Purdue University
Purdue e-Pubs

School of Engineering Education Graduate Student Series
School of Engineering Education

10-16-2019

Debugging: The Key to Unlocking the Mind of a Novice Programmer?

Anthony A. Lowe
Purdue University, lowe46@purdue.edu

Follow this and additional works at: https://docs.lib.purdue.edu/enegs

Part of the Engineering Education Commons

Lowe, Anthony A., "Debugging: The Key to Unlocking the Mind of a Novice Programmer?" (2019). *School of Engineering Education Graduate Student Series.* Paper 84.
https://docs.lib.purdue.edu/enegs/84

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

Debugging: The key to unlocking the mind of a novice programmer?

Tony Lowe
Purdue University
West Lafayette, IN
lowe46@purdue.edu

***Abstract*— Novice programmers must master two skills to show lasting success: writing code and, when that fails, the ability to debug it. Instructors spend much time teaching the details of writing code but debugging gets significantly less attention. But what if teaching debugging could implicitly teach other aspects of coding better than teaching a language teaching debugging? This paper explores a new theoretical framework, the Theory of Applied Mind for Programming (TAMP), which merges dual process theory with Jerome Bruner's theory of representations to model the mind of a programmer. TAMP looks to provide greater explanatory power in why novices struggle and suggest pedagogy to bridge gaps in learning. This paper will provide an example of this by reinterpreting debugging literature using TAMP as a theoretical guide. Incorporating new view theoretical viewpoints from old studies suggests a "debugging-first" pedagogy can supplement existing methods of teaching programming and perhaps fill some of the mental gaps TAMP suggests hamper novice programmers.**

Keywords*— *Computer Science, Programming, Education, Debugging, Bruner, Dual process theory

I. Introduction

This research-to-practice full paper suggests a new pedagogical approach to teaching novice programmers. A programmer probably spends more time debugging than writing code, so it follows that learning to debug is equally critical when learning to program. What if debugging is more than a complementary skill, but a more effective way to learning and integrate logic, language, and design? This paper makes a case for a "debugging-first" pedagogy inspired by a new theoretical framework modeling how programmers think, by extension and novices learn. The Theory of Applied Mind for Programming (TAMP), utilizes dual process theory as a replacement model of cognition. Traditional models employ only the 'logical side' of our brain, but TAMP includes intuition and automation as support of reasoning. TAMP leverages the mental representations model described by Jerome Bruner to refine the concept of the notional machine [1]. The next section introduces each theoretical foundation and a brief overview of TAMP before discussing "debugging-first". The goal of TAMP and debugging first is to revisit the conventional wisdom of programming pedagogy and consider new ways to support struggling learners.

II. Theoretical Underpinnings

A. Dual Process Theory

Dual process theory models cognition as possessing two mechanisms of thought: System 1 is fast and automatic where, System 2 is slow and reasoning [2], [3]. System 2 is responsible for mental tasks which require focus, attention, and integration of new ideas. When programmers trace through code to find bugs, they are primarily employing their System 2. System 2, however, possesses limited resources and demands substantial mental energy without support from System 1. Experience yields automation, which provides programmers with both speed and accuracy, even in simple programming tasks [4]. Researchers often describe "learning to program" in terms of understanding concepts and applying logic, with little emphasis on (but not ignoring [4], [5]) the value of building intuition and automating skills. Deliberate reasoning is vital to many aspects of programming. Our reasoning improves with experience because System 1 implicitly automates the tasks we repeat frequently.

System 1 provides fast, automated responses to situations we have repeatedly experienced. Most System 2 reasoning functions efficiently due to System 1 automation based on prior learning. When notices read code, the language centers in the brain activate to provide meaning [6]. Language processing lives in System 1. Until a novice quickly and easily processes the language, it will be difficult for them to tackle more complex aspects of coding [7]. When novices lack programming language understanding they fill in gaps with meaning from their natural language [8]–[10] or might experience *cognitive load* which overburdens their thinking as they attempt to juggle syntax, algorithm, and inputs [11]. Overburdening is a very apt way of describing some novice programmers. Burdened novices sometimes stop working or successively jump to alternative approaches rather than considering their mistakes [12]. Teaching should "stress continuous practice with basic materials to the point that they become overlearned" [4, p. 389] to alleviate cognitive load. "Overlearning" describes the creation of System 1 processes which are quick and 'costless' yet requires significant and deliberate repetition to develop.

Dual process theory challenges the epistemology of "knowing how to program". In many classrooms, success in CS1 is measured by how much students remember about programming, rather than how well they can build programs. McCracken et al. [13] reported it was "students' knowledge, rather than their skills, that enabled them to successfully complete their first-year courses" (p. 134). Becoming a

programmer requires more than understanding, but the application of concepts to design, implement, and test code based on a problem statement. Many educators promote a 'bottom-up' approach to learning [14], eventually building towards the entire skillset. Dual process theory may hint that these approaches risk developing inflexible skills and ones that have no context in the 'whole task' of programming. A learner must 'overlearn' critical skills, contextualized skills within the steps of programming. A well-designed 'bottom-up' approach may promote automation but defers integration of skills in authentic contexts until much later in the process.

B. Bruner's Mental Representations

Dual process theory better models cognition but lacks learning strategies and insufficiently describes the interplay between the two Systems during complex tasks. Many theories describe learning, but Bruner's representational theory offers a model that closely aligns to dual process theory. Bruner proposed three mental representations: enactive, iconic, and symbolic [15], [16]. Enactive representations form implicitly based on our experiences. We develop enactive representations throughout childhood, and each time we see the behavior of a computer. When our smartphone automatically corrects our typing, it sets the expectation that computers 'know what we are saying'. It might come as a shock later when the compiler is unable to understand and correct for our mistakes! Enactive representations are unconscious, context, bound, and automatic; just like System 1. They hold our intuitive prediction of how the world behaves. Enactive representations are tied to action yet are initially context-bound (only relate to a specific experience). They often disagree with the complex rules and concepts which govern the world forming misconceptions. Enactive representations form the foundation of our behavior and must reconcile with other forms of reasoning.

