Rethinking assessment in early computing courses

Dr. Tony Lowe
Jarvis College of Computing and Digitial Media
DePaul University
Chicago, IL, USA
https://orcid.org/0000-0002-5510-5235

***Abstract*— This research-to-practice paper considers the impact of theory and findings from cognitive science research on assessments in computing courses. While the pandemic-forced virtual learning interrupted many standard classroom routines, these interruptions also offered opportunities to innovate beyond common pedagogical tropes. One educational mainstay, the proctored test, became particularly difficult to administer in virtual environments. Ensuring that a test only reflects an individual's learning without external support became more cumbersome than the alternatives – alternatives that might offer better long-term options for assessing learning.**

Many educators question the equity and effectiveness of high-stakes testing, with assessments of computing skills introducing even more challenges. Tests typically require students to demonstrate learning apart from computers, assuming such knowledge transfers equally to pen and paper. Advances in cognition suggest that learning may not transfer between activities in a timely and effective manner, particularly when much of the work is hands-on. Paper tests may not accurately measure a curriculum's learning objects giving students inaccurate feedback on their abilities. Non-traditional students with professional experience in coding may be disenfranchised as their practical knowledge fails to help on tests. Some students with otherwise strong GPAs may become frustrated when unable to tackle open-ended problem-solving tasks in later courses. Both groups are learning but need assessments, policies, and grading approaches that measure all types of learning.

This paper describes integrating theory and research from the cognitive sciences with practice in the computing classroom. The foundation for the discussion is based on a new theory of programming cognition called TAMP. TAMP suggests that expert programmers gather and apply knowledge both explicitly and implicitly. Traditional testing may not investigate all types of knowledge and thus may miss important aspects of a new programmer's development. Theoretical advances drive new research and can have a vital impact on classroom practices. This discussion reflects on how theory (and the pandemic) drove changes to assessment in computing classrooms. The hope is that discussing how theory and research drive practice can inform future research and inspire additional practical improvements.

Keywords—Dual Process Theory, TAMP, Computing Education, Bruner, Student Assessment, Course Design

I. Battling a 'Content Squeeze'

Today's computing educators are blessed to teach at a time with a great number of educational resources yet with equally great expectations. The internet provides students with thousands of hours of tutorials and many documented technical fixes freeing educators from covering every basic concept and troubleshooting step. Our profession's potential means that students are flocking to computing degrees [1], and society is looking to broaden the pipeline [2], [3]. Parallel to increasing class sizes, including students with less prior exposure to computing, workplaces expect even more from graduates. Companies report that graduates are "unprepared" [4], expecting more than "the fundamental knowledge of algorithms, computer language theory & software development" [5]. Employers want students with technology experience but also demand less tangible skills. Tikayatray [6] suggests that graduates who are not decisive, comfortable with change, accountable, creative, and disciplined suffer in later career growth – all skills hard to measure using traditional methods. To meet the pressing demands from both ends of the pipeline, educators need flexible and inclusive teaching methods that promote a more expansive and perhaps slightly different skillset

Computing educators are essentially seeking to improve the throughput of a legacy educational delivery solution – to do more in the same timeframe and resources. A half-century of computing education research has provided many insights but has yet to deliver enough 'productivity increases' to offset this 'content squeeze'. Simplifying expectations may improve early pass rates and retention, yet removing too much complexity risks leaving students underprepared for the workplace. Customizing curricula to meet employer demands risks irrelevancy as technologies are often fungible and short-lived. Instead of incrementally changing, the computing classroom may need to transition its learning objectives into developing traits that programmers possess rather than knowledge gained. Employers want creative, disciplined, self-motivated workers who acquire knowledge independently and on-demand. Encouraging that particular skill set not only better prepares students for the workplace but also the classroom but requires significant restructuring.

