher. “Teaching an Object-oriented CS1 -: With Python”. In: *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’08. New York, NY, USA: ACM, 2008, pp. 42–46. DOI: 10.1145/1384271.1384285.

23. Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. “Teaching introductory programming: a quantitative evaluation of different approaches”. In: *ACM Transactions on Computing Education (TOCE)* 14.4 (2014), p. 26.

24. Laurie Williams and Richard L. Upchurch. “In support of student pair-programming”. In: *ACM SIGCSE Bulletin*. Vol. 33. ACM, 2001, pp. 327–331.

25. Charlie McDowell et al. “The effects of pair-programming on performance in an introductory programming course”. In: *ACM SIGCSE Bulletin*. Vol. 34. ACM, 2002, pp. 38–42.

26. Lynda Thomas, Mark Ratcliffe, and Ann Robertson. “Code Warriors and Code-a-phobes: A Study in Attitude and Pair Programming”. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’03. New York, NY, USA: ACM, 2003, pp. 363–367. DOI: 10.1145/611892. 612007.

27. Jan Chong and T. Hurlbutt. “The Social Dynamics of Pair Programming”. In: *29th International Conference on Software Engineering, 2007. ICSE 2007*. 29th International Conference on Software Engineering, 2007. ICSE 2007. May 2007, pp. 354–363. DOI: 10.1109/ICSE.2007. 87.

28. N. Salleh, E. Mendes, and John Grundy. “Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review”. In: *IEEE Transactions on Software Engineering* 37.4 (July 2011), pp. 509–525. DOI: 10.1109/TSE.2010. 59.

29. Jacob L. Bishop and Matthew A. Verleger. “The Flipped Classroom: A Survey of the Research”. In: ASEE Annual Conference & Exposition. Atlanta, GA, June 2013.

30. Joshua DeSantis et al. “Do Students Learn More From a Flip? An Exploration of the Efficacy of Flipped and Traditional Lessons”. In: *Journal of Interactive Learning Research* 26.1 (2015), pp. 39–63.

31. John D Bransford, Ann L Brown, Rodney R Cocking, et al. *How people learn*. Washington, DC: National Academy Press, 2000.

32. Susan A Ambrose et al. “Chapter 4: How Do Students Develop Mastery?” In: *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010, pp. 91–120.

33. TA Litzinger et al. “A Cognitive Study of Problem Solving in Statics”. In: *Journal of Engineering Education* 99.4 (2010), pp. 337–337.

34. RA Streveler et al. “Learning Conceptual Knowledge in the Engineering Sciences: Overview and Future Research Directions”. In: *Journal of Engineering Education* 97.3 (2008), pp. 279–294.

35. Carli D Flynn, Cliff I Davidson, and Sharon Dotger. “Engineering Student Misconceptions about Rate and Accumulation Processes”. In: ASEE Zone I Conference. Bridgeport, CT, 2014.

36. James D. Slotta and Michelene T. H. Chi. “Helping Students Understand Challenging Topics in Science Through Ontology Training”. In: *Cognition and Instruction* 24.2 (June 1, 2006), pp. 261–289. DOI: 10.1207/s1532690xci2402_3.

37. Bracha Kramarski and Zemira R. Mevarech. “Cognitive-metacognitive training within a problem-solving based Logo environment”. In: *British Journal of Educational Psychology* 67.4 (1997), pp. 425–445.

38. David Reed. “Incorporating Problem-solving Patterns in CS1”. In: *Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE

’98. New York, NY, USA: ACM, 1998, pp. 6–9. DOI: 10.1145/273133.273137.

39. Orna Muller and Bruria Haberman. “Supporting abstraction processes in problem solving through pattern-oriented instruction”. In: *Computer Science Education* 18.3 (Sept. 1, 2008), pp. 187–212. DOI: 10.1080/08993400 802332548.

40. J. Philip East et al. “Pattern-based programming instruction”. In: *Proceedings of the ASEE Annual Conference and Exposition, Washington DC*. 1996.

41. Orna Muller. “Pattern oriented instruction and the enhancement of analogical reasoning”. In: *Proceedings of the first international workshop on Computing education research*. ACM, 2005, pp. 57–67.

42. Eugene Wallingford. “Toward a First Course Based on Object-oriented Patterns”. In: *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’96. New York, NY, USA: ACM, 1996, pp. 27–31. DOI: 10.1145/236452.236485.

43. Eric S. Raymond. *The art of Unix programming*. Addison-Wesley Professional, 2003.

44. Matthias Felleisen et al. “A Functional I/O System or, Fun for Freshman Kids”. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’09. New York, NY, USA: ACM, 2009, pp. 47–58. DOI: 10.1145/1596550.1596561.

45. Marcus Crestani and Michael Sperber. “Experience Report: Growing Programming Languages for Beginning Students”. In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. New York, NY, USA: ACM, 2010, pp. 229–234. DOI: 10.1145/1863543. 1863576.