Iconic representations form as we reconcile the experiences captured in enactive representations. Iconic representations are associated with imagery, but not all images are iconic representations. Iconic representations are personal, formed by finding patterns in our experiences. A programmer who started their career in banking may view the proper design very differently than one who started in auto insurance. Bankers think in terms of immediate transactions, where insurance claims take days or weeks to reconcile. The way they see problems solved influences the way they will think to solve future problems. The iconic representations which form for each programmer come from their shared experience, but in some cases may have a common ground which we want to represent in standard terms. Our spoken language provides such a medium for sharing ideas, but we also use diagrams (e.g., flowcharts or UML) or the programming language to capture details efficiently. When iconic representations are shared between people by a formal drawing, they require each party to form a shared mental representation of the same idea. Bruner places formal shared understanding stored in his final representation, symbolic.

Symbolic representations form around shared rules and concepts. Programming languages are the ultimate symbolic representation; people use language as a symbolic shorthand to command hardware (also constructed by people). The symbolic representations in a programming language embody its author's experience, which may or may not align with our own experience. For instance, some novices initially struggle with loops treating "the WHILE loop as if it generated some kind of interrupt" where "the loop could terminate at the very instant that the controlling condition changed value" [10, p. 69]. Instructors certainly lectured novices on the behavior of the while statement, but the new symbol "while" conflicted with not only with the English word but the enactive and iconic representations that link the word "while" to its experienced behavior. Learning a language by memorizing its rules risks tying that knowledge to unrelated experience, or worse no experience.

Symbolic representations risk becoming inert unless tied to experience. In childhood, we learn first from our enactive experiences building iconic understandings which bind to symbols (words, drawings, signs) we learn from our loved ones and eventually school. Programming instruction tends to flip this sequence, starting by memorizing syntax rules followed with small disjointed examples. Bruner notes that learning from symbolic representations is possible, but "the learner may not possess the imagery to fall back on when his symbolic transformations fail to achieve a goal in problem solving" [17, p. 49] (emphasis mine). For Bruner, experience and knowledge must be tied together, which occurs in iconic representations. CS education literature is full of examples where students can answer questions about programming, but fail to apply that knowledge when most needed [13], [18]–[20]. Bruner's representations, in conjunction with dual process theory, can be used to enhance an existing mental model of programming, the notional machine.

C. The Notional Machine

Du Boulay, O'Shea, and Monk [1] described the notional machine as a programmer's mental model of a programming language. Every programmer forms their unique notional machine for the language they are using, which allows the mental execution of source code. The notional machine helps novices to read, trace, design, or write code [21]. Sorva [22] also refers to the notional machine as "programming dynamics". A programmer must be able to mentally model the execution of code at times to plan algorithms and predict the results of the written code.

Researchers have suggested many strategies for developing the notional machine, with limited success. Some suggest simple languages with clear syntax and limited ruleset are easier to learn [1], [23] since every language includes hidden actions that "have to be inferred by the novice unless special steps are taken" [1, p. 238]. Instructors should shield learners from unnecessary details while making the "hidden inner state" visible to show the connection between language and action. The notional machine seems to require both an understanding of the language (symbolic) tied to 'hidden behavior (enactive) yet without a clear role for the iconic representation Bruner describes as being critical for building problem-solvers. The next section looks to expand the notional machine to address this gap in learning.

D. The Applied Notional Machine

The Applied Notional Machine (ANM) reimagines the notional machine under Bruner and dual process theory. Programming knowledge is not a static schema of facts, but a mix of concepts and skills applied in dynamic novel ways. Intuition gives experts hints to solve problems based on past strategies [24], which the rational mind applies to details of the problem at hand. Using Bruner's representations (Figure 1), an experienced programmer has learned the syntax and semantics (symbolic), formed automated processes to read, write, and execute code (enactive), and can dynamically form a design/algorithm (iconic) utilizing the learned symbolic and enactive representations.

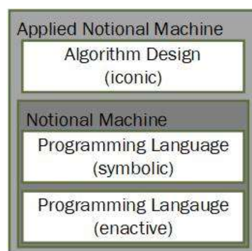

Fig. 1. The Applied Notional Machine

Each representation forms over time and continues to grow through new and varied experiences. Novices will struggle as they neither hold perfect understanding syntax, nor the experience which breeds automation and lack the repertoire of example solutions from which experts draw. The notional machine ignores the role of algorithm and context, which presents a challenging for novices [25]–[28]. The ANM provides a way to model the gradual transition from a 'monolithic' look at language and algorithm into discovering the function of language feature independent from contextualized use. The improved granularity of the ANM hints at both ways of teaching and assessing maturity in learners. Novices need to learn more than facts about the language to create enactive representations. But they also need to experience similar but varying code samples which help form the generalizations stored in iconic representations. The ANM provides a foundation for 'debugging-first' as a way of establishing and integrating each of these three representations

E. Theory of Applied Mind for Programming

The Theory of Applied Mind for Programming (TAMP) provides a model of the cognition of programmers using dual process theory and Bruner as a foundation[1]. An individual uses a theory of mind to "impute mental states to himself (sic) and others" [29, p. 515]. While there are undoubtedly infinite variations on how people program, the use of dual process theory, Bruner's representations, and the notional machine provide a vocabulary to describe the common mental structures that enable programming competency. Dual process theory more effectively explains how experts acquire and utilize knowledge. Bruner's representations provide a model for how established skills and emerging information are blended to produce working ideas. Programming demands interaction of very complex ideas in creative ways, often tacitly. When programming challenging tasks with unclear heuristics, computation mimics cognition using artificial neural networks [30], [31], yet we often attempt to teach the same brain by defining programming heuristics. TAMP attempts to differentiate logic-based programming tasks versus once that are driven by tacit knowledge.