Educators need new teaching models if we are to create 'super-learners'. Traditional assessments may not encourage the study approaches that build the lasting memories and habits of mind computing professionals need. The cognitive mechanics that support 'cramming' information for short-term retrieval may not yield the long-term memories or procedural expertise required for professional practice [7]. Past research [8], [9] and new theories in computing education [10] suggest that this practice (i.e., experience) separates experts and novices. This paper explores one attempt to apply such theory to the teaching and assessment of students during early programming coursework (e.g., CS1, CS2, and Systems). The first step is

understanding the relevant theory and research on how expert programmers think and novices' gaps in learning. Theory then informs how educators create assessments and classroom policies that reflect the desired outcomes. The goal is to start a discussion on how theory informs practice and, in turn, drives new research that quantifies the relationship between theories and practice.

II. A Cognitive Model of 'Expert' Programmers

The Theory of Applied Mind of Programming (TAMP) defines a cognitive model explaining the knowledge and skills that distinguish expert programmers [10]. Instead of a detailed concept inventory of programming topics, TAMP describes the cognitive mechanisms underlying such 'repositories' of knowledge and how they differ between experts and novices. For example, Youngs [11] noted in the early days of COBOL that programmers of all levels produced similar errors in their first run of a program, but experienced programmers fixed them faster. It was not that experienced programmers were necessarily better at avoiding mistakes but used repositories of knowledge that helped them to find and correct errors quickly. The experts used these repositories to fix bugs but did not to avoid them. The circumstantial recall of such information implies that such knowledge is either not simple facts or highly tied to specific stimuli. TAMP suggests that certain programmers learn certain strategies tacitly, not explicitly. Wiedenbeck [8] saw a similar phenomenon noting that experts were faster *and more accurate* than novices in recognizing syntax errors and coding patterns against her predictions. She reasoned that since students learned the same programming facts and studied them more recently than the experts, they would perform the same. Experts exceeded novices in all ways, suggesting practice and experience improved performance even on simple 'fact-based' tasks. TAMP leverages cognitive sciences to explain the mechanics behind why experts perform better and suggests ways to help novices develop such abilities.

TAMP applies the cognitive model from dual process theory to explain why experience leads to quicker and more accurate responses. Dual process theory describes cognition as two distinct mechanics: an automatic, implicit, and effortless path called System 1 and a conscious, effortful, and reasoning approach called System 2. Experienced programmers complete many aspects of programming, like writing correct syntax or recognizing error messages, automatically and with little effort. Yet these repositories do not automatically overlap, as Youngs observed. Finding errors at runtime seems to be a distinct knowledge repository than writing code. Selective recall is a trademark of System 1, which relies on stimulus to spark memories. Through a deliberate review of the code, programmers may catch errors proactively, but it is often simpler and less effortful to run the code and fix problems. TAMP suggests that System 1's capabilities translate to much more than procedural activities and influence creativity and problem-solving – but only when given exposure to relevant experiences.

According to TAMP, developers gradually acquire repositories of implicit design knowledge necessary for effective problem-solving. Such repositories help experts leap from problems to promising design approaches or even debugging strategies that often stymie novices. This explicit form of such knowledge is known as design patterns. Kolfschoten et al. [12] noted that "Design patterns seem especially useful to transfer ***tacit*** best practices and expertise" (p. 652, *emphasis added*). They investigated how experts and novices learn and use design patterns, finding that novices scaffold their early design attempts, while more experienced participants eschewed and "complained that they had to find the design pattern" (p. 659). TAMP suggests that new developers find patterns useful since they lack any intuitive starting point. In contrast, experienced programmers found the process annoying as it required them to make tacit knowledge explicit. While instruction on patterns seems a promising method for teaching design, explicitly teaching tacit abilities is not always effective [13] and does little to alleviate the 'content squeeze' in courses. Understanding the implicit nature of such knowledge helps inform new ways to create the distributed and deliberate practice required to teach new programmers implicit aspects of coding and wider problem-solving like design and debugging.

III. Designing Curricula to Support Implicit Learning

Traditional pedagogy and assessment approaches may not reinforce the good habits required for the types of learning TAMP suggests. When students' grades primarily derive from presenting the "correct answers", they may shortcut the intermediate experiences necessary to form lasting implicit memories, especially when working to a strict deadline. In his book on making, Adam Savage [14] described deadlines as a sieve that removes extraneous details, such as overly ambitious goals or perfectionistic tendencies, and gets projects done. Following Savage's logic, what do students 'sieve out' when completing assignments? When faced with a deadline, my students tell me they turn to the internet for solutions, seek answers from peers, or inevitably turn in whatever progress they complete for partial credit. They seldom return to '(in)completed' exercises to seek further understanding or for additional practice. They may review for exams but rarely fix code or even 'tinker' for practice and improved understanding. Their learning mostly stops when the points end.

