THE THEORY OF APPLIED MIND OF PROGRAMMING

by

Tony Lowe

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Engineering Education

West Lafayette, Indiana

May 2020

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Sean Brophy, Chair
School of Engineering Education

Dr. Ruth Streveler
School of Engineering Education

Dr. Michael Loui
School of Engineering Education

Dr. Geoffrey Herman
Department of Computer Science, University of Illinois

Approved by:
Dr. Donna M. Riley

To my family before (Stan, Donna, Ruby) and after coming to Purdue
(Kelly, Braeden, Maggie, Logan)
To my beloved dogs (Nikki, Jasper, Razzberry, Hank, Goose)
And in memory of my brother Mike

ACKNOWLEDGMENTS

I want to offer my sincerest thanks to my committee for their generous time and support in this endeavor. Dr. Brophy has been a constant companion in the depths of how programmers think and how we can aid in their learning. Dr. Loui's insights into debugging and as a generous test audience have helped to refine not only the quality of the ideas but also their clarity. You have Dr. Streveler to thank for TAMP in several ways. Not only did her offerings on neuroscience plant the seed of the theoretical foundations of TAMP, but she also instilled the value of theory and its powerful contributions. Dr. Herman came in later to the process but brought a combination of theoretical and practical thinking as well as an eager audience by generously letting me test my ideas with his students.

I also would like to thank those who have promoted the ideas of discipline-based educational research. While this work hopefully serves as proof of how much we can learn from educators, psychologists, neuroscientists, and other 'pure science' disciplines, it also may serve as an example of why we need disciplinary crossovers. It may be self-serving hubris, but I believe Engineering and Computing Education researchers serve in the same capacity as engineering disciplines in general – finding practical uses for scientific findings. We need people who understand both education and the concepts of a discipline to make the best use of each. Without the pioneers who took the risk of exploring disciplinary educational research, I would not have come nearly as far.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATED GLOSSARY

Term	Brief definition	Section
accommodation	A mode of learning from Piaget's theories where new information causes some restructuring of what a person 'knows'.	6.1.2
ANM	The Applied Notional Machine (ANM) is a theoretical construct of TAMP using Bruner's representations to add design to the mental representations surrounding the notional machine.	7.5.2.1 7.5
assimilation	A mode of learning from Piaget's theories where a person adopts new information into their existing knowledge on the topic.	6.1.2
Cartesian model of cognition	Rene Descartes famously described human reasoning as pure and separated from the body. His notion lends to a belief that people can think purely rationally and may distort many interpretations of human behavior by overlooking other modes of cognition.	4.1
cognitive load	A formal look at cognitive load measures the burden placed on short-term memory when a person processes a task. Cognitive load theory considers the educational ramifications of overloaded students, but just as often, authors invoke cognitive load simply to describe a complex task.	2.2.3
debugging	The process of identifying the source of an error in programming or other endeavors and making a change that removes that error from the designed product.	9.3.3.2
declarative memory	Memories that we consciously recall and use in our remembrances or reasoning	7.3.1
enactive representation	One of Bruner's representations describing our mental model of actions and experiences	6.3.2.1
episodic memory	A type of declarative memory that strings together facts and events in some timeline.	7.3.1
fragile knowledge	While Section 2.2.1 describes several types of fragile knowledge, the main idea is we sometimes know something but do not reliably recall that information when needed.	2.2.1 5.2.1
iconic representation	One of Bruner's representations that generalizes from enactive and blends with symbolic representations. Bruner categorizes iconic as 'imagery', though TAMP looks to add to this definition in Section 7.4.3.	6.3.2.2 7.4.3
iconic manipulation	The process of restructuring knowledge within an iconic representation to generalize from experience, solve problems, or otherwise make connections between other representations.	7.4.3.3
inner speech	Vygotsky believed that children learn language from others, that leads to egocentric speech, personal utterances, that eventually becomes inner speech, our conscious thought processes.	7.4.3
mental load	Mental load is similar to cognitive load, but mental load allows for 'non-cognitive' factors and focuses on operators completing tasks, often using technology.	6.1.2

misconception	A personalized belief or fact that does not align with a shared concept or belief.	2.2.2
'non-cognitive'	If 'cognitive' reasoning is logic separate from human concerns, 'non-cognitive' involves those human concerns like emotions, motivations, values, or attitudes.	2.2.4
nondeclarative memory	Memories that do not surface in our consciousness but drive much of our thinking and behavior. For instance, physical behaviors, perceptual traits, judgment without conscious rules are all learned, but we do not consciously consider them.	7.3.1
notional machine	A mental model that simulates the execution of the source code of a specific language on a specific computational device.	2.1.2
preoperational	One of Piaget's stages of development where a learner begins to blend their action-based knowledge with reasoning.	2.1.3 6.1.1
prospection	A neuroscience concept of using memories to plan future events.	7.3.2
Schema	A model of memory and knowledge proposed by Piaget but used by many traditions, including cognitive load theory. Schemas group and organize knowledge hierarchically around concepts.	6.1.1
semantic memory	A type of declarative memory for remembering facts.	7.3.1
symbolic knowledge	Information that comes from external sources (e.g., books, lectures) that form the basis of symbolic representations	7.4
symbolic representation	One of Bruner's representations describing facts and ideas that come from outside our experience, generally communicated by another person, perhaps using a unique system of symbols.	6.3.2.3
System 1	The intuitive and automatic mechanism of thinking from dual process theory.	4.2.1.1
System 2	The conscious and reasoning mechanism of thinking from dual process theory.	4.2.1.2
TAMP	The Theory of Applied Mind of Programming, a theory describing the way experts think about programming	1.1 7 9.1
Theory	A way of describing the world that looks to explain the interrelationships between observations and, in its best form, offers some predictive power.	3.1
Tracing	A programming activity where the programmer uses a set of inputs to a sample of code to predict the resulting execution. Tracing is a popular pedagogical activity primarily but has occasional use in professional practice.	2.1.3 5.2.2
Zone of Proximal Development (ZPD)	Vygotsky's theory described how two individuals who perform the same on an assessment can each show vastly different potential in future learning, particularly when given external support (latter named "scaffolding") from a more knowledgeable other.	6.2

ABSTRACT

The Theory of Applied Mind of Programming (TAMP) provides a new model for describing how programmers think and learn. Historically, many students have struggled when learning to program. Programming as a discipline lives in logic and reason, but theory and science tell us that people do not always think rationally. TAMP builds upon the groundbreaking work of dual process theory and classical educational theorists (Piaget, Vygotsky, and Bruner) to rethink our assumptions about cognition and learning. Theory guides educators and researchers to improve their practice, not just their work but also their thinking. TAMP provides new theoretical constructs for describing the mental activities of programming, the challenges in learning to program, as well as a guidebook for creating and recognizing the value of theory.

This dissertation is highly nontraditional. It does not include a typical empirical study using a familiar research methodology to guide data collection and analysis. Instead, it leverages existing data, as accumulated over a half-century of computing education research and a century of research into cognition and learning. Since an applicable methodology of theory-building did not exist, this work also defines a new methodology for theory building. The methodology of this dissertation borrows notation from philosophy and methods from grounded theory to define a transparent and rigorous approach to creating applied theories. By revisiting past studies through the lens of new theoretical propositions, theorists can conceive, refine, and internally validate new constructs and propositions to revolutionize how we view technical education.

The takeaway from this dissertation is a set of new theoretical constructs and promising research and pedagogical approaches. TAMP proposes an applied model of Jerome Bruner's mental representations that describe the knowledge and cognitive processes of an experienced programmer. TAMP highlights implicit learning and the role of intuition in decision making across many aspects of programming. This work includes numerous examples of how to apply TAMP and its supporting theories in re-imagining teaching and research to offer alternative explanations for previously puzzling findings on student learning. TAMP may challenge conventional beliefs about applied reasoning and the extent of traditional pedagogy, but it also offers insights on how to promote creative problem-solving in students.

1. INTRODUCTION

1.1 What is TAMP?

The Theory of Applied Mind of Programming (TAMP) defines a model of thinking and learning specific to the domain of computing education. A theory of mind describes the mental model one being has for the thought processes of another (Leslie, 1987; Premack & Woodruff, 1978). TAMP looks to model the way expert programmers think, and from that model describe the struggles novices face when learning. While computing education researchers have studied people of all ages for nearly a half-century, most of these efforts have paid little attention to theories of cognition and learning. In the introduction to the recently released Cambridge Handbook of Computing Education Research, the editors noted,

> Too much of the research in computing education ignores the hundreds of years of education, cognitive science, and learning sciences research that has gone before us. (Fincher & Robins, 2019, p. 3)

While I started the process of creating TAMP a few years before Fincher and Robin published their observation, TAMP follows their advice by blending classical theory with studies in computing education to create a discipline-based theory.

This dissertation does not attempt to 'solve' any specific problems in research or the classroom. In time, TAMP may be an inspiration or even drive empirical studies or classroom interventions, but these pages merely contain ideas, but not just any ideas. TAMP builds upon established, long-lived, and empirically supported theories of cognition and learning. This dissertation blends what we 'know' about how people think and learn with empirical and occasionally anecdotal observations from computing education research. In doing so, TAMP defines theoretical constructs that describe the mental activities of programmers, including how they use and form memories of programming knowledge and skills. TAMP's theoretical constructs provide researchers with the foundation to build theoretical frameworks for their studies and educators with practical guides for designing their curricula.

TAMP *is* a theory to guide research and practice. TAMP *is not* a set of heuristics or practices that shortcut the practices of research or teaching. Making the most of TAMP may require at least multiple readings of this work or visiting the original source materials referenced.

As described in Section 3.1.2.3, I do not see theory as a tool that practitioners pick up as needed, but a way of seeing the world. TAMP reflects the combination of neuroscience, learning theory, and practical experience teaching and working in programming with empirical research to propose a new way to evaluate how programmers think and students learn.

1.2 Organization of this dissertation

While many aspects of this work mirror a traditional literature review-methods-results-discussion format, the nature of the data and methodology add a twist. This chapter frames what to expect from this dissertation and its conventions. Chapter 2 resembles a more traditional literature review defining the 'problem space' – what literature says about training new programmers. It investigates what computing education literature says programmers need to learn, the struggles many students face during their learning, and the creative methods educators have proposed to improve early programming education. Chapter 3 captures several definitions of theory to establish the scope and contents of TAMP before defining a new methodology for theory building that drives the rest of this work.

The methodology of theory building proposed in Chapter 3 is a bit different than traditional empirical studies, and thus so is the remainder of this dissertation. While this dissertation is rich with data, it is not newly collected. Rather the data supporting TAMP comes from existing theories, empirical studies, and scientific findings on cognition and learning. Chapters 4 and 6 contain 'working literature reviews' that analyze the bodies of knowledge surrounding dual process theory and theories of learning and development, respectively. The dual purpose of these chapters is to summarize the pertinent features of several models of cognition and learning and rethink how they apply to programmers and learning to program.

The creation of TAMP primarily occurs within Chapter 7, leveraging theory, findings, and interpretations from the preceding chapters. TAMP's major contribution is applied theoretical constructs within the domain of programming, but Chapter 7 also proposes several models that capture and compare expert and novice programmer's cognition. Chapters 5 and 8 each revisit past studies under a new analytical framework. Chapter 5 considers the epistemological ramifications of dual process theory on programming education. It also reinterprets foundational studies on concepts like fragile knowledge and tracing through the lens of dual process theory.

Chapter 8 explores the cognitive model of expert design thinking from Chapter 7 by revisiting two studies that tested how novices design and write code. Chapter 9 considers the implications of TAMP on research and teaching.

1.3 Disclaimers

More than anything, this work looks to promote a dialog on the use of theory within computing and engineering education. I believe that the ideas in this work are promising, supported, and defensible, but I am not ready to say they are immutably correct. People are entirely too complex to fall into clean theoretical categories and predicted behaviors. New theories should wax and wane as better explanations emerge for specific aspects of education. It is the dialog, not the dogma, that should be the focus of theory. When I started this journey, I had an affinity for Jerome Bruner's research that I am still not sure was due to reason or undefinable preference. I was skeptical of Piaget and unsure of Vygotsky and skeptically wrote about many other theorists and theories. Possibly because of time spent in competitive debate, philosophy, or political science coursework, I started with an implicit mindset that theory was a competitive game rooted in outdoing the competition. At several points, I will say things like "TAMP offers better explanations than," which is both comparative and possibly competitive, but I believe factual. My goal is not to diminish other theories so much as the improvisational comedy mantra of "Yes and". Improvisation is said to be most entertaining when it builds upon the work of others. Likewise, I have come to believe that theory is best when built on the insights of prior theorists. In reviewing the work of others, I am generally looking to build upon the solid work they published and offer additional insights.

Theory building should be a collaborative effort over time that includes both hard data and squishy intuition. Anecdotal stories are not the materials for validating theory, but they can serve well in inspiring and explaining one. In this narrative, I will frequently share stories of personal experience to frame or elaborate on a point. I am not expecting my story to stand as proof of expert reasoning or novice struggles but as a starting point for seeing such phenomena in theory and data. Reading about dual process theory, may raise questions on how we make decisions, and when to feel confident about our knowledge. Theory is not about undercutting our confidence. Theory provides additional viewpoints to how we interpret what we see and make decisions in our practice.

Part of being a great practitioner is developing an intuition for our subject, but it may be that great researchers and educators also need to be aware of their own cognition and limitations.

I sincerely hope that some or many aspects of TAMP resonate with existing work. I regret to say, at this stage, even if TAMP inspires new arguments, it cannot act as proven science. My methodology looks to establish consistency in TAMP's propositions, but only further research can confirm my arguments. Chapter 9 describes some of the ways I plan to continue validating aspects of TAMP. New empirical data and analysis can help to validate TAMP further, and I look forward to seeing how others will make use (or alter) what I am proposing – and glad to collaborate as needed!

2. WHY DO NOVICES STRUGGLE TO LEARN PROGRAMMING?

2.1 What programmers "need to know"

Learners of any subject are judged proficient once they have developed a set of desired knowledge and skills, yet that set may differ based on their intended use. For example, I thought I was an excellent saxophonist based on my experiences in band class, concerts, and even competitions, up to the point where I had to play a solo in a Jazz band. I could sight-read nearly any song with few mistakes and, after a few classes, play pieces without error. Jazz required an entirely new set of skills for which traditional practice did not prepare me. Jazz demands musicality and improvisation. My experience strengthened the mechanics of playing the saxophone (seeing and playing notes) but taught nothing about understanding the composition of chords and rhythms required to improvise on the fly (or even prepare 'improvisations' ahead of time). Each time we reached the solo in the song, I had no idea what to do or where to start. My mechanical expertise, which was nearly flawless, did nothing to help me solve the problem of creating a new musical experience. Similarly, the skills taught in programming courses seem too often to build mechanical skill in producing code, yet fail to create innovative, applied, programmers. This section considers what computing education literature suggests that programmers must learn beyond the basic skills of coding, in the hopes of creating innovative programmers.

2.1.1 Programming versus coding

Many computing professionals use the terms *programming* and *coding* synonymously, yet accurately describing the knowledge and skills of a programmer seems to require a distinction. At a high level, it may be enough to say that coding is a subset of all that happens within programming, like typing sentences is a subset of all that happens when composing a novel. Becoming an excellent author (or programmer) requires skill in the vocabulary and grammar (syntax and semantics) of the language, but these skills are a small fraction of what it takes to tell a compelling story. Nearly every student studies the mechanics of their language, yet few people become authors, and even fewer are excellent authors. Becoming a computing professional often starts with mastering a specific programming language, but many excellent coders struggle in other

aspects of the work of programming. Some highly regarded computing professionals spend very much time writing code. Over the first decade of my career, I honed my skills as a coder and programmer to the point that I was 'promoted' to work on larger and more complex problems (often with the title of 'architect'). With the change in responsibilities, my employers no longer expected me to code, in one case, forbidden to do so as I was 'too expensive' to merely write code. I believe I completed many of the same tasks as programmers, except coding. What are the skills that a person must develop to not only code but program as well?

The computing education literature has captured the expected skillset of programmers since the early days of the technologies involved in programming. Some authors explicitly listed their thoughts on the concepts and skills required to be a programmer. Other researchers attempted to gather data to identify skills by watching experts while coding, or simply asking novices what they think they are learning. Ruven Brooks (1975) proposed a theoretical framework, describing "A model of human cognitive behavior in writing code for computer programs". His theory divided the writing of code from other programming tasks, such as debugging code. He went as far as to state, "a theory of program writing would have strong implications for a theory of debugging and ought to be developed first" (p. 137). He suggested that programmers make errors in one of three places corresponding to his three steps of writing a program.

1. Understanding – "Specifically, he must have representations of the initial state of the problem, the desired final state or goal, and one or more operations which he can apply, appropriately, to begin the transformation on the initial state" (p. 8)
   (e.g., solving the wrong problem)
2. Planning – "a method for solving the programming problem; it consists of specifications of the way in which information from the real world is to be represented within the program and of the operations to be performed on these representations in order to achieve the desired effects of the program" (p. 10)
   (e.g., choosing the wrong approach)
3. Coding – "using the plan to select and write a piece of code, assigning as effect or consequence to the code that has been written, and comparing the effect or consequence to the stipulations of the plan" (p. 16)
   (e.g., failing to realize the plan)

As early as 1975, Brooks is differentiating the role of coding as the act of translating a plan into the programming language. Long before even starting this task, the programmer must make sense of the problem and consider promising solutions.

Brook's early work included two specific themes that continue in the literature for decades, even perhaps after the industry abandons such ideas. Brooks described the programming process as sequential – understanding then planning then coding – that aligns with early software development processes. Until the advent of agile processes (Legaspi, 2014), most formal software development processes suggested strict ordering for building software. The sequential approach sometimes referred to as waterfall development, assumes that a programmer can fully understand a problem and conceive of a solution before starting to write code. There is some irony in this assessment as even Brooks "makes no claim about the understanding or planning process, [the process of coding] is not relevant to errors occurring in them" (p.138). Brooks seemed to imply that the details of coding and other aspects of programming are disconnected. If mistakes made in the coding process do not influence those of understanding or design, then these aspects of programming must require training that is distinct from that required for coding.

Later authors added their support to Brooks' idea to separate coding as only a part of programming skills. One summary added details and variety to the list of programming skills.

> "It is widely accepted that to program effectively one must:
> - have a good knowledge of the syntax and semantics of the target programming language (i.e., have an understanding of the conceptual machine supported by the programming language);
> - be able to debug programs;
> - and be able to analyze (complex) tasks and design algorithms aimed at solving these tasks." (Sleeman, Putnam, Baxter, & Kuspa, 1986, p. 6)

This list suggests that programmers must remember facts about the language and use this knowledge in analysis, design, and debugging. What these lists do not illustrate is how a student acquires strategies for analysis, design, and debugging that often feel foreign to students. Eckerdal and Berglund (2005) interviewed programming students who left with the impression that "learning to program is a way of thinking, which enables problem solving, and which is experienced as a 'method' of thinking" (p.141). Their students believe they needed to think in new and 'alien' ways to become a programmer. Many experts talk about programming in terms of processes (Sime & Arblaster, 1977) or cognitive processes (Brooks, 1977) rather than a 'way

of thinking' that is new or different. Where students saw unfathomable ways of thinking, these 'insider' experts see procedures. Pennington and Grabowski (1990), a psychologist and anthropologist, brought another perspective seeing programming as "a complex cognitive and social task composed of a variety of interacting subtasks and involving several kinds of specialized knowledge" (p. 47). Programmers seem to call upon knoweldge and skills that are various and seem unstructured.

Some researchers focused on the changes a novice undertakes in "becoming" a programmer. Benedict Du Boulay (1986) captured five "areas of difficulty" that programmers face.

1. Recognize programming's orientation ("finding out what programming is for, what kinds of problem can be tackled and what the eventual advantages might be of expending effort in learning the skill" (p.57))
2. Understanding the notional machine (*described later*)
3. Acquiring the notation of the language (syntax and semantics)
4. Discovering standard structures (patterns for solving everyday problems)
5. Programming pragmatics ("how to specify, develop, test, and debug a program using whatever tools are available" (p.58))

Du Boulay started with the realization that not everyone understands the value of programming and what advantages a computer offers. Forty years later, the value of computers may be an easier sell, but what they do and how they function are perhaps even less clear. Du Boulay described programming as a "Tool-Building Tool" (p. 59) that hides its inner workings from the user but must expose its secrets to the aspiring programmer. Du Boulay distinguished the notional machine, the functioning of a program, from the syntax and semantics of the language. The notional machine models a programming language in action, or as Sorva (2010) calls it, program dynamics, and Du Boulay splits code in action from the rules that govern that behavior. Once again, the primary focus of coding, learning the rules of a language, is only a small fraction of the skills of a programmer.

Computing education literature repeatedly suggests that becoming a programmer requires more than "learning to code" even if the programming language dominates early learning. Understanding the language and how to use it to write code is a subset of programming skills, but authors also regularly point to specifying (understanding) the problem, designing (planning) solutions, and debugging as critical skills. The amount of detail and the number of publications dedicated to these aspects of becoming a programmer is significantly less than aspects of coding.

Learning to code is a significant challenge for many new programmers, but without tackling other aspects of programming, students will struggle later in their education.

Sidebar on the two definitions of the notional machine

Depending on the application of the notional machine, the distinction between a mental model or a standalone mode may be negligible, but it is worth considering to understand how TAMP will apply the term notional machine. TAMP leans towards the use of *notional machine* as a mental model of a program's execution. When programming language designers create a new language, they define and codify a set of rules as to how their code behaves. In some cases, they control the behavior down to the chosen hardware, in other cases, an interpreter or virtual machine may introduce possible variances, but generally, their definition and implementation define the full bounds of the notional machine. In theory, a notional machine's rules and behavior can be in perfect alignment. The quality of the notional machine, in this sense, matters little to the programmer who is attempting to make their program function. Many programmers with incomplete or incorrect mental models of a language have produced a working program, despite their misconceptions. The code written by the designer of a programming language (who defined the notional machine) will still fail to run properly if their plan for the language (mental model) does not match the actual behavior. To the working and learning programmer, it matters little what the perfect model of execution looks like if there is a flaw in their mental model that results in an error. In computing education, it is the programmer's mental model that is of primary interest. In an ideal state, the programmer's mental model would perfectly match the execution model of the notional machine, which is the goal state of computing education. The notional machine, as currently defined, does little to measure the variance between a mental model and the ideal model (or even define the ideal model) a gap that TAMP seeks to fill.

2.1.2 The notional machine

The notional machine names a theoretical construct that is frequently mentioned in computing education literature, yet not clearly defined [1]. The notional machine encompasses the execution rules of a programming language at some level of abstraction. Du Boulay et al. (1981) distinguished that "the properties of the notional machine are language, rather than hardware, dependent," meaning the model of the notional machine abstracts specifics of the underlying hardware. In fact, in the age of virtual machines, the notional machine for languages like Python and Java might require few, if any, facts about the underlying hardware. Since each programming language may 'tweak' the behavior of familiar language constructs, a programmer must understand

[11] Szabo et al. (2019) conducted a review of the use of theory in computing education and noted the catgegory "theories for which we could not find a description, e.g., notional machines" (p. 104). The notional machine is widely invoked, and seemingly important, but this group of researchers did not feel it was adequately defined to include in their list of theories.

these subtle differences between their notional machines. The notional machine may even abstract further away from the language. Berry and Kölling (2016) created a tool that draws diagrams to visualize object-oriented concepts in the code, noting

> The notional machine does not need to accurately reflect the exact properties of the real machine; it presents a higher conceptual level by providing a metaphorical layer above the real machine (or indeed several such layers) that are hoped to be easier to comprehend than the real machine. (p. 54)

They stated that some instructors believed that students must understand every detail of the underlying language and hardware, but the notional machine allows for the abstraction of some of these details. The notional machine represents an abstract concept of the important details that a student must acquire as they begin to learn to program.

It is worth noting a small split in how the literature discusses the notional machine. Du Boulay, O'Shea, and Monk's original description never identifies where the notional machine 'exists' leading to some confusion and carefully selected language. Is the notional machine a document or tool that someone creates? Is the notional machine implied in the combination of language and hardware specifications? Is there a mechanism for validating the definition of a notional machine against reality? The vagueness of the notional machine may be the reason that "Notional machines, code tracing, mental models and program visualization are often appear conflated in the literature at least to non-experts." (Dickson, Brown, & Becker, 2020, p. 164). Many authors speak of the need for new programmers to build a mental model *of the* notional machine (Rountree, Robins, & Rountree, 2013; Sorva, 2013; Teague, 2014). Others speak of the notional machine as the mental model itself (Cunningham, Blanchard, Ericson, & Guzdial, 2017; Guzdial, 2015; Lopez, Whalley, Robbins, & Lister, 2008; Qian & Lehman, 2017; Xie, Nelson, & Ko, 2018).

> A programmer needs an accurate and robust notional machine and facility with defining the list of instructions for the computer to achieve desired ends. Programming is not the only reason to have a robust mental model of the computer. If we want people to be able to use the computer expressively, they need to know the computer's capabilities and limitations. The ability to use the computer to express ideas and to consume others' ideas is known as *computational literacy*. *(Guzdial, 2015, p. 2)*

Guzdial stated that programmers need a notional machine for almost every programming task and expands that need as a prerequisite for *computational literacy* as well. While there may be a

distinction between Du Boulay's original concept of the notional machine and a mental model of such an abstraction that a programmer holds, the value seems to be in the formation of mental models within new programmers.

Many computing education researchers place a mature mental model of the programming language as a cornerstone for mastering programming (Ahadi, Lister, & Teague, 2014; Cunningham et al., 2017; Du Boulay, 1986; Du Boulay et al., 1981; Garner, Haden, & Robins, 2005; Guzdial, 2015; Ma, Ferguson, Roper, & Wood, 2011; Mead et al., 2006; D. N. Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Qian & Lehman, 2017; Rountree et al., 2013; Sorva, 2010, 2013; Sorva, Karavirta, & Malmi, 2013; Teague, 2014). Khalife (2006) believed, "the first threshold students need to pass is to develop a simple but yet concrete mental model of the computer internals and how it operates during program execution" (p.246). Sorva suggested to instructors, "it is probably better to have learning about a notional machine as an explicit goal than an implicit one" (Sorva, 2013, p. 8:20). Authors point to several ways the notional machine helps students. Vainio & Sajaniemi (2007) said it was important for understanding the concepts of object-orientation and the associated diagrams. Several authors discuss the connection between the notional machine and tracing (Cunningham et al., 2017; Sorva, 2013). Teague (2014) included the "skills for reasoning about programs, sometimes referred to as the notional machine" (p. 155). Authors often tout the importance of the notional machine and seek ways to promote it.

Leading students to form a mental model/notional machine is challenging. Remember, Du Boulay (1986) suggested that the notional machine is distinct from syntax and semantics and it is vital to expose learners to the inner workings of a program (Du Boulay et al., 1981). The notional machine as a mental model is only measurable through secondary observations in tasks such as tests, tracing, or coding. While misconception literature (see Section 2.2.2) categorizes the mistakes students make in programming classes, the source of mistakes is unique to each student. For example, beginners often confuse programming syntax with their natural language origins (e.g., words like `while`, `switch`, or `try`), falling back on the colloquial meaning rather than that of the notional machine (Bonar & Soloway, 1985; Du Boulay, 1986; Pea, 1986). Instructors cannot hope to teach the notional machine through lectures alone, and each student may need different feedback to correct their very personalized mistakes.

The notional machine is primarily a theoretical construct used by researchers to describe learning and direct pedagogy, but the ambiguity of its nature introduces confusion. The discussion of the notional machine, this section being no exception, tends to treat the concepts as monolithic. The notional machine, like the language it models, is made up of numerous constructs that have many possible interactions. The knowledge required to describe the notional machine seems to have different forms and uses as well. The notional machine seems to offer an essential construct for describing the challenge of learning to program and perhaps can be more powerful still when enhanced by theory.

2.1.3 Tracing

Researchers often suggest that one measure of strong programming students is their ability to predict the execution of code, most often called tracing[2]. Tracing requires a programmer to determine the runtime flow/output given a snippet of code and set of inputs, often framed as a multiple-choice question to ease grading or provide instant feedback. The correct answer is only part of the feedback that tracing offers to researchers in studying the learning process. Some novices will take notes to help track the changes of values in the code (called sketches, doodles, or sometimes simply, their 'trace'). The presence/absence and quality of sketches is a common discussion point in the literature. Some instructors take time to teach a formal notation style (Xie et al., 2018) in which to track the code's execution, such as a trace table (see example in Figure 2.1). A trace table provides a handy format for tracking multiple variables, particularly involving iteration, but does not reflect the underlying hardware-execution mechanics. Novices seem to prefer an individualized sketching approach, as even when shown a specific method, not all students will readily adopt the technique (Cunningham et al., 2017). Researchers often point to sketching while tracing as a good indicator of ability (Cunningham et al., 2017; Lister et al., 2004).

[2] Tracing is the most common term, though some early literature uses different terms, such as close tracking (D. N. Perkins et al., 1986).

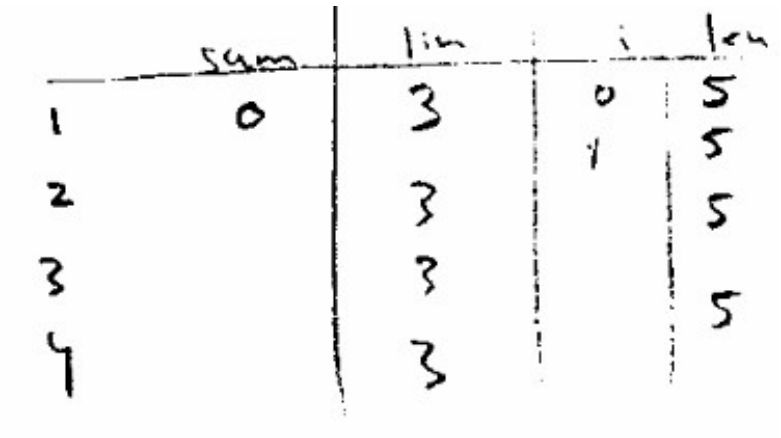

Figure 2.1 An example of a trace table that includes four variables: sum, lim, i, and len (Lister et al., 2004).

Tracing provides a moderately authentic activity[3] that provides significant scaffolding to help new programmers. Tracing does not require the programmer to write of perfect syntax, use complex tools, or even have a computer. Many tracing examples allow instructors to isolate specific language features and do not require the manufacture of real-world scenarios to elicit specific ideas. Many researchers point to tracing as a foundational skill on the path to becoming a programmer (Cunningham et al., 2017; Lister et al., 2004; D. N. Perkins et al., 1986; Sorva, 2013; Thomas, Ratcliffe, & Thomasson, 2004; Whalley et al., 2006; Xie et al., 2018). Raymond Lister (2011, 2016) and Donna Teague (2014) placed tracing at the heart of their Piagetian development construct focused on programming. Lister (2016) suggested tracing as "the path for acquiring [programming] knowledge" (p. 5).

Despite the encouragement of educators, students often find tracing arduous. Perkins et al. (1986) describe tracing as a "mechanical procedure" yet "mentally demanding" (p. 45). Few students in their study used tracing as a natural strategy for overcoming challenges in debugging their code, "sometimes simply that students do not even try to track" (p. 46) unless prompted. They proposed that their students were reluctant to trace because of several reasons.

1. motivational influences stemming from lack of understanding that tracking is important and lack of confidence in one's ability to track;
2. faulty understanding of how the programming language works;

[3] Cunningham et al. (2017) note "Sketching is not as frequently used or as successful on code fixing, code ordering, and code writing problems", but they suggest should be more common. Experienced programmers sometimes trace when debugging or during code reviews, but rarely do so in the absence of a computer to run the code.

3. projecting intentions onto the code so that one cannot objectively map the code as written onto the output;
4. cognitive style differences (p. 47-48)

Perkins et al. noted that students first need to view tracing as valuable and feel confident in doing so. Their students were working in code they wrote, yet still, they struggled to understand what their code was doing, and what it was supposed to do. Perkins et al. suggested that "students who naturally approach problems methodically and reflectively may be better [tracers] than those who approach their work in a more trial-and-error, or impulsive, fashion" (p. 47). Tracing may be easier than writing code, but it seems that new programmers still need a strong foundation in how the code behaves and how to consider code methodically.

The research into tracing is generally promising, but its role in programming is unclear. Lister et al. (Lister et al., 2004) demonstrated that students are generally learning to be reasonable good tracers, but only on certain types of problems. Xie et al. (2018) explicitly taught to students a 'lightweight' tracing approach involving tracking variables line-by-line and reported improvements in tracing skills, in contrast to the findings of Cunningham et al. (2017). They suggested that "completeness of tracing is more predictive of correctness than tracing strategy" (p. 171). The approach used by their students mattered less than their ability to complete the trace through the entire problem. Tracing was not a predictor of other programming abilities for Lopez et al. (2008). They tested their students' skills at tracing, explaining the meaning of code, and writing code, with no significant correlation between these skills. They saw students who were good tracers, but not very good at other skills and vice versa. While tracing seems to be a useful activity in teaching programming, the literature is not entirely clear on what skills it builds and how it supports other programming activities.