TAMP provides a model of learning based on new epistemological definitions of programming. Reading, parts of writing, and mentally executing code process in System for experienced programmers and only form with practice. Drilling activities alone, however, risk building skills which do not transfer when needed. A novice who uses an integer for every numerical variable may be doing so out of habit rather than evaluating the needs of the problem at hand. We saw earlier, Bruner suggests knowledge integrated with experience better supports problem-solving [17]. Automated skills only aid in creative tasks when they link a variety of experiences and the conceptual knowledge defined in symbolic representations. Dual process theory helps segregate subject mastery into essential skills and concepts, while Bruner's model helps bind knowledge to applied uses like analysis, design, testing, and debugging, to name a few.

TAMP suggests shifting the start of programming education away from the programming language. Programming languages are essential, but not the central skill in programming any more than literacy is the central skill in being an enthralling storyteller. Knowing the rules of syntax and semantics is like memorizing words of a second language without understanding the culture. To be a programmer, you must learn common patterns for solving problems, as good storytellers call upon archetypes and cultural icons to tell compelling stories. Young children learn the essence of a good story long before receiving formal language instruction. One of the most enduring memories of my toddler daughter is her spontaneously exclaiming "ICE" from the back seat as we sat at a gas station. She was years away from formal schooling yet used her "ABC's" and experience listening to hundreds of stories in recognizing the big red word on the side of a metal machine selling ice. TAMP suggests debugging may provide a critical pedagogy in developing a robust ANM and a less stressful path to learning to program.

III. Debugging as a Critical Pedagogy

It little matters how well students understand a programming language or even can write snippets of code if they are unable to complete, and retain the full programming skillset. Many pedagogical approaches in computing education (CEd) provide much-needed scaffolding to support students through the myriad of details required to start a basic program. These support mechanisms help get struggling novices through their first course, but in the end, a significant percentage of students still fail to meet basic expectations [13], [20]. Perkins [32] provided strategies for addressing these gaps as one of his seven principles of teaching: "playing the whole game". Playing the whole game suggests students see problems in authentic contexts and applying skills in meaningful ways. Debugging may be the last thing programmers do in the development process, but starting

[1] In this paper, TAMP is simplified as the amalgamation of the theories in this section but the full theory is forthcoming as part of my dissertation. The 'debugging-first' pedagogy is an offshoot of the pedagogical suggestions offered by TAMP, which also looks to provide aid in research.

by teaching debugging provides an authentic look at the whole game of defining, designing, writing, testing and of course fixing code. The "whole game" approach battles the perception that CS1 is only about learning a language[13], [33]. This section investigates the limitations of tracing as a "small game", uses TAMP to identify potential gaps in knowledge base on past studies, and describes how debugging better manages complexity and builds the missing skills novices require to excel at programming.

A. Authentic Experiences in Programming

Programming pedagogy strategies have bounded past stodgy lecture, but students still struggle to write whole programs despite improvements in conceptual understanding. McCracken et al. [13] listed problem-solving as the major limitation in novices' ability to create programs even after completing one or more courses on programming. They believed instructors do teach the entire process of software development, but students fail to employ the keys steps of defining a problem. Under traditional models of cognition, "being told" sets the expectation that the novice will use new information the next time they encounter a problem. Bruner suggests forming iconic representations is critical in problem-solving. Iconic representations are the integration of knowledge and experiences and are unlikely to develop from a lecture on strategy alone.

The ANM suggests designing an algorithm to solve a problem is guided by intuition and supported by reasoning. Novices must blend facts about a language with examples they have previously encountered, and do this best when this information is "overlearned". Before a novice acquires automated skills and forms a library of intuitive design approaches, their knowledge must be actively recalled by System 2. Researchers describe novices knowledge under these circumstances as 'fragile' [19]. Lister et al. [18] consider training a critical skill in building and measuring programming maturity but noted its 'fragility' in participants. Insufficient maturity and support from System 1 may be the root cause of the fragility [34]. Building mature iconic representations seems not only to be required for problem-solving but in precursor activities as well.

Computing educational research (CEdR) offers many approaches to introducing novices to complex ideas with varying levels of 'whole game' support. Literature describes worked examples [25], [35], [36], tracing [18], [37], [38] Parsons problems [28], [36], among others, but with little widespread, or in some cases even consistent, reports of long-term success. Each of these approaches focuses on a 'part of the game' and sometimes using less than authentic methods. A professional programmer may employ mental tracing, but most often pairs this with computer-supported log files or debugging. TAMP's alternative view of cognition suggests teaching methods that may feel less direct may actually construct vital mental structures for novices.

1) Why the 'Small Game' Fails

A common strategy for managing with complexity is decomposition. Instructors decompose complex concepts into simpler or less intertwined ideas and hope by dividing it eases the student's ability to concur the subject. Educators isolate pieces of a topic by carefully constructing 'scaffolding' [39], which focuses student work on the target ideas and completes unrelated pieces of problem-solving. Programming books typically introduce one language construct at a time using a nearly universal ordering of subjects [14], [40]–[43]. Assignments isolate constructs for demonstration purposes to drive conceptual knowledge (symbolic) but provide few authentic examples (enactive) the novice will encounter when it comes time to solve problems in code. Examples artificially imagine algorithms with no useful feature other than showing a specific function. Variables named "x" and "temp" dominate much of the sample code. The lack of specificity and context clues reduces the production of the iconic representations vital for deciphering patterns. A mature ANM needs conceptual understanding, but equally vital is the intuition only constructed within authentic contexts.