Traditional assessments do little to build the tacit knowledge of System 1. Quizzes and tests permit clever students to cram System 2 with recently learned facts, but this knowledge quickly fades without repetition (and lacks the context clues System 1 needs). New programmers need repetition and variety in every aspect of computing, whether syntax, design, or troubleshooting to form the repositories that make experts 'better'. Students can excel at some test questions only to mysteriously forget the same concepts later in the same test [15] - a phenomenon labeled fragile knowledge [16]. TAMP reframes fragile knowledge as a failure of System 1 to prepare (a.k.a., prime) System 2 with the appropriate context and facts. Students remember programming concepts, but their untrained System 1 cannot connect what they see to the related ideas. New programmers must ***cause*** compiler and runtime errors (many times!) before they will quickly jump to the necessary fix. Merely watching a worked example and even quizzing on syntax may not lead to practical knowledge since it is devoid of the context in which the errors typically appear. Experts are better at transferring concepts because of their exposure to many examples of the required knowledge in action.

TAMP suggests that students must participate in complex designs and problem-solving strategies to form repositories in those areas. McCracken et al. [17] witnessed the difference in students who had and lacked prior exposure to useful design strategies. Most participants in their study made little progress in building a simple calculator application, yet one subgroup did significantly better. Their "instructor had given the students an example to study, which was a complete answer to a similar problem" (p. 132). The mere exposure to a similar design vastly improved those students' progress (scoring nearly triple the others), but not universally so. Despite seeing a similar solution, the students still failed to earn even half of the possible points for the assessment. Seeing a similar problem did not guarantee success, but it helped those students progress, whereas others failed to produce even compiling code. Fostering excellent programmers may require students broadly train their System 1 using various problems, not merely targeted examples that are excellent at exposing how computers function.

Pedagogy and assessments that enforce repetition help move new programmers beyond ephemeral into ongoing competency. Ensuring that new programmers see many problems and code varying solutions while making and correcting errors, finding and debugging mistakes is vital beyond abstract interconnected knowledge. The brain is not a database to fill but living hardware that responds to its environment. Novices need repetition before their eyes recognize salient information, and hands make quick corrections. Repetition also breeds familiarity, which in turn builds confidence. One group of researchers [18] found that conceptual understanding alone did not always lead to programming prowess. Traditional tests have their place, but it is "probably important that the teaching process stress continuous practice with basic materials to the point that they become overlearned" [8, p. 389]. Instructors can help students develop robust "System 1 knowledge" by choosing grading schemes and assessments that promote more practice. I use two approaches to build an integrated skillset that I call coding small and coding large.

IV. Theory inspired Changes

A. Coding Small

The simple and impractical answer to getting students to 'practice more' is to assign them more work. Merely adding assignments risks alienating students who list workload as a reason for leaving computing degrees, particularly among women [19]. Students need targeted work with a clear purpose, not "pointless busywork" (p. 405). Instructors can avoid such perceptions by creating focused and scaffolded assignments that tie to clear learning objectives. Scott Portnoff [20] introduced his inner-city high school students to programming syntax by asking them to memorize, character-by-character, and reproduce code. While bordering on "busy work", memorization trained his students' eyes to identify the tiny details that lead to compiler errors. Portnoff's approach shows the value of implicit learning, but instructors may achieve similar results while sticking to higher-level programming tasks.

Students can learn the syntax and identify the meaning of errors through simple directed coding activities that only require them to produce code snippets. Novices must learn to write perfect syntax before seeing a simple program execute, but compiler errors are notoriously difficult to understand [21]. I often see beginners ignore compiler errors and attempt to plow on with the complete assignment, perhaps hoping they go away. Writing a dozen makes little difference when they don't understand one error. Since they must untangle the errors, these simple syntax issues derail further progress. As an alternative, instructors can include platforms that present 'small' coding exercises that require a code snippet of just a line or two, such as the following.