2.1.4 Patterns of use

One aspect that researchers focus on that is not language-centric is the strategies commonly used to solve problems. Discussions on learning to program often overlook the pedagogy of design, but one place that it does manifest is in the study of software *patterns*. Patterns are common strategies that designers customize to a given problem. For instance, a civil engineer might choose from a series of patterns for entering and exiting a highway, such as the ubiquitous cloverleaf. Each construction pattern offers advantages for managing the flow of traffic, the space required,

and other high-level principles that the engineer uses to guide the specific plan for that specific exit. Computing researchers capture and categorize "patterns of use" (Falkner, Vivian, & Falkner, 2013; Sajaniemi, 2002; Sajaniemi & Kuittinen, 2005; Shneiderman, 1977; Soloway, 1986; Soloway & Ehrlich, 1984; Wiedenbeck, 1985) and some curricula include coursework that explicitly teaches such patterns as a means for teaching design. Soloway (1986) described programming experts as having "built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them" and suggests teaching students "about these libraries and strategies for using them" (p. 850). Wiedenbeck (1985) noted that experts automate such libraries and "have overlearned certain stereotyped patterns to the point where recognition and generation of the patterns can be done without thought" (p. 389). Pattens seem to be a type of knowledge that experts have, and novices need to learn.

Patterns seem to play an important role in how experts approach programming. Soloway and Ehrlich (1984) asked the question, "what is it that expert programmers know that novice programmers don't?" (p. 595), arriving at two categories of knowledge: programming plans and rules of programming discourse. Programming plans "represent stereotypical action sequences" (p. 595), which are customized to meet the details of the current problem, such as using a loop to search for an item in a list. Rules of discourse act as heuristics for writing quality code such as, "variables are usually updated in the same fashion as they are initialized" or "programmers do not like to include statements that have no effect" (p. 595). Soloway and Ehrlich also tested their ideas by presenting sample code to experts and novices with a missing line that the participants needed to complete. Four of the code samples they created conformed to the standard programming plans they proposed while the other four did not conform to their standard plans and violated one or more of the rules of discourse. As they expected, the experts performed better than novices (by 18%), but their advantage dropped for the non-conforming code (to 6%). Soloway and Ehrlich's summarized, "advanced programmers have *strong* expectations about what programs should look like, and when those expectations are violated – in seemingly innocuous ways – their performance drops drastically" (p. 608). Both novices and experts struggled with non-conforming problems, missing those problems more than two times as much as conforming code. Patterns seem to offer experts quite an advantage and promoting them in novices may be critical.

A few researchers have attempted to teach patterns explicitly. Sajaniemi (2002) built a pattern library describing the use of variables in code. He suggests that "variable plans and roles

are tacit knowledge that is not mentioned explicitly in teaching" (p. 37). Sajaniemi later joined with Kuittinen (2005) to test the impact of explicitly teaching variable roles to students. They split students into three groups: a control, a group that received lectures on the roles of variables, and a group that used an animation tool that presented the roles of variables. They reported the approach "seemed to foster the adoption of role knowledge" (p. 80), and students were better at comprehending code with attention on patterns, yet their analysis is mixed. They reported little or no difference across the performance of the three experimental groups in their conclusion. They noted that the animation group's average final grades were lower, which they attributed to the method of grading by instructors. It seems that discovering patterns of use is important for novice programmers, but there is no clear pedagogy to promote such knowledge.

2.1.5 Summarizing what programmers should know

While researchers may occasionally favor their specific area of study, the literature is generally consistent in describing what novices are expected to learn. Novices should understand the purpose and function of computers, know the programming language and how it executes, strategies for solving problems, and acquire several more supporting skills that are essential (e.g., analysis, tracing, debugging, use of tools). Each topic seems to bring new stumbling blocks for some subset of students, as does the eventual integration of knowledge and skills. The next section investigates some of the more significant novices struggles that literature describes.

2.2 How literature describes novice struggles

From the earliest reports looking at how people learn to program, the literature has reported that many, if not most, students face significant struggles[4]. The performance of College-bound students in math and literacy skills tend to demonstrate a 'normal distribution' - most students scoring in the middle with a few strong or weak learners (College Board, 2018). Some studies report that the performance in programming courses is bimodal - students either struggling or

[4] High fail rates are the most extreme indicator of struggling, some studies report as many as one-third of all first year students leaving computing (Stephenson et al., 2018) and another that sees only a handful of students who start their program succeed (Beaubouef & Mason, 2005). Struggle is not just about quitting though, but in facing situations where pedagogy leaves students unprepared for programming work. Literature shows that some seemingly simple assessments confound students (McCracken et al., 2001) and even when students are successful in some areas they struggle in others (Lister et al., 2004; Lopez et al., 2008).

excelling, and few in between (Dehnadi & Bornat, 2006; Robins, 2010; Rountree et al., 2013). Perkins et al. (1986) described this phenomenon as "Johnny can do anything, but Ralph just can't seem to get the hang of it" (p. 37). Some postulate that a "geek gene" causes this bimodality, where a combination of talent and motivation sets some apart, yet this view is also challenged (Lister, 2010). Looking at the two AP Computer Science exams, for instance, the test on principles demonstrates characteristics of a normal distribution, while the core test slightly favors higher-scoring students ("AP Score Distributions – AP Students | College Board," n.d.). Identifying the struggles that novices face is not merely about tackling the seeming barriers to learning but improving the entire process of learning; learn more, faster, and with less unneeded toil along the way.

Programming's very nature presents the first struggle for new programmers. Novice's code is first judged by the unforgiving compiler and runtime rather than a sympathetic mentor. Programming tools do not recognize 'close enough' answers or offer partial credit. A program must be syntactically perfect, logically sound, and represent the desired behavior for a learning programmer to succeed. Few disciplines demand this level of perfection so early from their learners. By comparison, it seems that new programmers struggle more than those in other disciplines. The computing education literature breaks down several themes describing the struggles that novices face.

2.2.1 Acquiring and applying conceptual knowledge

When students are unable to complete seemingly simple programming activities, the natural question is: *are they learning anything at all*? In a critical study of programming abilities, McCracken et al. (2001) reported, "first and most significant result was that the students did much more poorly than we expected" (p. 132). Very few students in their study came up with promising approaches, much less wrote working code, to realize a simple calculator. They suggested that in "future studies, we might specify the level of prior programming experience or the specific programming knowledge that the students are assumed to have for each exercise." (p. 134). The assumption that seems to be behind this suggestion is that conceptual knowledge was lacking in their students and resulted in poor performance when writing code. Taking up the challenge offered by McCracken et al., Lister et al. (2004) designed a study to assess if students

were learning programming concepts. Using tracing as the primary test of understanding, they reported more promising results. Their students showed they were learning basic programming concepts, though they sometimes forgot to apply them on the more challenging questions. Conceptual understanding, it seemed, was dependent on the use as well as the concept.

Researchers and instructors have long lamented student forgetfulness. Perkins and Martin (1985) coined the term "fragile knowledge" to describe the tendency of novice programmers to incorrectly, inconsistently, or forget to apply concepts that they have previously demonstrated. Perkins and Martin used a clever strategy for testing their participants' knowledge while completing coding exercises. A researcher watched the student work and was to offer advice that frequently 'activated' the forgotten information, demonstrating that students may not have forgotten concepts, but merely failed to use them when needed. Guzdial (2015) described the problem as "[w]hen a student *knows* something but seems unable to access that information in a new context, we say that there is a lack of knowledge transfer" (p. 28). Transfer is a seemingly elusive goal in teaching, where students can apply concepts into new tasks or domains. Hatano and Inagaki (1984) described the challenge of transfer as adaptive expertise that requires both conceptual understanding but also some degree of procedural expertise. Conceptual knowledge, it seems, is not merely "acquired or not" but contextualized to specific tasks, at least at first.

2.2.2 Misconceptions

Where a faulty computer program contains bugs, researchers often describe a struggling programmer as holding misconceptions. Pea (1986), in fact, described novice misunderstandings as *conceptual "bugs"* in his work capturing programming misconceptions. Qian and Lehman (2017) defined, "misconceptions are the flawed ideas held by students, often strongly, which conflict with commonly accepted scientific consensus" (p. 2). Misconceptions are problematic as they are barriers to creating useful computing solutions. Studies frequently look to document, categorize, and remediate misconceptions in both novices and experts. The literature categorizes misconceptions into various topics like language syntax (Du Boulay, 1986; Khalife, 2006) and constructs (Du Boulay, 1986; Garner et al., 2005; Khalife, 2006; Spohrer & Soloway, 1986), or identifying the problem to be solved (Garner et al., 2005; Kwon, 2017; Spohrer & Soloway, 1986), assembling complex programs (Spohrer & Soloway, 1986) or even how

computers execute programs (Kwon, 2017; Pea, 1986). Misconceptions provide an intriguing window into how novices think, particularly in comparison to experts, and some believe understanding and teaching to misconceptions can yield improvements in learning.

2.2.2.1 Common Programming Misconceptions

The first major source of misconceptions seems to be the lack of, or misplaced information students bring about computers and how they function. Even when students do not think they understand a programming language, they seem to fill in missing information with similar but unrelated concepts. Pea (1986) explained, "novice programmer works intuitively and pursues many blind alleys in learning the formal skill of programming" (p. 26). Several researchers have pointed to the tendency of new programmers to try to interpret programming languages through the misleading meaning of their English counterparts (Bonar & Soloway, 1985; Du Boulay, 1986; Kwon, 2017; Spohrer & Soloway, 1986). For instance, many programming languages use the syntactic keyword `while` to declare an iteration construct (i.e., loops). The term sometimes confuses novices, who "treated 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" (Du Boulay, 1986, p. 69). It is not just English that poses such problems, as the symbols and operations used in algebra can also interfere with the core concepts of variables and assignments (Du Boulay, 1986; Khalife, 2006; Kwon, 2017; Sorva, 2013). Researchers report that novices are far from an empty vessel to fill with information, and instructors may need to refine prior knowledge to meld with programming concepts.

Misconceptions are far from universal in students as prior knowledge makes each learner unique. One cohort of students may suffer from misconceptions that never appear in another. Spohrer and Soloway (1986) classified the bugs they captured from their students into categories, yet seemingly paradoxically claim both that:

> misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. (p. 624)
>
> *and*
>
> Most bugs arise because novices do not fully understand the semantics of particular programming language constructs (p. 627)

Their students did not hold as many misconceptions about language constructs as expected, yet gaps in knowledge about the language caused most of a program's bugs. They seemed to be missing required knowledge as often as applying knowledge incorrectly. Spohrer and Soloway explained that novices struggled to create a proper plan for how to solve the problem. Novices did not struggle with individual constructs, rather they struggled to *apply and combine* constructs to solve problems. It seems that the rules and use of language constructs are two types of knowledge that may form independently. For example, Kwon (2017) studied how pre-service teachers plan algorithms on paper, despite having no prior instruction in programming. Kwon's pre-service teachers demonstrated strategies for solving problems but struggled to do so with well-defined commands at the level of abstraction of a programming language. On the one hand, Spohrer and Soloway's students received instruction on the language but struggled to use that knowledge to create a functional design. On the other hand, Kwon reported that his students proposed reasonable designs but struggled to use individual language constructs properly.

A major stumbling block for many novices is translating their rough plans into language constructs and understanding the details of what they created. Kwon's participants could express their ideas in any desired format, yet "[w]hile people are intelligent interpreters of conversations, computer programming languages are not" (Pea, 1986, p. 26). At one point in time, computers were strange machines with arcane interfaces, but modern devices seek to mimic human interactions. Even when computers used rudimentary interfaces, Pea noted that novices sometimes believed that a computer was an intelligent agent. They seemed to think the computer knows and secretly is hiding the 'right answer' if only the novice can uncover the truth. Pea named this novice mindset the "superbug" and believed it leads to other misconceptions. It is not that novices consciously attribute intelligence to the computer. Instead, they fail to recognize their agency in causing (and thus fixing) bugs and hope in vain their disparate fragments of code will produce the intended behavior.

2.2.2.2 Misconceptions in experts and novices

If programming mistakes are sometimes rooted in misconceptions, experts are not immune. Youngs (1974) captured the errors that novices and experts make, finding that both groups produced the same number of errors on a first run of their new program, but experts fixed their

mistakes quicker. The experience obtained by Youngs' experts did not prevent mistakes, but it did help them identify errors quicker and to fix more of them in response compared with novices. Misconceptions among working programmers have influenced the design of programming languages as well. Early programming languages contained a construct that no longer seems to exist in programming, the `goto` statement. A `goto` statement allowed a program to jump to any line of code in their program, a powerful but potentially problematic construct as it can lead to unexpected jumps in the flow of code. Some experts suggested that `goto` statements allow for simpler logic in certain instances (Du Boulay, 1986), but they were against the tide. Sime and Arblaster (1977) conducted a series of tests with experts and novices to see if a structured approach to creating logic reduced the number of errors produced (it did). Inevitably, language designers deemed the `goto` statement's risks outweighed its advantages.

One possible reason the `goto` statement was problematic was the lack of conformity to predictable patterns. As described in Section 2.1.4, Fix, Wiedenbeck and Scholtz (Fix, Wiedenbeck, & Scholtz, 1993) noted experts' "skill at recognizing basic recurring patterns, skill at understanding the particular structure inherent in a program text, skill at recognizing the links tying the separate program modules together" (p. 78). People may be less likely to fall to misconceptions when they are following well-trained patterns. It may be that `goto` statements presented a challenge since the patterns obvious in looping and other such constructs are harder to identify. The research into experts does not seem to indicate a lack of misconceptions, rather that experts are better at identifying and fixing mistakes.

2.2.2.3 Approaches to address misconceptions

The next step after identifying misconceptions is finding ways to correct, combat, or prevent them from forming. In the early days of computing education, misconception researchers proposed many improvements to the language and tools. Computer scientists created some of the first languages, "such as BASIC (1964) and Pascal (1970)… explicitly to ease learning how to program" (Fincher & Robins, 2019, p. 11). The Logo language, developed in 1967, targeted young people to enhance and motivate the learning of mathematics (Feurzeig, Papert, & Lawler, 2011) and continued as a way to introduce programming (Seymour; Papert, 1978; Seymour Papert, 1987; Pea, 1983, 1987; Weyer & Cannara, 1975). Logo neither revolutionized the teaching of math and

programming, nor improved the learner's general problem-solving skills, as hoped (Pea, 1983)[5]. Other researchers tried simplified syntax that better aligns with natural language (Soloway, Bonar, & Ehrlich, 1983) or even removing text in favor of "block-based" programming languages that use pictures instead. Graphical programming languages seem to allow novices to start quicker, but in the long run, did not radically change the learning outcomes compared with text-based languages (Weintrop & Wilensky, 2015). A debate as to the best 'teaching language' has waxed and waned over the decades, but no single language has proven to be vastly superior to another.

Other efforts looked to improve programming tools to facilitate learning. The compiler has long been a source of frustration, and studies have sought to improve its communications and assistance (Becker, 2016; Becker et al., 2019; Youngs, 1974). The compiler is a primary source of feedback for students, so this research seeks to improve the error messages produced. Some researchers seek to create tools to help new programmers visualize code and how it executes (Bednarik & Tukiainen, 2006, 2008; Ma et al., 2011; Sorva et al., 2013). These efforts seek to present the structure of code during execution and, as Du Boulay suggested, expose the inner workings of the notional machine. Efforts to create a friendlier language or improve tools have demonstrated limited success but may take students only so far as misconceptions "have less to do with the design of programming languages than with the problems people have in learning to give instructions to a computer" (Pea, 1986, p. 26). Inevitably instructors must also consider misconceptions in their teaching.

One approach that educators use to combat misconceptions is attacking them as or before they form. Garner et al. (2005) identified 27 categories of programming problems in programming and noted the number of times students sought help in each. They believed that better data on the errors that students make could help improve learning during an ongoing course.

> A valid, reliable analysis of programming students' problems would have many potential applications. On the basis of such an analysis we can adjust the amount (and kind) of attention devoted to various topics in lectures, laboratories and other resource materials. (p. 173)

By categorizing misconceptions, educators can plan their work to combat them directly. Garner et al. took a tactical look at misconceptions, where Spohrer and Soloway (1986) considered the

[5] This finding was publicly debated (Seymour Papert, 1987; Pea, 1987), and while descendants of Logo exist, they are far from the forefront of early childhood learning, much less current efforts to teach programming.

larger picture. They looked beyond individual issues to help students "put the pieces together" by "[f]ocusing explicitly on specific strategies for carrying out the coordination and integration of the goals and plans that underlie program code" (p. 632). Spohrer and Soloway proposed that the root cause of a misconception is not a lack of understanding, but an inability to assemble pieces of a language. They described earlier work, where "experts have and use such concepts [design strategies], although only on an implicit level… Students typically must acquire this tacit knowledge by induction from their teachers and their textbooks" (p. 632). They indicate that the strategies that experts use to solve problems successfully are not conscious or something they will learn in traditional pedagogies.

Much of the misconception literature describes misconceptions as muddled thinking that with the proper training and knowledge, novices can explicitly correct. Even ideas like simpler language, easier to understand tools, and directly addressing misconceptions in pedagogy seem to assume that the problem is one of consciously managing complexity and correcting unhelpful memories. Spohrer and Soloway suggested that experts transcend conscious planning strategies and perform much of these tasks implicitly. Researchers frequently point to the role of tacit knowledge and intuition in expert programming, as Section 2.3.3 will discuss further.

2.2.3 Cognitive load

Cognitive load is frequently invoked within computing education literature, but often without definition or theoretical underpinnings to inform how it impacts novices [6]. Cognitive load captures that familiar feeling when struggling with a complex task, but the science behind the concept has evolved dramatically. For the sake of scope, this work will not tackle in detail analysis of cognitive load in light of the theory proposed by TAMP, but it still seems useful to describe the basics of cognitive load and Cognitive Load Theory as it is very influential in computing education.

Programming tasks can quickly become complicated as the mind tackles intricate rules, tracks numerous data points, and carefully reasons through exacting logical statements and

[6] The term cognitive load is often used without any references in computing literature (Bednarik & Tukiainen, 2006; Denny, Luxton-Reilly, & Simon, 2008; Kwon, 2017; Robins et al., 2003; Vainio & Sajaniemi, 2007) and even sometimes when the authors cite theory they do not explicitly describe how that theory influences their work (Guzdial, 2015; McCartney et al., 2013; Muller et al., 2007). I am not critiquing the inclusion of cognitive load but pointing out that many authors assume that readers are familiar with a concept that is used in many ways and has evolving definitions.

calculations. Two theories attempt to model the impact on an overloaded mind – *mental load* investigates people's interactions with machines (Moray, 2013) while *Cognitive Load Theory* (CLT) focuses on instruction (Plass, Moreno, & Brünken, 2010; Sweller, 1994; Tindall-Ford, Agostinho, & Sweller, 2019). Understanding the mental load, the precursor to CLT (Plass et al., 2010) is helpful since authors frequently use the term *cognitive load* as a catch-all any intense period of mental effort and the same theoretical constructs may or may not apply.

Cognitive load occurs when a task overloads a person's available short-term memory (STM) with more facts than they can currently manage. When reaching the limits of memory, a person may feel strained and begin to forget details. Sweller (1994) looked at the impacts of cognitive load on learning specifically. His early theory proposed that when STM is overloaded, the learner no longer has the resources to learn, a process he called germane load. Germane load, however, may not be a tenet of CLT, as the authors do not mention it in the most recent publication on CLT (Tindall-Ford et al., 2019). The more salient factor in CLT might be the automation of knowledge and skills. Automation enables the execution of rules "without conscious control" (Sweller, 1994, p. 297), freeing up the mind for other tasks. CLT addresses some aspects of problem-solving, but primarily in the classroom and considering pedagogical approaches, where mental load considers a wider scope.

Mental load includes more factors than cognitive load, which focuses primarily on 'pure' cognition, excluding 'non-cognitive' factors. Unlike in the CLT model, an individual's mental load can be alleviated or exacerbated by the thinker's motivation, innate abilities, prior learning, or even their current physical status, such as general stress or fatigue. CLT included only prior learning as a salient factor[7]. Moray (2013) considered mental load as a limiting factor in a person's ability to complete tasks which require interaction with a machine effectively. He modeled the operator workload to include 1.) the inputs to the task, 2.) the operator effort, which 3.) result in some performance output, often becoming inputs to other processes. Moray and CLT each consider the design of inputs (e.g., the machine's interface, pedagogy) and the complexity of the task as a source of external load. Moray's model also considers internal factors "such as psychophysical characteristics, general background, personality; and fluctuating ones such as experience, motivation and attentiveness" (p. 4). While Moray's theory seems to focus on

[7] The exclusion of 'non-cognitive' factors is questioned in more recent work (Plass et al., 2010), but not included in the CLT model

production line workers, the concepts may more closely match the tasks required of programmers. Programmers are highly dependent on the design of tools (e.g., language, compilers, libraries), processes, and problem domain all relate to inputs and different individual demonstrate a broad spectrum of performance (Klerer, 1984; Raymond, 2005).

2.2.4 Emotion, motivation, and 'non-cognitive' factors

It is with great regret that I will only give lip service to the impact of 'non-cognitive' factors in this work. 'Non-cognitive' factors include emotions, but also ideas like motivation or persistence that is not part of traditional cognition. You may note that I prefer to add quotes to 'non-cognitive' as the research in neuroscience indicate that our cognition relies on emotions as a core part of our reasoning (Immordino-Yang & Damasio, 2007). Computing education research has long noted the impacts on the emotional states of learners as, "[s]tudents in introductory programming courses have misconceptions about computers and the programming process which produce anxiety" (Shneiderman, 1977, p. 193). More recently, Eckerdal et al. (2007) reported students frequently using "emotionally laden terms" (p. 128) in describing their learning experience. 'Non-cognitive' factors in computing classrooms may be of similar import as other research since according to Wilson (2010), "[c]omfort level in the computer science class was the best predictor of success in the course" (p. 153). While the scope of this work does not allow for a deeper discussion of the intersection TAMP and 'non-cognitive' factors, this brief section looks to acknowledge the import of such work and act as a placeholder for future efforts.

2.3 How to teach programming

Having considered the type of knowledge and skills that programmers need to acquire and some of their struggles, the last important detail is the ways and means by which educators teach programming. Pea (1983) considered Computing education different from other subjects, saying, "it is necessary to develop an instructional science for teaching programming and to rethink the educational goals programming is meant to fill" (p. 1). While traditional teaching methods are common, the literature suggests ways to support, augment, or even entirely discard the historical methods of instruction.

2.3.1 Critical content and common pedagogy

The backbone of any programming curriculum is the programming language, and researchers have offered advice on methods to introduce its concepts. Du Boulay (1986; 1981) believed instruction needs to be careful, consistent, and simple. Many researchers (Guzdial, 2015; Khalife, 2006) have advocated for teaching languages with simpler notional machines to introduce the concepts of programming. The use of 'blocked-based' languages offers another alternative, though with mixed results (Weintrop & Wilensky, 2015). Paired with the syntax and semantics of programming languages, researchers suggest methods to expose the inner-workings of code through tools (Guzdial, 2015; Ma et al., 2011; McCauley et al., 2008; Sorva et al., 2013), diagrams (Thomas et al., 2004) or tracing (Cunningham et al., 2017; Xie et al., 2018). Becoming familiar and proficient with the language is only a small portion of the challenge of instruction.

Researchers discuss the need to engender problem-solving skills using the programming language. Teague (2014) describes two dimensions of the programming curriculum. Instructors must teach syntax and semantics, but the "more neglected dimension comprises the skills for reasoning about programs" (p. 268). She advocated tracing as a way to improve reasoning, but novices must eventually apply their knowledge to solving problems. Several studies have tested the novice's ability to interpret meaning from code (Bayman & Mayer, 1983; Fuller et al., 2015; Whalley et al., 2006), noting that many students struggle to derive meaning from code. Mayer (1981) pointed out that many novices lack sufficient domain-specific expertise to solve problems. He suggested that "learner should be able to describe the effects of each program statement in his own words" but needs domain knowledge to do so. Learning about constructs alone does not seem to provoke a learner to discern the intent of those constructs as part of reading sample code. One approach researchers suggest to promote such skills is creating tasks that challenge students to explain the purpose of code snippets (Lopez et al., 2008; Whalley et al., 2006).

Some advice in the literature focuses on fine-tuning the practices already in place. Sleeman et al. (1986) suggested that code reading competency comes from a combination of knowing a language's syntax and semantics combined with debugging skills. They advocated for the formation of mental models to provide a means of organizing and accelerating learning but did not have new strategies for promoting them, "other than by extensive practice" (p. 6). Wiedenbeck (1985) also advocated that "it is probably important that the teaching process stress continuous

practice with basic materials to the point that they become overlearned" (p. 389). Repetition may be a mundane but important aspect of mastering foundational skills.

Computing education research creates, revisits, and refines teaching practices that target various aspects of programming. Table 2.1 compiles approaches starting with basic knowledge and moving towards encouraging the learner to complete basic programming tasks.

Table 2.1 Suggested learning activities from computing education

Intervention	Description
Worked Examples	From CLT, learners watch an expert complete an activity, either live or more often recorded. Each worked example focuses on some aspect of the subject and may repeat a single or show multiple examples to cover the breadth and depth of the subject thoroughly. (Caspersen & Bennedsen, 2007; Guzdial, 2015; Morrison, 2015)
Mimicry	The learner will follow, most often precisely, the work of another to rehearse the steps they will eventually be expected to take independently. (Eckerdal et al., 2007; Sorva, 2010)
Parsons problems	Learners receive a sequence of code statements, but they are out of order and require arrangement to complete the specified behavior. (Guzdial, 2015; Lopez et al., 2008)
Fill-in-the-blank	Learners receive partially completed code and must either choose from a list of options or write the appropriate line(s). (Muller, Ginat, & Haberman, 2007; Soloway & Ehrlich, 1984)
Live coding	Similar to a worked example, but more interactive with a live audience of learners. (Morrison, 2017)
Subgoal labels	As an extension of worked examples, subgoals focus on metacognitive awareness of the purposes of each problem-solving step. Subgoals apply to code, defining a 'plain English' explanation of the underlying algorithm (Morrison, 2015) or the role of variables (Sajaniemi & Kuittinen, 2005), for example.
Pointed examples	Some examples worked or otherwise, are formulated to draw out misconceptions learners may hold (McCauley et al., 2008)
Focus on patterns	After/within examples, focus on the patterns of use that may exist in the chosen solution (Caspersen & Bennedsen, 2007)
Scaffolding	Scaffolding simplifies complex tasks by reducing the learner's workload to one or a few steps of the full problem. In programming, the design, code, test cases, or step-by-step instructions provided to help the novice get started. (Caspersen & Bennedsen, 2007; Guzdial, 2015)

Continued on next page

Table 2.1 – continued from previous page

Intervention	Description
Faded guidance	Faded guidance gradually removes the scaffolding provided to a learner challenging them to complete more steps within complete tasks. (Caspersen & Bennedsen, 2007)
Peer instruction	Learners engage first with materials at 'home' and spend time in class engaging with peers to answer questions and discuss the subject in more detail (Guzdial, 2015)
Labs	Learners must complete all or some of a code problem, typically starting with some statement of the problem and requiring them to complete a full solution. (McCracken et al., 2001; Robins, Rountree, & Rountree, 2003; Utting et al., 2013)
Pair programming	A lab, or similar coding activities, is completed by a pair of learners, who trade-off in coding and helping, fulfilling formal roles in coding and learning. (Cliburn, 2003; Cockburn & Williams, 2001; McDowell, Werner, Bullock, & Fernald, 2002; Salleh, Mendes, & Grundy, 2011; Van Toll, Lee, & Ahlswede, 2007)
"Authentic" tasks	Authentic tasks make up a whole array of possible activities which better resemble the work professionals do, rather than what learners perceive as contrived 'classroom activities' (Guzdial, 2015)

2.3.2 On the order of instruction

Many subjects offer a natural order for introducing various topics. For example, learning to recognize the alphabet seems a natural precursor for learning to read, as, in mathematics, addition is a natural precursor to subtraction or multiplication. Many computing researchers have proposed an optimal ordering for teaching programming concepts (Berges, 2015; Lister, 2016; Ma et al., 2011; Mead et al., 2006; Shneiderman, 1977), with slightly overlapping but also differing results and priorities. Spohrer and Soloway (1986) noted that many classrooms use a construct-centered ordering, "reinforced by the structure of most introductory programming textbooks" (p. 626). The early textbook authors may have stumbled on an optimal order of instruction, or we simply follow their example by habit.

Raymond Lister (2016) described two very different approaches to instruction, *bottom-up* and *top-down*. He stated a preference for bottom-up while admitting the need for both. His bottom-up approach would resemble that of phonics in reading, focusing on "the relationship between written letters and spoken words" (p. 6). Lister advocated that programmers should learn

the mechanics of the language first. The top-down approach that allows readers to "make guesses about words they do not recognize, based on clues such as the context of the word in the text" (p. 6). Along the same vein, 'block-based' languages like Scratch allow very young programmers to build simple programs quickly with little concern for syntax. The carefully shaped blocks only assemble in legal ways, so they can create programs without even fully understanding the component pieces. Advocates of the top-down approach focus on the value programming offers and the processes of problem-solving. Bonar and Soloway (1985) suggested introducing problems using step-by-step natural language before dealing with specific programming knowledge, similar to Kwon's (2017) work described in Section 2.2.2.1. Spohrer and Soloway (1986) promoted a "goal/plan" approach rather than teaching construct-by-construct. They suggest that focusing on goals rather than isolated constructs helps the learner to assemble programs from the pieces.

> Students typically must acquire this tacit knowledge by induction from their teachers and their textbooks" (p. 632).

While Lister has his preference, students may need both methods to promote different aspects of programming.

Many researchers suggest it takes more than a single pass to teach programming concepts. Mead et al. (2006) suggested teaching using a *spiral curriculum*. A spiral curriculum (Wood, Bruner, & Ross, 1976) introduces then revisits key ideas multiple times with greater depth and challenge, eventually integrating with other concepts. Shneiderman (1977) offered an early proposal for using a spiral curriculum based on work by Bruner (1976a). Shneiderman suggests a "parallel acquisition of syntactic and semantic knowledge in a sequence which provokes student interest by using meaningful examples, builds on previous knowledge" (p. 193). His top-down approach looks to head off the anxiety that "keypunching" activities might induce. Schneiderman believed that focusing on low-level details might cause stress where meaningful examples instead build confidence. He suggested that syntax knowledge will follow if "frequently rehearsed and is anchored by repetition" (p. 197) and semantic knowledge will be "resistant to forgetting but it must be presented in small units which are either subtle variations or higher level organizations of previously acquired knowledge" (p. 197). Thuné and Eckerdal (2010) echoed the role of variation in learning within their phenomenographic look at programming concepts and learning. If a natural ordering to programming concepts exists, these researchers suggest that variation and repetition are essential in forming pedagogy.

Instructors must inevitably choose some order to present programming topics to students, but that choice may impact the quality, efficacy, and success of learning. Spohrer and Soloway are not alone in pointing out the need to teach not only the composite 'pieces' of programming but also help learners to develop strategies to assemble the pieces. Clancy (2004) emphasized the need to integrate knowledge to battle misconceptions. Pea and Kurland (1984) stated,

> "programming" is not a unitary skill. Like reading, it is comprised of a large number of abilities that interrelate with the organization of the learner's knowledge base, memory and processing capacities, repertoire of comprehension strategies, and general problem-solving abilities such as comprehension monitoring, inferencing, and hypothesis generation. (p. 144)

Pea and Kurland drew a parallel between reading and programming knowledge. Literacy and programming each demand that learners acquire and integrate not just information but a variety of skills. The resulting abilities are only useful, though, when a person can apply them in new ways. Instructors should select topics and order their curriculum as to build skilled problem-solvers.

2.3.3 An undercurrent of tacit knowledge and skill

Already in reviewing the literature of what programmers do and how to teach them, the notion of tacit knowledge, implicit skills, and intuition has regularly entered the conversation. Many authors point to intuition as a marker of expertise, but few pedagogies seem to mention, much less explicitly promote such abilities. A critical aspect of TAMP is the role of intuition in programming, so this section looks to capture additional appearances of intuition in computing education literature beyond that already mentioned.

Researchers often note the importance of intuition in design. Brooks (1975) said that knowledge of the programming language is "usually acquired from formal study of manuals and similar materials which give the grammatical rules for the language" (p. 140), but understanding and planning skills are acquired "almost exclusively by experience in programming in these domains" (p. 139). Since even mastering code "is acquired by direct experience with using these structures and operations in writing programs" (p. 6), experience seems vital to many aspects of programming. Mayer (1981) agreed, comparing the experience of programmers to that of chess masters, whose advantage seems to lie in their pattern recognition (Chase & Simon, 1973).

Researchers who compare experts and novices often identify intuition as an important difference. Wiedenbeck (1985) compared the speed and accuracy of novices and experts on simple code compilation rules and categorizing the purpose of code. She "predicted that even novices would be very accurate, as accurate as experts in these tasks" (p. 388), yet slower, only to find that experts were both faster *and* more accurate. Wiedenbeck's initially believed novices and experts would show similar accuracy since they have learned the same information. The results showed that experience adds to the quality and speed of knowledge leading to her conclusion that it is "probably important that the teaching process stress continuous practice with basic materials to the point that they become overlearned" (p. 389).

Implicit skills are not the only 'hidden' advantage of experts described in the literature. Soloway (1986) noted that "teaching the syntax and semantics of a programming language is not enough" (p. 858) because "[e]xperts are not necessarily conscious of the knowledge and strategies they employ to solve a problem, write a program, etc." (p. 851). He challenged researchers to identify the various types of tacit knowledge so educators could make them explicit – a task TAMP seeks to undertake. Eckerdal and Berglund (2005) pursued one type of tact knowledge by seeking the implicit canonical procedures that programmers develop. Hazzan (2003) originally defined a canonical procedure as "a procedure that is more or less *automatically triggered* by a given problem" (p. 108, emphasis added). Hazzan implied that programmers may not consciously decide the procedures they use in design. Intuition seems to influence many areas of programming, and the inability to reliable create such abilities in students may be a major limitation of current pedagogy.