Tracing is a common activity in programming education. Tracing provides interactive code examples to execute and predict the outcome mentally. Tracing continues to be a standard pedagogical tool and research topic, despite students distaste for the activity [12], [22] and research showing it may [28] or may not [37] help students write code. Tracing, done well, seems a perfect method for maturing the ANM, as it demands exercising conceptual understanding, form mental models of the algorithm, and experience the code's execution. The trouble with tracing is twofold: 1) without engaging the computer, it is less authentic and may not invoke shared iconic imagery, and 2) it can be accomplished line-by-line [38] circumventing the creation of deep mental models of algorithms.

Educators may hold similar misconceptions about novices that novices hold about computers. Pea [8] described the "superbug", where novices attribute the computer with greater understanding due to its sometimes clever results. Computers process code one line at a time, but do not form greater meaning from the process. Some effort has been started to teach novices to 'reverse engineer' meaning [25], [26], but TAMP suggests this is an implicit skill best learned within the 'whole game'. The biggest obstacle in tracing is the lack of built-in feedback. How does the novice know if they traced successfully and where they went wrong? It is understandable why instructors wish to avoid using the computer until novices are more proficient, but the cost is authenticity, which may diminish the quality of learning.

2) Playing the Whole Game

Debugging provides an authentic context for many contributory programming skills such as tracing. While debugging a programmer must form and manage numerous mental representations, as shown in Figure 2. Tracing short-circuits a number of these representations less by scaffolding them than obfuscating them. Most tracing problems ignore the very existence of a larger design, motivating problem, and even specific test case which inspired the inputs for the trace. Debugging, even if scaffolding the learner with the details of these representations, contextualizes tracing within the larger process.

Debugging provides intrinsic motivation to tracing, that provides immediate feedback. Tracing problems must hide the correct answer to test the learner. Debugging provides the

answer, so the learner knows their trace is incorrect until they can produce the same answer. Successful debugging goes further in demanding the novice to form a mental model of the actual and expected execution and reconcile the differences.

Moving from tracing to debugging may build 'the right types' of mental models but at the cost of added difficulty. In tracing the learner can at least guess and answer and be finished, where debugging goes on until solved. Novices can be scaffolded to start debugging using the same strategies for teaching coding.

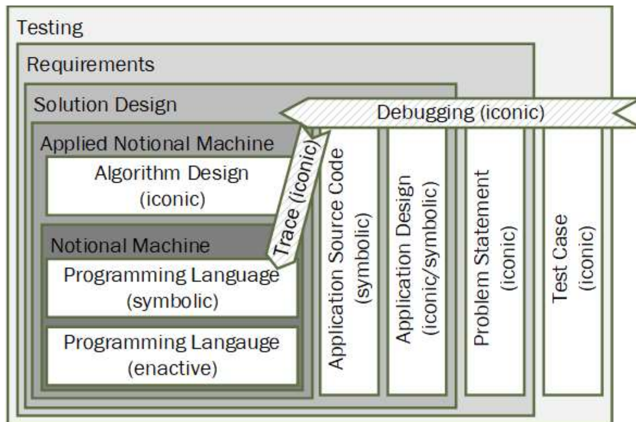

Fig. 2. Mental Representations in Debugging vs. Tracing

Worked examples demonstrate writing code [44], which easily translate to debugging. Debugging examples can present strategies for debugging (also tacit knowledge) where the demonstration follows an expert as they trace variables comparing expected and actual results. Debugging examples would be markedly easier than writing code as they can ignore syntax outside any changed lines! Debugging worked examples can instead promote forming high-level mental models by describing the test case, significant functional steps, and annotating assumptions. At some point, a novice will need to acquire the details of the language, but their first experience can be less stressful and more closely resemble the authentic tasks of professional programmers.

B. Evidence from Debugging Literature

CEdR includes streaks of interest in debugging reporting common themes yet resulting in seemingly little impact in the classroom. This section looks at the sometimes-contrary findings from various studies on debugging through the lens of TAMP.

1) TAMP and Debugging

Some researchers point to misconceptions as a roadblock in learning to debug. Many CEd researchers seek misconceptions to aid novices [10], [21], [45], [46]. Identifying misconceptions may not help novices, however. Ben-Ari [47] counters that "merely listing misconceptions is fruitless; a description of the underlying model and a prescription for constructing a modified one must be given" (p. 258). Ben-Ari describes building the same type of model described in the ANM, to encompass the algorithm and language. McCauley et al. [48] summarized that other research "[debunked] the notion that misconceptions about language constructs are the cause of most bugs… educators can help novices by making them aware of the types of non-construct-based problems they may experience" (p. 70). Unfortunately, this advice is only partially helpful. If addressing misconceptions about a language does not universally fix problems, will address misconceptions about the construction of algorithms?

TAMP supports Ben-Ari's view that 'knowing' misconceptions is secondary to correcting the mental representations that spawn them. Think of a misconception as a bug in a novice's mind. As with any bug, we hope a single process contains the bug, and a single change corrects all instances. Unfortunately, our brain does not structure knowledge so cleanly. A misconception can only be explicitly corrected when newly learned and when no prior knowledge exists. Since most action is supported by System 1, only varied repetition can replace the old habits and triggers. Telling a novice their mistake in a for loop may fix that code, but may not fix their next for loop, or the same type of mistake in while loops or recursion. Debugging activities may prove slightly more transferrable as they start with the nature of the problem in which debugging occurred, not the surface details of the solution. Being aware of a problem does not correct "bad habits" which require replacement through better enactive representations.