> Assume your program has a variable `printMe` containing a string that the program needs to output to the user. Write the code that will display the contents of this variable to the user.

Code snippet tools allow the novice to focus on a single concept and its related syntax. It limits the number of places the beginner can introduce errors that distract attention from the targeted lesson. This 'language in pieces' approach scaffolds System 2 to tackle problems analytically while giving System 1 at least basic feedback to help train "eyes and hands". By keeping problems small, students can tackle many of them in a short amount of time and also see a concept applied in many ways. In my CS1 Python course, students can access 362 similar problems across eight topics. They can tackle as many as they desire, averaging 250 problems per student when I only suggest 120. The newest programming students often state their appreciation for such tools as a way to become familiar with the language and a way to build early confidence.

Small coding problems come in many flavors and are common in many classrooms but have drawbacks. The primary disadvantage of any tool is the resources required to buy or create one. Home-grown solutions demand a significant investment of time to develop and maintain but allow specific customizations. Off-the-shelf tools balance affordability with control. Existing tools offer the infrastructure and sometimes even questions, but answers are potentially available online. Plagiarism does occur, but less in students motivated towards deep learning and can be detected (discussed more later). The main goal of 'small coding' is to encourage the practice that trains System 1 to 'see' computing details and 'do' computing tasks. 'Small coding' alone is not likely enough to prepare students to tackle more complex problems that require a larger perspective.

B. Coding Large

If knowledge about languages is an incomplete foundation, students need instruction/activities to fill gaps in knowledge. TAMP suggests that new programmers need experiences promoting the instincts that drive design, testing, and debugging. "Not knowing" the fundamentals of a language is undoubtedly a limitation to writing complex code. The literature often reports students who demonstrate such foundational knowledge yet fail to apply it – a.k.a. fragile knowledge [16]. Fragile knowledge seems to occur when System 1 fails to activate the appropriate knowledge for System 2 to make decisions. Perkins and Martin showed how students could remember helpful facts after receiving hints, essentially letting the research stand in for an undertrained System 1. Students need to learn programming facts and see that knowledge in context, or better yet, many contexts. The more problems a beginner sees (particularly with

many different contexts), the more likely they will remember the required information when facing open-ended problems. Adding more work to struggling students would seemingly add to their currently insurmountable challenge.

Instructors can prepare students to be strong designers by gradually exposing them to complex problems. System 1 does analytically dissect complex situations but 'notices' patterns in the complexity. Experts often forget that to a beginner, everything is a vague abstraction! Programmers write code as a series of magic words and rituals in the eyes of many beginners. Offering instruction that logically deconstructs abstractions helps support System 2 learning, but only repetitive practice trains System 1. System 1 does experience mental load but can benefit from focused attention. Knowing what to look for makes it easier to see the patterns. Teaching to System 2 makes sense analytically and works for 'gifted' students but may leave many others behind on its own. Teaching System 2 without giving System 1 time to automate the lessons leaves the brain less able to deal with increasing complexity.

Dual process theory suggests that rather than avoiding complexity, we can embrace it. System 1 does not require logic but experience. Experience that instructors can provide by focusing the learner's attention on specific goals. Gradual exposure to 'complex and messy coding processes' can help students as much as a reasoned explanation. Worked examples [22] provide one such approach where students follow along with a video or live demonstration of a task. Such 'mimicking exercises' should ideally be rewarded tasks, not merely instructional devices. Giving points for even mimicking work encourages repetition and broad exposure. Too many guided examples will not likely lead to truly independent work, but students tend to lose interest in mimicking as they mature [23]. Students become better at tackling problems when they see various 'complex' scaffolded exercises that increase both challenge and autonomy.

C. *"Additional" Skills*

One frequently under-served computing topic is testing, a shockingly straightforward way to introduce novices to complex programming processes without much foreknowledge. I often forget that creating appropriate tests is neither natural nor easy for most humans. If curricula contain directed Testing coursework, it is often late in the progressions of courses or focused on System approaches, not lines of code. Simple introductory exercises in testing can help fill a massive gap in understanding and introduce programming concepts in a practical setting.