2.4 The state of teaching and learning in programming

It would be unfair to say there is a crisis in the teaching of programming, but there seem to be significant gaps in educational practices that leave some students underserved. This chapter introduced a sliver of the research on learning to program as a foundation for further discussions. Before leaving this review of the literature, it seems helpful to highlight a few points important for the formation of TAMP. First, while learning to code (i.e., write syntax, perform basic tests and debugging) is an essential first step, becoming a programmer requires applying these skills to identifying, analyzing, and solving problems. It may not be reasonable to expect that students will

acquire this full skillset in a single class or even year of schooling but identifying the need and a potential path seem to be the first steps.

The other critical role this chapter plays is identifying the big questions that new theory should address. Why do some students seem so much better suited to programming than others? Fragile knowledge is a problem, but why is knowledge fragile, and what can instructors do about it? What role does intuition play in expert thinking, and how do we prepare novices to think like experts? Later chapters will revisit some of the studies mentioned here and others to tackle such questions and offer evidence for how TAMP proposes to model how programmers think and learn.

3. THE CONSTRUCTION OF THEORY

The goal of TAMP is to define a new theory, but while many educational theories exist, the theorists at best hint at the process of their creation. The closest any methodology comes to claiming to build theory is the qualitative method called grounded theory. Glaser and Strauss (1967), the creators of grounded theory, defined grounded theory to create what that called substantive theory, "developed for a substantive, or empirical, area of sociology inquiry" (p. 32). They contrast the theories created using grounded theory with formal theory, "developed for a formal, or conceptual, area of sociological inquiry" (p. 32). Grounded theory helps researchers to build new ideas based on never-before-seen phenomena. TAMP looks to build formal theory by applying existing theory to computing education. This chapter defines the methodology for building such a theory, which leverages elements from philosophy, grounded theory, and a formal method for using existing data as the testbed for new theories.

3.1 What is theory?

Theory should play a central role in rigorous research, yet disciplinary education researchers often lament how little it appears in publications. Section 1.1 quoted Fincher and Robins, who directly called for the greater inclusion of theory in computing education research. Section 3.1.2.3 will describe research on the use of theory in computing education, but before considering the need for theory, it may be helpful to define what theory is for TAMP. Part of the challenge in understanding theory is its pervasiveness and grand role.

> The basic aim of science is theory. Perhaps less cryptically, the basic aim of science is to explain natural phenomena. Such explanations are called "theories." (Kerlinger & Lee, 2000, p. 11)

Educational researchers construct and use theories to help students. Engineers use theory to inform their designs and help improve the human condition. The nature of theory is scientific, a way of abstracting various behaviors and events into a simpler model of how the world may function. While the heart of theory may be easily enough defined, how researchers use theory varies dramatically.

3.1.1 Exploring formal definitions of theory

Quantitative research has a defined role for theory, yet often oversimplified compared with the nuance required in the most challenging problems in education. Creswell (2008) stated, "[a] theory in quantitative research explains and predicts the probable relationship between independent and dependent variables" (p. 131). Quantitative researchers invoke theory to select the variables that likely have relationships worth studying. Simple theories predict how one variable impacts the other. Richer theories also explain the relationship offering guidance for applying theory. Kerlinger and Lee noted, "the very nature of a theory lies in its explanation of observed phenomena" (p. 12). For example, reinforcement theory originally predicted that rewards lead to a specific behavior but did not explain how. The lack of an explanation could lead to the assumption that continuing rewards will *always* elicit the desired behavior. Eventually, neuroscience identified the effect of dopamine within the rewards center of the brain (Eagleman & Downar, 2016), explaining why the impact of rewards might fade, and offering better ways of battling addiction. When theory also contains explanations, it allows for more refined studies and discussion in research.

Qualitative researchers use theory for some of the same reasons as quantitative, but also hold different perspectives on theory. Creswell (1997) described qualitative researchers using "social science theories [to] provide an explanation, a prediction, and a generalization about how the world operates" (p. 84). Qualitative methods do not look to prove causality through statistical inference, but instead, seek to describe relationships between observations. Kerlinger and Lee (2000) provided a more comprehensive view of theory

> (1) a theory is a set of propositions consisting of defined and interrelated constructs, (2) a theory sets out the interrelations among a set of variables (constructs), and in doing so, presents a systematic view of the phenomena described by the variables, and (3) a theory explains phenomena; it does so by specifying which variables are related to which variables and how they are related, thus enabling the researcher to predict from certain variables to certain other variables (p. 11).

The strength and validity of qualitative research may rest on the details of what theory can provide. Robust theories offer clear definitions, widely observed relationships, and rich explanations. Qualitative traditions have radically different views on the role of theory, as we will see in the next

section. When used in qualitative research, theory offers support through detailed descriptions that simplify the task of seeking order in data.

Theorists agree on the core purpose of theory as an explanation, but some proposed additional caveats and addendums. Dubinsky and McDonald (2001) created a theory to describe learning mathematics, including the criteria that theories (italics added)

- "support prediction,
- have explanatory power,
- *be applicable to a broad range of phenomena*,
- *help organize one's thinking about complex, interrelated phenomena*,
- serve as a tool for analyzing data, and
- *provide a language for communication of ideas about learning that go beyond superficial descriptions*" (p.275)

Kerlinger and Lee (2000) agreed theories should apply "to many phenomena and to many people in many places" (p 13). If reinforcement theory only described a relationship between rats and the amount of time they spend in a maze, the utility of such a theory would be limited. Theories may start from such observations but gain power when they are shown to apply to a broader range of phenomena. Dubinsky and McDonald also note the value theory has in organizing and communicating complex ideas.

Theory defines a new framework, and even vocabulary, for discussing ideas. Dubinsky and McDonald suggest their theory offers a new language for discussing learning, built on robust descriptions. Theory can provide a shorthand for complex ideas helping to simplify discussions on research. The framework and vocabulary also help to discuss the theory itself. As Kerligher and Lee (2000) put it, "Theories are tentative explanations" (p. 13). A theoretician should not expect to perfectly describe all aspects of the world in their first theory. New theories lead to discussions on their validity use, and refinement. An initial theory may not correctly capture a phenomenon but may spark interest in under-researched ideas by offering a new perspective and vocabulary. Utilizing theory, according to Silver (1983), takes effort. She believed "understanding [theories] requires more than simply memorizing the terms, their definitions, and their interrelationships" (p. 8). Silver (1983) describes theory as personal and transformative.

> A unique way of perceiving reality. An expression of someone's profound insight into an aspect of nature. A thought system that reaches beyond superficial experience to reveal a deeper dynamic than people usually perceive. A fresh and different perception of an aspect of the world we inhabit (p.4)

Silver views theory less as a tool for using, and more of a pattern of thought.

> To understand theory is to travel into someone else's mind and become able to perceive reality as that person does. To understand a theory is to experience a shift in one's own mental structure and discover with startling clarity a different way of thinking. To understand theory is to feel some wonder that one never saw before what now seems to have been obvious all along. This interpretation of the nature of theory does not sound very scientific. More formal definitions are dry, however, robbing theory of its beauty, its emotional significance, its importance in everyday life. A theory is a distinctive way of perceiving reality. (p. 4)

Silver describes theory in much more demanding terms than other sources. Theory is less a tool for Silver than a mindset. This view resonates with many qualitative traditions but may seem extreme for most other researchers. Theory seems most influential when it challenges conventional views and offers new perspectives on research and the application of knowledge.

3.1.2 Why is theory important?

Researchers probably speak the most about theory, but Silver's definition of theory makes the case that educators can also benefit from a strong theoretical grounding. Theory not only informs the construction of studies but curricula. Theory is not merely a tool to occasionally inform a specific step thought; it can serve as a set of values and a viewpoint for interacting with participants or students.

3.1.2.1 Theory and research

The quality of research and practice improves when grounded in theory. Kerlinger and Lee (2000) noted that many researchers complete valuable studies without a theoretical basis, but such studies report on "shorter-range goals of finding specific relations", and "ultimately most usable and satisfying relations… are those that are the most generalized, those that are tied to other relations in a theory" (p. 13). Theory allows researchers to link findings across studies with a consistent rationale and extend one investigation into the next, either confirming, refining, or possibly refuting the propositions of theory. In quantitative studies, theory informs the data to be captured and its analysis.

> A theory in quantitative research explains and predicts the probable relationship between independent and dependent variables… Not all quantitative studies employ a theory to test, but doing so represent the most rigorous form of quantitative research. It is certainly better than basing variables on your own personal hunches that are subject to challenge by other students and professors. (Creswell, 2008, p. 131)

Theory informs the research protocol to direct the study at data most likely to be useful. Theory acts as a backstop against data mining practices gone awry such as "p-hacking". P-hacking occurs When a researcher "[records] many response variables and deciding which to report postanalysis [or decides] whether to include or drop outliers postanalyses" (Head, Holman, Lanfear, Kahn, & Jennions, 2015, p. 1) thereby achieving a statistically significant, but potentially meaningless result. Theory not only identifies proper measures but to provide enlightening reasons they matter. Does an average improved test score of 2% indicate stronger comprehension? Did the intervention help students even though their scores are failing? The proper inclusion of theory enables a more vibrant and consistent story regardless of the outcome.

Qualitative researchers share less consensus on the role of theory, yet theory still contributes in many useful ways. Anfara and Mertz (2014) classified three viewpoints qualitative researchers hold:

> first, that theory has little relationship to qualitative research; second, that theory in qualitative research relates to the methodology the researcher chooses to use and the epistemologies underlying that methodology; and third, that theory in qualitative research is broader and more pervasive in its role than methodology (p. 7)

These viewpoints are not mutually exclusive; many researchers flow between each position. Researchers who see little value in theory during qualitative research are not diminishing theory, but feel theory stems from qualitative research. 'Methodology theorists' believe theory establishes the epistemology, nature, and source of knowledge that guides qualitative research. Theory acts as a lens for evaluating data (Anfara Jr & Mertz, 2014; Creswell, 1997), influencing the methodology and analysis. The final group believes theory plays a role implicitly even when not explicitly included. Anfara and Mertz summarized about qualitative researchers, "[t]heory has a place—an unavoidable place for all but a few of the authors we reviewed—and plays a substantive role in the research process" (p. 14). The direct role of theory in conducting qualitative research may vary, but its link to theory is undeniable.

3.1.2.2 Theory and practice

Educators also lean on theory as a guide toward classroom practices. Dubinsky and McDonald (2001) argued the "[d]evelopment of a theory or model in [disciplinary] education should be… part of an attempt to understand how [a discipline] can be learned and what an educational program can do to help that learning" (p. 275). The development and validation of theory in research may be an exercise in 'pure science', but educators can and should use theory to inform practice. Pellegrino (2002) leveraged models of cognition as a foundation for assessing "what students know". Svinicki (2004) promoted the use of cognitive theories that align instruction to the ways people learn. General theories of cognition and learning are prevalent in education, but as Dubinsky and McDonald point out, customized theories for a specific discipline provide learning strategies tailored to that subject. Computing education research increasingly sees the need for inclusion of theory yet struggles to do so in literature and practice.

3.1.2.3 Theory in computing education research

Computing education literature has historically leveraged little theory in publications. Joy, Sinclair, Sun, Sitthiworachart, and López-González (2009) reported: "the proportion of papers with a largely theoretical education focus is currently small overall (8.9% for journal papers and 3.3% for conference papers)" (p. 120). Other fields also struggle to including theory, but the 25% presence in other disciplines still dwarfs computing education publications. Joy et al.'s survey cast a wide net, though, including venues covering both research and practice, which may not expect rigorous research and the use of theory. Lishinski, Good, Sands, and Yadav (2016) completed similar research focused on computing education research's 'elite' conference (ICER) and journal (Computer Science Education). These two publications included theory in 66% and 77% of the respective articles – a much better saturation of theory, but with the very shallow threshold "if they contained at least one citation to a reference on learning theory from outside CS education" (p. 165). Lishinski et al. did not look at how the author incorporated the referenced theory, if theory informed the methodology or analysis, or if the authors merely mentioned the citation as part of a literature review. Even at the 'top' it there might be room for improvement in the use of theory in research.

One of the most significant impacts of the underutilization of theory is the inability to translate research into practice. Take, for example, the research around tracing. Green (1977) first identified tracing as a crucial skill in learning to program back in the '70s. Perkins and Martin (1985) documented how novices struggle to trace yet touted its value in learning to program. Despite early identification of tracing as a valuable pedagogy, two decades later, researchers reported inconsistent tracing skills (Lister et al., 2004) and could not connect tracing to other crucial programming skills (Lopez et al., 2008). Only in the last decade has theory emerged, describing the value of tracing (Lister, 2011, 2016; Sorva, 2013; Teague, 2014) and how it might promote overall programming skills and many other themes in research can also benefit from theory.

The story of tracing seems to reflect what happens when theory is missing. Researchers identified the importance of tracing forty years ago. I would argue researchers still cannot entirely explain what makes tracing critical, how to mature tracing skills, and how (or if) they relate to other aspects of programming. Practitioners who do not study educational theory and research are even less likely to pinpoint the aid struggling students need.

3.2 How is theory constructed?

The classical theories in education (e.g., Piaget, Vygotsky, Bruner) are well-known, but even though we have a chain of publications that describe these theories, the theorists do not share the step-by-step process of their creation. Theories seem to emerge from a series of studies and publications rather than an intentional effort rather than a specific process or method. TAMP may be different in that I am setting out to build theory as a purpose. TAMP seeks to apply existing theory in new ways. TAMP applies theories of cognition and learning to the discipline of computing education, and from that application considers new theoretical elements. While a guidebook for building theory may not exist in full, many authors speak to the building blocks of theory and the ways that theorists present and validate their work. This section considers how theorists use rhetoric to define their theory and how a technique from philosophy can help to hone such arguments. It also considers concepts from grounded theory as a means of validating the emerging theory.

3.2.1 Rhetorical methods for building theory

Within education, most theory comes to us in the form of books or long articles that spell out the theorist's ideas using a combination of rhetorical arguments, anecdotal stories, and descriptions of research. John Dewey's (1938) description of the role of experience in education is almost entirely a reasoned argument with little empirical data. Jean Piaget (1970) famously built his learning theory based on observations of his children and expanding his theory through decades of pivotal studies. Lev Vygotsky (1978, 1986) made a case for the social aspects of learning in response to Piaget, and Jerome Bruner (1966c) refined Piaget's and Vygotsky's ideas with small studies. Reading these works, the theorists never seemed to be formally constructing a theory, but certainly, that is the result.

Often the challenge in reading these theorists is identifying the exact definitions and relationships, if not the components of their theory. We hear terms like schema, assimilation/accommodation, the zone of proximal development, iconic representation, or spiral curriculum, but seldom do the authors stop to provide direct definitions. Often other authors seem to codify these terms and their relationships. With time, many authors have considered the construction of theory and its building blocks. Glaser and Strauss (1967) suggest that researchers using grounded theory define categories, properties of categories, and hypotheses to describe the groups of people under research. While these building blocks make sense for substantive theories, particularly for sociology, broader theoretical building blocks are also available. Kerlinger and Lee (2000) and Silver (1983) included a different set of three building blocks for constructing theory: *concepts*, *constructs*, and *propositions*, as visualized in Figure 3.1 visualizes and defined in Table 3.1.

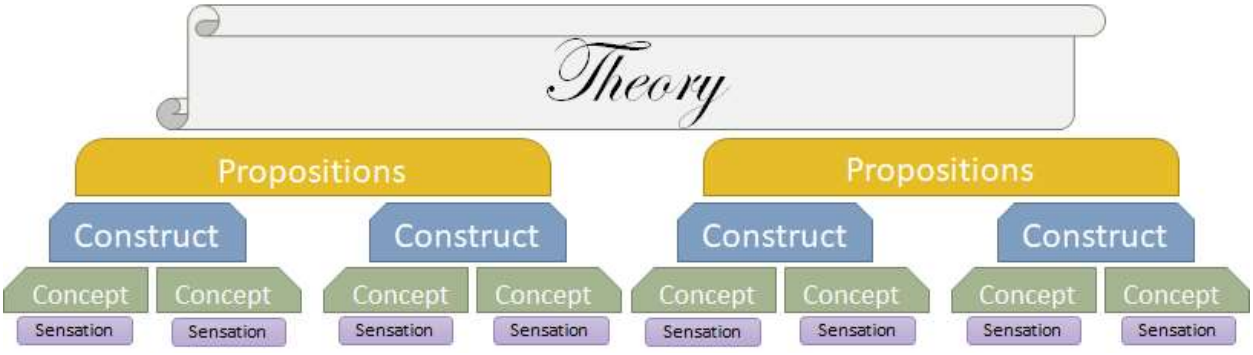

Figure 3.1. Concepts, constructs, and propositions as a building block of theory

Table 3.1 Silver's (1983) definitions for building blocks of theory

Idea	Definition
Concept	"a mental representation of a unit of concrete experience" (p. 5)
Construct	"a mental representation of a cluster or blend of concepts" (p. 5)
Proposition	"a statement of relationships between or among constructs" (p. 6)
Theory	"A set of logically interrelated propositions" (p. 6)

Science provides many examples of these four building blocks of theory. One such example that is likely familiar to many technology students is the theory of mechanics. Mechanics deals with the broad space of bodies in motion and contains many propositions, such as the conservation of energy, momentum, and angular momentum. These propositions rely on various constructs such as energy (e.g., potential or kinetic) and describe the relationships between these constructs (e.g., a ball that transitions from resting high up on a tower to falling to the ground). The constructs of energy are derived based on directly observable concepts. Kinetic energy builds upon the concepts of mass and velocity (which also includes the notions of distance and time). The physical science of mechanics provides a concrete example of theoretical building blocks that may help to translate into the construction of theory in social sciences.

Concepts provide the basic building blocks of a theory that capture what we can see in the real world. "A concept is a mental device for interpreting a unit in the stream of sensations we experience" (Silver, 1983, p. 5). Concepts define observable experiences as abstractions that can project into the past, present, or future. Concepts generally, but not always, relate to words – a single word captures a single concept. Some concepts transcend a single word, though, or since language is ambiguous, words might differ across contexts. For example, a person's age is a directly observable concept by comparing dates (birth date with some future date), but intelligence requires some interpretation based on the nature of the testing. The word 'age' perfectly parallels the concept, where intelligence is contextualized depending on the type of testing, as we will see later with the notion of fluid intelligence. Concepts are not all simple traits, but concepts all should tie to some sensation or experience.

Theorists define constructs to represent complex ideas involving multiple concepts. Constructs "cluster and merge [concepts] into a higher unit of thought" (Silver, 1983, p. 5). Constructs capture ideas that cannot be directly perceived but can be inferred based on the included concepts. Silver used IQ (the Intelligence Quotient) as an example of a construct. Intelligence is

a concept describing the results of testing, where IQ captures the notion that intelligence changes as people mature, thus combines intelligence and age. We expect children to know less than an adult, thus a high-scoring child would possess a greater IQ than an adult with an equal score. Constructs generally represent abstract ideas, blending experiences with a slight separation from perception. Silver noted that constructs often have 'made-up' names (like IQ) as words already exist to describe concepts, so constructs need new monikers. Theorists create better constructs by linking to and defining the relationship between concepts[8].

Just as concepts combine to form constructs, propositions blend constructs to offer, what Silver called "a unique conceptualization of reality" (p. 6). Propositions are helpful when "an individual conceives of a pattern of relationships among several constructs and can articulate these relationships clearly, logically and convincingly" (Silver, 1983, p. 6). Take, for example, the proposition: education improves IQ. Education and IQ are constructs that are not directly observable, but since they include observable concepts, educators can take measurements and test the proposition. It seems apparent that this proposition makes sense, but investigating the proposition is made easier since its constructs, tied to concepts, are all measurable. The construction of propositions upon well-defined constructs moves beyond the quality of rhetorical logic and provides a pathway to empirical exploration.

Turner (2003) described four schemes for producing propositions, one of which seems to overlap the type of theory TAMP is seeking to produce. His meta-theoretical scheme describes a process for asking retrospective questions about existing theories to enhance a theory or investigate its gaps. Theorists sometimes perform "a reanalysis of previous scholars' ideas in light of [various] philosophical issues… to summarize the metaphysical and epistemological assumptions of the scholars' work and to show where the schemes went wrong and where they still have utility" (p. 9). Turner did not describe a formal structure of meta-theoretical propositions, except to rethink the existing theory, but his scheme offers justification for defining such a process here.

Theory offers new ways of perceiving reality, but methodically constructing that reality makes it easier to share with others. Silver advised, "[constructs] must be defined and explained

[88] It may be helpful to note that once a concept becomes very familiar, we may implicitly blur constructs with concepts. Once familiar with the concept of IQ, or even before, you may jump directly from displayed intelligence to IQ implicitly knowing that a very young child's spark will likely outshine the older person's equally competent answer. The razor between concepts and constructs is one of direct observability, which requires a shrewd interpretation of what is perceived versus what is derived.

carefully so that others, who may have tended to think in terms of concepts or of different constructs, can grasp their meaning" (p. 6). Readers can misconstrue a theory if the author does not differentiate their definition of concepts from existing ones. The theorist should take care when offering new propositions to ensure they are clear and built upon distinct constructs. Silver suggested that the process of building theory "is the culmination of a highly abstract thought process whereby ideas are removed in successive stages from the world of immediate experience" (p. 6). Each step away from experience risks introducing imprecision, so precisely defining concepts, constructs, and propositions help to produce a clear and cohesive theory.

3.2.2 Formal philosophical reasoning

The challenge with rhetorical methods of theory building is providing clear and concise propositions amid the flurry of data. Most scientific fields rely on empirical data as the primary source of evidence, but philosophers must rely on arguments and logical reasoning. Philosophers rely on the structure of their arguments as much as their content. Educational researchers can rarely make irrefutable arguments by logic alone but can improve their discussion by borrowing the structure and style of philosophical arguments. While this dissertation is a requirement for a Doctor of Philosophy, it is not a Doctorate *in* Philosophy, so for the benefit of the author and reader, this section details the bare bones of formal reasoning[9].

The core of philosophical reasoning is defining an argument for your proposition. Talbot (2014b) used an example taken from a Monty Python sketch to define an argument:

> An argument is a connected series of statements to establish a definite proposition. (location 141)[10]

In the context of philosophy, people make arguments *for* something – an idea they wish you to accept. Philosophers convert prose arguments into a 'logic-book style' (location 594) to aid in their analysis of its quality. The logic-book reconstructs an argument into a series of premises leading to a conclusion. Figure 3.2 provides an example of an argument in the logic-book style.

[9] For readers who seek more on formal reasoning, this section is entirely based on the continuing education series (Talbot, 2012), podcast (Talbot, 2014a), and accompanying book (Talbot, 2014b) by Marianne Talbot.
[10] The Kindle edition of Talbot's book does not include page numbers but "locations".

Premise one: It is currently the morning
Premise two: I am starting to get hungry
Premise three: In the morning it is customary to eat breakfast food
Conclusion: I shall stop writing this example and grab some breakfast

Figure 3.2. An example of a logic-book style argument

Restructuring an argument into a logic-book style has several advantages. While prose allows for flexibility and occasionally beauty in describing an argument, the simplicity of the logic-book and its strictures offer clarity and concision. Each premise stands independently. The logic-book deconstructs complex sentences into simpler premises with distinct statements of truth (or not). As will be discussed in a moment, arguments are easier to assess when written in this format. The logic-book format also helps to identify ambiguities in the argument. Talbot notes three types of ambiguities: structural, lexical, and cross-reference (when reading argument, spoken word ambiguities are not present). Structural ambiguities occur due to the poorly ordered words. Talbot provided the example:

> So the sentence 1 (Every good girl loves a sailor) has two interpretations
> 1a. There is a sailor such that every girl loves him
> 1b. Every girl is such that there is a sailor she loves. (location 608)

Lexical ambiguities appear when a word can have multiple meanings. For example, Talbot uses the example "I went to the bank" (location 641), which could identify either the edge of the river or a financial institution. Cross-reference issues occur when a sentence is not clear on the use of pronouns or other words that refer to another part of the sentence. I often abuse pronouns in first drafts such as "Researchers presented participants with assessments which *they* found difficult to analyze." Was the analysis difficult for the researcher, students, or both? Logic-books add value since "every sentence is such that we can offer reasons for believing that it is true, and every sentence can be a reason for believing the truth of another sentence" (location 746). Logic books help to form well-structured, disambiguated, and easy-to-follow arguments.

3.2.3 Grounded theory

Qualitative researchers often seek to describe areas of experience that yet to be formally described by theory. Creswell (1997) describes five traditions of qualitative research, as listed in Table 3.2. Qualitative studies in education seek to gather rich descriptions of students, educators, the classroom, or some other aspect of learning. Generally speaking, most qualitative traditions gather similar data using similar methods and vary in their analysis and goals. Qualitative research gathers data primarily through observation. A researcher may observe people at work or play, conduct interviews, read documents, or watch videos. Some qualitative traditions use theory to guide their observations and analysis. Grounded theory researchers prefer to observe with an 'open mind', unbiased by existing theory or expectations. Grounded theory has a specific goal of generating theory, but as noted early, a specific type of early targeted at filling gaps based on new or unexplored phenomena.

Table 3.2 Qualitative traditions of research as identified by Creswell (1997)

Tradition	Typical focus	Used to describe
Biographical	Single person	An individual portrait
Phenomenological	Several individuals	A concept or phenomenon
Grounded Theory	Several individuals	A theory
Ethnography	A group	A cultural group portrait
Case Study	Bounded system	Portray a case

The grounded theory process may be easier to understand by following an example from computing education. Kinnunen and Simon (2010a, 2010b, 2010c, 2011, 2012) employed grounded theory to "[shed] light on the various processes and contexts through which students constantly assess their self-efficacy as a programmer" (p. 1). They interviewed nine students five different times across a ten-week term. They defined an initial interview protocol and immediately began analyzing the responses to find patterns in student responses, a process called *open coding*. Open coding helps researchers to see patterns in the data that transcend individuals and may lead to a category that provides an abstraction of particular phenomena, a process referred to as axial or analytical coding (Merriam & Tisdell, 2016). The collection of categories, their properties, and the relationships between categories form the beginning of a new theory. A robust theory is unlikely to emerge from a single round of data collection, however. Grounded theory suggests that the research team repeatedly refine their data collection approach to elicit additional targeted

information to describe their emerging categories better and test their bounds. Kinnunen and Simon (2012), for instance, modified their interview and "added questions concerning students' emotional reactions towards the process of doing the programming assignments to our protocol" (p. 5) since emotions emerged as a significant category. By returning to their students with the emerging importance of emotions as part of programming, Kinnunen and Simon discovered more than they initially expected from their data.

Grounded theory does not seek to 'prove' a theory but seeks to manage its consistency. Grounded theory seeks to manage the emerging theory's internal validity, deferring the later generalizability and external validity to future work. Zetterberg (1966) described the difference between internal and external validity as follows.

> The major difference is that the former expresses a "logical" relationship, while the latter expresses an "empirical" relationship. Internal validity, in other words, can be appreciated without empirical studies, while the determination of external validity is a test of a hypothesis. (p. 115)

A major strategy within grounded theory for ensuring the integrity of its data analysis is the *constant comparison* of data.

> Constant comparison is an inductive (from specific to broad) data analysis procedure in grounded theory research of generating and connecting categories by comparing incidents in the data to other incidents, incidents to categories and categories to other categories. (Creswell, 2008, sec. 443)

The initial analysis seeks to create abstractions from data. Later rounds of analysis confirm, refine, or eliminate aspects of the theory as dictated by the growing set of data. The repeated comparison challenges the researchers to consider if their theory explains existing and new rounds of data. Researchers continue to gather new data until they find no new information, a point called saturation.

Grounded theory provides a systematic approach to exploring an area with little understanding but is only the fledgling start of building theory. Kinnunen and Simon studied just nine programming students at one university. Substantive theory risks being 'localized' to the participants, course, school, country, or some other subgroup that may not generalize in future studies. Grounded theory seeks to identify new theory, but as Creswell (2008) described, "After developing a theory, the grounded theorist validates the process by comparing it with existing processes found in literature" (p. 450). Grounded theory researchers prefer to start without

consulting the literature to reduce the chance that it biases their analysis. Comparisons from literature at the end of the process allow the researchers to compare the emerging theory with other studies and theories.

Grounded theory contains important concepts and methods for theory-building, I argue the entire process may not fit with the approach needed for TAMP. Novice struggles in computing education are hardly unexplored territory, nor theories of cognition and learning. Collecting new data not only ignores the rich sources only touched upon in Chapter 2, but new data comes with no guarantee of generalizability. TAMP seeks to build, using Glaser and Strauss's (1967) phrasing, a formal theory that leverages existing theories and data that already contributes to substantive theories in computing education. While a traditional application of grounded theory may not apply, some of its methods will when defining a new method of theory building in Section 3.2.4.

3.2.4 An approach to building theory

Theory construction, like any creative task, blends inspiration with established knowledge, typically requiring some process or procedure. Personal experience, whether first-hand observation or through literature, motivates, and guides theory.

> Formal theory formulated directly from comparative data on many substantive areas is hard to find, as have noted earlier, since stimulation and guidance, even if unacknowledged, have usually come from substantive theory. However, it is possible to formulate formal theory directly. The core categories can emerge in the sociologist's mind from his reading, life experiences, research and scholarship. He may begin immediately to generate a formal theory by comparative analysis, without making any substantive formulations from one area (Glaser & Strauss, 1967, p. 90)

The educational theorists mentioned earlier used their experiences as inspiration for their theories and used these stories to guide and inform their audiences. The genesis of a theory is less important than the follow-up process by which that theory is elaborated, refined, and validated.

3.2.4.1 Making clear arguments

A perfect argument is only helpful to readers if they find it impactful and actionable. Readers often prefer a few concise lines that summarize the findings within an empirical study, but pithy quotes rarely capture the nuance required to describe the complex arguments of theory.

The methodology in theory building must, therefore, support both validity and challenge the reader's perspective. Silver (1983) defined theory's value, not by its profound implications, but how it transforms perspective. Theory may hold the most value when it not only proposes new ideas but considers how people use those ideas. In addressing why technology often fails, Safi Bahcall noted:

> It's not about the supply of the idea or the creation of the idea, innovation usually fails in the transfer. And it's not just transfer in one way, but it's feedback the other way. (TechNation Radio Podcast, 2019, sec. 16 minutes 38 seconds)

Theory loses value if it is unable to guide readers to a new perspective. Logical reasoning is important but must be supported by rich descriptions, examples, and details if the reader is to enter the theorist's mind. The use of logic-books adds a level of transparency to major propositions. While not every argument needs a logic book, the most important theoretical constructs and propositions can benefit from a clear and concise summary. The logic-book inserts, in the style and format of Figure 3.2, signify an important concept and, in this work, act as an advance organizer (Ausubel, 1960) for the coming argument. An explicit method of theory-building must present arguments that are both defensible and enable readers to form lasting impressions.

3.2.4.2 Validating theoretical assertions

Grounded theory offers a systematic framework for validating theory. The constant comparison approach enforces the gradual integration of new ideas with iterative checks for consistency using various sources of data. New data serves to refine a theory until the theorist can either confirm or reject their proposed theory. At this point, grounded theory suggests a process for a 'final' confirmation called *discriminant sampling*. Discriminant sampling offers one last comparison of the new theory against a previously unanalyzed set of data as a measure of its descriptiveness and accuracy in modeling the target phenomena. This reinterpretation approach is not without precedent as not every grounded theory study collects data from new sources. Bowen (2009) used documents as part of his study's data and noting, "entire studies can be conducted with only documents" (p. 34). The fundamental difference in this new approach and traditional grounded theory is the source of data and the type of theory produced. Where grounded theory seeks to avoid the bias of existing ideas to build substantive theory, this approach embraces

existing ideas to build formal theory. These two methods from grounded theory offer a rigorous approach to considering substantive theories that also seem to apply to formal theory creation.

The proposed methodology for theory building borrows the methods of constant comparison and discriminant sampling, refined to work with existing theory and data. Existing 'data' – in the form of past studies – replaces newly collected data. The 'theory under test' may take the form of existing theory applied to a new domain (e.g., how well dual process theory describes computing education issues like fragile knowledge) or to test new theories derived from existing. Instead of axial coding, the new method is a reinterpretation of a past study under the guise of a new theory. Section 3.3.6 describes the details of this reinterpretation method, but the concept is simple. If the new theory offers a better explanation than the old after reanalyzing a study, it is a promising replacement. Figure 3.3 presents a high-level view of the TAMP approach compared to a traditional grounded theory approach.