46. Tamar Vilner, Ela Zur, and Judith Gal-Ezer. “Fundamental Concepts of CS1: Procedural vs. Object Oriented Paradigm - a Case Study”. In: *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’07. New York, NY, USA: ACM, 2007, pp. 171–175. DOI: 10.1145/1268784.1268835.

47. Fred Paas, Alexander Renkl, and John Sweller. “Cognitive load theory and instructional design: Recent developments”. In: *Educational psychologist* 38.1 (2003), pp. 1–4.

48. Michal Armoni. “On Teaching Abstraction in Computer Science to Novices.” In: *Journal of Computers in Mathematics and Science Teaching* 32.3 (2013), pp. 265–284.

49. Mitchel Resnick et al. “Scratch: Programming for All”. In: *Commun. ACM* 52.11 (Nov. 2009), pp. 60–67. DOI: 10.1145/1592761.1592779.

50. Stephen Cooper, Wanda Dann, and Randy Pausch. “Using animated 3d graphics to prepare novices for CS1”. In: *Computer Science Education* 13.1 (2003), pp. 3–30.

51. Ben A. Calloni, Donald J. Bagert, and H. Paul Haiduk. “Iconic Programming Proves Effective for Teaching the First Year Programming Sequence”. In: *Proceedings of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’97. New York, NY, USA: ACM, 1997, pp. 262–266. DOI: 10.1145/268084. 268189.

52. Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. “Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 16.3 (2009), p. 13.

53. Jason Snook et al. “Incorporation of a 3d interactive graphics programming language into an introductory engineering course”. In: *age* 10 (2005), p. 1.

54. Vinod K Lohani et al. “Reformulating General Engineering and Biological Systems Engineering Programs at Virginia Tech”. In: *Advances in Engineering Education* 2.4 (2011), pp. 1–30.

55. Susan A Ambrose et al. *How learning works: Seven research-based principles for smart teaching*. John Wiley & Sons, 2010.

56. Raymond Lister. “On Blooming First Year Programming, and Its Blooming Assessment”. In: *Proceedings of the Australasian Conference on Computing Education*. ACSE ’00. New York, NY,

57. USA: ACM, 2000, pp. 158–162. DOI: 10.1145/359369.359393.

58. Des Traynor and J. Paul Gibson. “Synthesis and Analysis of Automatic Assessment Methods in CS1: Generating Intelligent MCQs”. In: *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’05. New York, NY, USA: ACM, 2005, pp. 495–499. DOI: 10.1145/1047344. 1047502.

59. Michelene T. H. Chi et al. “Self-explanations: How students study and use examples in learning to solve problems”. In: *Cognitive Science* 13.2 (Apr. 1, 1989), pp. 145–182. DOI 10.1016/0364-0213(89)90002-5.

60. Brenda Cheang et al. “On automated grading of programming assignments in an academic institution”. In: *Computers & Education* 41.2 (Sept. 2003), pp. 121–131. DOI: 10.1016/S0360-1315(03)00030-7.

61. Raymond Lister. “Objectives and Objective Assessment in CS1”. In: *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’01. New York, NY, USA: ACM, 2001, pp. 292–296. DOI: 10.1145/364447. 364605.

62. Judy Sheard et al. “Exploring Programming Assessment Instruments: A Classification Scheme for Examination Questions”. In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER ’11. New York, NY, USA: ACM, 2011, pp. 33–38. DOI: 10.1145/2016911.2016920.

63. Petri Ihantola et al. “Review of Recent Systems for Automatic Assessment of Programming Assignments”. In: *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. Koli Calling ’10. New York, NY, USA: ACM, 2010, pp. 86–93. DOI: 10.1145/1930464.1930480.

64. Jens Bennedssen and Michael E. Caspersen. “Abstraction Ability As an Indicator of Success for Learning Computing Science?” In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER ’08. New York, NY, USA: ACM, 2008, pp. 15–26. DOI: 10.1145/1404520. 1404523.

65. Herbert A. Simon. “The role of attention in cognition”. In: *The Brain, Cognition, and Education*. Ed. by Sarah L. Friedman, Kenneth A. Klivington, and Rita W. Peterson. New York: Academic Press, 1986, pp. 105–115.

66. Lynn Andrea Stein. “Challenging the computational metaphor: Implications for how we think”. In: *Cybernetics & Systems* 30.6 (1999), pp. 473–507.

Biographical Information

Darren Maczka is a Ph.D. student in Engineering Education at Virginia Tech. His background is in control systems engineering and information systems design and he received his B.S. in Computer Systems Engineering from The University of Massachusetts at Amherst.

Jacob Grohs is an Assistant Professor in Engineering Education at Virginia Tech with Affiliate Faculty status in Biomedical Engineering and Mechanics and the Learning Sciences and Technologies at Virginia Tech. He holds degrees in Engineering Mechanics (BS, MS) and in Educational Psychology (MAEd, Ph.D.).