Asking novices to create inputs and evaluate outputs provides a practical reason to explore the logic of computers. One such exercise presents a decision tree within a flow chart and asks students to 'code' inputs for testing each flow. Students set values to True and False within a list and run the program to see if their tests pass. This simple activity asks them to:

- Consider the logic of a decision tree and possibly review its corresponding code/syntax
- Trace that code to determine appropriate inputs
- Compare their predicted trace with the actual output
- Consider what it means to test code

The active analysis of code in action reinforces System 2 concepts while providing System 1 with practical experience. The tracing and running code combination help build System 1 'code rules repositories'. First, the output is predictable as it is the correct code, not a bug-ridden novice attempt. Second, the exercise provides a context that focuses attention beyond just achieving an output towards understanding design and test strategies. Testing tasks are low-stake and relatively quick problems that span the entire programming lifecycle while promoting basic coding skills.

An equally underserved and possibly more difficult topic to teach is debugging. Debugging requires programmers to juggle multiple mental representations, including the problem, design, test case, the program's outputs, what they think their code is doing, and what it is actually doing. It is another area where experts forget that novices do not know even simple strategies, like printing out data. Like the testing exercises, instructors introduce rich examples seeded with bugs to fix. Rather than asking for a full solution, novices focus on specific aspects of the existing code to seek the root cause of the error. Targeted bugs might prey upon misconceptions and teach students to overcome their expectations using data – a process I call being a computer ***scientist*** instead of a computer ***philosopher***. Debugging exercises bring forth implicit strategies and assumptions to form habits that might overcome our natural cognitive biases that make debugging particularly difficult. TAMP suggests that testing and debugging activities help create the implicit connections between concepts that experts need.

TABLE I. EXAMPLE EXERCISES AT VARIOUS LEVELS OF SCAFFOLDING

Scaffolding	Exercises
Very High	• Code snippet exercises – complete just a few lines of code in a highly controlled environment • Mimic exercises – Complete an exercise in a full environment by following a complete demonstration • Testing exercises – create inputs that fully validate working code
Moderate	• Debugging – fix code that mostly works but has targeted bugs • Intro mimic – Complete one piece of a problem using mimic, but then complete more independently
Low	• Open-ended – complete a problem to satisfy an existing set of test cases based on a description of the need
None	• Code from scratch – build a solution from the description alone

D. *Coding Small and Large*

Strategically cycling between small and large assignments may offer a way to do more in the same amount of time. 'Small' assignments promote repetition of specific ideas, whereas 'large' ones place ideas in context and link them to other important ideas. Instructors can employ a spiral curriculum [24] to cycle between small and large activities. Each 'spiral' can integrate new concepts and demonstrate how each concept

behaves in authentic solutions. The 'spiral' philosophy has two essential ideas: integration and repetition. Instead of isolating each concept, a spiral curriculum demonstrates rich integrations but focuses on specific ideas *shown in all their complexity*.

Avoiding complexity helps when training System 2 but presents decontextualized facts that may not help learners "play the whole game" [25]. Perkins discusses the need of students to see their subject as something more than the sum of its parts. His suggestion maps well to System 1, which requires training in the noisy complexity of reality to respond appropriately in similar future situations. A spiral should not necessarily start with simple tasks but rather with 'noisy' problems supported with a lot of scaffolding. Successive steps then gradually reduce scaffolding until students work independently. Table I. shows some examples of exercises at various levels of scaffolding.

E. System over Pedagogy

My early innovations in teaching focused on finding 'just the right way to explain' complex computing topics. Over time I recognized that even my best-conceived examples would not resonate with everyone. Finding the right activity might set people on the right path to discovery. The educational psychologist Jerome Bruner suggests that teachers put students at the forefront of their learning.