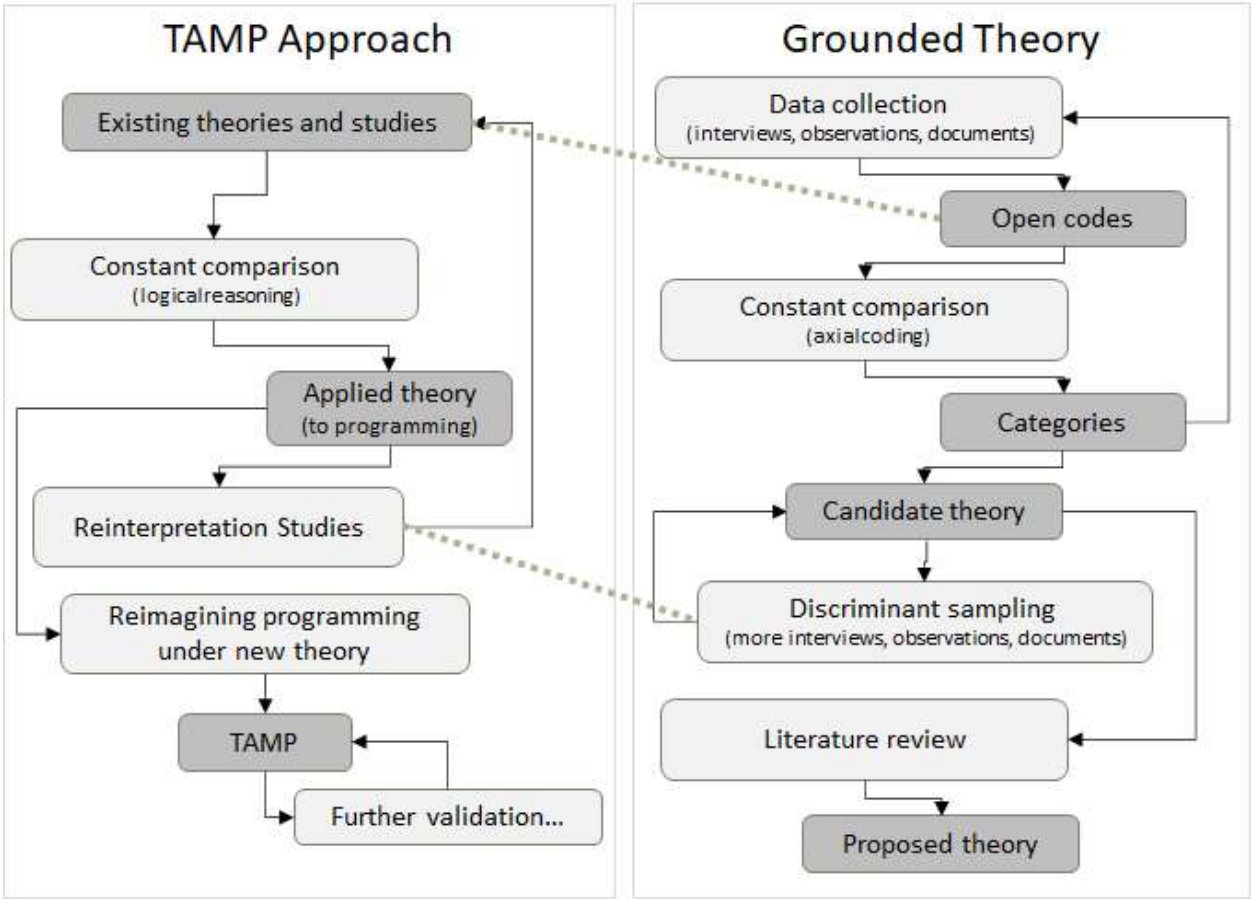

Figure 3.3. TAMPs stages of analysis in comparison to grounded theory

3.3 Methodology for constructing TAMP

Philosophy offers some guidance for making rhetorical arguments, and grounded theory offers an approach for validating theory, so this section combines these elements to propose a formal methodology for building theory. This section includes not just general methods for creating applied disciplinary theories, but the specifics of TAMP.

3.3.1 Scoping TAMP

TAMP's inevitable goal is the creation of an expansive theory of how programmers think. The full model is far beyond the scope of what a single dissertation can accomplish. This work will introduce the scope of what TAMP might eventually accomplish but inevitably focuses on a small fraction of the full theory of mind, specifically looking at how programmers design as a substantive contribution.

3.3.2 What is a theory of mind?

Some an educators or researchers may wish to "get in that person's head" to see what they were thinking? The task of teaching or researching education would be so much simpler if we could see the proverbial "gears turning" and map out the moment comprehension goes awry. The desire to understand another's thinking is the foundation of the theory of mind.

> In saying that an individual has a theory of mind, we mean that the individual imputes mental states to himself and others (either to conspecifics or to other species as well). A system of inference of this kind is properly viewed as theory, first, because such states are not directly observable, and second, because the system can be used to make predictions, specifically about the behavior of other organisms. (Premack & Woodruff, 1978, p. 515)

Premack and Woodruff used a "theory of mind" to understand the mental life of chimpanzees, but the construct is equally helpful in understanding students (Leslie, 1987). We apply our theory of mind when using words like *believes*, *thinks*, *doubts*, *guesses*, and ***knows*** when speaking about others. Educators regularly assess what students 'know'. Researchers try to capture what participants are guessing or what they believe. Both routinely apply a theory of mind without realizing, much less documenting their assumptions on how people think.

The concept of theory of mind not only applies to students, but to the work of educators and researchers. Premack and Woodruff described all humans as inferring a theory of mind about each other (and often animals as well!). Our theory of mind influences how we perceive the actions of others, yet how does a theory of mind form?

> Although it is reasonable to assume that [inferences about others depend] on some form of experience, that form is not immediately apparent. Evidently it is not that of an explicit pedagogy. Inferences about another individual are not taught, as are reading and arithmetic; their acquisition is more reminiscent of that of walking or speech. Indeed, the only direct impact of pedagogy on these inferences would appear to be *suppressive*, for it is only the specially trained adult who can give an account of human behavior that does *not* impute states of mind to the participants. All this is to say that theory building of this kind is natural in man. (p. 525)

It is safe to assume anyone practicing or studying education has likely spent time as a student in a classroom. Educators and researchers automatically and often unconsciously infer student's motivations, internal processes, and capabilities. Premack and Woodruff contended such assumptions are natural, unavoidable, and individualized based on experience – perhaps making them untrainable. A person can be taught to suppress their assumptions, but not consciously reform them[11]. The question is not *if* educators generalize, just in what way. A theory of mind exposes our implicit ontology, epistemology, axiology, and judgments of students.

A complete theory of mind for learning is a slippery concept, but thankfully TAMP is focusing on modeling the mind only when applied to programming. This type of applied thinking was used by Leslie (1987), who investigated the role of pretending in cognition. Theory of mind acts both as a construct that children form to "comprehend opaque states in oneself and in others" (p. 421) as well as a construct for describing a child's thoughts while pretending. Leslie compared children with and without autism, noting "autistic children should also show serious impairment in their later theory of mind" (p. 423). Theory of mind acts as a means for comparing the two groups of children. TAMP, like Leslie's study, looked to model select aspects of cognition. A theory of mind describing how trained programmers think can be compared with novices to see what knowledge and skills must be acquired.

[11] The nature of which seems to align with the contents of Chapter 4.

3.3.3 TAMP's scope for "What programmers need to know"

Chapter 2 provided a baseline set of skills that a programmer should possess. Rountree, Robins, and Rountree (2013) extended the discussion of what a programmer should know, to the types of mental structures they need, in extending the theory of Threshold Concepts within computing education[12]. A novice who masters a threshold concept gains a perspective that accelerates and enhances future learning. Threshold concepts are compelling in computing education as a solution to the alleged 'bi-modality' seen in learners (see Section 2.2). Rountree, Robins, and Rountree described three areas of required learning.

> passing the threshold requires the successful acquisition and internalization not only of knowledge, but also its practical elaboration in the domains of applied strategies and mental models. (p.286)

They suggested that threshold concepts are not just about remembering but applying knowledge. Students must develop "schema and automatization" (p. 283), which we will see in Chapter 4 aligns well with dual process theory. Rountree, Robins, and Rountree do not discuss the content of learning to program but focus on ways of knowing as a threshold concept. The gap that remains in theory and literature is what that 'way of knowing' looks like and how novices might acquire it.

This dissertation covers a few focused areas that provide the foundation for the cognitive model of TAMP. The first key contribution will be establishing the role of dual process theory in describing programming cognition. To tackle the process of learning and problem-solving, this work combines Jerome Bruner's representations with dual process theory and neuroscience to establish a set of refined constructs to model mental activity. Returning to computing education, TAMP revisits the notional machine and defines a new theoretical construct focused around the mental model programmers form around the notional machine, the Applied Notional Machine (ANM) as a computing education-specific educational construct. This work concludes by defining two proposed models that use elements of the ANM to describe how programmers read code and come up with designs.

[12] TAMP does not address threshold concepts (TCs). TCs are compelling, but as Rountree, Robins, and Rountree discussed, the definition of TCs is very difficult to nail down in programming. TAMP may align with or totally mitigate TCs, but the overhead of making that case on top of the bevy of literature and theory only serves to complicate the discussion. As most practitioners will never have heard of TCs, they felt extraneous and complicated with little added value.

3.3.4 Contributing theories and literature

TAMP builds upon existing theory and past studies from computing and other educational literature. Chapter 4 focuses on dual process theory as a foundational framework replacing traditional models of cognition. Chapter 6 revisits theories of learning and development, specifically Piaget, Vygotsky, and Bruner. While findings and studies from neuroscience appear throughout several chapters, Section 7.3 specifically investigates specific findings from neuroscience and their relationship to computing education. Table 3.3 provides an overview of the included literature across the chapters described above as well as existing theories or commonly used within computing education.

Table 3.3 Contributing theories and studies with references to their core literature

Purpose	Sources
Dual process theory	Kahneman (1973, 2011; 1983), Stanovich (2012), Evans (2009)
Neuroscience	Burton (2009), Eagleman & Downar (2016), Immordino-Yang & Damasio (2007), Kandel (2009), Ledoux, Brown, Lau, & Mobbs (2017), Squire & Kandel (2003)
Contributory Learning Theories	Piaget (1970; 1952), Others on Piaget (Ginsburg & Opper, 1988; Inhelder, Chipman, & Zwingmann, 1976; Piaget & Inhelder, 1967), Neo-Piaget (Case, 1996; Fischer & Bidell, 2007; Lister, 2016; Morra, Gobbo, Marini, & Sheese, 2008; Teague, 2014), Vygotsky (1978, 1986),
Computing education	Notional Machine (Du Boulay et al., 1981; Sorva, 2013) Cognitive Load Theory (Morrison, 2017; Morrison, Dorn, & Guzdial, 2014; Plass et al., 2010; Sweller, 1989, 1994)
Bruner's representations	Bruner (1966a, 1966b, 1966c, 1971, 1979a, 1979b, 1997; 1956)

3.3.5 Methods of building and documenting TAMP

The theory-building toolbox contains tools for theory creation and content for TAMP, as shown in Table 3.4. All that remains is assembling the pieces into a plan.

Table 3.4 Tools available for constructing TAMP

Theory creation devices	TAMP-specific content
Rhetorical argument (Table 3.1)	Theory of mind (Section 3.3.2)
Logic-book format (Figure 3.2)	Scope of TAMP (Section 3.3.3)
Modified grounded theory (Figure 3.3)	Literature (Table 3.3)

The next chapters each tackle a step building and validating TAMP (see Figure 3.4). Chapter 4 describes dual process theory, and Chapter 5 provides a series of reinterpretation studies acting as a constant comparison for how well dual process theory explains the struggles of novices. Chapter 6 introduces theories of learning and development, including Bruner's representations and new theoretical constructs of TAMP. Chapter 7 assembles the theoretical constructs and propositions of TAMP, including a detailed vignette of how experts and novices think when designing code. Chapter 8 performs the equivalent of the discriminant sampling step from grounded theory. Chapter 8 validates TAMP's propositions by revisiting three studies that tested students' ability to design and write code. Chapter 9 provides one last form of validation by looking at potential research and pedagogical implications of TAMP.

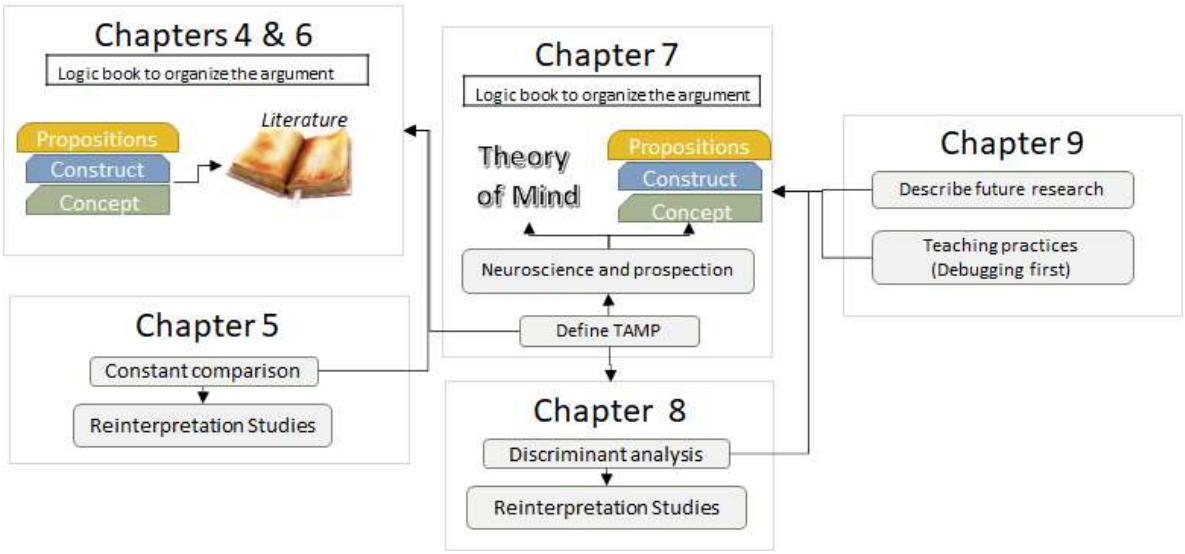

Figure 3.4. Overview of the argument structure for the dissertation

3.3.6 Past studies as data and reinterpretation as validation

Premise 1: A replacement theory provides a better model compared to the original or alternative theories.
Premise 2: Experimental studies offer a strong methodology for comparing two alternatives.
Premise 3: The selected theory is the only dependent variable
Premise 4: Reinterpreting past studies, whose methods do not change under either theory, allows comparison of the dependent variables with other variables controlled.
***Conclusion*:** Past studies provide an effective methodology for comparing theories

Before describing the methodology of past studies, it is first essential to establish why they are superior in building TAMP. To get in the habit of using and reading logic-book arguments, this section includes one to present its case for reinterpretation studies. The validity of theory must be weighed both against how well it describes its target and does it do so better than existing theory. Researchers typically choose experimental methods to compare alternatives, in this case, competing theories. The best experiments tightly control every aspect of the study such that the only variation is the treatment. Any variance in the procedure risks obfuscating the differences between the alternatives. Establishing control between experimental groups is notoriously difficult, particularly in educational studies. Even studies that merely replicate past studies exactly struggle to replicate results. In a study to compare theories, the only variable of interest is the chosen theory, but how can you vary theory in a study?

A study with theory as the dependent variable seeks to control every other variable, but what variables interact with theory? Theory generally appears in the planning and analysis phases of a study. Theory informs methodology, yet so long as methods do not vary based on theory, any methodology and resulting data will suffice. Theory plays a more significant role in analysis, so a 'controlled' study of theories can branch at the phase of analyzing data and reporting results. Using past studies also allow exploring multiple phenomena in many studies with various methods. Each reinterpretation shares any faults from the original study's data collection and methodology holding TAMP as the only changing variable.

One limitation of the reinterpretation approach is the lack of 'original' data. Most studies publish only a fraction of data, often filtered through statistics and the author's perceptions. The original dataset might allow TAMP to make unreported findings, but the comparison of TAMP as a theory to the original theoretical framework only requires better, not additional insights. At this phase, establishing TAMP possesses a superior interpretive power than the original publication demonstrates validity. The studies included in the following chapters were selected because their authors report something paradoxical they could not explain sufficiently. The reinterpretation studies used here are not looking to contradict the initial findings, but to explain the paradox, augment suggestions, or suggest how follow-up studies can go further. Theory is always evolving, and future research might confirm or refute TAMP. Using reinterpretation studies to show TAMP is plausible, consistent, and worth pursuing provides success at this stage.

An additional advantage of reinterpretation is the low cost of comparing multiple past studies across different phases of learning, types of research questions, and theoretical approaches. The selected reinterpretation studies vary in content, contain valid methodologies, cogent analysis, and clear reporting. Each also documents gaps in understanding about novices and their struggles. These are not 'bad' studies with hopeless findings, but well-constructed with sensible conclusions yet perhaps limited due to limitations in the theoretical framework. Reinterpreting these studies provides the reader with a demonstration of how a new theoretical viewpoint can open doors in teaching and researcher.

The process for conducting a reinterpretation study mirrors that of any article, except its data collection is already completed. The responsibilities and steps of the study include:

1. Describe the past study, its theory (if any), methods, and report findings
2. Describe any paradoxical findings or inconsistencies
3. Present the relevant alternative theory
4. Apply the alternative theory to analyzing #2

Some of the reinterpretation studies are standalone publications, while other studies are as part of this narrative. When the studies are standalone, the chapter will provide a summary, but the original publication will hold many of the details.

3.3.7 Theoretical architecture for TAMP

This chapter established a mechanism for capturing theory, its construction process, and the methods of validation. Before diving into TAMP, it is worth considering its architecture. I have worked as a 'software architect' for the better part of two decades, so let me share my perspective of the purpose and value of architecture. Architecture provides an enabling technology from which the actual functionality emerges. As a software architect, I would choose between frameworks, languages, tools, and, most importantly, standards (driven by values) that describe how these things combine. As mentioned in Chapter 1, I am passionate about how people first come to learn to program, but TAMP is using early programming research as a vehicle for defining its architecture, not its end goal. Learning to program is one application of TAMP, but the concepts underlying that argument apply to many computing education challenges.

Dual process theory (Chapter 4) and the learning theories (Chapter 6) are the foundations upon which the rest of TAMP builds. The dual process perspective is critical to understanding the role of intuition and tacit knowledge in programming. Without dual process theory, the rest of the insights of TAMP are useful but tenuous. Dual process theory explains both the advantages experts hold over novices and the baffling struggles novices often exhibit. Dual process theory provides an alternative model of cognition to explain programmers in action, but alone does little to inform how people learn. The theories of learning and development proposed by Piaget, Vygotsky, and Bruner encompass many of the observations of dual process theory, yet from different perspectives. Piaget wrote deeply about implicit learning and individual development, where Vygotsky focused more on burgeoning reason and the social impacts of learning. Reviewing each of their theories provides insights on how people learn explicitly and implicitly. Bruner's theories are critical in explaining how people blend each type of knowledge and ability to solve problems.

Dual process theory and the collection of learning theories provide a theoretical architecture from which to pursue any learning endeavor. I chose programming because that is what I am the most familiar with, and because it offers the most 'self-contained' discipline. I cannot think of any other design discipline that allows every student access to every tool and skill with essentially no cost. Even art requires basic supplies and time to clean up, where once a student has a computer, the only other cost is typically an electric bill. While I am not attempting to generalize the findings of TAMP to other disciplines, if the architectural foundation holds for programming, it may hold for other disciplines, STEM and beyond. Programmers solve very complex sociotechnical problems and generally get to see the final product and receive feedback. Programming languages provide a medium from which to capture the nuance of design decisions at every level. There are always tacit elements to design, but programming goes further than nearly any other discipline in capturing the thinking of practitioners simply because of the nature of the product – a socially constructed set of rules defining a software program. While TAMP focuses on early programming/CS1 type questions, it is important to remember that such questions are merely just one amongst countless cases of how people learn complex problem-solving activities.

4. DUAL PROCESS THEORY

This chapter explores the cognitive model proposed by dual process theory and its applications in programming. A model of cognition describes how the human mind processes and retains information as well as responding to stimuli. Most models of cognition also imply or directly describe the mechanics of learning, offering guidance to educational researchers who embrace them. This chapter starts by describing several models of cognition, some explicitly used in computing education research, before considering dual process theory. The tenets of dual process theory challenge the traditional epistemic definition of knowing that rely on logic alone as a source of knowledge and reason. The remainder of the chapter considers the possible influence by revisiting fragile knowledge and tracing through the lens of dual process theory.

Premise 1: Using dual process theory as a model of cognition enhances the epistemology of programming to align with updated models of how people use knowledge and experience
Premise 2: Dual process theory explains the nature of fragile knowledge, which can lead to better approaches to manage it.
Premise 3: Dual process theory provides a better explanation of the tracing behavior of students than the authors in Lister et al.'s study, rationalizing why students 'decide' to stop using notes to aid their tracing and the resulting change in performance
Premise 4: Cognitive load theory only helps explain novice struggles in the narrowest of senses, particularly in comparison with dual process theory
Premise 5: Dual process theory, in conjunction with the 'feeling of knowing', can help model non-cognitive impacts and offer support for improving curriculum
***Conclusion*:** Dual process theory provides a model of cognition that explains both how experts use knowledge and several examples of why novices struggle to learn programming

4.1 Traditional models of cognition

The stereotypical programmer is someone who is analytic, pays attention to detail, and driven by logic and reason ("5 Personality traits every new programmer should have," 2014; "Top 10 qualities of information technologists," 2018; Dunn, 2013; James, 2008; Sloyan, 2017). While programmers may demonstrate mastery of logic and reason in their work, many believe that logical reasoning is not just a characteristic of programmers, but humankind. For centuries philosophers and economists have positioned all humans as above all analytical thinkers who make logical decisions, perhaps most famously Rene Descartes whose famous words "Cogito, ergo sum," translated to "I think, therefore I am." Perhaps arguably (or not) programmers are stereotyped as

being highly logical thinkers. Software programs must be precise, attend to arcane details, and implement flawless logic. Daniel Kahneman and Amos Tversky (1973) coincidentally used the stereotype of programmers as the content of one of their tests. They presented the following to a group of participants.

> Tom W is of high intelligence, although lacking in true creativity. He has a need for order and clarity, and for neat and tidy systems in which every detail finds its appropriate place. His writing is rather dull and mechanical, occasionally enlivened by somewhat corny puns and flashes of imagination of the sci-fi type. He has a strong drive for competence. He seems to have little feel and little sympathy for other people, and does not enjoy interacting with others. Self-centered, he nonetheless has a deep moral sense. (Kahneman, 2011, p. 147)

Participants overwhelmingly predicted that Tom W was "more likely to study computer science than humanities or education, although they were surely aware of the fact that there are many more graduate students in the latter field" (Kahneman & Tversky, 1973, p. 239). Kahneman and Tversky's test tells us two things, most people stereotyped programmers as driven by order and logic and easily forgot to apply the logical rules of probability in making their assessment. Do programmers think differently than other people making them immune from such mistakes? Are programmers, by training or nature, scions of rational thinking beyond that of other people and professions?

Programmers probably are not fundamentally different from other people. Papert (1988) observed, "[c]omputers are a domain where everyone expects that analytic to reign supreme, yet this situation makes it especially clear that for certain children, the development of intelligence and programming expertise can reach high levels without becoming highly analytic as well" (p. 12). Papert reported that children could become good programmers without being notably logical thinkers. So, what is the relationship between rational cognition and programming? Is there a better assumption about how people think if they are not entirely rational? To best understand the impact dual process theory might have on how we research and teach computing, it is useful to understand the legacy of Descartes' philosophy and how it has influenced conventional beliefs about cognition.

Mathematician and philosopher Rene Descartes established the modern view of cognition. Descartes believed that physics was the root of a tree of knowledge with branches, including morality, medicine, and mechanics (Ariew, 1992). Three principles underly his philosophy:

1. To employ the procedure of complete and systematic doubt to eliminate every belief that does not pass the test of indubitability (skepticism).
2. To accept no idea as certain that is not clear, distinct, and free of contradiction (mathematicism).
3. To found all knowledge upon the bedrock certainty of self-consciousness, so that "I think, therefore I am" becomes the only innate idea unshakable by doubt (subjectivism). ("The rationalism of Descartes," 2019)

Descartes believed philosophers should be logical but skeptical. His view of this 'highest form of thinking' seems to have seeped into the Western world's definition of the everyday nature of thought. Many seem to translate rationality as an existential differentiator of humanity. Human reasoning is what set us apart from the rest of the animal kingdom, and our civilized behaviors were far beyond that of other creatures. As Frans de Waal (2016) described, each time a study showed how animals expressed behaviors that met the current definition of *culture*, some scientists would redefine culture to a higher standard. The last several decades of research have demonstrated animals with many 'human-like' abilities, but far from undercutting our humanity, these insights help to explain our occasionally irrational behaviors. Philosophers continue to debate Descartes's assertions (Hintikka, 1962), yet while rationality *may* be distinctly human, modern research shows that humans are far from being distinctly rational.

Many people, even scientifically trained ones, seem to hold implicit views that trace back to a Cartesian model of cognition. It is not that our teachers explicitly teach us a model of how people think and learn, but we seem to acquire subtle hints of a Cartesian view of rationality handed down through the centuries. Donald MacKay (2019) saw "Rene Descartes [as] the main reason why twenty-first century scientists find it difficult to imagine" (p. 96) the recent findings on memory, perception, and behavior. MacKay described the Cartesian model of cognition as flowing through "sensory perception, comprehension-and-thinking, memory-storage, memory-retrieval, and muscle-movement" (p. 96). He used the analogy of a train to describe the order of each stage of thinking, as shown in Figure 4.1.

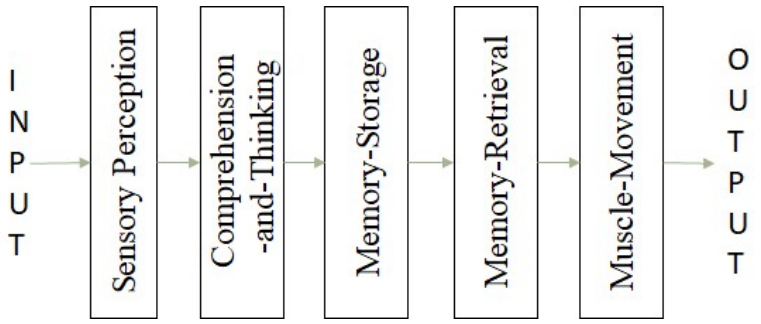

Figure 4.1. MacKay's view of the Cartesian train of thinking.

MacKay's *Cartesian train* describes the (mostly implicit) model of how we make decisions that misleads our understanding of cognition. For most of history, people believed the brain contained one type of memory for remembering everything. MacKay recalled Plato describing the brain as a piece of wax upon which any memory could be impressed. Memories under the Cartesian train model do not form until after we have considered what to remember, perhaps implying we only learn what we choose to and thus we should not remember wrong ideas (e.g., "I told the students the bad aspects of this analogy, but they still remembered them"). The Cartesian train model is not entirely without modern twists. Each 'car' within the train might describe "episodic versus semantic memory, explicit versus implicit memory, sensory and perceptual versus modality-independent memory, procedural versus declarative memory, reference versus working memory, short-term versus long-term memory and long-term versus very long-term memory" (MacKay, Burke, & Stewart, 1998, pp. 29–90)[13]. By accommodating various types of memory, the Cartesian train model seems flexible and powerful, but MacKay found deficiencies in how the linear model of thinking describes cognition in action.

The beauty of the Cartesian train is its simplicity and alignment with a centuries-old narrative of how people think. The challenge of sticking to Descartes view of cognition is how quickly it breaks down when researching the details of how people think and learn. It is reassuring to believe the engineers of an airplane and legislators of public policy use flawless logical reasoning at each step of the creative process. We know, however, that professional programmers make numerous mistakes along the way to constructing their 'logical' applications. Teaching seems easier when the goal is merely creating a perfectly logical description from which students learn. At some level, the model from Figure 4.1 even feels right, as it often resembles the stories

[13] The details of some of these types of memories are discussed further in Section 7.3.1.

we might tell of making conscious decisions. Carefully constructed empirical studies poke holes in this model, leaving room for alternatives that better match observations on cognition. MacKay's (2019) book speaks about his work with the famous patient H.M. (Henry Molaison), who battled severe problems forming memories after a surgery to stop life-threatening seizures removed portions of his hippocampus and amygdala. When Henry lost his amygdala, he lost the ability to form new memories, even forgetting people he just met when they left the room. Henry dedicated the rest of his life to participating in perhaps thousands of studies to help the scientists understand the brain. Many studies reported that Henry suffered from long-term memory loss alone (Milner, Corkin, & Teuber, 1968). MacKay believed Henry struggled with language issues at the least, yet he did not lose all types of learning, something not easily explained in the Cartesian train model.

4.2 Dual process theory

Dual process theory formalized the role of automation and intuition within cognition. Under dual process theory, the mind utilizes two mechanisms, commonly named System 1 and System 2 (Evans & Frankish, 2009; Kahneman, 2011). System 1 includes mental processes that are intuitive, swift, automatic, and complete without focused attention. Examples of System 1 in action are our ability to instantly read a billboard as we speed by on the highway or complete a simple math problem (e.g., 6 x 7). System 2 performs tasks that require attention, focus, and effort, such as complex calculations and logic, or waiting at the starting line for the official to start the race. Table 4.1 provides an overview of the two Systems. Separating the two Systems is helpful for description, and later learning, yet we will see thinking, 'rational' or not, relies on both Systems working in concert.

Table 4.1 Overview of the two Systems in dual process theory

	System 1	System 2
Traits	Quick and automatic	Logical/Rational
	Unconscious and supportive	Requires attention, focus, and effort
	Learns slowly	Updates quickly yet can be ephemeral
A.K.A.	The Autonomous Set of Systems (TASS)	Type 2 processes
	System X (refleXive)	System C (refleCtive)
Examples	Understand simple sentences	List your social security number
	Find a strong chess move	Fill out a tax form
	Read "camel case"[14] words in code	Trace an entire algorithm

[14] Camel case is a style for naming things in code (e.g., numberOfCharacters or printSalesReport).

Many theorists have contributed to dual process theory. This introduction primarily follows the phrasings and descriptions used by Daniel Kahneman in his book "Thinking Fast and Slow" as a way of making the concepts more accessible. Kahneman's book targets a general audience and avoids the often jargon-filled descriptions used in literature aimed at psychologists on dual process theory. Section 4.2.2 will visit some of the additional literature to refine Kahneman's basic descriptions of System 1 and 2.

4.2.1 Introducing dual process theory

The most significant contribution of dual process theory might be the inclusion of intuition, tacit knowledge, and implicit skills as mechanisms supporting rational thinking. Before diving into the details of System 1 and 2, it is important to remember that these are parallel systems that are highly intertwined. As an analogy to set the frame for dual process theory, consider a hybrid vehicle. It is easy to talk about a *gas engine* versus an *electric engine*, yet if you have ever looked under the hood of a hybrid, it is not so easy to pick apart these major mechanisms – it looks like just a regular car, not two side-by-side devices. A driver has little control over which fuel system activates at any given time; the vehicle's computer decides. Dual process theory helps to remind us that our brain holds two mechanisms for thinking and learning but does not create distinct 'easy-to-work-on' Systems any more than a hybrid. Like the hybrid-trained mechanic, an educator must be aware of the two Systems of dual process theory but must consider them together in action to get the best results. It is helpful to remember that dual process theory gives us a useful vocabulary for describing cognition, but it is often counterproductive to take this separation too literally or forget they are mostly inseparable.

4.2.1.1 Introducing System 1

Some of our most impressive mental feats involve our automatic and subconscious System 1. A very counterintuitive example of System 1's prowess is mastering the game of chess. Chess, at least in western culture, is often held up as the epitome of strategy, logic, and planning. One chess website offers the following advice to novices.

> There are many novice players out there, who don't feel like calculating deeper than one move ahead. They use their *intuition* to judge if the move is good or not. Although, the *intuition* can be a valuable resource for a chess player, calculation of variations is something that will help you win a lot more games, than solely relying on *intuition* and luck! (Markushin, 2013 emphasis added)

People believe that chess masters look deeper into the future than lesser players, weighing each move and possible countermove. Markushin's description asks novice players to not only consider their next move but that of their opponent and then possible combinations of moves beyond that point. The ever-expanding tree of possible moves easily overwhelms new player's memories, so perhaps chess masters have vastly superior memories than regular people. The IBM artificial intelligence (AI) program, Deep Blue, used its vast memory stores and the ability to compute each move to push the game towards promising future moves that eventually defeated one of the top chess masters of the day (Ellis, 2008). Deep Blue used Markushin's strategy as the algorithmic approach for playing chess, yet it took fifty years of advancement in electronics hardware to achieve. At that time, Moravec (1998) suggested that Deep Blue probably possessed $1/30^{th}$ of the computing power of a human, and thus perhaps computing finally 'caught up' to the processing power of the brain, at least for this one task.

The surface description of Deep Blue aligns well with the conventional wisdom of how to play chess like an expert but does not tell the whole story. Chase and Simon (1973) used chess pieces in a famous study investigating short-term memory. Researchers asked participants to memorize pieces on a chessboard then recreate the scene. Participants who regularly played chess remembered significantly more pieces than those who had not, but only when the pieces fell into recognizable chess patterns. When the pieces are placed randomly (in unlikely game positions), skilled chess players lose their advantage. Chase and Simon postulated the expert advantage "derives from the ability of those players to encode the position into larger perceptual chunks, each consisting of a familiar subconfiguration of pieces" (p. 80). They suggested that chess players recognized and thus recreated patterns better than non-chess players. According to one of its creators, Murray Campbell, Deep Blue used the same shortcuts for matching human players.

> One, in particular, was to help with the opening library, which every chess program uses in order to save time and make sure it gets into reasonable positions.

> And then we increased the chess knowledge of the system by adding features to the chess chip that enabled it to recognize different positions and made it more aware of chess concepts. (Greenemeier, 2017)

The Deep Blue team programmed their AI with a library of opening moves honed through centuries of games and created special hardware to recognize patterns. Chess is not just a matter of logical deduction, but experience and near-instant recognition of board positions that are advantageous or risky. Markushin suggested that new chess players must develop both planning and intuition to improve their game. Like we saw in Section 2.3.3, the concept of intuition is easily evoked, but often much more difficult to define.