A programmer's ability to build robust mental models seems to be an indicator of success. Katz and Anderson believed "a person must first come to understand or have a representation of the device being repaired" [49, p. 352] to troubleshoot any problem. They compared the time participants took to debug their own versus another person's LISP code. People debugged their code significantly quicker, but only because many failed to find bugs in other people's code. When participants found bugs in their peer's code they were only slightly slower (statistically insignificant) than their own. When people are debugging their code, they not only have a mental model of their anticipated behavior, they also hold one for their test cases. Many of the participants could not derive sufficient test cases for their peer's implementation in the 20-minute time limit for the activity, yet when the program was similar to one they have worked on previously, they did better. Forming and holding mental models proved a significant factor in the speed of debugging.

A programmer forms the same types of mental models in each phase of programming. Vessey [50] captured the strategic goals of debuggers:

- Determine problem
- Gain familiarity
- Explore program structure and function
- Repair the error (p. 471)

Vessey's steps form the same mental representations from Figure 2 that programmers form in constructing code. In fact, the "difficulty of debugging is not in repairing the error but rather in the earlier stages of the troubleshooting process" [48, p. 78]. The required skills Vessy and McCauley et al. describe creating a "chicken-and-egg" scenario for novices. To be a reliable debugger they must be good at each step leading up to

debugging, but without debugging, how do you correct mistakes in design and code?

2) Using Debugging to Teach Programming

Starting with debugging seems counter-intuitive as a way of teaching programming. How can someone fix problems in a language they do not understand? Understanding the programming language does not mean you create code any more than learning Greek prepares you to challenge Socrates or Plato in philosophy! Experts, tend to forget that novice is unable to see the nuanced between language constructs, it is all strange and new. They tend to assume one skill translates right into the next, but Ahmadzadeh et al. [51] reported a substantial number of their "good programmers" were not "good debuggers". Being a programmer requires the application of knowledge, not just remembering.

A debugging pedagogy should promote building the mental representations required to understand a software solution. The 'bottom-up' approach attempts to develop mastery of basic skills before integration. Debugging-first offers a 'top-down' approach that focuses on building understanding contextualize skills. Table 1 shows a series of activities focused on building the representations from Figure 2.

TABLE I. ACTIVITIES TO FORM MENTAL REPRESENTATIONS

Activities	Representation
Define test cases and specific test scenarios	Test Case
	Problem Statement
Debugging example isolating the area of concern	Application Design
Tracing code execution to locate the error	Source Code
Fix the error	ANM

Defining test cases forms and tests understanding of the problem as the novice must consider helpful inputs paired with expected outputs. Instructors can provide full working systems in which to gain hands-on experience with the solution, seeded with strategic bugs. Before 'mundane tracing' even beings, the novice is motivated to solve a problem, not seek an answer. Finding a bug requires the novice to investigate the implementation, deriving meaning, not simply following code line-by-line. The problem may demand close tracing of inputs to seek the failure, but the goal state is known, and always available to provide feedback. Best of all, the novice can do this with a live system, building experience into the 'hidden machine', rather than hiding the computer to prevent 'cheating'.

Debugging pedagogy can utilize the same scaffolding approaches as writing code. Many classrooms start with a "hello world" task where the instructor guides novices through building a tiny example to introduce the toolset and produce a working if rather a useless program. Novices quickly discover the finicky syntax, the compiler is their enemy, and it will be a long time before they get close to the cool things they hope to be able to do with code. Starting with debugging allows novices to experience robust, complex applications immediately yet can still narrow down to small specific features. Worked examples can introduce the process followed by partially completed examples [52] where the novice can perform any step in the process. Coding cannot progress without mastery of syntax, but debugging can jump between defining test cases, tracing, isolating the wrong line, or correcting individual lines of code. Many problems can use the same scaffolding by introducing new bugs.

For instance, a list of names may be skipping the first item when displayed (e.g., the loop goes from 1..length rather than from 0..length). This bug investigates the rules of how the language stores data in an array/list and how to construct loops. A follow-up can change the requirement present the list from Z-A rather than A-Z demonstrating how loops can navigate a list in either direction. A further bug can be introduced in the creation of the list to break the proper sorting. Each exercise builds upon the same mental model of both the problem statement and intricate architecture that many novices may not even see through CS1. Since examples progressive introduces novices to the complexity of the solution without requiring mastery of each construct involved, or even proper construction of subprograms, yet novices come to see complex applications as the norm, not the exception in code examples. From the beginning, pedagogy novices how to navigate and catalog complex code.

If tracing is a critical pedagogy, debugging provides a stronger form of tracing. Debugging motivates tracing as a natural activity rather than the unpopular one noted earlier. Debugging provides tracers with better feedback, they know the goal state, but also helps to address mental blocks in our cognition. Clancy [45] describes the impact of the confirmation bias amongst programmers. Kahneman [2] describes confirmation bias, when we "seek data that are likely to be compatible with the beliefs [we] currently hold" (p. 81). If we believe a piece of code is well built, it takes strong evidence to convince us otherwise. Novices may struggle to 'see' their mistakes when tracing on paper alone. Katz and Anderson [49] noted that students "seem to be working from a mental representation of the program as well as the listing of the program itself" (p. 363). Kahneman describes System 2 as being "lazy", meaning it may not double-check things System 1 'knows' to be true.

Experts suffer from, but learn strategies to combat confirmation bias. They use debugging tools, or even simple logging to validate their flaws in their mental model. Novices tend to use "the output data in developing a hypothesis about a bug; few reported using the input data" [48, p. 76]. Novices both work a potentially flawed mental model, and do not fully trace from input to output even when debugging! Debugging allows not only for the computer to be available to provide granular data at each step, but to encourage teaching strategies that combat our confirmation bias. Coaching good tracing habits in debugging activities can motivate students and teach them how to mature their mental models with data from execution.