> The tutor must correct the learner in a fashion that eventually makes it possible for the learner to take over the corrective function himself. Otherwise the result of instruction is to create a form of mastery that is contingent upon the perpetual presence of a teacher. [26, p. 53]

Students need feedback, but instructors may limit progress if students rely on their instructors too much. The most common feedback offered is grading, but instructors also can become central by how they select assignments, pedagogy, and timelines. Rethinking assessment should be more than merely the activities students complete, but what we expect of students and how we 'reward' them for their work.

Students need a course structure that encourages work that builds System 1 competency, grounded in repetition and variety. The structure of early computing courses should encourage 'eyes and hands' training over feats of memorization. I created an effort/results-driven environment through a few policies: 1.) students earn grades from an accumulating point total, 2.) they have unlimited resubmissions of work, 3.) most work is due at the end of the course, and 4.) a 'trust-but-verify' system of academic integrity.

Accumulating points encourages students to complete many exercises at their chosen level of complexity. My students must earn, for example, 1000 points for an A, 850 for a B, and 700 for a C. To discourage 'point hacking', I require a minimum number of points in each major concept area of the course (e.g., 100 points from Decisions, 75 points from Loops), but they can choose assignments from more than 1500+ points available in a class of 1000 points for an A. They can select tasks at many levels of scaffolding, from mimicking to creating apps from scratch on various topics that may appeal to their goals. Allowing students agency allows them to decide how deeply they engage with each topic and does not make them feel inadequate when unable to complete mismatched assignments. If they are struggling, they can merely try another assignment!

Allowing unlimited resubmissions encourages a deeper understanding and a growth mindset, not to mention mirrors authentic practice. Many students inexplicably expect to get every answer right the first time. They need to see that mistakes – and fixing them – are a critical part of programming. Unlimited resubmissions set this expectation and replace the notion of 'partial credit' with the idea of iterative improvement. Students now have a reason to work through the problems giving System 1 invaluable experience in the implicit aspects of coding.

Deferred due dates have the same advantages as unlimited resubmissions but with a dark side. Allowing students to pick their due dates removes the burden to progress at the same rate as their peers or limitations imposed by external factors. Students can spend time mastering challenging tasks and go back at any time for further practice. Flexible deadlines reduce the impact of extracurricular disadvantages (e.g., illness, need to work, gaps in prior learning) that may lead to a feeling of exclusion and lack of belonging. The challenge in offering this flexibility is ensuring procrastinators stay engaged – a topic beyond any reasonable scope here. Open deadlines may allow some students to fall behind, but teaching self-directed work is a critical professional skill that students need to learn. Like most design tasks, deadlines are a tradeoff that educators must balance within their greater learning goals.

Giving students great agency requires changes to assessments and the measures of academic integrity. Many of these ideas may seem impossible without the inclusion of technology. Computing educators have an advantage as our subject is much easier than many to automate feedback (and we have the skills to do it!). Immediate automation gives System 1 the best chance to make accurate connections (we can easily form bad habits without feedback!). Technology also reduces the manual effort of grading and offers automated measures of honesty. Many automated grading systems include plagiarism checking, which helps but only on the final results. To identify if students are engaging with the 'coding slog' to get their final results, I wrote code that captures a "ledger" of their work. Every time a student runs their code, my ledger captures the time and their code. I can see how much time they spend and, if suspicious, review each step of their work. The ledger identifies students who spend hours on assignments and those possibly cutting corners. The ledger measures integrity on each assignment but is not the only check of individual learning.

I avoided traditional tests for most of my first two years of 'pandemic teaching'. The logistics involved in administering tests were too complex, adding to my apprehension about creating a fair assessment. Instead, I used one-on-one behavioral interviews, a vital industry measure that students may never get to practice in school. The interview was a backstop where students told me stories of their work that I correlated with their ledger results and earned points. Rarely did students struggle to prove they had completed work, though it did open conversations with some students who were 'cheating'. I felt confident in these interviews, having interviewed hundreds of industry candidates, though interviewing is as much an art as a science.

I added a 'paper exam' for the last term focused on interview-style questions relevant to the course materials (e.g., draw a linked list, trace some code, answer conceptual questions). Students must exceed 50% on this test before earning the grade dictated by their point total. The exam served several purposes, hopefully without being too high-stakes. First, the notion of a test is familiar and ideally would motivate procrastinators (though I remain doubtful). Second, it tested 'System 2' expressions of knowledge frequently reviewed during interviews. Finally, it measures the overall effectiveness of the teaching and individual learning.