System 1 provides a model for not only describing intuition in action but also provide hints at is formation. Computing education researchers have often cited Chase and Simon's work in noting the importance of pattern matching (Brooks, 1975, 1977; Crk, Kluthe, & Stefik, 2015; Mayer, 1981; Pea & Kurland, 1984; D. N. Perkins et al., 1986; Soloway, 1986; Teague, 2014). Pattern matching is a core part of developing expertise.

> However, expert knowledge depends not on the prowess of some general memory talent but on highly specialized abilities, acquired through experience, to encode and organize particular kinds of information. These abilities give experts the ability to recognize quickly a large number of patterns. (Squire & Kandel, 2003, p. 73)

Experts are not necessarily supremely gifted with an expansive memory but develop *specialized abilities* to recognize patterns in a particular area given enough practice. Despite the popular belief that *chess makes you smart*[15] (C. Chabris, 2016), it seems that the experts benefit as much from pattern matching as other cognitive abilities, based on Chase and Simons' results.

Dual process theory formalizes the research around tacit knowledge and implicit skills into the theoretical construct that is System 1. System 1 provides a proverbial Swiss-army knife of cognitive support tools, including quick access to information, instant computation, and under the right circumstances, a cognitive autopilot. Kahneman (2011) gave examples of System 1 such as,

> Read large words on a billboard
> Answer to 2 + 2 =?

[15] Seymore Papert (1987) and Roy Pea (1987) similarly debated if programming would improve general problem solving skills. Papert believed that programming in Logo could create general problem-solving skills in children. Pea's work tended to argue against, and it seems the neuroscience favors Pea's position. In so far as patterns and knowledge help expert problem-solving it seems no single field will help another. It may be that in certain problem-solving approaches may transfer, but this is probably not due to 'generally being smarter'.

Find a strong move in chess (if you are a chess master) (p. 21).

One of System 1's major roles is to automate activities that frequently occur within the brain. In many ways, our brain evolved just like we suggest that software programs should; if a sequence of code is reusable, it is better to encapsulate that algorithm in its own container. The same way a programmer builds a subprogram to allow reuse and optimization, System 1 builds memories that provide fast, automatic, and essentially effortless responses to familiar experiences. Given time, the strange lines and curves that are letters not only become easier to discern but become impossible not to recognize. The multiplication tables are cumbersome to memorize for many children, but the work pays off later in life in instantly computing the practiced calculations. System 1 also tackles more complex aspects of cognition that Kahneman described as intuition.

System 1's processing is 'invisible' to the rest of our cognition, so intuition provides an apt description for how it influences our thinking. Kahneman (2011) noted, "most of what you (your System 2) think and do originates in your System 1, but System 2 takes over when things get difficult, and it normally has the last word" (p. 25). The Cartesian view of cognition is compelling because reason is the final arbiter of our choices, even when it does not fully understand the framing of those choices. As we will see further in the next section, System 1 influences how we make decisions by preprocessing what we sense and offering System 2 filtered information based on our experience, a process called *priming*. Priming takes advantage of the patterns recognized by System 1 to reduce the effort required by System 2. Chess masters, for instance, can 'see' further into the future because System 1 offers immediate feedback if certain moves are good or bad. A new player, on the other hand, must spend significantly more effort evaluating threats and opportunities and thus considers far fewer possible future moves.

System 1 offers speed, accuracy, and automaticity that improve the quality of other types of reasoning, but all these advantages come with a cost. System 1 processes may be quick to process but are relatively slow to form. People do not learn to read or become great chess players overnight. In most cases, direct instruction seems to have much less benefit to System 1 learning than simple repetition. Rote repetition, though, may not develop the type of skills desired. System 1 learning is not limited to behaviors we find desirable and helpful; we can learn bad habits as easily as good ones. One simple example, I tend to mistype the word 'language', swapping the 'u' and 'a' when I am typing quickly. I assume that this is because the smallest finger on my left hand rests on the 'a' key while my right index finger must move up to the 'u' key. Regardless, I must

correct this misspelling quite often even though I never misspell 'language' in any other venue. Knowing the possible cause of my mistake has done nothing to correct the problem alone, as fixing the problem will require retraining System 1 to properly sequence the letters of the word 'language'[16].

System 1 processes must also account for a variety of experiences to provide reliable priming and behavior. System 1 requires not only repetition but variety in its training. Artificial intelligence offers another example of the nature of System 1 and the risk of training a process without enough variety. Cognitive models inspired many of the approaches and algorithms of machine learning (Sun, 1997), specifically System 1. As described further in Section 4.2.1.3, machine learning iterates through large sets of data to gradually train computers on complex tasks like facial recognition. Recent studies have shown that facial recognition solutions are significantly worse at recognizing the faces of people of color because these groups are underrepresented in the training set (Simonite, 2019). Intensive training of neither AI models nor System 1 guarantees that their behavior will extend beyond the variety of the input/experience. System 1 behaves based on what it has experienced in the past, leaving System 2 to deal with new circumstances and make appropriate choices.

4.2.1.2 Introducing System 2

System 1 supports our mental processing of familiar tasks, but System 2 kicks in when things are new or strange, and prior experience provides no clear guide. Once System 2 takes on a challenge, it requires focus and attention. When distracted, System 2 risks forgetting what it was considering. Kahneman (2011) provided examples of System 2 tasks like

> Look for a woman with white hair
> Tell someone your phone number
> Check the validity of a complex argument (p. 22).

To search a crowd, System 2 focuses on the specific feature, white hair, filtering out other stimuli while combing the masses of people. System 1 continues to function and may interrupt System 2 if it sees a familiar face, but System 2 can also choose to filter out all other stimuli other than its

[16] System 2 can help this retraining effort by calling attention to my mistake, but only through repetition will I come to consistently type the word 'language'.

current task. Kahneman described the study by Chabris and Simons (2011; Simons & Chabris, 1999), who asked participants to count the number of passes made by a basketball team during a short video. Many of the viewers were quite successful at counting the correct number of passes but failed to notice a woman in a gorilla suit walk through the game in progress. System 2 gives us the concentration to handle new and unusual tasks, yet in doing so, narrows our focus and perception.

System 2 leverages the speed and automation of System 1, so much so it is challenging to describe System 2 fully in isolation from System 1. In many ways, System 2 acts as an emergency backstop for System 1.

> The main function of System 1 is to maintain and update a model of your personal world (Kahneman, 2011, p. 71).

System 1 efficiently manages our response to the world so long as it conforms to our model. When what we are experiencing violates the normal responses of that model, System 2 has the option to change our response. Our brain handles decisions like that of a medieval castle: the gate guards (System 1) handles everyday transactions with merchants and peasants where the king (System 2) is summoned only when receiving an unexpected and important visitor. Like the king, System 2's time is too precious to waste on mundane matters. Kahneman (2011) goes so far as to call System 2 *lazy* and not just for simple decisions.

> when people believe a conclusion is true, they are also very likely to believe arguments that appear to support it, even when these arguments are unsound. System 1 is involved, the conclusion comes first and the arguments follow. (p.45)

Priming not only prepares System 2 for making decisions, but System 1 also offers a degree of confidence in that decision. Most of the time, System 2 "adopts the suggestions of System 1 with little or no modification" and "is activated when an event is detected that violates the model of the world that System 1 maintains" (p. 24). System 2 generally accepts System 1's suggestions until it is 'uncomfortable' that the current situation does not match past experiences sufficiently. System 2 might choose to offer an alternative response or keep the chosen response of System 1.

System 2's laziness can manifest itself as an unwillingness to challenge System 1, or merely that System 2 has better things to do. Kahneman (2011) offered the example, "Drive a car on an empty road" (p. 21). Many drivers have become lost in thought or enjoying the radio while

driving down a long stretch of familiar road. Minutes or even hours may seem to pass with little awareness of the minor adjustments to the steering wheel or gas pedal required to stay in your lane and avoid cars. On many occasions, I become so lost in contemplation that I navigated towards a habitual, yet unintended destination only later recognizing my mistake. System 1 can manage complex tasks in parallel with other complex deliberations of System 2. A sudden change to the familiar routine (e.g., seeing a deer or police car) will snap System 2 back into the (in this case, literal) driver's seat. Our brain is often content to let System 1 make major decisions when they are routine.

The interplay between System 1 and 2 helps to define each of the Systems and offers insights when our cognition goes awry. Writing code requires a programmer to remain focused and attentive, yet as we saw in Section 2.3.3, experts are often unaware of their reasoning. They learn to see patterns in code, but not through instruction, rather through experience. Design seems to rely as much on intuition as it does on deliberate planning on which algorithm to choose and how to translate that algorithm into code. System 2 relies on System 1 for contextual priming to jumpstart reasoning. System 2 rarely (if ever) begins its work tabula rasa. Before System 2 engages, System 1 activates memories and offers insights based on collective experience. For example, fill in the missing letter of the following word:

CO_E

What word or words came to mind? Did a word immediately spring to mind? At any point, did you search your memory for words that begin with a CO and end with an E? Dual process theory suggests it is likely the word CODE came quickly to mind since the start of this paragraph primed your System 1 to recognize that word over several other alternatives[17].

System 2 is more effective when leveraging the networks of information System 1 has gathered from experience than when it must acquire knowledge on its own. The prior paragraph created an *associative activation* between the story of how experts design and the word code, intentionally used several times. System 1's priming activates one or more related networks of information related to the stimulus within System 2. Reading about computing education made *code* a more likely choice than if you were reading about soft drinks (coke), road construction

[17] In my search of dictionary.com, you could have chosen the words coke, cole, come, cone, cope, core, cose, cote, cove, or coze

(cone), or apples (core). While System 2 is free to pursue any line of inquiry to solving a problem, the effects of priming create a strong attraction to solutions we have seen in the past, whether these solutions are eventually helpful or not. Priming provides a powerful shortcut for cognition, but for the inexperienced, it can confound System 2 more than it helps. Priming impacts our thinking in unexpected ways.

> When we meet or are reminded of an acquaintance, an unconscious mental process may begin that "primes" us to initiate behaviors characteristic of that individual. Some studies have shown that college students exposed to descriptions associated with the elderly—"Florida," "gray," "bingo," and so on—subsequently walk down the hall more slowly after the experiment is finished, in line with the stereotype of the elderly as slow and weak. Similarly, "priming" words or images related to the stereotypical idea of a nurse leads to greater helping behavior, and cuing stereotypes associated with politicians results in more long-winded speeches. All these effects appear to occur unconsciously, without the participants being aware of how their behavior has been influenced. (Bargh, 2014)

Bargh's research shows that recent experiences change the way we respond in very unconscious ways. Our perceptions trigger memories of past experiences that influence our next actions.

The computing education research discussed in Section 2.3.3 seemed to have identified this exact reaction amongst experts. Rather than creating designs based on superior analytical skills or learned processes, experienced programmers seem to use patterns learned through experience in making decisions. For example, a decade ago, I moved from a position as a software architect at a bank to the same role in a health insurance company. My recent experience at the bank was in bill payments and credit cards, contrasted with my new team, who held decades of collective experience automating insurance claims. My manager assigned me to a project that proposed to use a specialized debit card to make individuals more aware of their spending on health care[18], an interesting bridge between these two otherwise separated worlds. The main difference is one of timing. Debit card payments happen (relatively) instantly for a known amount due. Insurance payments may take a week or more after the time of service to reconcile since they must account for many additional factors (e.g., negotiated rates, deductibles, co-insurance, out-of-pocket limits) that can change daily. The current team struggled to see a viable solution to blend a banking solution within the health insurance paradigm.

[18] The policymakers did not seem to do the research on how credit cards vs. cash influence spending (Abramovitch, Freedman, & Pliner, 1991), but the program was soon scrapped anyway.

Experience had primed my new teammates to think of the problem in terms of health insurance solutions, where my experience was that of banking instead. Under the rules of insurance claims, providers are paid only after adjudicating a series of rules to ensure accurate information and fair compensation. Paying a provider before considering these rules was an anathema to the entire foundation of insurance claims. My time in the world of finance suggested this problem was nothing more than shuffling money between ledgers of credits and debts. The transaction was not an official claim yet, but merely an advance on a forthcoming claim. The provider could receive funds today as a credit, which would balance upon approval fo the claim; otherwise, future claims would settle against the credit already extended. An existing process was already in place to handle such adjustments to payments, except for the new debit card as a means of transferring money. The necessary changes were far from trivial, but the required processes existed. The team simply lacked the experience to frame the problem as banking rather than insurance. System 2 is often blinded to options, as Kahneman (2011) put it, "because you were not aware of the choice or the possibility of another interpretation" (p. 80). I was able to offer my team a new choice based on the happenstance of my prior employment. Given that slightly new perspective, their expertise in insurance took over, and we created a successful solution.

Education tends to value the 'clear' rational thinking of System 2, but dual process theory tells us that System 2 is nothing without the right support from System 1. Researchers speak of intuition, but often only as the mysterious advantage that experts possess, and novices must someday acquire. Dual process theory suggests intuition is not just useful in experts but essential for *knowing how to program.* Researchers often overlook the importance of System 1 when describing the Cartesian view of rational thinking[19], but dual process theory helps us to see where intuition and 'non-cognitive' factors play into our reasoning.

4.2.1.3 Better understanding System 1 and 2 through an AI example

The science behind dual process theory provides abstractions of brain functions that are not only useful for describing human cognition but in developing AI. AI research benefits from dual process theory (Sun, 1997; Sun, Merrill, & Peterson, 2001; Sun, Slusarz, & Terry, 2005), and dual process theory has evolved with insights from AI (Evans & Frankish, 2009). Considering

[19] Perhaps because they lacked a way of describing it?

an example from AI further illustrates dual process theory's concepts. AI researchers simulate the interactions between neurons in the brain by creating an artificial neural network (ANN), a field often called *deep learning*. The mechanisms modeled in deep learning are the same as those in System 1, so we can simultaneously learn about an important aspect of computing and computing education at the same time.

An ANN uses simple math to represent the exchange of electrical impulses and chemical stimulations between neurons in living brains. AI developers 'teach' neural networks by repeatedly presenting inputs (e.g., pictures of faces) and comparing the ANN's result to the desired outcome (e.g., the identity of the matching person). Whereas neurons learn through chemical changes and growing new connections, ANN learning occurs mathematically. AI programmers write algorithms that slightly tweak a matrix of numerical values (i.e., weights) to better approximate the desired answer from the last input. A training session typically includes many thousands or millions of input/desired output combinations to train an ANN properly. Each new input brings the ANN a bit closer to the desired behavior, but the more complicated the problem, the more time it takes to learn.

Traditional programming languages act like System 2. Code represents a logical flow of decision making, and we can see and fix errors directly. System 2 is similar, in that, you can correct your memory if you attribute the wrong name to a person[20]. Instructors can access the 'inner state' of System 2 thinking because students are aware of their reasoning. Neither AI programmers nor educators can debug the inner working of neural networks except to offer refined training cycles that better reflect the desired output. As noted in Section 4.2.1.1, the reliability of a neural network's output requires a comprehensive dataset. If a training set (e.g., the diversity of faces, the types of problems assigned for homework) do not cover all intended uses of the neural network, it will not respond as desired in all circumstances. When new inputs extend beyond training, System 1 returns its best answer and leaves it up to System 2 to make the final decision. System 2 can sometimes infer the correct path but is at a disadvantage when System 1's priming does not actually align to the needs at hand. Without useful priming, System 2 may have the wrong knowledge activated that may not transfer to the current problem. Asking students to *transfer* their

[20] Such mistakes are not always so easy, however. I played Ultimate with an undergraduate and for some reason I picked up the wrong name of Curtis, his name was Ethan. It took me a few weeks of conscious effort to stop calling out Curtis during the game when my System 2 was focused on playing and System 1 clung to the wrong name.

learning is challenging since transfer often relies on either relevant prior experience acquired by System 1 or a particularly alert and clever System 2 to move beyond insufficient or inappropriate priming. Dual process theory suggests that instructors should only expect transfer in specific circumstances.

Cognition and computation solutions use a blend of explicit and implicit logic to tackle complex problems. Kahneman used the example "Look for a woman with white hair" as a function of System 2. Searching a crowd for a particular person requires concentration but leverages System 1's ability to match patterns (e.g., look for something white or shaped like women's hair). System 2 determines the best search pattern (e.g., broadly scan the crowd or methodically look through each area of people). System 1 supports this strategy by indicating when to ignore an area or drawing attention when spotting the desired patterns. System 2 would quickly grow tired with the arduous task of carefully evaluating each person spotted (and spotting a person already leverages System 1). Utilizing System 1's prodigious pattern matching abilities with a flexible strategy for searching based on the environment maximizes the likelihood of finding the desired woman while minimizing the required effort. Programmers use a similar strategy to find specific features within computer vision systems.

Some AI solutions can require users to provide very specific inputs, making their work simpler. For example, a facial recognition system could use either driver's license or passport photos that all have specific dimensions and fill the picture mostly with a person's face as a reliable form of input. Such systems are good at matching identities, but only if the input also is similarly framed. To create a solution to find a person in a crowd is significantly more complex. Imagine trying to teach a computer to find the famous character from children's books, Waldo. The illustrator of "Where's Waldo" books is notorious for creating densely packed scenes with many people, objects, animals, and somewhere in the chaos, the distinctly dressed Waldo. Waldo's iconic striped shirt, blue jeans, hat, and glasses make him instantly recognizable, yet deceptively hard to find at a glance. While I always hope to spot a flash of stipes or his red and blue theme, my success usually requires a systematic search that focuses my pattern matching to likely areas where he might be lurking. An AI solution would take a very similar approach. One deep learning approach for finding specific content within images is to break the full image into small pieces (see Figure 4.2) and submit each sub-image to an ANN trained in 'seeing' the target. The result

of each sub-piece is a numerical value that represents the likelihood of the target (e.g., Waldo) being in that image. The image with the greatest value is most likely the place to find Waldo.

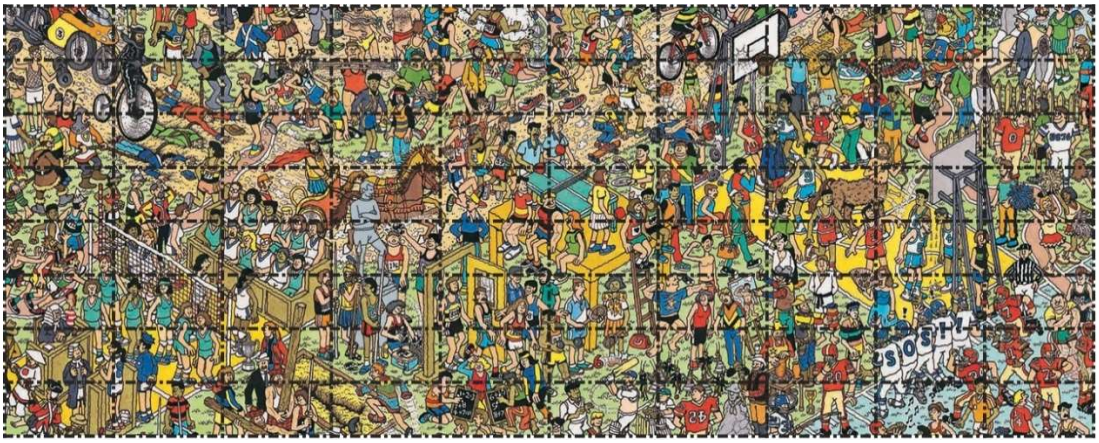

Figure 4.2. A sample search grid of a "Where's Waldo?" puzzle [21]

Earlier, we saw how the simplified depiction of Deep Blue, using a computer's vast memory to consider all possible moves, overlooks the role of System 1. Deep Blue's chess strategy relies on vital information obtained through experience in addition to logic (e.g., opening moves, patterns of pieces). Visually identifying an object in unpredictable space opens up entirely new challenges compared with automating chess. A chessboard contains only 64 potential locations, and the computer knows perfectly where each piece resides. Computer vision, even the simple example of finding Waldo, introduces uncertainty beyond simply which of all known possible moves the opponent will make. Waldo's illustrator is generally kind enough not to overly obscure his image, but computer vision in the real world must contend with weather, motion, changes in appearance, and the uncertainty of similarly looking targets or simply a profile view rather than a direct image.

Engineers creating ANN solutions seek large and diverse sets of data from which to train their solutions, with one facial recognition company claiming to have amassed over 3 billion images to train its network (O'Flaherty, 2020). Teaching neural networks complex tasks requires repetition and variety, and even then, are only as 'good' as their training. Evolution has provided living systems with specialized subsystems for processing different aspects of vision (e.g.,

[21] Modified image retrieved from http://clmmag.theclm.org/home/article/Wheres-Waldo. For those purists, the actual algorithm would likely use overlapping segments instead of the easier to draw clean grid, but that imagery is more difficult to render in a static image. Oh, and Waldo is in the first column, sixth row if you did not find him.

focusing on shapes, color, movement), yet even a human's powerful ability to recognize faces breaks down when the faces do not meet our expectations. Research shows that people have trouble recognizing faces when the arrangement of eyes, nose, and mouth no longer conforms to what we expect, for example (Dowling, 2018). System 1 processes, like ANNs, are not particularly flexible beyond their training.

While cognition as a whole can be very flexible, skills that reside within System 1 generally do not transfer beyond their training. Avi Karni and Dov Sagi (1991) tested the *plasticity* of the visual processing center of the brain. Plasticity describes the restructuring of the brain's network in response to learning and when accommodating for injury. At the time of the study, neuroscientists believed that "visual processing is… 'hard-wired' in adult mammals" (p. 4966), Karni and Sagi sought to prove that the brain continually learns even in its core areas involved in vision. Their test created a simple 'texture' on a computer screen of vertical or horizontal dashes and asked participants to identify three successive virgules (/ / /). As quickly as possible, the participants needed to find the outliers and indicate if the three characters aligned or horizontally (see a. and b. respectively in Figure 4.3) with only a brief glimpse at the screen.

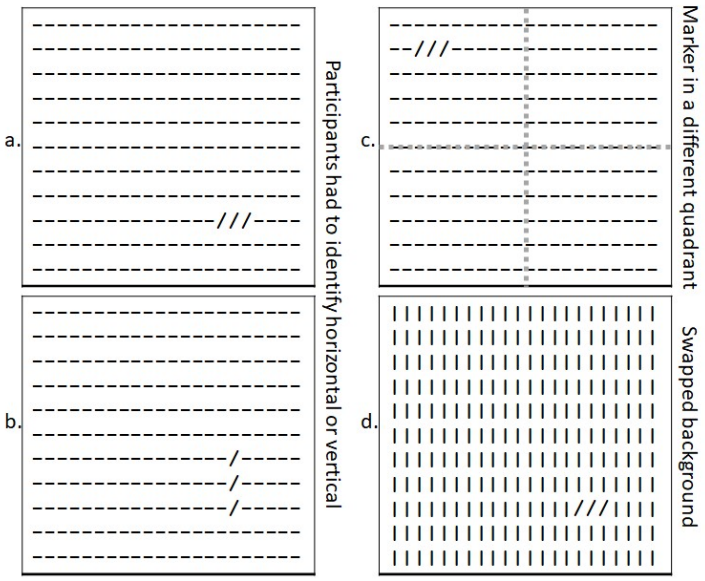

Figure 4.3. Replicated displays from (Karni & Sagi, 1991)

Karni and Sagi found their participants indeed improved over time, a sign of continued learning long after physical maturity. With practice, they identified the orientation of the outliers

more accurately, even when seeing the image for less time. The participants responded very quickly (< 200 ms), meaning there was no time for System 2 deliberation, yet the smallest changes to the test erased any gains the participants made through changing. To perform well, each eye required individual training. Merely moving quadrant where the outlying slashes occurred (e.g., a versus c. in Figure 4.3) erased the speed and accuracy gains of training, as did swapping top-to-bottom (a versus b) or the texture (d). Karni and Sagi's research show that some aspects of learning are highly bound to specific stimuli and do not transfer to even the same stimuli occurring in a slightly different place. This type of behavior and learning is typical of System 1 and points to the need to not just train for localized mastery, but any expected use of a skill. If, for example, when students only ever use geometry in the classroom, they may not think of the Pythagorean theorem to ensure the deck of a porch rests at a right angle. A well-trained System 1 is fast and accurate but often limited in how it supports System 2. Conversely, System 2 is flexible and creative, but often relies on System 1 to tackle (easily) problems that we struggle even to describe, much less solve programmatically.

4.2.2 Refining the dual process model

The prior section used the descriptions of Daniel Kahneman (2011) to provide an introduction to dual process theory. This section further explores the dual process model to revisit the main ideas (i.e., help train our System 1 through repetition) and expand on specific areas that may relate to education and programming.

4.2.2.1 The many processes of System 1

Creating awareness of System 1 is the most valuable addition that dual process theory offers, yet System 1 is also almost entirely a vague abstraction. We can talk quite a bit about what System 1 does, but little about how. Reber (1989) extensively researched intuition, offering numerous compelling studies yet concluded by saying:

> There is probably no cognitive process that suffers from such a gap between phenomenological reality and scientific understanding. Introspectively, intuition is one of the most compelling and obvious cognitive processes; empirically and theoretically, it is one of the processes least understood by contemporary cognitive scientists. (p. 232)

Reber has little doubt that intuition exists, is important, and should be fostered. At the time of his writing, and maybe even today, we have little understanding of what goes on behind the curtains of System 1. Some dual process theorists contend that the moniker 'System 1' is too generalized and potentially misleading. They fear using it leaves the false impression of a single region of the brain where all such behaviors exist, preferring to describe *Type 1* and *Type 2* ***processes*** rather than *Systems*. Some dual process theory literature replaces System 1 with a collective group of Type 1 processes called *The Autonomous Set of Systems* or ***TASS*** (Evans & Frankish, 2009, p. 56). The difference between System 1 and TASS is mere nomenclature, but the implication is notable. System 1/TASS is far from monolithic, and each process is distinct and potentially unrelated unless trained otherwise. It is like assuming that sending one employee to project management training will make every coworker a project manager. True mastery comes with integrating skills across a variety of experiences. The introduction of System 1 serves to remind educators of the importance of experience and practice as well as gathering knowledge.

Experience defines an individual as much or more than genetics. Our brains are certainly 'hardwired' to perform certain behaviors and perform them effectively. Squire and Kandel (2003) noted that DNA guides some of the early formations of our nervous system. "A given neuron will always connect with certain neurons and not others" (p. 35), but whether these connections flourish or wither depends on their use. Experiments on animals showed that when the optic nerve is detached and later reconnected, the animal recovers vision, but the brain needs time to relearn and may never be as adept (Dowling, 2018). System 1 requires time to learn, and while those skills fade slowly, they diminish if unused. When a complicated skill requires some aspect of System 1, System 2 can only compensate so much until the System 1 process matures. One such example of the importance of System 1 is the tragic tale of Genie and the quest to teach her to speak.

> Genie was born in April 1957. When we first encountered her, she was 13 years and 7 months old-a painfully thin child who appeared six or seven years old. When hospitalized for malnutrition, Genie could not stand erect or chew food; she was not toilet trained; and she did not speak, cry or produce any vocal sounds. The reconstruction of her previous life presents a bizarre and inhuman story. From the age of 20 months, Genie had been confined to a small room under conditions of apparently increasing physical restraint. In this room she received minimal care from a mother who was herself rapidly losing her sight. She was physically punished by her father if she made any sounds. Most of the

> time she was kept harnessed into an infant's potty chair; otherwise she was confined in a homemade sleeping bag in an infant's crib covered with wire mesh. She was fed only infant food. (Curtiss, Fromkin, Krashen, Ringler, & Ringler, 1974, p. 529)

Genie's parents denied her the basic human interactions that help our brain to communicate and form bonds with others. Her cruel deprivation left her brain deprived of the experiences it needed to acquire spoken language and possibly many other typical human behaviors.

> When Genie was first admitted to the hospital, there was little evidence that she had acquired any language; she did not speak. Furthermore, she seemed to have little control over the organs of speech…
> It appeared, therefore, that Genie was a child who did not have linguistic competence; i.e., who had not yet acquired language. (Curtiss et al., 1974, p. 530)

Genie initially did not seem to comprehend the speech of her caregivers and therapists. With time and training, she started to understand and eventually speak, but her path and proficiency varied significantly from traditional language development in children.

While Genie eventually started to speak, sadly, she is not the only example of a child suffering such treatment. A young Florida girl, Dani, was found in a similar condition, yet while she matured in many aspects of cognition, she had not started talking as Genie did (Degregory, 2017). Even while our brains are predisposed to certain types of learning, such as language, we still need experience and practice. Physical traits (e.g., genetics or brain damage) ***and*** experience alter cognition and learning. Dani's story might lead some to believe that after a 'prime' period, learning is difficult or impossible, where Genie's story showed that growth is possible, if slow. Genie and Dani's story of isolation and neglect provide heart-rending accounts of how vital experience is in learning. Many aspects of language processing are automatic and occur within System 1 (see Section 7.3.1), and System 2 can only go so far to compensate.

Studies of behavior combined with neuroscience support the role of experience as a critical guide in learning. In his chapter on dual process theory, Lieberman names the two Systems as X and C, "RefleXive and RefleCtive" (Evans & Frankish, 2009, p. 45). He describes system X as residing within the basal ganglia within the brain, tied to implicit learning. System C lives within the temporal lobe and hippocampus, associated with explicit learning[22]. While much of dual

[22] The hippocampus is also involved in processing novel experiences (MacKay, 2019)

process theory describes how the two Systems model *thought*, Lieberman also looked at how System 1 acquires skill by discussing a study by Knowlton, Mangels, and Squire (2016). The study tested the relationship between implicit and explicit learning by comparing participants with amnesia (who are unable to form explicit memories) or Parkinson disease (who have difficulty creating implicit memories) against a control group of 'unafflicted' individuals. Researchers showed participants a series of cards with geometric shapes and asked if it would rain. The shapes equated to a probabilistic pattern telling if it would rain, but participants received no instructions and could only guess if it would rain. During the trials, participants might derive a pattern in the cards forecasting rain (either explicitly or implicitly) or continue to make chance guesses. After many iterations of guesses, amnesic and control participants began to correctly predict rain around 70% of the time, showing they were learning what the symbols meant to some degree. The Parkinson's participants made correct guesses at the same level of chance (50%), showing they were not learning the patterns. Since the amnesic patients could not explicitly learn such information, it confirmed not only the presence of implicit learning but its import in making good decisions in the absence of explicit knowledge. System 1 can learn without exposure to the rules or even conscious awareness that learning occurred.

System 1 not only acts but learns implicitly. Knowlton, Mangels, and Squire showed that learning happens without explicit knowledge, but other studies show that explicit training is not always beneficial. Berry and Broadbent (1988) used several computer simulations to test implicit versus explicit learning. One simulation, for example, asked participants to control a sugar factory by altering the number of workers at the plant. The participants either received explicit instructions on how the factory operated or jumped directly to the simulation without instructions. Berry and Broadbent both compared the participant's performance at the job (implicit skill) and their ability to answer questions about the task (explicit knowledge). They reported that "providing an individual with detailed verbal instruction that is understood and later remembered is not necessarily sufficient to improve task performance," in fact, "verbal instruction significantly improves ability to answer questions, yet it has no effect on control performance" (p. 229). The untrained participants seemed to implicitly learn the task, doing so better than their explicitly taught peers. The trained peers had the advantage in describing the process, yet this advantage did not make them better at the task. Berry and Broadbent's study supports the "dual epistemology" idea that 'knowing' is not just remembering, but also how knowledge is applied.

A mature and robust System 1 is important yet forms in ways that may not be covered by current pedagogy, particularly in computing education. Squire and Kandel (2003) commented, "The remarkable feature of learning is that it is often highly specific to the task and to the specific way in which the training is carried out" (p. 165). The implications from neuroscience and dual process theory test the efficacy of traditional learning theory, as will be discussed in the next chapter. Advances in the understanding of memory and cognition should alter how we perceive the acquisition of knowledge and skill. Squire and Kandel included a quote by Henri Bergson who described habits, the worker-bees of System 1.

> [It is] a memory profoundly different... always bent upon action, seated in the present and looking only to the future... In truth it no longer represents our past to us, it acts it; and if it still deserves the name memory, it is not because it conserves bygone images, but because it prolongs their useful effect into the present moment. (p. 175)

Each process within System 1 represents the sum of our weighted experience, not a memory of history or rules. System 1 may or may not provide the best response, but it offers a familiar reply from which System 2 can methodically analyze the best course of action.

4.2.2.2 Refining System 2 – Algorithmic processing

While System 1 describes a collection of various processes, System 2 may share underlying and multi-purposed resources, yet some dual process theorists describe two distinct 'parts'. Stanovich (2012) described System 2 as having "the algorithmic level and the reflective level" (p. 57). The algorithmic level represents the logical thinking that lies at the heart of System 2, where reflective processes include an individual's beliefs and goals. The reflective side is the topic of the next section, while this section examines how Stanovich's algorithmic System 2 provides cogent insights into programming.

Algorithmic System 2's main function is managing mental models of what might be. Programmers regularly work with mental simulations across numerous tasks. They must consider potential designs and algorithms, weighing their flaws and merits. Testers translate the requirements into the anticipated behavior that makes up test cases. Debuggers compare the predicted implementation against the erroneous execution to pinpoint faults in code. A critical part of the work of programming is creating mental models or *representations*. A primary

representation relies on sensory information, with each sight, sound, smell, and touch adding to a temporary mental model. Primary representations are grounded in sensory information, but algorithmic System 2 focuses on building and manipulating *decoupled representations*. Leslie (1987) proposed that children create decoupled representations to separate ideas from the concrete world so that they can pretend during play. When pretending, a child can control factors they cannot in the real world, testing their understanding of how the world operates. Stanovich proposed that System 2 forms such secondary representations to consider novel situations or alternative responses. Designers, testers, and debuggers must create decoupled representations to handle novel encounters.