Debugging also provide a mechanism of forming a notional machine, particularly as described within the ANM. Initially, a novice struggles to separate language from the algorithm. The ANM says instruction builds the symbolic aspects of the notional machine, and practice builds the enactive, where the iconic forms through variation. The example just described provides a means of merely introducing such variety while managing complexity. Anderson and Jeffries [53] note, "error

frequency is affected by the complexity of the components of the problem to be solved" (p. 129). When the majority of examples novices first see are isolated and simple their notional machine will be limited. It takes the rare individual who can take two very complex ideas (like loops and conditional statements) and blend them in precise ways without seeing examples of their combined use first. In school, I was a proficient saxophonist, yet despite being adept at reading music, the piano confounds me. I can understand the basics of melody a single hand, but the saxophone did little to prepare me for each hand independently producing chords. Expertise at small aspects of coding may not translate into the process of programming without explicit guidance.

Programming requires mastery within both System 1 and 2, which changes the nature of 'knowing'. Instructors cannot assume that knowledge is consistently applied once 'learned'. Anderson and Jeffries [53] observed: "a given subject will not make the same error on all problems of a given kind; he or she is much more likely to give different kinds of responses to equivalent problems." They add to the many descriptions of fragile knowledge, which TAMP begins to explain and address. Intuition drives maturity in novices more than memory. Building robust intution requires diverse experiences in authentic programming actiities. Debugging tasks allow instructors to target lessons within complex code by baking in strategic bugs. These lessons cater to the 'hidden machine' and build the System 1 processes/enactive representations required to support complex thinking. Just like reading storybooks to children, novice programmers can engage with complex ideas long before they are mature enough to produce such stories themselves.

C. Building Debugging Pedagogy

The teaching through debugging shifts the early workload onto the instructor. While any pedagogy that provides scaffolding by definition shifts work from student to instructor, debugging-first means instructors are building more extensive, complex applications rather than simple, isolated exercises. An entire working application (or several) needs constructing, before creating variants to place interesting, but not too interesting, bugs. The code should ideally display the preferred style and design as each example is unconsciously adopted as 'correct' by System 1. TAMP can help to guide pedagogy in aligning with how people think and learn.

1) Selecting a project

Selecting the appropriate domain for coding examples is trickier than it may seem. McCracken et al. [13] thought a calculator would be a simple challenge as any college student taking a programming class must have used a calculator, yet ironically Du Boulay et al. [1] warned twenty years prior that most people have little understanding of the inner workings of calculators. McCracken et al.'s students did not demonstrate an in-depth understanding of any part of the problem space, and some struggled even to start. Any chosen domain could alienate some group of students as not everyone likes sports, music, has held a bank account or shared cultural experiences. Debugging-first alleviates some of these difficulties since it starts with testing a full application to learn about the domain. The first task of the professional programmer is familiarization with the domain long before considering technology. When novices start with testing a system, nearly any domain is acceptable. The choice of the project should focus on creating a testable interface that includes the desired language constructs, algorithmic approaches, and common bugs covered in the target lessons.

2) Inserting bugs

The application presented to novices should represent exemplary code occasionally 'corrupted' with intentional bugs. In the earliest exercises, the nature of the bugs is less critical than strategies to single out discrepancies and bugs that demonstrate the nuances of language constructs. An excellent example of an early lesson might be how temp variables can be incorrectly applied to swap two values. Swapping demands a mental model of variable storage, assignments, and the order of execution in a nontrivial way. The variables can even reside within other logic so long as that logic is intuitive to read, as shown in Figure 3. Figure 3 presents an example swap method includes a simple bug caused by the order of the assignment statements hidden within other logic.

```
private static void swapOwnership(VehicleSwapAgreement agreement)
{
  if (agreement.isOwner1Registered() && agreement.isOwner1Registered())
  {
    if (agreement.isValidTitleVehicle1() && agreement.isValidTitleVehicle2() ||
      agreement.isBothPartiesWaveTitles())
    {
      LegalEntity tempOwner = agreement.getVehicle1().getOwner();
      agreement.getVehicle2().setOwner(tempOwner);
      agreement.getVehicle1().setOwner(agreement.getVehicle2().getOwner());

    } else
    {
      agreement.setStatus("Title Issue");
    }

  } else
  {
    agreement.setStatus("Registration Issue");
  }
}
```

Fig. 3. Example of a simple bug

A simpler exercise might present just the three 'swap' lines, but the debugging-first approach manages the complexity by ensuring the novice understands the context. Starting with a user experience teaches the novice the domain and builds expectations for what the code 'should do'. Before taking on the 'bugged' test case, the novice should witness positive test cases where the swap successfully occurs. Once the intent of the code is understood, and the focus can turn to the implementation and bug. The novice tackles bug within an authentic if scaffolded, context where the 'complexity' of the object-oriented design is 'normal'. The carefully named variables and methods become as readable as a paragraph, and the exact nature of each construct is as easily abstracted away as the mechanics of a print statement.

Bugs should drive authentic interest and experience in wanting to observe the execution of code. Why do both vehicles end up with one owner? If I am the owner without a car, it becomes a big concern! An even better exercise allows for examples that can be easily manipulated to see how the execution changes as a result. Novices can tweak individual lines of code and test their suspicions. If they fail badly, they can revert to the full buggy version, rather than the tendency of

novices to start from scratch when they get stuck [12]. Each successive attempt refines their mental model, rather than starting fresh. Debugging exercises provide self-regulated feedback. The desired outcome is known (from the test case), and the task continues until the code meets that expectation.