I do not describe my course design as a template to follow but as a thought piece. While it would be tempting to throw stats around about success rates (nearly 50% earn an A or A- versus less than 20% fail rates), such details are very contextual, and like most classes, the data is neither systematic nor controlled. Instead, I hope to present a radical theory-driven course structure that students generally enjoy and find valuable and allow instructors to consider appropriate changes in their courses.

V. Takeaways

While there are practical measures to take away from small and large coding, the underlying theoretical insights may be the most important. Learning about the brain has drastically altered how I see student struggles. TAMP identifies the previously undefinable benefits experts gain through experience: the skillset described by System 1: automation and intuition. Furthermore, it provides a roadmap to purposefully passing these abilities on to novices. Recognizing implicit learning allows educators to test new ways to encourage System 1 growth. My course design is an example of one such attempt to use theory and research to inform classroom practices.

Many instructors must balance their preferences with the needs of the broader curriculum. Instructors must cover the minimum content in sequenced classes and ideally have similar workloads and outcomes. Students sometimes chafe when the expectations change between classes. Further research is needed to compare the outcomes of different approaches and the impacts of moving through multiple systems on students. This work intentionally avoided gathering and analyzing research data, preferring to focus on the connections between theory and practice, but that is a critical next step.

One future research goal would be to track and compare students longitudinally. One limitation of the proposed grading system is the lack of a clear 'ranking'. Students earn points on both effort and ability, and the point totals tend to plateau after students achieve an A. Traditional testing provides valuable research data even if ranking is not a classroom priority. Instructors must balance the time/effort taken to prepare and conduct testing versus the learning opportunities lost to the testing process. The best grading system seems to mix both, but researchers need a window into each for the best insights.

Higher education for computing is experiencing stressors on all fronts. Industry wants more of our students, society wants us to serve more people, entrepreneurs are offering quick alternatives to degree programs, and many professors are still focused on advancing research agendas. It seems that radical changes may be the only approach to tackling these demands. Google executives Eric Schmidt and Johnathan Rosenberg [27] believe that incremental change makes radical improvements difficult to accomplish. Academic freedom blesses most educators with the ability to make radical changes if in a limited sphere of control. But it is a place to start. While my startup effort was substantial, the maintenance remains minimal, and my workload during the term is nearly exclusively helping students. Despite allowing unlimited resubmissions without deadlines, I forgo the available graders. Grading is minimal effort and gives a deeper insight into student progress. Rethinking everything from the ground up with theory as a guide has allowed my students and me to complete more in the same amount of time with more satisfaction and anecdotally better results.

Acknowledgments

This work is a blending of years of training intersecting with practical realities of teaching. The theoretical side comes from many long conversations with Sean Brophy, Ruth Streveler, and Micheal Loui, among others. Equally important is the patience, feedback, and forgiveness of my students. They have struggled through technical innovation, well-intended yet less well-thought-out assignments, and demanding work. Their perseverance and accomplishments drive me to work hard for the next term.

References

[1] S. Hambrusch, "NAS Report Investigates the Growth of Computer Science Undergraduate Enrollments," *Computing Research Association*, 2017. [Online]. Available: https://cra.org/crn/2017/11/nas-report-investigates-growth-computer-science-undergraduate-enrollments/. [Accessed: 05-Apr-2022].

[2] "Computer Science for All," *National Science Foundation*, 2020. [Online]. Available: https://beta.nsf.gov/funding/opportunities/computer-science-all-csforall-research-and-rpps. [Accessed: 05-Apr-2022].

[3] "STEM + Computing K-12 Education (STEM+C)," *National Science Foundation*, 2018. [Online]. Available: https://beta.nsf.gov/funding/opportunities/stem-computing-k-12-education-stemc. [Accessed: 05-Apr-2022].