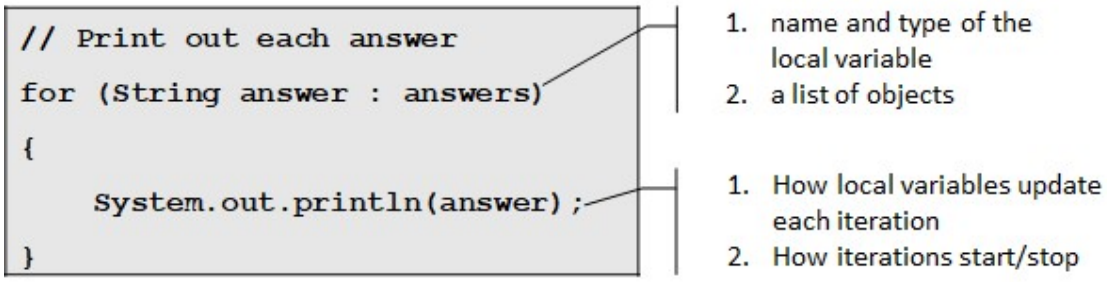

Figure 4.4. An example code snippet to demonstrate decoupling representations

Programmers create decoupled representations when they read code, but code presents many layers of potential representations. Determining the intent of the code in Figure 4.4 seems obvious – the preceding comment (//) states the end result – but Stanovich suggests that the novice programmer must blend two primary representations into a single mental model. The code itself forms the first primary representation, with the output of the running code being the second. The novice System 2 must align each line of code with its corresponding output, which does not seem terribly demanding in this simple example. Yet, as the notes within Figure 4.4 suggest, it still contains several important concepts to discern. Identifying each 'moving part' is critical to understanding the full purpose of the construct. Even a simple task, like reading code, demands that algorithmic System 2 manage several representations. Experienced programmers have an easier time reading code, as we will see in Section 7.6.2, since they have the support of System 1 to interpret syntax, analyze output, and even divine purpose.

What makes programming particularly challenging is the need to create, manipulate, and reconcile multiple decoupled representations. When presented the example in Figure 4.4, a

programmer must simultaneously interpret the syntax, compute the output, and even divine the purpose of the algorithm to fully 'understand' it. Experts do this quickly and effortlessly, but we saw in Chapter 2 that novices often have difficulties with any of these, much less all of these at once. Figure 4.4 even shows code that compiles, runs, and completes its intended purpose. A programmer's System 2 may face its most difficult test when the code does not operate as expected because it has a bug. Debuggers must consider both the code they believe should work and the output that says it does not. "[W]hen considering an alternative goal state different from the current goal state, [a debugger] needs to be able to represent both" (Stanovich, 2012, p. 63). Debuggers must simultaneously hold a representation of the code they read/wrote and the observed output of the buggy code. The challenge of debugging is "the latter must not infect the former while the mental simulation is being carried out" (Stanovich, 2012, p. 63). It is too easy for the expectation of what the code 'should do' to corrupt the mental model of what the code is doing. Kahneman (2011) called the tendency for us to overlook conflicting information – such as an incorrect execution of our code may not convince a programmer their code has a bug – as the *confirmation bias*. The confirmation bias makes us less likely to seek evidence that disturbs, and instead seek information that confirms, our mental model. Rather than tracing their code, a programmer may first blame the test case, as an example. Clancy (2004) placed the confirmation bias as a severe challenge for programmers. Debugging can be challenging for competent programmers who have the skills to eventually look beyond their confirmation bias. Novices carry the additional burdens of managing their fledgling language skills, understanding algorithms, and how to debug, so they may already doubt their knowledge. One reason that intelligence is highly valued amongst programmers may be how a strong algorithmic System 2 helps in managing representations in the absence of System 1 support.

Algorithmic System 2's aptitude can vary across individuals depending on their *fluid intelligence*. Fluid intelligence measures a person's ability to find relationships between seemingly unrelated items and solve problems without first requiring mastery of the subject. Fluid intelligence's counterpart is *crystallized intelligence,* which reflects the sum of individual learning[23], or more simply, what they know. In some ways, these constructs resemble those of

[23] Crystallized versus fluid intelligence looks to separate the available educational opportunities from the capabilities of physiology (e.g., genetic predisposition, brain damage). Intelligence testing use this distinction to divide intelligence based in educational opportunities from 'raw talent'. Tests of fluid intelligence focus on talents in

Hatano and Inagaki (1984). A learner develops procedural expertise (crystallize intelligence, System 1 processes) through study and practice. Algorithmic System 2 is critical in helping adaptive experts extend their mastery to new problems. Stanovich might suggest adaptive experts are better at forming decoupled representations of problems.

> I have conjectured that the raw ability to sustain such mental simulations while keeping the relevant representations decoupled is likely the key aspect of the brain's computational power that is being assessed by measures of fluid intelligence (Stanovich, 2012, p. 63)

To think creatively, a person must parse sensation into a working model and maintain the separation between observation and plan while manipulating their model towards the intended goal. The scope of fluid intelligence also integrates factors beyond memory, such as attention and resistance to distractions, each critical for complex manipulations within System 2. System 2 demands focus because any distraction risks disrupting representations in short-term memory. Without System 1 support, some new programmers may struggle compared with their peers who possess a greater fluid intelligence.

To be the most effective, System 2's algorithmic thinking needs support from System 1. The literature on tracing in Chapter 2 reported many examples where novices struggled to trace code. Several of the authors from Chapter 2 also discussed the role of cognitive load during tracing and aligns well with Stanovich's model of managing decoupled representations. Without mature System 1 support, novices must juggle multiple primary representations of the code and evolving trace. The next chapter explores an example of a student juggling mental models while tracing and the evidence that their thinking requires frequent shifts in context (Section 5.2.2). Stanovich's algorithm level of System 2 fits well in explaining many types of novice struggles. What the algorithmic level does not spell out is why some novices who excel tasks like tracing still struggle to write code.

---

grouping, classification, analogy, forming relationships and other mental task not focused on semantic knowledge alone. For more see (Horn & Cattell, 1967)

4.2.2.3 Refining System 2 - Reflective processing

Stanovich (2012) presented an enticing example demonstrating the two levels of reasoning within System 2 that aligns with the logic-book concept presented in Chapter 3. He points to a study which asked if the following conclusion is valid:

> Premise 1: All living things need water.
> Premise 2: Roses need water.
> Therefore, Roses are living things. (p. 60)

If you agree with the 70% of university students that this is valid, then you have succumbed to a failure in your reflective processing. A philosopher would break down the argument as:

> Premise 1: If P then Q
> Premise 2: Q
> Therefore, P

When phrased as such, the argument is easier to see as invalid, falling to what Talbot (2014b) called the "fallacy of affirming the consequent" (location 3010). "Clearly, the believability of the conclusion is interfering with the assessment of logical validity" (p. 61). Stanovich argued that the truth of the conclusion makes the argument seem valid. Consider this argument.

> Premise 1: All living things need water.
> Premise 2: Tile saws need water
> Therefore, tile saws are living things.

It is much less likely that you thought this argument was valid since the statement that tile saws are living things seems wrong, where a rose felt right. The argument violates norms, which Stanovich placed as a value that people hold. Rather than fluid intelligence, the reflective mind processes tasks based on an individual's disposition. Rather than working from pure logic alone, reflective System 2 alters its response based on our values.

The algorithmic and reflective levels of System 2 coordinate their work, yet each has a distinct mission to complete. On the surface, both levels of System 2 are similar as "the algorithmic and reflective mind will both share properties (capacity-limited serial processing for instance) that differentiate them from the autonomous mind" (Stanovich, 2012, p. 58). The algorithmic level works with System 1 to build decoupled representations and complete simulations of alternative realities to test what could be. The reflective level performs critical thinking tasks, perhaps comparing simulated results with the results System 1 predicted from

experience or other simulations. A person's beliefs and goals add weight to the evaluation and subsequent decision to form a response. The response could be to go with experience (System 1's "gut response") or replace it with an alternative.

Stanovich touted hypothetical thinking as a critical capability of the reflective System 2. "All hypothetical thinking involves the analytic system, but not all analytic system thought involves hypothetical thinking" (p. 68). Stanovich claimed the algorithmic level is responsible for overriding System 1, yet this is not hypothetical thinking, but a direct comparison. Besides being lazy, algorithmic System 2 can also engage but do so using false pretenses. Say a programmer starts a code trace with a faulty decoupled representation of the code (e.g., they misread a function name) or 'bad' System 1 prompt (e.g., they execute a line of code incorrectly). Unless the reflective level catches the error using hypothetical thinking, the algorithmic level will simulate the code using the wrong information, never realizing the problem. An experienced coder might be more likely to detect something that went wrong, but all programmers are vulnerable. System 2's algorithmic processing, in conjunction with System 1, does not consider what is 'right', simply the task at hand. It is up to the reflective processing in System 2 to discern if the results are meaningful and appropriate. Stanovich's two levels of System 2 might explain how some people can recover from misdirected System 1 prompts while others cannot or take much longer to do so.

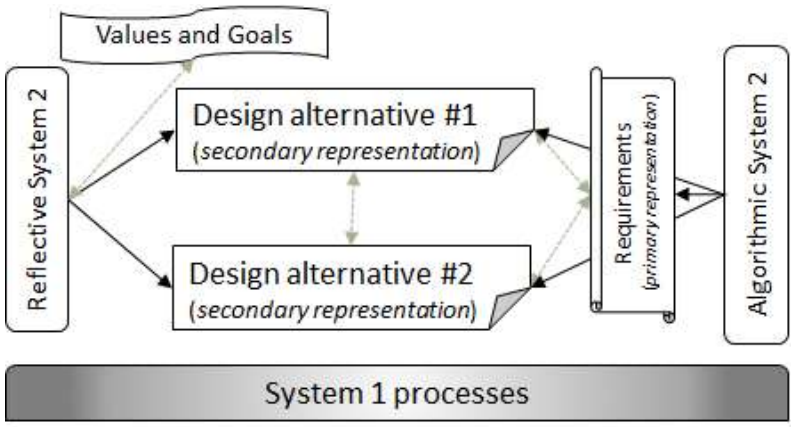

Figure 4.5. Mental representations involved in design trade-offs

Design demonstrates another example of hypothetical thinking. Designers use algorithmic processing to form alternatives decoupled from their mental model of the requirements (see Figure 4.5). Reflective processing engages in hypothetical thinking to compare design alternatives, influenced by the designers' personal views of what makes a 'good' design. Does the designer seek the most efficient algorithm or the simplest? Do they minimize the use of memory

or maximize throughput? What programming language do they choose? The designer's preferences influence the reflective mind, combined with logical reasoning, to pick their preferred alternative. The inclusion of goals and beliefs explains why two equally informed and rational designers can choose different alternatives based on the same requirements; they hold different beliefs/goals.

Hypothetical thinking builds upon the decoupled representations introduced in algorithmic processing. Each design alternative becomes a "secondary representation – the decoupled representations that are the multiple models of the world that enable hypothetical thought" (Stanovich, 2012, p. 63). A designer must balance at least three representations in order to weigh options and trade-offs. At a minimum, the designer must consider the needs for the design and compare two or more alternatives. Designers working in complex systems or comparing many options need exceptional fluid intelligence or System 1 support to manage the permutations!

It is at this point that Stanovich's model might break down, at least in aiding a discussion of programming education. Stanovich talked about how a conflict between the two Systems favors beliefs over intuition. For example, I am not fond of spiders, particularly when startled by one, and my instinct when startled is to slap and swat at whatever landed on my body. A long time ago, I learned about how spiders not only prefer to avoid me but are very helpful at managing other pests. Now when I encounter a spider without being startled, my intuition might say swat at it, but my acquired value for not needlessly killing things[24] generally wins. I instead escort my arachnid housemate outside. It is one thing to talk about personal beliefs such as valuing all forms of life, but where do programmers acquire their beliefs about preferred software design? Is it reason or habit that tilts a programmer to prefer Java or Android over C or iOS? Software design seems to hold many reasonable alternatives for choosing an appropriate design, so where do beliefs and goals 'live' in memory? Stanovich's model holds compelling ideas about how programmers make design choices, and Section 7.6.3 considers how dual process theory impacts the way designers think and thus how novices learn this critical aspect of programming.

[24] Mosquitos being the exception as I am sure they have violated a pact somewhere

5. PROGRAMMING AND DUAL PROCESS THEORY

At this point, it is hopefully becoming clear how the dual process theory viewpoint of cognition and might offer alternative explanations for performance differences between experts and novices. The rational decision-making System 2 should feel familiar, plus the formation of mental models for those familiar with the notional machine, though the next chapter will cover that in some depth. For some, System 1 may be new while others may finally have a name to the oft-reported yet indefinable sense of intuition possessed by experts. Dual process theory presents a formal framework from which to deconstruct programming thinking and remodel it with better precision and explanatory power.

This section will revisit the contents of Chapter 2, this time through the lens of dual process theory. This section should provide further examples of the two Systems and how they relate to the tasks of programming. As discussed in Chapter 3, the developing argument for TAMP will continue to emerge making the case that dual process theory provides a superior model of cognition which starts to explain the nature of expertise and the obstacles that novices face in acquiring it.

5.1 The epistemology of what programmers "need to know"

Premise 1: Epistemology has a basis in individual cognition
Premise 2: Dual process theory changes the model of cognition typically assumed by researchers and instructors in studying/teaching programmers
Conclusion: Using dual process theory as a model of cognition enhances the epistemology of programming to better align with how professionals use knowledge

Dual process theory does not change the content of what programmers need to know but may change the meaning of 'knowing'. Dual process theory challenges the conventional epistemology of what it means to "know how to program". Epistemology is critical to philosophy and theory, yet not an everyday subject for even most academics. Before diving into how dual process theory changes epistemology, it seems helpful to define what is meant by it, particularly as in the current discussion has a slightly different flavor. Epistemology generally describes the reasons and method used to believe knowledge to be true.

> Defined narrowly, epistemology is the study of knowledge and justified belief. As the study of knowledge, epistemology is concerned with the following questions: What are the necessary and sufficient conditions of knowledge? What are its sources? What is its structure, and what are its limits? As the study of justified belief, epistemology aims to answer questions such as: How we are to understand the concept of justification? What makes justified beliefs justified? Is justification internal or external to one's own mind? Understood more broadly, epistemology is about issues having to do with the creation and dissemination of knowledge in particular areas of inquiry. ("Epistemology," 2005)

Unlike most subjects, programming has a clear and easy to trace epistemology; many of the creators of programming languages and environments are still alive! Programming is entirely artificial, thus being altogether made by humans here is no higher and unknowable natural law to consider, and if you feel strongly enough, you can create a new language of your own. For programming, and specifically programming education, Goldman's (1986) definition of epistemology is more descriptive.

> Epistemology, as I conceive it, divides into two parts: individual epistemology and social epistemology. Individual epistemology- at least primary individual epistemology- needs help from the cognitive sciences. Cognitive science tries to delineate the architecture of the human mind-brain, and an understanding of this architecture is essential for primary epistemology. Social epistemology needs help from various of the social sciences and humanities, which jointly provide models, facts, and insights into social systems of science, learning and culture. (p. 1)

Goldman splits epistemology into what the individual knows, possibly opposite societal consensus. Programming languages, at least popularly used ones, generally are formed based on a social agreement for their foundation, construction, and the rules of use. What is especially interesting in Goldman's split is that individual epistemology based on *cognition*. Programming epistemology has always included a dimension of knowledge and skills as part of 'knowing how to program'. Chapter 2 saw multiple authors describing programming as an active enterprise with as many verbs (e.g., understand, plan, code, trace, debug) as concepts (e.g., syntax, semantics, notional machine, patterns). In their descriptions of novice struggles, we can see that the cognitive model that many researchers use allows for only 'knowing' or 'not-knowing' programming and its parts. Fragile knowledge cataloged how novices toggle between these states at different times, yet cognition is absent from the discussion as a possible cause. Using Goldman's notion of individual

epistemology rooted in cognition opens the door to a new definition of 'knowing how to program' when changing the presumed model of cognition.

Under dual process theory, 'knowing how to program' involves forming knowledge and skills across System 1 and 2, and integration of cross-System skills. The main limitation of the current epistemology is the number of students who are successful in their CS1 experience yet seem to forget their programming, or even coding skills when they reach their next course. The remainder of this section describes how dual process theory accounts for such observations and reinforces the need for an epistemological shift in programming[25].

5.2 Novice struggles viewed through dual process theory

Dual process theory offers a new perspective for evaluating novice struggles. At a minimum, it offers a new way to view how information is learned and used. Within dual process theory, it is not unexpected for students to learn something yet sometimes forget. Misconceptions are not a failure of learning, but a core part of how our brain is designed to compensate within an unknown world. The theoretical frameworks of past studies have been useful in exposing the struggles novice face, but often unable to entirely explain the nature of the struggles, much less providing guidance to overcome them. This section will revisit four aspects of novices struggles to show the explanatory power of dual process theory and create a foundation for improvements to pedagogy and curriculum.

5.2.1 Fragile knowledge - Perkins and Martin (1985)

Students in many disciplines confound their instructors with the fragile knowledge they display. Fragile knowledge, when students seem to know something at one point, only to forget it later, was first discussed within computing education by Perkins and Martin (1985) and through the years has become a popular way of describing novice learning (Ben-Ari, 2004; Berges, 2015; Eckerdal et al., 2007; Fitzgerald et al., 2008; Ko & Myers, 2005; Lister, Fidge, & Teague, 2009;

[25] What the shift in epistemology looks like is the subject of Chapters 5 and 6. As a brief preview for the curious/impatient, the ripples of dual process theory should start at the top. It may mean shifting the expectations of the content of CS1 courses and certainly alters the assessments to determine if novice programmers are 'ready'. It certainly does not throw out most of what is taught or even how it is taught, instead suggesting a restructuring and ways of augmenting practice.

Lister et al., 2004; Lopez et al., 2008; McCauley et al., 2008; Qian & Lehman, 2017; Robins et al., 2003; Sajaniemi & Kuittinen, 2005; Sorva, 2013; Teague, 2014; Vainio & Sajaniemi, 2007; Whalley & Kasto, 2014; Xie et al., 2018). Fragile knowledge is a useful construct for naming a specific type of struggle but naming the issue has not led to pedagogical interventions to fix it. Dual process theory offers a new take at the observations of Perkins and Martin and perhaps the alternative explanation it offers can invoke new ways of compensating for fragile knowledge.

Premise 1: An insufficiently trained System 1 leads to inert and misplaced knowledge
Premise 2: Conglomerate knowledge is exacerbated by inert and misplaced knowledge, requiring System 2 to compensate and risk become overloaded
Premise 3: Dual process theory leads to strategies for building knowledge in System 1 and 2
***Conclusion*:** Dual process theory does a better job explaining fragile knowledge, which can lead to better approaches to manage it.

5.2.1.1 Summary of the original study

Perkins and Martin (1985) investigated why some learners struggled to learn to program. Their inquiry differentiated two areas of student learning: knowledge of the programming language and problem-solving[26]. They described the leading problem that novices have in learning to program is a fragility in their knowledge. Fragile knowledge represents the middle ground of understanding between comprehension and ignorance. "Rather, the person sort of knows has some fragments, can make some moves, has a notion, without being able to marshall [sic] enough knowledge with sufficient precision to carry a problem through to a clean solution " (D. Perkins & Martin, 1985, pp. 6–7). Perkins and Martin have captured many educators' experiences with novices who may know how they want to solve a problem, but not remember the appropriate constructs. Or recognize the general construct to use but forget the correct syntax. Their study used a brilliant methodology to draw out examples of forgetting to seek its source organically.

The cornerstone of Perkins and Martin's study was allowing students to work freely yet having a researcher available to support as needed. The study included twenty high-school students learning the BASIC language. Each student selected one of eight progressively more difficult problems that was "challenging, not too easy, but not too hard" (p. 10) A researcher then observed

[26] The focus on problem-solving seems to continue the study of programming as a tool to build general problem-solving abilities (Seymour Papert, 1987; Pea, 1987; Pea & Kurland, 1984), though little of their report focuses on that aspect.

the student's work, occasionally asking questions to illuminate the student's thinking or to offer aid if they seemed to struggle. The questions were purposefully formulated to probe at the nature of the struggle, either an issue in problem-solving or with the language.

The protocol called for three specific levels of intervention by the researcher. The first level of probes was to ***prompt***. To create a prompt, the researcher would ask "high-level strategic questions one might ask oneself" (D. Perkins & Martin, 1985, p. 10), to trigger memories that may otherwise be inert. If the student continued to languish after several prompts, the researcher provided a ***hint***. Researchers offered hints based on their understanding of the student's current problem and the best path to the desired solution. If hints offered no help, the researcher ***provided*** the student with step-by-step instructions towards until the student could continue independently.

Perkins and Martin tallied the *prompts*, *hints*, and *provides* that the researchers offered to students. Since researchers attempted to always start with a prompt before moving to hints and finally provides, accompanied by questions, they could explore where the student struggled, and test what knowledge they were using or lacked. A prompt might trigger unused, but available knowledge from their mind. A hint might provide a missing problem-solving strategy but allow the student to show their knowledge of the underlying constructs. Providing the next few steps permitted the student to move past one issue and continue if they were able. Perkins and Martin captured qualitative data from audio recordings as well as quantitative data by tracking the progression of prompts, hints, and provides. They categorized fragile knowledge into four types.

- Partial- never remembered or never was learned/taught
- Inert- recalled at times, but not applied when needed
- Misplaced - used in a way that does not apply to the need
- Conglomerated – "code that expresses loosely the intent without following the strict rules that govern how the computer actually' executes code" (D. Perkins & Martin, 1985, p. 19)

The description of the four categories of fragile knowledge provides numerous insights into the novice mind. It is easy to assume that all fragile knowledge is partial, students learn some pieces of the puzzle, but the missing ones are entirely absent. The fact that simple prompt could invoke knowledge implies it was there, just not at first active. Perkins and Martin offer ideas on why this is so.

> The causes of such fragile knowledge seemed varied but comprehensible. Among the factors discussed were a sparse network of associations,

> underdifferentiation of commands binding of commands and programming plans to customary contexts without recognizing their generality, treating a programming language more like a natural language where one can say what one means in many ways, and, of course, underuse of general strategic questions to prompt oneself to better marshall one's knowledge. (p. 25)

Why is knowledge fragile? Students fail to connect facts about programming, struggle to decouple what they see, confuse programming with speaking habits, and do not self-regulate to "prompt oneself" to activate relevant tacit knowledge. But why do they do this? How do you teach them to stop? The prevailing view of cognition at the time of Perkins and Martin's study would not have answered any of these questions, but thankfully, dual process theory does.

5.2.1.2 Applying dual process theory to the study

Dual process theory can go a long way to explain the causes of fragile knowledge. Perkins and Martin do not express any specific cognitive model as part of their research, so given the period and lack of explicit description, a traditional model of memory and Cartesian thinking was probably implicit[27]. The characteristics of the sub-types of fragile knowledge are insightful and align well within dual process theory. The causes are likewise well thought out, but the explanations offer more abstract reasons that lead to suggestions of vague actions. Dual process theory can dissect each type of fragile knowledge and explain why it occurs.

The nature of fragile knowledge

Premise 1: Knowledge is inert when its recall is unreliable
Premise 2: Knowledge is misplaced when recalled in the wrong circumstances
Premise 3: System 1 recalls knowledge based on observations from the closest relating experience, which may or may not match the current situation
***Conclusion*:** An insufficiently trained System 1 leads to inert and misplaced knowledge

Partial, inert, and misplaced knowledge seems to be rooted in System 1. Partial knowledge is what it is – a novice remembers only parts of the required knowledge. Whether the student has forgotten or never learned the content, the only answer is to fill in the gaps. In many instances,

[27] Perkins and Martin may or may not have held a wider view of cognition, but the general reader likely did. Perkins certainly went on to write influential and informative works outside the 'mainstream' views (David N Perkins, 2010; David N Perkins & Salomon, 1992)

the only way to compensate for partial knowledge during a programming activity is to look up the information and hope that System 2 retains it the next time it is needed. The just-in-time acquisition of information is helpful for more than simply rectifying partial knowledge. It may help System 1 to prime System 2 when similar circumstances occur in the future with the newly acquired knowledge. While Perkins and Martin could only distinguish partial and inert knowledge in instances when the student never seems to remember the required information, dual process theory reminds us that System 1 needs trained context to aid System 2 properly. Students may have no memory of important facts, but it may also be difficult to distinguish when knowledge is partial versus inert.

Inert and misplaced knowledge likely represent that beginning knowledge forming in System 1. Perkins and Martin noted the "sparse network of associations" that might explain inert knowledge. For example, a novice might remember the `if` statement, but not immediately recall the details of the `else` branch. Dual process theory suggests System 1 was able to identify the need for an `if` statement but probably has not seen that specific problem used with an `else` branch. The novice might search their mind knowing the concept of `else` is on the tip of their tongue. They might look at other examples or read the book looking for a trigger to invoke the memories of the `else` option. If they find the correct prompt, they will have the "Aha!" moment that signals remembering. A better student might return to their book in depth to relearn about decisions, yet at the cost of attention and focus required to keep System 2 in the moment for solving the specific problem. Thus, many students may remain in limbo hoping to stumble across the answer, particularly if they are focused on completing the task rather than filling in the missing knowledge.

Misplaced knowledge represents the opposite deficiency within System 1. Some aspect of the problem triggers System 1 to prime System 2 with the wrong information, leading to a framing error. Most of Perkins and Martin's exercises focused on printing out a specific pattern using asterisks (*). They described one student, Stan, who remembered that a formatting command was used to convert data into printable formats and was determined to use it to print a column of stars, even though that approach is difficult or even impossible. A conceptual review of the format command says that it is used to turn the variable `amount_due` holding a value of `10.0` into the formatted string `"Amount Due: $10.00"`. If Stan was using System 2 to apply his conceptual knowledge, he might not have made a mistake. More likely, Stan's undertrained System 1

associated every printing problem with this command and primed his System 2 with this knowledge. Stan would remain unable to solve the problem until his System 2 could break through the framing error that the formatting command was needed.

If partial, inert, and misplaced knowledge are issues of System 1, correcting these types of fragile knowledge requires repeated and varied exposure. Programming lecture and examples tend to focus on code snippets, but System 1 need to connect not just code with behavior but constructs with the needs they satisfy. The protocol of the study encouraged researchers to wait until the student was stuck, far past the point where the initial prompt may have misdirected the student. At that point, their strategies for correcting fragile knowledge are limited. Perkins and Martin pulled from literature to suggest strategies for overcoming inert knowledge in the moment, "[c]onventional tactics of fluency such as, brainstorming ideas seem to offer little help; however the strategy of listing words that might be used in an essay considerably increases students' retrieval of relevant information" (D. Perkins & Martin, 1985, p. 13). Listing words is a superior strategy to open brainstorming as it has a better shot of triggering System 1, where brainstorming likely would continue in the 'wrong' framing under which the problem started. Brainstorming is more productive in experts who have a broader pool of experience from which to draw. Novices, under traditional pedagogical approaches, spend significantly more time looking at decontextualized simple code examples, derived to isolate individual elements within language construct. Presenting many small examples may support the automaticity required to promote System 1, but neglects to connect code constructs with the open-ended problem statements and sometimes even useful algorithms needed to solve problems.

Conglomerated knowledge seems to be the catch-all for any other cases, and as such, offers little explanation. By its nature, conglomerated knowledge makes little sense to experts. Experts are generally baffled when novices are unable to piece together perfectly logical rules into a simple working program.

> The remaining question asks why programmers take such stabs rather than doing the "right thing?" Several answers seem relevant. First of all, the "right thing" often involves knowledge inert or not possessed at all, leaving the programmer no proper recourse. Second, the programmer often works from an underdifferentiated [sic] knowledge base, leading to misplacements that yield conglomerates. (D. Perkins & Martin, 1985, p. 21)

Perkins and Martin listed possible reasons the programmers struggle to pull together a complete program. It starts with fragile knowledge. If a novice begins with a shaky plan to solve the problem, they are more likely to start throwing whatever pops to mind into their code. Their lack of expertise in syntax makes the code even more incomplete and disjointed. The weak System 1 forces System 2 to conglomerate knowledge in a valiant attempt to make do with what knowledge is available. System 2 must fill in the gaps for what is not offered by System 1. The effort to do this, whether searching memory or relearning from resources, takes attention away from the problem-solving. System 2 must therefore frequently context shift, making it more likely the results are disjointed as information is lost and reloaded into short-term memories.

Premise 1: Knowledge becomes conglomerated when various pieces of information are associated improperly and/or incompletely
Premise 2: System 2 is responsible for dealing with novel situations and forming mental models that allow for hypothetical thinking
Premise 3: System 1 supports System 2 by automating skills and providing fast recall
Premise 4: Inert knowledge adds to the burden of System 2 to either force the novice to search for missing knowledge, or to temporarily 'relearn' within System 2.
Premise 5: Misplaced knowledge can disrupt System 2 with incorrect primes
***Conclusion*:** Conglomerate knowledge is exacerbated by inert and misplaced knowledge, requiring System 2 to compensate and risk become overloaded

When a novice programmer tackles a complex problem without the aid of a well-trained System 1, they are more likely to suffer from conglomerated knowledge. System 1 supports System 2 by activating relevant knowledge, which helps programmers, for example, to complete the required syntax of a language construct without adding any burden to System 2.

> Conglomerated knowledge signifies situations where a student produces code that jams together several disparate elements in a syntactically or semantically anomalous way in an attempt to provide the computer with the information it needs. (D. Perkins & Martin, 1985, p. 7)

Perkins and Martin provide an example of conglomerated knowledge where a student attempted the syntax `PRINT "*" * X` to print out X asterisks in a row. The student conjoined the concept and syntax from multiplication with the syntax for printing in a clever, but still illegal syntax. Dual process theory explains that the student may have followed a misplaced System 1 prompt that

System 2 failed to reconcile with its knowledge of programming syntax. Conglomerated knowledge may seem illogical when you know the rules, but through the lens of dual process theory, such creations seem less erratic.

Perkins and Martin noted that conglomerated knowledge not only impacts writing code but other areas of programming as well. Without support from System 1, a novice programmer must simultaneously manage their plan for solving the problem, the appropriate language constructs to realize that plan, and the syntax for those constructs. When the problem exceeds the fluid intelligence of a programmer's System 2, they may end up demonstrating conglomerated knowledge. Continuing from Perkins and Martin (1985) earlier quote,

> Third, the programmer fails to close track tentative conglomerates or may be unable to do so with precision. Fourth, the programmer lacks the general critical sense that one simply cannot expect to throw things together in a programming language and have them work. (p. 21)

They noted that their students struggled to *close track* (i.e., trace) code, and did so infrequently. Students initiated tracing without prompting only 20% of the time, and only 50% of those attempts were successful. Furthermore, Perkins and Martin implied that their students were making irrational plans that had little logical chance of succeeding. Not only were students 'inventing' new syntax but optimistically believing their various efforts would result in success. Conglomerated knowledge seems to describe failures within System 2, yet many of these issues may start in System 1.

The aid offered to novices by the researchers might have worsened some cases of conglomerated knowledge. When offering a *prompt* or *hint*, the researcher triggered System 1 to prime the required information. The good news is that the student recalled the knowledge they needed to proceed. The bad news is that they had not yet associated the current problem with that knowledge, so System 2 must now reconcile its original plan with the new prompt. The guidance from the researcher may keep the novice moving, but their decoupled representation may not include the new information. If a researcher, for example, *prompted* a student to use a loop, the resulting priming may be enough to complete the syntax of a loop successfully but does not assure the student understands the purpose of the loop. If the resulting code does not work, the student is now debugging the researcher's plan, not their own. A student may be less likely to consider tracing when working with the researcher's plan instead of their own.

The seemingly haphazard planning that Perkins and Martin observed also may come from System 1. Experienced programmers have spent years working at the algorithmic level (or higher), where most of a new programmer's experience may be at the construct level. Textbooks are full of examples that isolate individual programming constructs, and only later will students see robust examples of loops, decisions, and other constructs working to solve complex problems. When a novice programmer "throws things together," they are likely following multiple System 1 prompts, yet none of these are sufficient for solving the entire problem. Without a concerted effort to reconcile the various prompts, System 2 may never create a cohesive mental model of the design approach. The result, as Perkins and Martin captured, is a jumble of disconnected, seemingly nonsensical pieces, possibly made worse by the advice given by the researchers. Fragile knowledge may not merely be a problem of recall, but also in the contextualization of the knowledge offered by experience.

Reconciling fragile knowledge

Perkins and Martin's definition captures fragile knowledge's symptoms but does not seem to explain its nature. They provided excellent examples for each type of fragile knowledge as case studies yet did not explain the origins of fragile knowledge. They offered three suggestions to instructors.

1. Teach things such that knowledge is not fragile
2. Encourage exploration in learning to program
3. Promote basic problem-solving approaches

As the last section demonstrated, dual process theory can explain the nature of fragile knowledge, and thus can refine Perkins and Martins' advice to teachers.

After crafting an exemplary methodology and offering insightful analysis, the first suggestion sadly amounts to *teach clearly*. Perkins and Martin (1985) suggested that teachers should "convey an understanding of exactly what commands do" (p. 30). Their advice echoes the Cartesian model of cognition, where the rational mind will think logically so long as the thinker is disciplined and remembers all the required information. The epistemological shift of dual process theory reminds us that logical thinking is not just about having or lacking information but also about having the right *kind* of information contextualized appropriately. An accurate understanding of a language construct helps only if that knowledge is activated and if similar

knowledge (e.g., multiplication of asterisks) does not interfere. Instructors cannot simply do what they are already doing better. Dual process theory suggests that programmers need variety and possibly new pedagogical techniques to promote the integration of knowledge required within both Systems 1 and 2.