3) Supporting Development

One of the critical roles of a programming instructor is monitoring and supporting the progress and development of novices. The literature within this paper only touches the surface of the ways novices thinking can go awry. The power of programming comes from its unlimited potential, yet this also applies to the ways of 'doing it wrong'. As a group, researchers classify novice errors, but this does little to help the individual without guidance on their mistake. Left to their own devices, novices often fall to mimicking or modifying existing samples, equating learning to completing projects. Long term success, particularly as a debugger, may depend on learning to form robust mental models of the problem and solution [48], [49], [54], [55]. Debugging-first exercises focus on understanding, not results. It may be easier to cheat if the task stops with correcting the bug, but a simple remedy is to require the learner to explain the cause of the bug and the rationale of the fix. Being asked to explain code makes learning personal. It is not about producing code by any means, including unintentional plagiarism, but about your personal understanding of the process, both technical and domain.

Instructors can leverage debugging to discuss requirements, design, writing code, and testing to place the programming language as a means, not an end. Language skills are essential, yet introduced too early can promote anxiety [42]. By starting with debugging, learners achieve small wins that appear significant because of the broader context. Rather than printing a trivial message, they prevented a person from losing their car! By showing complex examples like Figure 3, new constructs become familiar even before they are fully understood. Novices become accustomed to reading code as functional chunks, discovering patterns of use, while building an understanding of language constructs. By starting with purpose, as described by domain and test cases, the top-down approach becomes a more natural and comfortable task compared with explaining the purpose of a handful of decontextualized lines of code. Interpreting code is "an accurate indicator of debugging strategy and speed" [48, p. 81] and TAMP suggests also contributes to the intuition required to be a successful designer.

IV. Next Steps

This paper proposes a debugging-first pedagogy as a way of helping struggling novices. Theory drives the concept and application of debugging-first, yet no matter how well-supported still requires empirical evaluation. Next steps include creating full 'debugging-first' exercises and measuring student responses. Such a study should ideally compare against traditional pedagogy using experimental, pseudo-experimental, or single-subject approaches. Statistical comparison alone may or may not reveal the nature of the intervention using conventional measures of classroom progress. The inevitable goal is to promote the full skillset of programming and a metacognitive understanding of the process. If these measures do not already exist as quantitative measures, a corresponding mixed-methods approach could capture quantitative measures of progress. Specifically, measures of self-efficacy, motivation, frustration, and attitudes towards learning would provide interesting comparisons on the 'non-cognitive' impacts of each approach.

Acknowledgment

This paper is an offshoot of my dissertation research, focusing on the formation of TAMP. I have significantly benefited from the advice of my advisor and committee: Drs. Sean Brophy, Michael Loui, and Ruth Streveler. I also want to thank my wife for continued patience and apologize to my dogs for many missed walks.

References

[1] B. Du Boulay, T. O'Shea, and J. Monk, "The black box inside the glass box: presenting computing concepts to novices," Int. J. Hum. Comput. Stud., vol. 51, no. 3, pp. 265–277, 1981.

[2] D. Kahneman, Thinking, fast and slow. Macmillan, 2011.

[3] J. S. B. T. Evans and K. E. Frankish, In two minds: Dual processes and beyond. Oxford University Press, 2009.

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

[5] E. Soloway, "Learning to program = Learning to construct mechanisms," Commun. ACM, vol. 29, no. 9, pp. 850–858, 1986.

[6] J. Siegmund et al., "Understanding understanding source code with functional magnetic resonance imaging," pp. 378–389, 2014.

[7] 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.

[8] R. D. Pea, "Language-Independent Conceptual 'Bugs' in Novice Programming," J. Educ. Comput. Res., vol. 2, no. 1, pp. 25–36, 1986.

[9] J. Bonar and E. Soloway, "Preprogramming knowledge: A major source of misconceptions in novice programmers," Human-Computer Interact., vol. 1, pp. 133–166, 1985.

[10] B. Du Boulay, "Some Difficulties of Learning to Program," J. Educ. Comput. Res., vol. 2, no. 1, pp. 57–73, 1986.

[11] B. B. Morrison, B. Dorn, and M. Guzdial, "Measuring cognitive load in introductory CS," Proc. tenth Annu. Conf. Int. Comput. Educ. Res. - ICER '14, pp. 131–138, 2014.

[12] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons, "Conditions of Learning in Novice Programmers," J. Educ. Comput. Res., vol. 2, no. 1, pp. 37–55, 1986.

[13] 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.

[14] R. Lister, "Toward a Developmental Epistemology of Computer Programming," Proc. 11th Work. Prim. Second. Comput. Educ. ZZZ - WiPSCE '16, pp. 5–16, 2016.

[15] J. S. Bruner, "On cognitive growth," in Studies in cognitive growth: A collaboration at the center for cognitive studies, Wiley and Sons, 1966, pp. 1–29.

[16] J. S. Bruner, "On cognitive growth II," in Studies in cognitive growth: A collaboration at the center for cognitive studies, Wiley and Sons, 1966, pp. 30–67.

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

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

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

[20] I. Utting et al., "A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations Ian," in Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports, 2013, pp. 15–31.

[21] J. Sorva, “Notional machines and introductory programming education,” ACM Trans. Comput. Educ., vol. 13, no. 2, pp. 1–31, 2013.

[22] J. Sorva, “Reflections on threshold concepts in computer programming and beyond,” Proc. 10th Koli Call. Int. Conf. Comput. Educ. Res. - Koli Call. ’10, pp. 21–30, 2010.