[4] M. Cooney, "IT graduates not 'well trained, ready to go,'" *InfoWorld*, 2012. [Online]. Available: https://www.infoworld.com/article/2623183/it-graduates-not--well-trained--ready-to-go-.html. [Accessed: 05-Apr-2022].

[5] "Why Do Tech Companies Not Hire Recent Computer Science Graduates," *SynergisticIT*, 2021. [Online]. Available: https://www.synergisticit.com/tech-companies-not-hire-computer-science-graduates/. [Accessed: 05-Apr-2022].

[6] L. Tikayatray, "This Is Why Most Programmers Fail to Become Senior Developer," *Level Up Coding*, 2022. [Online]. Available: https://levelup.gitconnected.com/this-is-why-most-programmers-fail-to-become-senior-developer-143bc8c1342d. [Accessed: 05-Apr-2022].

[7] L. R. Squire and E. R. Kandel, *Memory: From mind to molecules*,

vol. 69. Macmillan, 2003.

[8] S. Wiedenbeck, "Novice/expert differences in programming skills," *Int. J. Man. Mach. Stud.*, vol. 23, no. 4, pp. 383–390, 1985.

[9] S. Wiedenbeck, V. Fix, and J. Scholtz, "Characteristics of the mental representations of novice and expert programmers: an empirical study," *International Journal of Man-Machine Studies*, vol. 39, no. 5. pp. 793–812, 1993.

[10] T. Lowe, "The theory of applied mind of programming," Purdue University, 2020.

[11] E. A. Youngs, "Human errors in programming," *Int. J. Man. Mach. Stud.*, vol. 6, no. 3, pp. 361–376, 1974.

[12] G. Kolfschoten, S. Lukosch, A. Verbraeck, E. Valentin, and G. J. de Vreede, "Cognitive learning efficiency through the use of design patterns in teaching," *Comput. Educ.*, vol. 54, no. 3, pp. 652–660, 2010.

[13] D. C. Berry and D. E. Broadbent, "The combination of explicit and implicit learning processes in task control," *Psychol. Res.*, vol. 49, no. 1, pp. 7–15, 1987.

[14] A. Savage, *Every tool's a hammer: life is what you make it*. Atria Books, 2020.

[15] R. Lister *et al.*, *A multi-national study of reading and tracing skills in novice programmers*, vol. 36, no. 4. 2004.

[16] D. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," Washington, DC, 1985.

[17] M. McCracken *et al.*, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," *ACM SIGCSE Bull.*, vol. 33, no. 4, p. 125, 2001.

[18] M. Lopez, J. Whalley, P. Robbins, and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," in *Proceeding of the fourth international workshop on Computing education research - ICER '08*, 2008.

[19] M. Biggers, A. Brauer, and T. Yilmaz, "Student perceptions of computer science: A retention study comparing graduating seniors vs. CS Leavers," *SIGCSE'08 - Proc. 39th ACM Tech. Symp. Comput. Sci. Educ.*, pp. 402–406, 2008.

[20] S. R. Portnoff, "The introductory computer programming course is first and foremost a language course," *ACM Inroads*, vol. 9, no. 2, pp. 34–52, 2018.

[21] B. A. Becker, "A new metric to quantify repeated compiler errors for novice programmers," in *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 2016, vol. 11-13-July, pp. 296–301.

[22] J. Sweller, "Cognitive Technology: Some Procedures for Facilitating Learning and Problem Solving in Mathematics and Science," *J. Educ. Psychol.*, vol. 81, no. 4, pp. 457–466, 1989.

[23] B. B. Morrison, "Computer science is different! Educational psychology experiments do not reliably replicate in programming domain," in *ICER 2015*, 2015, pp. 267–268.

[24] D. Wood, J. S. Bruner, and G. Ross, "The role of tutoring in problem solving," *J. Child Psychol. Psychiatry*, vol. 17, no. 2, pp. 89–100, 1976.

[25] D. N. Perkins, *Making learning whole: How seven principles of teaching can transform education*. John Wiley & Sons, 2010.

[26] J. S. Bruner, *Toward a theory of instruction*, vol. 59. Harvard University Press, 1966.

[27] E. Schmidt and J. Rosenberg, *How google works*. Grand Central Publishing, 2014.