Perkins and Martin's second suggestion, having students experiment with programming, is not as easy as suggested. Papert (1978) also believed that experimentation could help novices learn to program but found that students could not do this unaided. When teaching students to program in Logo, he noted that "Dan tells them to 'experiment' but it seems to me that they don't know what that means" (p. 70). He contrasted systematic experimentation with "messing about" – an unstructured attempt to derive meaning from actions. Papert suggested that children need the experience of "messing about" before they can begin to experiment, invoking Piaget's work (see Section 6.1) to explain why. Experimentation requires conscious goals and measured responses that may be difficult without, as Du Boulay (1986) referred to, an "orientation" for the types of problems a programming language can solve. Like Berry and Broadbent's (1988) work with the sugar factory, some tasks need an intuitive response before reasoning can help performance. Without developing some affinity for programming within System 1, it may be extremely difficult to conceive of experiments that expand a new programmer's understanding about programming.

One study (T. A. Lowe, 2018) sought to teach children basic programming concepts by starting with rudimentary instruction and open play using a robot mouse (see Figure 5.1). The researcher demonstrated the mouse's basic abilities to move forward or back and swiveling left or right, as well as how to store and clear a sequence of such commands. After this quick explanation, the children directed their mouse across a grid to a plastic block of cheese in any way they saw fit. The children commanded the mouse like a remote-control toy, rather than an autonomous device. If their programmed path did not reach the cheese, they simply cleared the commands and added new ones until they reached their goal. Without intervention, they did not develop the concept of planning a program to run from the same spot independently and repeatedly but seemingly viewed the mouse as an extension of their will.

Figure 5.1. A robot mouse

The open play by these children amounted to "messing about" and helped in some aspects but not in others. Open play seemed to increase their intuitive sense of the motions associated with commands but did not prepare them for the spatial reasoning[28] required to navigate a mouse programmatically. A common challenge in programming such a device is a shifting frame of reference as the mouse turns. From the perspective of the programmer, what might be a left turn becomes a right turn after the mouse completes its initial moves. Since during open play, the children created new paths relative to the mouse's final position, their System 2 did not need to reconcile the turning of the robot. They came to understand the relationship between the buttons and the movements but did not learn anything about how to construct a program. Worse, their open play developed a few habits that were, at least for a while, counterproductive.

The basic intuition for interacting with most devices is fundamentally different from the task of writing a program. The children's intuitive open play demonstrated that they were actively considering each step rather than creating a decoupled representation (System 2 plan) for what their mouse should be doing. The open play did not require the children to debug their previous plan, so they instead cleared the existing program and made a new one after each motion. Some children only entered a few steps at a time until they reached their goal. None of them arrived at the notion that they could, or should, plan and program complete paths with no intervention. Even after the researcher introduced this concept, a few of the students still wanted to move their robot incrementally. The children only arrived at the nature of programming after receiving clear feedback and direction from the researcher. After this point perhaps, allowing the children to experiment may have yielded better results.

Perkins and Martin's final recommendation is to promote problem-solving because novices do not know how, or they forget to use common approaches like tracing. They suggested that students may benefit from being more metacognitive.

> As our data demonstrate, students would gain by prompting themselves more often with simple strategic questions such as "what does the program need to do next," "what command do I know that might help to do that," "what will what I have written really do," or "how did my program get that wrong answers?" (p. 31)

[28] Section 6.3.2.2 talks more about how such shifts in spatial reasoning requires the support of mental models to reconcile.

Considering such questions requires the attention of a novice's System 2, which may be consumed with the details of the syntax, error messages, or other new concepts. They may not have the bandwidth to consider such matters until a maturing System 1 unburdens System 2 of mundane tasks like producing proper syntax. Without writing code, how will they gain the experience that System 1 requires to not only become familiar with coding constructs but create rich associations with other concepts? Perkins and Martin long ago captured the need, and TAMP hopes to capture strategies for creating such experiences for new programmers.

5.2.1.3 Reinterpretation Summary

The advice given by Perkins and Martin may not be perfect, but their observations provided a solid foundation for understanding this type of novice struggle. The construct of fragile knowledge influenced how many computing educators and researchers view and assess novice programmers. Revisiting their study through the lens of dual process theory reinforces their findings yet adds a new dimension of why fragile knowledge exists. Instances of inert and misplaced knowledge occur when System 1 is not yet fully mature for that content. For example, a student who has used loops to print items in a list, may not think of using loops as a mechanism for the validation of user input until they see an example. Providing novices with more practice that emphasizes a variety of examples of language constructs in action may help reduce the likelihood of knowledge being misplaced or being left inert. Conglomerated knowledge arises when gaps in System 1 confound an overburdened System 2. System 2's tendency to be lazy means that it is happy to follow promising, but maybe incomplete prompts from System 1, resulting in disjointed solutions that may not make sense to experts. A combination of reflection and applying metacognitive strategies may help reduce the impact of conglomerated knowledge, but fully understanding the nature of this complex interaction between the two Systems will continue to emerge over the next few chapters.

Most importantly, dual process theory offers educators a new perspective on fragile knowledge. Rather than being a handicap, fragile knowledge is an expected state of learning as novices transition from tackling simple problems using mostly System 2, to tackling complex problems that require support from a fledgling System 1. Making this transition through coding exercises alone proved challenging for students, even with the aid of a dedicated researcher. In

the decades that have passed, computing educators have proposed numerous intermediate pedagogies to promote basic skills that may encourage the greater automaticity and intuition of System 1. In addition to writing code, students can gain experience with a programming language through Parson's problems, tracing exercises, worked examples, or other such pedagogies, as discussed in Section 2.3.

Perkins and Martin's study showed that knowledge is useful only if it is triggered when needed. They introduced a valuable methodology (i.e., the use of *prompts*, *hints*, and *provides*) that looked beyond 'forgetfulness' and determined what knowledge students are acquiring, even if it is fragile. Fragile knowledge, when considered as a byproduct of System 1, hints at gaps in traditional models of learning. How can a novice programmer gain the necessary experience for building new solutions, when experience is the only way to acquire that knowledge? Answering *that* question requires a better grasp of the interplay between experience (System 1) and traditional ways of knowing (System 2).

Counterpoint!
Dual process theory seems to provide a better explanation of the nature of fragile knowledge, but some could claim the study by Perkins and Martin holds two specific nuisance elements that could explain fragile knowledge: • The difficult task of coding from scratch, perhaps before coders are 'ready' • The help provided by researchers could have extended students beyond what they have learned thus far, thus creating the overload by introducing students to unprepared content and challenges As noted, conglomerated knowledge could be the result of cognitive overload, which could explain such struggles without the existence of System 1. Cognitive load theory might interpret conglomerated knowledge as an inability to manage short-term memory, and thus, the learner is struggling to keep up, much less learn new ideas. Perhaps the students needed simpler tasks with more support? The case studies by Perkins and Martin do not provide insight into the expected progress the student should make. Remember, students could start with any of 8 tasks and progress to more difficult ones, but how far should the students have been expected to go? Perkins and Martin noted the challenges students had in tracing, yet the struggles in tracing could be related to an inability to fix bugs introduced after the help offered by the researchers? Since the original data are unavailable, it is unknowable. Fragile knowledge, and thus the insights gained through dual process theory could be merely observations of how unprepared students flounder due to inadequate instruction. Thankfully, Lister et al. (2004) completed a study on the tracing abilities that the next section will revisit under dual process theory.

5.2.2 Tracing - Lister et al. (2004)

In 2001 an international group of computing education researchers created a tool to assess programming abilities. McCracken et al. (2001) asked students to implement a simple calculator (i.e., addition, subtraction, multiplication, and division). As in the study by Perkins and Martin (1985), McCracken et al. expected students should be able to analyze, design, code, test, and debug an application. Unlike Perkins and Martin, the students taking this test were completing their initial collegiate experience in programming (i.e., CS1), thus in theory better prepared and 'better' students in general. Chapter 8 will return to the details of McCracken et al., but for this section, the most notable finding from McCracken et al. was that "students did much more poorly than we expected" (p. 132). The fact that most students struggled, and some never even got to the point of writing compilable code, or any at all, led to the question: what, if anything, were students learning?

Lister et al.'s (2004) removed the need for students to create working programs and merely trace code. Where McCracken et al. exposed a significant gap in the abilities of new programmers, Lister et al. believed that students were still learning, and a different type of assessment could demonstrate it. Their test not only demonstrated the types of knowledge and skills that new programmers acquired but offered insights into their thinking process by tracking their work. The findings suggested that students are much more capable than reported by McCracken et al., but they still suffered from fragile knowledge. Lister et al.'s work provides another example of the role that System 1 plays in basic programming abilities and how dual process theory can help to explain confounding findings in computing education.

Premise 1: The lack of driving theory, much less theory of cognition left many unexplored avenues and gaps that dual process theory could have better informed the protocol and analysis
Premise 2: Some tracing is helpful, but too much shows that System 1 is underdeveloped
Premise 3: Many students did not trace on Question 8 at all, but instead used System 1
Premise 4: Tracing presents a more manageable task for novices than writing code
***Conclusion*:** Dual process theory provide a better explanation of the tracing behavior of students in Lister et al.'s study, which closes the gap for the complexity of the programming task as a factor in fragile knowledge

5.2.2.1 Summary of the original study

Lister et al. (2004) wanted to test the conceptual understanding of students by using multiple-choice questions rather than having them write code. Multiple-choice questions remove the need of the test-taker to write syntax, allowing them to focus on concepts. Rather than asking students questions about language constructs, the researchers presented sample code to trace or fill in missing lines. Lister et al. created an intermediary assessment that required students to apply knowledge about programming languages, yet only through tracing, not creating new code. Their goal was to test whether students were indeed learning from their CS 1 experience, even if unable to create new applications 'from scratch'.

The multiple-choice questions confirmed that students were learning basic ideas about programming, yet still struggled in many ways, including instances of fragile knowledge.

> One observation we make about the middle 50% of students is that, by virtue of the fact that they answered some questions correctly, they have demonstrated a conceptual grasp of loops and arrays. Therefore, the weakness of these students

> is not that they do not understand the language constructs. Their weakness is the inability to reliably work their way through the long chain of reasoning required to hand execute code, and/or an inability to reason reliably at a more abstract level to select the missing line of code. Based on the performance data, it is not possible to draw any firm conclusions as to the exact cause of this weakness (p. 128)

Lister et al. report both quantitative details about each question and discuss qualitative examples for some of the questions. Some students did quite well on the exam, though roughly half of the students answered 7 or fewer of the 12 questions correctly. The individual questions varied in difficulty; 73% of the students answer the easiest and 35% the most difficult questions correctly. As Lister et al. mentioned, the results seemed to indicate that most students had gained a conceptual understanding of the non-trivial concepts of loops and arrays, but the quantitative data is insufficient to describe the nature of the weakness when tackling the more complex questions.

The most compelling data providing evidence to the maturity of the student's programming skills was their *doodles* (i.e., notes taken by students while tracing). Doodles (also referred to as sketches or notes in the literature) are not typically a formal part of the answer (though some instructors may require them). Lister et al.'s study used doodles as an additional piece of data for a subset of the submitted problems in addition to the multiple-choice answer. Figure 5.2 presents a sample doodle taken from Lister et al. and further annotated within Lowe (2019). Lister et al. tagged each submission's doodles using qualitative coding categories, the ones of interest here shown in Table 5.1. These categories included submissions that did not include any doodles (*Blank Page*) as well as those with relatively unique or unidentifiable markings (*Extraneous Marks* and *Odd Traces*). Lister et al. identified the most 'formal' doodling approach as *Synchronized Trace* with the remainder being relatively ad hoc – tracing aids for basic concepts (*Position*) or tracking the changing values of variables and state (*Number*, *Keeping Tally, Trace*).

Lister et al. analyzed the available doodles for insight into student's thinking, using a qualitative coding system to categorize the doodles into different tracing strategies and which strategies seemed to indicate success. Overall, students who did not doodle answered 50% of their questions correctly, that Lister et al. suggested offered "very useful statistics for teachers to quote to their students" (p. 129). Lister et al. implied that doodling equated to better performance, based on this correlation, suggesting that students should adopt some form of doodle to become better tracers. Overall, students chose not to doodle on 39% of the submissions, and paradoxically

included fewer doodles on more difficult questions. Lister et al. mentioned that only 20% of the students failed to doodle on easier Question 2 than the 55% who left harder Question 8 blank except or their answer. They offered three potential reasons why students did not doodle.

> 1) The MCQ [multiple choice question] is relatively simple.
> 2) The student has internalized a sophisticated reasoning strategy for answering that type of MCQ.
> 3) The student is either guessing, or has heuristics for selecting a plausible answer without genuinely understanding the MCQ, which is essentially an educated guess. (p. 132)

Each of these options seems grounded in Cartesian views of cognition. They each imply a process or strategy for coming to an answer, including gaming the nature of multiple-choice questions that offer a 25% chance of discovering the correct answer.

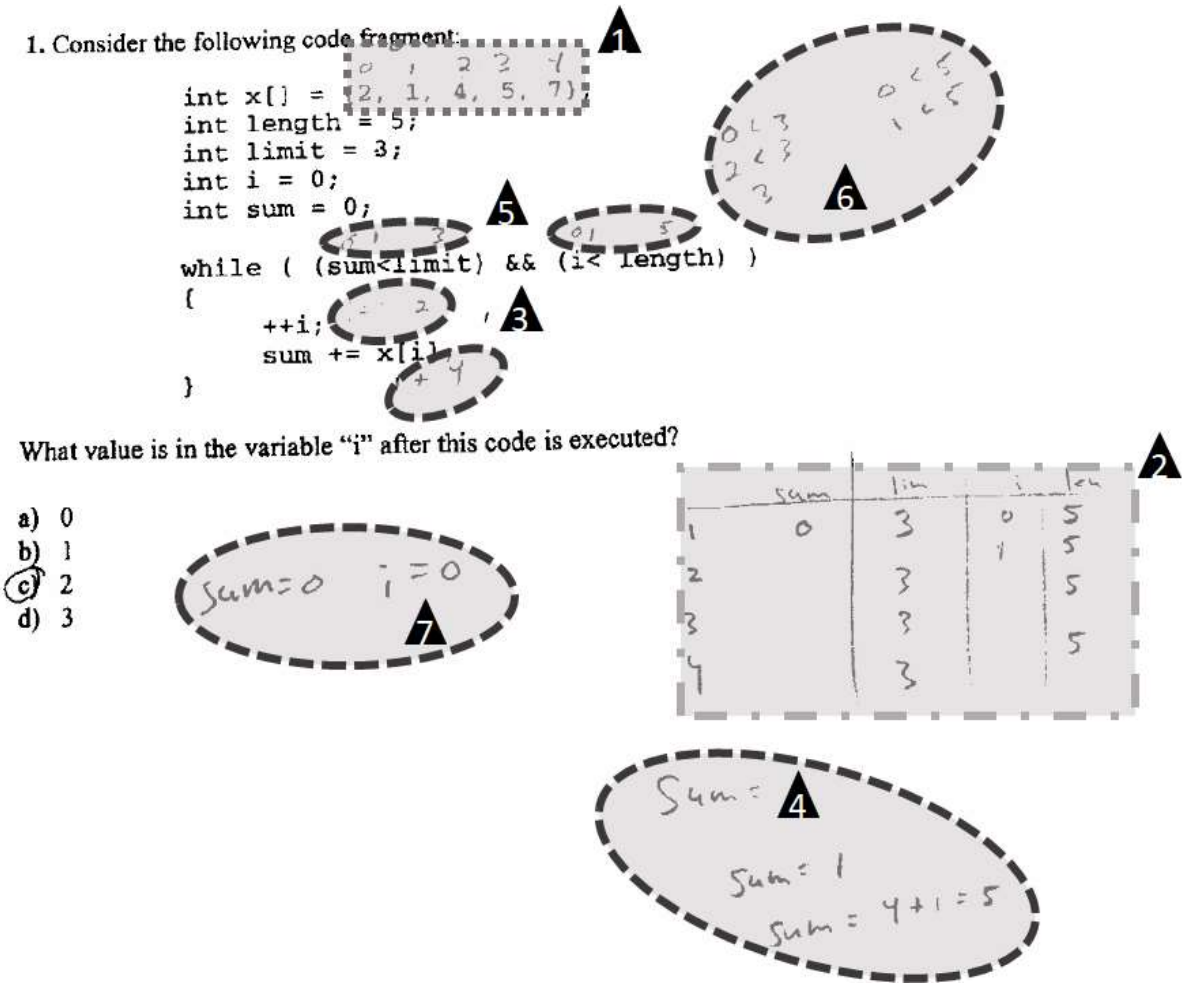

Figure 5.2. An annotated example of student work from (T. Lowe, 2019)

Lister et al. captured a robust data set and provided a cogent analysis of the students based on statistics, examples, and literature, yet were unable to peer through inconsistent behaviors by the test-takers. Why is it that students doodled less often on harder questions? How do they manage to answer twice as often as chance without taking notes? Lister et al. investigated whether the struggles McCracken et al. found indeed had to do with problem-solving or, as they suspected, fragile knowledge. Lister et al.'s study provides a rich dataset for exploring how dual process theory can answer questions, such as "the exact cause of this weakness" (p. 128) posed by the authors.

5.2.2.2 Applying dual process theory to the study[29]

The analysis by Lister et al. tells seemingly compelling stories about novices and tracing but given the lack of an explicit theory of cognition, their best advice seems to be to *tell students that when they trace, they might do better*. Applying dual process theory to Lister et al. helps to draw out three specific arguments that help better explain student behaviors.

Revisiting the quantitative analysis

Premise 1: The frequency of the coded occurrences of tracing can tell a story about the novice's cognition
Premise 2: Blank doodles do not seem to directly correlate with failure, thus may not support the notion that doodling is an essential part of success
Premise 3: Retroactively applying CLT to doodling values of variables invokes gaps in understanding, but without describing any reasons
Premise 4: The ultimate haphazardness of the doodles was worth investigating
***Conclusion*:** The lack of driving theory, much less theory of cognition left many unexplored avenues and gaps that dual process theory could have better informed the protocol and analysis

In many ways, Lister et al.'s study extends the concept of fragile knowledge into tracing as well as writing code. The authors cited Perkins et al., yet seemed to consider fragility as a fault in learning, rather than a useful construct for informing their study. They stated that correctly

[29] The following both summarizes and extends upon an existing analysis (T. Lowe, 2019). The two narratives are intended to be complimentary, though this work is the newer and may superseded any prior analysis. It is not required to read the other work to understand this section but doing so may provide additional details of the study and rational for the analysis.

answering some of the questions proved that students must have formed a conceptual understanding of loops and arrays. Had they planned for fragile knowledge, Lister et al. may have tracked individuals across their submissions to see, when, and where the students successfully applied conceptual understanding. What Lister et al. captured and reported was how frequently codes appeared in submissions and the corresponding percentage of students who answered correctly. Table 5.1 restructures their original data to emphasize the codes that appeared most frequently.

Table 5.1 Coding summary from traces performed by Lister et al. (2004)

Category	Answered correctly (%)	Total instances	Frequency in submissions (%)
Blank Page	50	259	39
Trace (updated values)	75	215	32
Number	70	189	28
Extraneous Marks	57	89	13
Position	64	75	11
Synchronized Trace	77	73	11
Odd Traces	78	23	3
Keeping Tally	100	6	1

Sorting the table by frequency rather than the percentage correct helps to put each strategy for tracing (identified by the coding category) in context. For example, *Keeping Tally* appeared important since 100% of the submissions included the correct answer, but occurred only *six* times in 672 submissions (perhaps for only one student!). The *Keeping Tally* strategy merely entails using tally marks to track how many times a loop has run, so it hardly seems a comprehensive strategy that might yield regular success in students. *Keeping Tally* might, at best, identify naturally disciplined students yet teaching others to keep tally may not instill discipline in other aspects of tracing. Likewise, students who included *Odd Traces* generally answered correctly but it seems of little value to teach a strategy coded with the name *Odd*.

By far, the most frequent coding category was *Blank page* occurring in nearly 2 out of 5 submissions (remember, to be *Blank* means no other categories could be present). While a fifty-percent success rate seems poor for blank submissions, it is still better than the twenty-five-percent rate that would be change guessing. Knowing the characteristics of non-doodlers would provide valuable insights into this behavior. Were there groups of non-doodlers who consistently scored

well and others who seemed to be guessing? Which students doodled on some problems and not others versus those who never doodled? Dual process theory might offer insights to why students could be successful without ever doodling. The black-box nature of System 1 suggests that strong students might trace successfully without the need to take notes. If they had automated the mental execution of loops and other required language constructs, they may have arrived at answers from System 1 without access to the intermediate computations that would be annotated in doodles. Dual process theory would suggest looking for different types of 'non-doodlers': a group that is highly successful and a group that is essentially guessing.

The categories where students doodled frequently offer additional insights. A researcher could apply multiple codes to each submission; for example, the student in Figure 5.2 seemed to include at least *Numbers*, *Trace*, and *Synchronized Trace*. Overall, the students averaged 1.6 doodle categories per submission, but more accurately, averaged over 2 categories after removing those without any doodles. The numbers indicate that students seemed to use multiple strategies, and did not seem to use the same ones, while tracing. Some researchers have suggested teaching formal notations for tracing (Cunningham et al., 2017; Xie et al., 2018). Cunningham et al. reported that novice programmers (and teaching assistants) tended to use ad hoc methods over the ones demonstrated in class, where Xie et al. made the formal use of tracing a part of their pedagogy and saw better conformance to their preferred style. Lister et al.'s data seems to support this notion. Their coding category *Synchronized trace* described a formal tracing approach like that demonstrated in Figure 5.2 under the triangle marker 2. Trace tables are a common systematic approach to tracing (basically what was taught by Xie et al.), yet after combining the 39% of *Blank Page* and 11% of *Synchronized trace*, half of the submissions used an ad hoc approach. Students seem to prefer doodles of their own devising when they doodle at all.

Dual process theory explains student's aversion to formal tracing. Without System 1 support, students might find that formal tracing adds to their cognitive load rather than reducing it. The trace table may seem intuitive to experienced programmers, but only after extensive practice (hence Cunningham et al.'s observation that even teaching assistants avoid their professor's notation style). Until the formal tracing notation becomes automatic, it is unhelpful for novices; thus, only 11% of the students used *Synchronized Trace*, and as we will see in the next section, some do so incorrectly. The doodles used seem to be manifestations of System 2's processing that is unique to each student but may seem common due to prior experiences in mathematics education.

If you remember, Perkins and Martin (1985) noted that their younger students were also reluctant to trace, possibly exacerbated by less overall experience completing math problems. These younger students may have lacked an example of what to write down and thus felt overwhelmed by the prospect of tracing their code. Formal tracing syntax seems to require the same dedicated pedagogy as any other aspect of programming languages, as further elaborated in Section 6.3.2.

If most students are not using predefined methods to trace, the type of information they are documenting may provide a window into their thinking. Some of the most frequently used traces seem to aid in the management of short-term memory by capturing the values of variables. Sometimes students do this in an orderly manner (43% for *Trace* and *Synchronized Trace*), but more often, they write down nothing (39%) or disjointed *Numbers* (28%). Once again, formal tracing methods do not seem to indicate dramatically better outcomes, as in Table 5.1 shows that any approach that includes documenting values (*Numbers*, *Trace*, *Synchronized Trace*, and *Odd Trace*) show relatively similar success rates in answering questions (70-78%). It seems that any means of offloading the burden of System 2 (i.e., freeing up short-term memory) is as helpful as the more meticulous note-taking strategies.

It would again be helpful to see how the traces are used across the various problems, rather than merely the summary statistics. Highly meticulous doodlers may need to do so because their System 2 requires the most support, where less organized (or non-) doodlers only need to capture the occasional value between bursts of System 1 automation. The statistics on doodling do not seem to tell a consistent or informative tale on their own. As we will see in the next section, the presence or absence of doodles may have less to do with a programmer's maturity than their choice whether they feel it is helpful. Some students may need doodles to manage even simple tasks as they have yet to automate the mental execution of even basic language constructs. Other students may choose to take detailed notes (perhaps out of habit) even if these doodles will not inevitably help them arrive at the correct answer. Dual process theory suggests that researchers might need more than the presence or absence of doodles to describe the mental activity of programmers when tracing.

A Cartesian view of cognition overlooks the seeming inconsistencies in how students doodle. While half of the students who did not doodle answered incorrectly, it was only a bit worse than students characterized as making *Extraneous Marks* that appeared 13% of the time. With only 11% of students using a formalized doodling strategy (*Synchronized trace*), it would

seem that the students did not learn a formal method of tracing, or they chose to ignore it in their doodles. Cunningham et al. (2017) support this view, as they noted that students and even teaching assistants do not follow the strategy for annotating their traces explicitly demonstrated by their instructor. The ad hoc nature of tracing seems to indicate that System 2 is only marginally effective at tracing without support from System 1. The drop in accuracy when the problems become more complicated suggests that System 1 is reducing the demands on System 2 for some students but not others. Furthermore, Cunningham et al. also noted that students who doodled (sketched, in their terms) took significantly more time – nearly double – to complete code fixing, ordering, writing, and even reading tasks. Some of this may be due to students being meticulous in their work, but it also hints at students relying on System 2 in the absence of fast System 1. When analyzed using dual process theory, the quantitative numbers from Lister et al. leave as many questions as answers.

Revisiting the qualitative analysis

Premise 1: The student in Figure 5.2 wrote down 'simple' calculations to aid in the formation of System 2 decoupled representations and manage short-term memory
Premise 2: System 1, being unconscious, would not allow the student to write down intermediate variables, so the trace is not being completed by System 1
Premise 3: Many students were able to answer the question successfully without tracing
***Conclusion*:** Some tracing is helpful, but too much shows that System 1 is underdeveloped

What novices write down in their traces only reflects part of their growing maturity in programming. Figure 5.2 presents an original example of doodles from Lister et al. with additional annotations. The student in this problem used at least five of Lister et al.'s coded categories, including *Position*, *Number*, *Trace*, *Synchronized Trace*, and *Computation*. A few of these categories seem confusing when observed in action, though. Why are students writing down computations such as `4 + 1= 5`, as seen in Figure 5.2 at triangle 4? It seems a college student in a programming class should quickly perform such operations without needing to take notes? My in-depth analysis (T. Lowe, 2019) suggested that this student used the observed mix of doodling categories as a means of supporting System 2 when context shifting between different language constructs. For example, the expression `sum += x[i]` may have been unfamiliar enough that writing down `4 + 1= 5` helped to reassure the student of the `+=` operator's function.

The student also seemed to excessively copy variable values between locations (*triangles* 3-6 in Figure 5.2) to support the math operations, decisions, and loop portions of the algorithm. Considering that System 1 does not need 'scratchpads', nor follows explicit rules that would allow the student to write down intermediate variables, the evidence hints that the student is tracing using their unsupported, or unsure System 2.

The doodles shown in Figure 5.2 demonstrate a student who has learned the concepts of loops and arrays but is still working to automate them. This student's doodles seem to represent a form of conglomerated knowledge (D. Perkins & Martin, 1985) – but since the problem only demands tracing, disorganized knowledge seems sufficient. Their doodles include a wide array of unrelated notes that we can guess, but only guess at what the student was thinking while tracing. My guess at deciphering the process, as detailed further in (T. Lowe, 2019), suggests that the student wrote down each separate doodle in the order dictated by the triangles in Figure 5.2 as they considered each successive language construct. Notably, this student included a *Synchronized Trace* (triangle 2), but the table is not only incomplete but incorrect – the trace does not require all 4 iterations. The trace table is another example of conglomerated knowledge, something they did out of ritual, rather than as a contribution to their tracing process. Given time and practice, dual process theory suggests that many or most of these traces would fade. At a minimum, triangle 1, 4, 5, 6, and 7 seem to reinforce easily automated calculations or offer redundant tracking of variables that occurs at triangle 3. As the student feels confident, they may either embrace the trace table (triangle 2) as the primary mechanism for tracing or stick with an ad hoc process (triangle 3) or for simple problems stop tracing altogether.

Neither Lister et al. nor to my knowledge, any researcher, has asked experts to trace code and compared their results to novices. In my two decades as a working programmer, I rarely traced code, and do not remember many instances of doing so manually with notes. The activity of tracing seems to be bound to the classroom, either as an effort to teach basic concepts or to avoid using the computer yet still test or exercise programming knowledge. I suspect that most experts, given their robust System 1, would take ad hoc notes, like novices, or none at all. Educators, on the other hand, may embrace formal doodling techniques since they often use such examples with students. The point of this analysis is not to deride tracing or doodling, so much as consider the mechanisms involved. Doodling, like showing one's work on long division, seems to help most in the formative stage of learning and offer insights into a student's thinking. Asking a

programmer to trace in a specific manner may add to their System 2 workload, particularly if they must unpack the automated answers coming out of System 1. Educators should consider how a learner's thinking changes as System 1 takes on more of the mundane tasks, and System 2 might forget some of the details it once knew. Another thread of analysis conducted by Lister et al. demonstrates how the interplay between the two Systems might be very different depending on a programmer's expertise.

Intuition as a driver of programming decisions

Premise 1: Lister et al. must presume that tracing is the only way to solve Question 8, so when students do not doodle they must be traced in their head
Premise 2: System 1 intuition offers a simpler explanation, a student may see a pattern
Premise 3: The statistics do not support the assumption that more students mentally traced
***Conclusion*:** Many students did not trace on Question 8 at all, but instead used System 1

Lister et al. included questions that seemed to demand tracing, yet dual process theory suggests that these questions could be solved intuitively. Some questions (like that in Figure 5.2) specifically asked for the execution results; thus, tracing offered the best method to determine the answer reliably. Other questions asked the students to complete a blank space in the presented code using one of the four multiple-choice answers (see Figure 5.3). It is these types of questions that seem to defy the need to trace. It would seem that Lister et al. expected their students to consider each answer in turn and use a combination of analysis of the code and tracing to select the one that is appropriate to produce the results described in the problem statement. My experience (T. Lowe, 2019) in answering the question was entirely different. I read the question and looked through the choices and immediately picked an answer (System 1). I double-checked my answer by considering the bounds of the loop I selected in the context of the code (System 2, mostly). I considered tracing to validate my answer, but only because I was reading a paper on tracing and then only with a cursory triple-check. By that point, I was supremely confident and confirmed that I chose the correct answer, but I could have saved several minutes – especially on a timed test – by going with my intuition (I was pretty sure at that point already). While my experience is definitively anecdotal, the data published by Lister et al. shows signs that many students also followed their intuition, but their novice intuition sometimes led them astray.

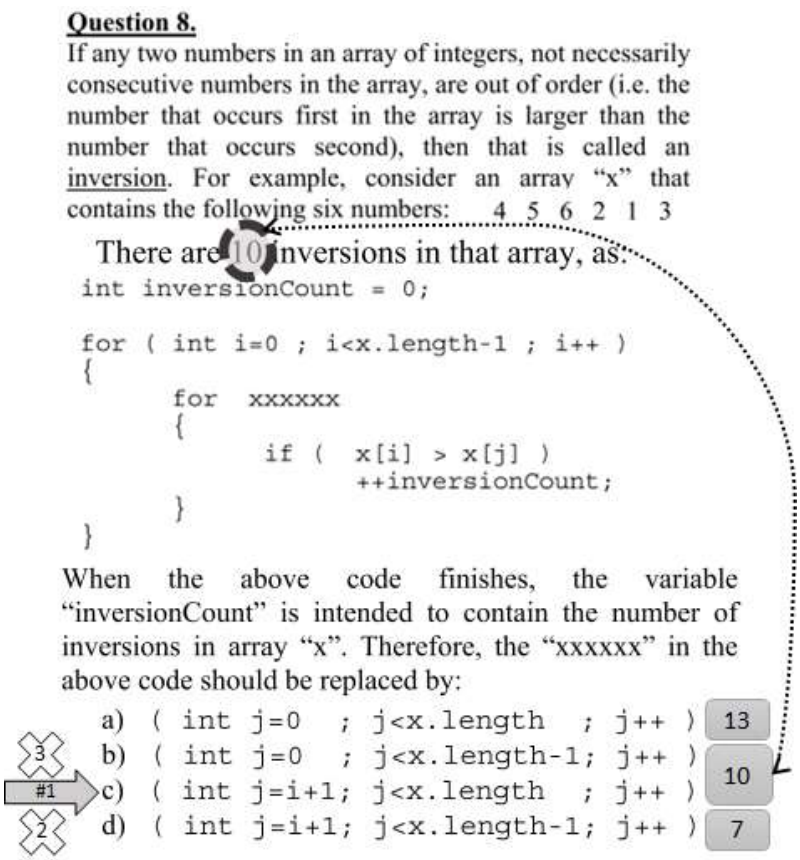

Figure 5.3. Question 8 from Lister et al. (2004), including some results information

In their analysis, Lister et al. compare the results of the easier Question 2 to tougher Question 8, shown in Figure 5.3. They noted that significantly fewer students not only answered Question 8 correctly, but they also doodled less (see Table 5.2). They speculated that on the tougher Question 8, students chose to trace the question *mentally* with 25% fewer students doodling. This analysis seems problematic in several respects, beyond my experience answering the question intuitively. On Question 2, the students would have had to conduct a single trace, and 80% chose to doodle. On Question 8, they may have had to conduct up to 4 separate traces, yet did not feel the need to document any of these traces? Lister et al. described Question 8 as more difficult because fewer students answered it correctly, yet they chose not to use a strategy that had helped them succeed on earlier questions. Dual process theory suggests their choice was neither irrational nor likely conscious.