[23] J. T. Khalife, “Threshold for the introduction of programming: providing learners with a simple computer model,” 28th Int. Conf. Inf. Technol. Interfaces, no. September 2006, pp. 71–76, 2006.

[24] J. C. Spohrer and E. Soloway, “Novice mistakes: are the folk wisdoms correct?,” Commun. ACM, vol. 29, no. 7, pp. 624–632, 1986.

[25] B. B. Morrison, “Computer Science Is Different ! Educational Psychology Experiments Do Not Reliably Replicate in Programming Domain,” pp. 267–268, 2015.

[26] J. Sajaniemi and M. Kuittinen, “An Experiment on Using Roles of Variables in Teaching Introductory Programming,” Comput. Sci. Educ., vol. 15, no. 1, pp. 59–82, 2005.

[27] P. Bayman and R. E. Mayer, “A diagnosis of beginning programmers’ misconceptions of BASIC programming statements,” Commun. ACM, vol. 26, no. 9, pp. 677–679, 1983.

[28] 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.

[29] D. Premack and G. Woodruff, “Does the chimpanzee have a theory of mind?,” Behav. Brain Sci., vol. 4, pp. 515–526, 1978.

[30] R. Sun, P. Slusarz, and C. Terry, “The interaction of the explicit and the implicit in skill learning: A dual-process approach,” Psychol. Rev., vol. 112, no. 1, pp. 159–192, 2005.

[31] R. Sun, E. Merrill, and T. Peterson, From implicit skills to explicit knowledge, vol. 25. 2001.

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

[33] A. Eckerdal and A. Berglund, “What does it take to learn ‘programming thinking’?,” Proc. 2005 Int. Work. Comput. Educ. Res. - ICER ’05, pp. 135–142, 2005.

[34] T. Lowe, “Novice Struggles in two parts,” in ITiCSE’19, 2019.

[35] M. E. Caspersen and J. Bennedsen, “Instructional design of a programming course: a learning theoretic approach,” Proc. third Int. Work. Comput. Educ. Res., pp. 111–122, 2007.

[36] M. Guzdial, “Learner-Centered Design of Computing Education: Research on Computing for Everyone,” Synth. Lect. Human-Centered Informatics, vol. 8, no. 6, pp. 1–165, 2015.

[37] K. Cunningham, S. Blanchard, B. Ericson, and M. Guzdial, “Using Tracing and Sketching to Solve Programming Problems,” 2017, pp. 164–172.

[38] B. Xie, G. L. Nelson, and A. J. Ko, “An Explicit Strategy to Scaffold Novice Program Tracing,” Proc. 49th ACM Tech. Symp. Comput. Sci. Educ. - SIGCSE ’18, pp. 344–349, 2018.

[39] J. S. Bruner, The process of education. Cambridge, MA: Harvard University Press, 1965.

[40] L. Ma, J. Ferguson, M. Roper, and M. Wood, “Investigating and improving the models of programming concepts held by novice programmers,” Comput. Sci. Educ., 2011.

[41] J. Mead et al., “A cognitive approach to identifying measurable milestones for programming skill acquisition,” Work. Gr. reports ITiCSE Innov. Technol. Comput. Sci. Educ. - ITiCSE-WGR ’06, no. May 2014, p. 182, 2006.

[42] B. Shneiderman, “Teaching programming: A spiral approach to syntax and semantics,” Comput. Educ., vol. 1, no. 4, pp. 193–197, 1977.

[43] M. Berges, “Object-Oriented programming through the lens of computer science education,” no. April, 2015.

[44] B. Skudder and A. Luxton-Reilly, “Worked examples in computer science,” in Conferences in Research and Practice in Information Technology Series, 2014, vol. 148.

[45] M. Clancy, “Misconceptions and attitudes that interfere with learning to program,” Comput. Sci. Educ. Res., pp. 85–100, 2004.

[46] A. Mühling, A. Ruf, and P. Hubwieser, “Design and First Results of a Psychometric Test for Measuring Basic Programming Abilities,” WiPSCE ’15, 2015.

[47] M. Ben-Ari, “Constructivism in computer science education,” ACM SIGCSE Bull., vol. 30, no. 1, pp. 257–261, 2004.

[48] R. McCauley et al., “Debugging: a review of the literature from an educational perspective,” Comput. Sci. Educ., vol. 18, no. 2, pp. 67–92, 2008.

[49] I. R. Katz and J. R. Anderson, “Debugging: An Analysis of Bug-Location Strategies,” Human—Computer Interaction, vol. 3, no. 4. pp. 351–399, 1987.

[50] I. Vessey, “Expertise in debugging computer programs: A process analysis,” Int. J. Man. Mach. Stud., vol. 23, no. 5, pp. 459–494, 1985.

[51] M. Ahmadzadeh, D. Elliman, and C. Higgins, “An analysis of patterns of debugging among novice computer science students,” ACM SIGCSE Bull., vol. 37, no. 3, p. 84, 2006.

[52] J. L. Plass, R. Moreno, and R. Brünken, “Cognitive Load Theory,” Americas (Engl. ed)., vol. 32, pp. 10013–2473, 2010.

[53] J. R. Anderson and R. Jeffries, “Novice LISP Errors: Undetected Losses of Information from Working Memory,” Human-Computer Interact., vol. 1, pp. 107–131, 1985.

[54] A. J. Ko and B. A. Myers, “A framework and methodology for studying the causes of software errors in programming systems,” J. Vis. Lang. Comput., vol. 16, no. 1-2 SPEC. ISS., pp. 41–84, 2005.

[55] V. Fix, S. Wiedenbeck, and J. Scholtz, “Mental representations of programs by novices and experts,” in Proceedings of the INTERACT’93 and CHI’93 conference on Human factors in computing systems, 1993, pp. 74–79.