Table 5.2. Analysis of Questions 2 and 8

	% Correct	% Used Doodles	By Quartile (%)			
			1	2	3	4
Question 2	65	80	87	76	56	28
Question 8	51	55	81	47	30	20
Difference	-14	-25	-6	-29	-26	-8

Fill-in-the-blank problems would seemingly add to the burden of System 2. To cover every possibility, students might need to conduct up to four potential traces and do so for an algorithm that has a big blank spot in its middle! The student would need to juggle a decoupled representation of the code with each substitution from the possible answers in turn. *And while doing so, choose not to take notes on up to four traces*. The greatest paradox is that tracing, however mundane or arduous, is a sure way to answer this problem. Question 8 provides the answer (circled in Figure 5.3) and only answers b. and c. arrive at that answer. A student could discard any trace that did not arrive at an answer of 10 as a possible answer, thus eliminating a. or d. (the most popular incorrect answer). Tracing would have also exposed the incorrect answer b., though this would have required that they double-checked their intermediate steps, not just the final answer, something easier for students who doodled.

Like my experience, it seems that many students followed their intuition to answer Question 8. System 1 adopts patterns that it sees regularly, and mere familiarity provides positive feelings (Zajonc & Rajecki, 1969) that likely misled some students. Lister et al. commented that they were very careful in creating what they called "distractor" answers. The distractor answers to Question 8 had a few 'familiar' items that could have tricked novices. Java loops typically start with `0` and go to `length`, but the last element of an array is always `length - 1`. Answers a. and b. seem to be very convincing distractors for this reason. Answers c. and d. are more typical of the bounds of inner loops (i.e., starting with the index of the outer loop). The choice between these two comes down to how well the test-taker understood the problem statement and its typical implementations (at least until they traced to check their answer). My System 1 had encountered such problems many times and thus leapt to the correct answer. Students who answered d. may have suffered from conglomerated knowledge (D. Perkins & Martin, 1985), conjoining several of the distractors and accurate ideas above. System 1 seems to offer a better explanation than the possibility that students abandoned doodling as they tackled a more difficult problem.

Dual process theory explains how intuition could have provided an answer, and one with a reasonable degree of confidence that would not have felt like a total guess. The use of intuition is not entirely without support, as the results from Table 5.2 offer quantitative support for this analysis. Compared with Question 2, 14% few students answered Question 8 correctly but the drop disproportionately impacted students in the middle two quartiles. Despite that 25 % fewer student doodled, the drop in scores was only 14%. Many students did not doodle and still arrived at the correct answer. While it would be great to know, particularly by quartile, who doodled or not and their result, it seems that students either had another strategy outside tracing or the doodles are not essential for tracing across all students. Given the number of students taking this test, different groups probably arrived at correct and incorrect answers from either System 1 or 2. Some students probably used detailed tracing to discover the correct answer, while others made mistakes while tracing. Some students intuitively chose the wrong answer, while others guessed the correct one. What dual process theory offers is an explanation of why students decided to abandon a strategy that seemed to work on prior questions yet answered the more difficult question correctly.

While a Cartesian view of cognition does little to explain how students may have tackled these different types of problems, dual process theory suggests that intuition helps programmers. The top quartile continued to thrive, likely because they held a combination of System 1 intuition and a robust System 2 to support tracing activities. The middle quartile likely lacked both System 1's automaticity and what priming they had might have been misleading as much as helpful. The familiarity with the distractor answers could have seeded a false sense of confidence that, combined with time constraints, led them to feel confident in their answer without conducting a thorough chase with doodles to check that answer. System 1 offered them a quick answer, but the quality of that answer was at the mercy of their experience in working with nested loops. Lister et al.'s data demonstrated that students are learning many useful concepts about programming, but like Perkins and Martin (1985) noted, it is occasionally fragile. The concepts of dual process theory seem to provide better explanations for the performance of novice programmers than those originally offered.

5.2.2.3 Reinterpretation Summary

Lister et al.'s (2004) study provided additional insights on fragile knowledge as well as hinting at how intuition may support different programming tasks. Beyond Lister et al., several studies point to the benefits of taking notes while tracing (i.e., doodling or sketching) (Cunningham et al., 2017; Xie et al., 2018). Dual process theory suggests that the mere presence of accurate doodles is not a sign of expertise, but educators could use doodles to track a developing programmer's maturity. The newest students may not doodle as they are not sure what to write down. As they see examples and practice, they may take more notes, perhaps in an ad hoc manner, before adopting more rigorous methods to document their trace. Eventually, doodling tails off, particularly for simple problems. Over time a chart of doodling might appear as a 'normal curve', with the most notes taken when System 2 is driving the tracing tasks but tail off as these abilities grow in System 1.

Dual process theory informs how to conduct future studies on tracing. Researchers conducting a future tracing study might define their protocol to explore the relationship between tracing and doodling. By 'forcing' all students to document their tracing on some problems, the study could evaluate the pattern of tracing across different skill levels. Then by 'banning' doodles/sketching on other questions, they could explore the change in the quality of answers. Researchers could further capture metacognitive reflections of why participants sketch (or not) when given a choice. Novices may use doodles to capture their progress and manage short-term memory as existing literature suggests but might also do so out of habit or as a self-soothing ritual. Some may claim the problem is 'too easy' to require notes, not know how to take notes, or merely forget to do so. In an academic setting, doodling may seem a logical activity for ensuring correct answers, but dual process theory suggests people develop habits, or not, for many reasons.

The last, accidental, and perhaps most important finding of revisiting Lister et al.'s study is how programmers may decide to construct their algorithms. Across Chapter 2, we saw evidence that experts use intuition in ways novices cannot, particularly around design. The analysis of Question 8 suggests that novices are starting to use the same mechanisms, but lack of training makes their intuition unreliable. Question 8 was not merely about intuition, though, as System 2 had every opportunity to step in and validate any intuitive decision making. Dual process theory does not devolve thinking into either intuitive or rational; each System works in parallel to

optimize the cognitive process. The challenge in tackling complex activities like programming lies in maturing both Systems and ensuring they work together collaboratively.

5.3 Next Steps for Dual Process Theory

Counterpoint!
Dual process theory provides insights on developing the basic skills a programmer needs, but these are a small portion of what a programmer does. Fast and automated mental execution of code does not necessarily mean a programmer will become a strong designer. The core of programming is problem-solving. Whether designing a solution or finding bugs, a programmer must learn quickly and manipulate abstract ideas which automation supports but cannot replace. Students must still learn concepts, and blend these ideas with automation, and dual process theory offers little insight on how to teach or how people learn.

Dual process theory goes a long way to describe the mind of a programmer, but most of the focus is on the demarcation between the Systems, with less attention given to the interaction between the two Systems. Stanovich discussed the value of decoupled representations as a working ground for mentally manipulating the world but offered little guidance on how to promote the development of such knowledge. Fortunately, learning theorists focus on exactly such manipulations. The next chapter explores how traditional learning theories complement dual process theory before focusing on the work of Jerome Bruner as a guide for describing and forming decoupled representations.

6. THEORIES OF DEVELOPMENT AND LEARNING

Dual process theory should prompt educators to consider an epistemological shift of what it means to know a subject upon completing a course of study. Dual process theory confirms the importance observed in the undercurrent within computing education literature (Section 2.3.3), but knowing the importance of intuition does not immediately provide insights on how to promote it. Dual process theory is neither a theory of learning nor development, so it offers few insights to education. Education research has a rich history of theory, with many theorists who speak of the role intuition in children but less so in the complex thinking of adults. This chapter revisits the work of three of these theorists. Computing education researchers sometimes invoke Jean Piaget and Lev Vygotsky's theories and offer the foundations of many programming pedagogies. The educational theorist that offers the most to the types of learning helpful in computing is Jerome Bruner. Bruner's model of mental representations, in particular, helps to integrate dual process theory with educational practice and help to define useful constructs for the construction of TAMP.

6.1 Jean Piaget as an influencer in computing education

Jean Piaget is a renowned theorist whom computing education researchers have often looked to for direction into how to teach programming. Piaget, a French psychologist, developed a model of human learning and development that grew from observations of his infant children to a model of cognitive development that exposed several quirks in reasoning as people mature. While later research challenges aspects of Piaget's model, its influence on modern education, and computing education specifically, are such that understanding his basic ideas offers insights to implicit beliefs that influence many classroom practices.

6.1.1 Piaget's model of human development

Piaget's primary influence on education comes from his model of cognitive development that defines stages of mental competence and ability. Piaget tied his four stages – sensorimotor, preoperational, concrete operational, and formal operational – to physical maturation that comes with age, though later neo-Piagetian theorists suggested that factors other than age may drive development (Morra et al., 2008). Understanding Piaget's developmental stages as he saw them

in child development provides insights into his model that mirrors the cognitive model proposed by dual process theory.

The sensorimotor (originally sensory-motor) stage models a preverbal child's intelligence and learning based on their actions. Piaget's model of development and learning seems to start with the formation of implicit knowledge and skills that mirror System 1.

> Language appears somewhere about the middle of the second year, but before this, about the end of the first year or the beginning of the second year, there is a sensory-motor intelligence that is a practical intelligence having its own logic- a logic of action. The actions that form sensory-motor intelligence are capable of being repeated and of being generalized. (Piaget, 1970, pp. 41–42)

Piaget noted that even before language develops, people think logically and learn through interacting with their environment. He observed infants tugging blankets as tools to to pull toys close enough to reach. As a child grows, "the practical logic of sensory-motor intelligence goes through a period of being internalized, of taking shape in thought at the level of representation rather than taking place only in the actual carrying out of actions" (p. 45). Piaget suggested that mental activity grows out of our mind's representations of action and model our impact on the surrounding world.

The action-driven sensorimotor stage leads to the preoperational stage when a child's mental model operates independently of physical activity. Preoperational mental representations are still quite immature. Driven by experience, mental manipulations are limited to simple observations without considering the logical consequences of manipulating an object. By the time a child reaches the concrete operational stage, they will understand "the identity of an object across transformations in its appearance or in the actions we perform upon it" (Bruner, 1997, p. 66), but until then their thinking still seems rooted in experience. For example, a preoperational child who watches a ball of clay smashed into a pancake shape will state that it now contains less clay than when it was a ball, rather than knowing the clay merely changed shape. Piaget called this concept, understanding that shape is not the same as quantity, *conservation*. Preoperational children consider only the shape of the clay in each form, failing to deduce the amount of clay does not change. The rules of conservation dictate that shape does not change the amount, but preoperational thinking seems to rely on perception, not logic.

Piaget documented many logical inconsistencies preoperational children exhibit. Preoperational children can classify jumbles of geometric shapes (e.g., circles, squares, triangles)

into distinct piles yet cannot articulate the relationships which distinguish the sorting process or logical operations on the groups. Their reasoning tends to deal with observable matters at hand.

> A child of this age will agree that all ducks are birds and that not all birds are ducks. But then, if he is asked whether out in the woods there are more birds or more ducks, he will say, "I don't know; I've never counted them." (Piaget, 1970, pp. 27–28)

Piaget referred to thinking at this stage as "semilogic," which he "used to call this articulated intuitions" (p. 50). Piaget's semilogic seems to have much in common with System 1, and later research shows that children who fail in conservation tasks seem to struggle to suppress their System 1 response (Houdé & Borst, 2015). Their thinking still requires visual evidence, not making leaps from 'rules' alone.

A child reaches the operational stages when their thinking grows beyond mere perception and starts to consider logic. Concrete operational thinking conjoins multiple attributes (e.g., height *and* width of the clay) in making judgments. Ginsberg and Opper (1988) summarized

> The concrete operational child focuses on several aspects of the situation simultaneously, is sensitive to transformations, and can reverse the direction of thought. Piaget conceives of these three aspects of thought- centration-decentration, static-dynamic, irreversibility-reversibility- as interdependent. If a child centers on the static aspects of a situation, he is unlikely to appreciate transformations. If he does not represent transformations, the child is unlikely to reverse his thought. By decentering, he comes to be aware of the transformations, which thus leads to reversibility in his thought. (p. 155)

Becoming a concrete operational thinker requires the integration of several complex types of thinking, but this thinking tends to be bound to a specific example. Piaget tested not only the conservation of solids (e.g., clay) but also used other media such as liquid in glasses, for example. Formal operational thinkers move beyond concrete operational reasoning by employing logic towards abstract and hypothetical thinking about the underlying principles.

> Confronted with a scientific problem, he begins not by observing the empirical results, but by thinking of the possibilities inherent in the situation. He imagines that many things *might* occur, that many interpretations of the data *might* be feasible, and that what has actually occurred is but one of a number of possible alternatives. (Ginsburg & Opper, 1988, p. 201)

Formal operational thinkers transcend specific predictions about a contextualized example (e.g., conservation within balls of clay) and instead deal with propositions (e.g., conservation of mass).

Observations inform thinking, but logic drives alternative explanations of the past or future for operational thinkers.

The formal operational stage of thinking is where Piaget's developmental model may break down for describing 'generalized' cognition. Adults do not always apply formal operational thinking in all activities. "Piaget does not mean to say that the typical adolescent of the formal stage *always* employs all or some of the formal operations,… but rather that he is *capable* of doing so" (Ginsburg & Opper, 1988, p. 203). Piaget sidestepped the question of age by proposing an alternative explanation.

> all normal subjects attain the stage of formal thought if not between 11/12 to 14/15 years in any case between 15/20 years. However this type of thought will reveal itself in the different activities of the individual according to their aptitudes and their professional specialisations (advanced studies or different types of apprenticeship for the various trades); the way in which these formal structures are used, however, is not necessarily the same in all cases. (Piaget, 1972, p. 10)

Piaget conceded that, for example, a lawyer would reason very differently than a physicist, about their respective disciplines. He suggested that each may 'forget' the required foundational knowledge they earlier learned when they later reason about topics outside their expertise. Piaget's explanation may sound familiar, as many educators are apt to point to 'forgetfulness' as a reason for student struggles.

6.1.2 Piaget's model of learning

Piaget wrote significantly more on his stages of development but also offered a model for learning. Piaget's driver of learning is "disequilibration, a process created by the relation between two component processes" (Bruner, 1997, p. 66). Learning happens when our model of the world conflicts with some new events or information.

> Disequilibrium, or imbalance, occurs when a person encounters an object or event that he is unable to assimilate due to the inadequacy of his cognitive structures. In such situations, there is a discrepancy or a conflict between the child's schemes and the requirements of experience. This is accompanied by feelings of unease. (Ginsburg & Opper, 1988, p. 227)

Piaget suggested that we learn when what we 'know' is challenged, resulting in discomfort. Learning is a natural response to better model our surrounding world. Disequilibrium does not

require conscious reasoning to learn, but our mind implicitly matures through experience. Ginsburg and Opper's summary of disequilibrium includes a few critical terms from Piaget's model: *scheme* and *assimilate* that are also essential to his model of learning.

Schemes (a.k.a., schemas) are the mind's mechanism for structuring knowledge. Piaget proposed that schemes generate and mature through a logical process of restructuring the network of knowledge and behaviors.

> Any given scheme in itself does not have a logical component, but schemes can be coordinated with one another, thus implying the general coordination of actions. These coordinations form a logic of actions that are the point of departure for the logical mathematical structures. For example, a scheme can consist of subschemes or subsystems. If I move a stick to move an object, there is within that scheme one subscheme of the relationships between the hand and the stick, a second subscheme of the relationship between the stick and the object, a third subscheme of the relationship between the object and its position in space, etc. (Piaget, 1970, p. 42)

The 'logic' that goes into structuring schemes is not conscious reasoning, but a side effect of the complex reorganization of simpler facts and actions. Complex actions originate from the coordination of many simpler actions. When a baby waves a rattle in the air, they are learning to control an object in their hand. When they poke other objects with that rattle, they are learning how to extend the reach of their hand. These precursor skills go into the future ability to use a stick to drag a faraway object within reach. The child's schema combines existing knowledge in new ways and with new ideas to mature their schema.

Piaget's definition of the schema is essential for his model of learning. Schemes drive learning, implicit or explicit, since "new structures are continuously being created out of the old ones and are employed to assist the individual in interaction with the world" (p. 23). For Piaget, memories build upon existing knowledge rather than forming anew upon instruction. Long before formal schooling, a child brings a head full of knowledge grounded in action and experience. New information, as a response to disequilibrium, updates our memory in one of two ways: *assimilation* into an existing schema or *accommodation* by recategorizing information to form new branches of our schema. Guzdial (2015) summarized, "Assimilation is the process by which new memories get added to existing networks… Accommodation is when you realize that the way you thought about the world does not work for the new memory" (p. 27). The process of learning seems individualized as a reflection of prior knowledge. For instance, Chapter 4 probably builds upon

existing knowledge for readers who had previously read Kahneman's (2011) book. Readers who have never heard of Kahneman or dual process theory had no existing scheme to extend, so they may find it more difficult to contextualize such details at first. Guzdial's description of accommodation, intentionally or not, implies some conscious awareness of the need to restructure knowledge. Piaget never detailed the mechanisms of learning to such a level of detail, that often leads to confusion about the differences in assimilation and accommodation.

The concepts of assimilation and accommodation are equal parts straightforward and cryptic. Ginsburg and Opper (1988) described Piaget's two mechanisms of learning further:

> Assimilation involves the person's dealing with the environment in terms of his structures, while accommodation involves the transformation of his structures in response to the environment (p. 19)

When a person is assimilating information, they are viewing the information as similar to what they already know, thus classifying it through that lens. For learning to be transformational, the learner must recognize a stark difference in what they are witnessing. A transformational lesson to one learner is a small change to another. For example, I learned to play the tenor saxophone starting in sixth grade, so five years later, it was a minor adjustment play the smaller soprano saxophone. The switch between saxophones only required minor physical adjustments to grip and embouchure (assimilation). In college, however, I taught myself to play the guitar. Playing the guitar not only required different hand movements but added the concepts of chords on top of the already familiar concepts of playing music. I had to accommodate not just new physical actions but also musical concepts in my learning. Accommodation is not an entirely separate process, but a means of establishing equilibrium, "there is no assimilation without accommodation" (Inhelder et al., 1976, p. 18).

The bulk of Piaget's theory of learning and development discusses infants and young children, but the same model likely applies through childhood and into adulthood. Piaget's model highlights the role of action and implicit learning as driver of development. Piaget's work long preceded the advent of the computer, though its ideas have influenced many computing education researchers. One such was Seymore Papert, who studied in Geneva with Piaget, who trusted Papert as an authority on Piagetian theories (Thornburg, 2013). Papert introduced many of Piaget's ideas into computing education and programming as a model for learning mathematical concepts through his work in creating and promoting the Logo language ("Logo history," 2015; Seymour;

Papert, Watt, DiSessa, & Weir, 1979). Papert (1991) also promoted his own extensions/adaptions to Piagetian theory. Raymond Lister (2016) and Donna Teague (2014) have promoted a Piagetian model for the developmental stages of programming. Piaget's influence on computing education, however tacit, is present in the literature, but educators lose many of his key observations in the disconnect between his model of disequilibrium, schema, assimilation, and accommodation and how they manifest in learners. TAMP looks to offer theoretical constructs that are better at connecting a model of thinking and learning with novice programmer's behaviors.

6.2 The Social Constructivism of Lev Vygotsky

Lev Vygotsky's tragically shortened career sparked intriguing ideas about learning and development that both complement and challenge the work of Piaget. Vygotsky worked through the Russian revolution balancing the political demands of the Marxist Revolution with flaws he observed in "Marxist psychology" (Vygotsky, 1986). His contributions, however influential, lay undiscovered for years due to –ironically– language, culture, and politics. Vygotsky (1962) felt, "Psychology owes a great deal to Jean Piaget" (p. 9), and he used Piaget's observations and analysis often in his work. The major division came in how the two interpreted language's role in development. Vygotsky proposed that language, not age, drove development. Vygotsky's best-known idea may be the Zone of Proximal Development (ZPD), which appears frequently in computing education publications (Berges, 2015; Chetty, 2015; Grover, 2014; Kalelioglu, Gulbahar, & Kukul, 2016; Kinnunen & Simon, 2012; Lister, 2016; Pea & Kurland, 1984; Robins, 2010; Strawhacker & Bers, 2014; Szabo et al., 2019; Teague, 2014; Werner, Ruvalcaba, & Denner, 2016; Whalley & Kasto, 2014; Williams, Layman, Osborne, & Katira, 2006). It is Vygotsky's thoughts on how language influences development that is the most salient for now.

6.2.1 Language and logic development

Vygotsky believed that language is a primary catalyst for intellectual growth. He stated, "an important concern of psychological research [is] to show how external knowledge and abilities in children become internalized" (Vygotsky, 1978, p. 91). Vygotsky believed that learning, not aging, advances development.

> Thought development is determined by language, i.e., by the linguistic tools of thought and by the sociocultural experience of the child. Essentially, the development of inner speech depends on outside factors; the development of logic in the child, as Piaget's studies have shown, is a direct function of his socialized speech. The child's intellectual growth is contingent on his mastering the social means of thought, that is, language. (p. 51)

Vygotsky believed that language is a forerunner for logical thinking, which he called *inner speech*. Vygotsky and Piaget arrived at drastically different conclusions from the same basic observations of children's egocentric speech. Egocentric speech is the tendency of younger children to verbalize their actions with themselves as the center. Piaget viewed egocentric speech as a sign of immaturity with no behavioral purpose that children grow out of as they mature. Vygotsky recategorized egocentric speech as a window to the development of reasoning in young children. He suggested that egocentric speech becomes inner speech. Vygotsky's work largely focused on children after they have reached the stage of verbal reasoning.

Vygotsky differentiates classroom learning from understanding developed through experience. He adopts Piaget's *spontaneous* concepts – those formed through their personalized mental effort and observation – and *non-spontaneous* concepts – those formed through the influence of another person. Concepts taught in the classroom tend to be non-spontaneous.

> In the case of scientific thinking, the primary role is played by initial verbal definition, which being applied systematically, gradually comes down to concrete phenomena. The development of spontaneous concepts knows no systematicity and goes from the phenomena upward toward generalizations. (Vygotsky, 1986, p. 148)

Vygotsky noted that people learn scientific ideas, in particular, from external sources such as instruction. Instruction begins with an organized and systematic sharing of the rules and concepts that eventually lead to concrete examples. Spontaneous concepts, on the other hand, generalize from a person's experiences. Vygotsky describes learning as an intersection between these two types of learning.

Vygotsky warned that instructors should not solely relying on either a 'top-down' or 'bottom-up' approach to learning. Depending on instruction alone promotes an "empty verbalism, a parrotlike repetition of words by the child, simulating a knowledge of the corresponding concepts but actually covering up a vacuum" (p. 150). Vygotsky suggests that instruction alone risks

leaving students who can speak to a subject, but devoid of practical uses for knowledge. Equally problematic is practical knowledge without considering its potential.

> The weak aspect of the child's use of spontaneous concepts lies in the child's inability to use these concepts freely and voluntarily and to form abstractions. The difficulty with scientific concepts lies in their verbalism, i.e., in their excessive abstractness and detachment from reality. At the same time, the very nature of scientific concepts prompts their deliberate use, the latter being their advantage over the spontaneous concepts. (pp. 148–149)

Vygotsky recognized that having a mental model of how the world behaves does not always lead to the abstract thinking required to apply that knowledge in new ways. Truly mastering a concept requires more than a 'parroting verbalism' or mindless automaticity of action, but integration of each.

> [A] concept is more than the sum of certain associative bonds formed by memory, more than a mere mental habit; it is a complex and genuine act of thought that cannot be taught by drilling, but can be accomplished only when the child's mental development itself has reached the requisite level. (p. 149)

Vygotsky seemed to foreshadow concepts were later captured in dual process theory in noting the role of "mental habit" separate from the "genuine act of thought" that requires more than rote practice. He suggested that further development is required, but perhaps never clearly establishes the nature of such development, which perhaps dual process theory might elaborate on.

6.2.2 Ramifications of Vygotsky's observations about language

Piaget and Vygotsky each advocated different sides of a 'chicken and the egg' argument: which comes first, language or understanding? Piaget suggested that children's early utterances are vacuous, and their real logic emerges from action. Vygotsky proposes that such early speaking is the progenitor of reasoned thinking. It would seem that words must have a well-defined meaning or, as Vygotsky (1962) put it, "A word without meaning is an empty sound, no longer part of human speech" (p. 5), yet people routinely use words they do not fully understand.

> The data on children's language strongly suggests that for a long time the word is to the child a property, rather than a symbol, of the object; that the child grasps the external structure word-object earlier than the inner symbolic structure. (p. 50)

Vygotsky suggested that children associate words with objects long before the significance of the word emerges. For example, think of how easily a brand name supplants the functional description: Kleenex for facial tissue, Rice Krispies instead of rice cereal, or in some regions of the United States, Coke for every type of soda/pop. We frequently use words that blend the name and concept in familiar consideration that may lose the significance of our experience compared to that of others.

Vygotsky spoke to the need to build language and their associate concepts upon experiences. Experience, as modeled by System 1, offers a rich and instant model of the world, but since System 2 is not always aware of the work of System 1, the formation of concepts from language takes a different path of acquisition.

> Closer study of the development of understanding and communication in childhood, however, has led to the conclusion that real communication requires meaning – i.e., generalization – as much as signs. According to Edward Sapir's penetrating description, the world of experience must be greatly simplified and generalized before it can be translated into symbols. Only in this way does communication become possible, for the individual's experience resides only in his own consciousness and is, strictly speaking, not communicable. (Vygotsky, 1962, p. 6)

Vygotsky points to the need to integrate experience and language. Language requires the reconciliation of two people's experiences into a common set of symbols. Learning through language is difficult or impossible if the learner has no personal experience with the concept behind the word. Twenty years ago, for example, my toddler daughter used to sing,

> Twinkle twinkle little star
> How I wonder what you are
> Up *a bubble* world so high
> Like a *diaper* in the sky

Notice how she substituted the word *bubble* and *diaper*. Her world was full of bubbles and diapers, so she associated those ideas rather than the similar-sounding and proper words *above* and *diamond*.

Vygotsky suggested that learning new words and concepts at the same time is particularly difficult. Dual process theory explains why. My toddler daughter remembered words she was familiar without considering the underlying concepts (e.g., It may make sense that a bubble is "so high" after all, but a diaper in the sky!?). One could argue that she was a toddler after all, why

should she consider a word's meaning? Age may be less of a factor than the mechanics of our brain suggested by dual process theory. Take, for example, the video of an 85-year-old grandmother's attempt to use Google's smart device captured on YouTube (Actis, 2017)[30]. Her family patiently instructed her to say either "Hey Google" or "OK Google" to activate the smart device before asking it a question. In a rather amusing sequence, she called out several combinations of "Hello *goo-goo*", "OK *goo-goo*", and "Hey *goo-goo*" before asking, "He want to know is the weather… tomorrow". She also adopted a habit of tapping the device, perhaps because touching devices is a normal mode of interaction or because the device flashed when she did so earlier. Eventually, the device responded with the weather, to which she grasps her husband's hands and says, "I'm scared… I'm scared… It a mystery… Oh my gosh!" This hilarious video offers a look at how people respond when language is unsupported by experience.

Technologists build a smart device to mimic natural interactions, but Vygotsky may have told them that the concept of interacting with a machine might still be a stretch for some people. In theory, a user simply needs to say a simple recognition phrase and speak in natural language, but despite careful instructions, our grandmother struggles in a very entertaining way. We do not know if she is familiar with the company Google, but despite her relatives repeating the name quite clearly, she persists in saying *goo-goo*. Like my daughter, her mind clings to the closest sounding word, seemingly baby-talk. Before recent advances, most electronics technologies required touch, so she may have associated the accidentally timed flash of lights and her touch and continued to do so while she tried voice commands as well. She is literally unable to speak *its* language properly.

The grandmother, as Vygotsky phrased it, parrots empty verbalism in repeating her relative's instructions. Despite her thick accent, she seems fluent in English, so the issue is understanding Google Home's communication protocol. The alienness of the device and its purpose makes even mimicking a challenge. Her example offers a wonderful depiction of how even a wise and revered adult can lose footing when language and action are devoid of referential experience. The makers of smart devices seek to remove technology as a barrier by allowing for natural language as an interface – something that should be 'intuitive' to everyone. Our grandmother knew the language, what she was asking it to do (retrieve the weather report), had the help of informed mentors, and still struggled to produce the required grammar for several tries

[30] For quick reference: https://www.youtube.com/watch?v=e2R0NSKtVA0

and was amazed when the device performed the simple task. Perhaps her response is exactly what some novice programmers experience when they first encounter the foreign languages of programming?

Vygotsky's theories hold many valuable insights into learning and development, but for now, a brief look at his work on language and logic helps to inform the struggles novices may face when learning to program. Piaget emphasized the importance of action in the early stages of learning, as captured within System 1. Vygotsky explored the emergence of conscious reasoning and the intersection of experience and formal education, which seems to map to System 2. Even when these two disagree, their arguments may simply be describing the different aspects of our brain's mental mechanism as much as a fundamental schism in their models. Both Piaget and Vygotsky, however, focus most of their work on the very early stages of childhood development and thus only take computing educators so far in tackling the much more complex task of learning to program.

6.3 Jerome Bruner and his model of cognitive problem-solving

Jerome Bruner combined elements of Piaget and Vygotsky into a model that combines Piaget's action and Vygotsky's symbols. Bruner (1966c) believed that humans "developed three parallel systems for processing information and for representing it – one through manipulation and action, one through perceptual organization and imagery, and one through symbolic apparatus" (p. 28). Bruner's three mental representations – enactive, iconic, and symbolic – describe different ways of knowing yet also offer a model for how people combine experience and formal education and use their knowledge to solve problems. Revisiting Bruner's theories from the perspective of dual process theory not only offers insights into Bruner's descriptions but provides a vocabulary to describe the complex interplay between intuition (System 1) and reason (System 2).

6.3.1 Bruner's view of 'knowing' and development

Bruner's work challenges traditional measures of classroom learning. Too often, assessments of knowledge devolve to simple remembering; can the student recall and apply the proper facts. Bruner (1966a) described three ways our mind represents information.

> There are two senses in which representations can be understood: in terms of the medium employed and in terms of its objective. With respect to the first, we can talk of three ways in which somebody "knows" something: through doing it, through a picture or image of it, and through some such symbolic means as language. (p. 6)

Bruner's three "ways of knowing" reminds us that knowledge is only useful when it aligns with the intended purpose. We can see a carpenter 'knows' construction as they build things. Many may not 'read' blueprints or understand every rationale for each habit they employ in their trade, yet those habits passed down by example still lead to safe structures. The architect may never pick up building tools but can envision the scope of the work and formulate plans to build trusses, stairways, or other elements in a building. They work in the space of functionality and aesthetics yet turn over certain tasks to engineers. The engineer tends to dwell in the symbology of construction (e.g., free body diagrams, mathematical computations) to ensure proper safety and durability. They can calculate the precise load a beam must carry but may never see the final project after construction. While some people learn to master every aspect of a discipline, often the role a person plays may influence the type of knowledge they must acquire to do a job.

Knowledge is often most valuable when contextualized to need. Many years ago, I partnered with some friends to renovate a neglected Victorian home (circa 1890). I brought financing, organization, planning, and thirst to learn more about construction. My partners brought practical construction experience (much more useful in this case). One of our first projects pitted my symbolic education as an engineer against their practical knowledge of construction. We were replacing the deck on my side porch but first needed to ensure the frame was square (formed a right angle) with the house. My twelve-year-past training as an electrical engineer triggered Pythagoras as a procedure for ensuring a right angle, so I proceeded to precisely measure the three sides to see if *a-squared* and *b-squared* indeed totaled *c-squared*. After a few minutes on my calculator resolving the decimal to fractions of an inch on the tape measure, I was satisfied we were square. My two partners, probably hiding their eye-roll, quickly marked out three feet on one board, four feet on the next, and checked if the distance between those points was five feet, effectively making a 3-4-5 right triangle in a fraction of the time and with much less computation. Either approach to ensure the corner was square could work, but one is significantly easier to learn, execute, and perfectly acceptable for the situation.

Bruner's representations remind educators that not only do we have different mechanisms for storing knowledge, but each mechanism also plays a specific role in how we later use that information. To be the most effective, educators should understanding how each type of representation forms, how our mind uses them, and how to promote the types of knowledge required for the desired task. Computing educators have the additional burden in that the discipline rarely allows for a 'clean' separation of duties as the earlier example of the carpenter, architect, and engineer. Often a programmer must be an expert in all three, conceiving, building, and testing their solution entirely on their own.

6.3.2 Bruner's three representations

Unfortunately, many who summarize Bruner's representations provide such simple of his theory as to seems trivial. One website presents the following.

- Enactive representation (action-based)
- Iconic representation (image-based)
- Symbolic representation (language-based) (McLeod, 2019)

As a very high-level summary, McLeod's bullet points align with Bruner's descriptions. Enactive representations store implicit actions we take based on our perceptions of events. Bruner described iconic representations as imagery but personalized and abstracted from experience. Symbolic representations may be languages, but they could be diagrams, programming instructions, or merely a new set of facts using existing words. The oversimplification of Bruner's ideas may be the reason researchers overlook the power of his representation to describe the acquisition of complex ideas and abilities.

6.3.2.1 Enactive Representations

Enactive representations capture the logic behind our actions grounded in experiences, much like Piaget described.

> By enactive representation I mean a mode of representing past events through appropriate motor response. We cannot, for example, give an adequate description of familiar sidewalks or floors over which we habitually walk, nor do we have much of an image of what they are like, yet we get about them without tripping or even looking much. Such segments of our environment—

> bicycle riding, tying knots, aspects of driving—get represented in our muscles, so to speak. (Bruner, 1964, p. 2)

Enactive representations form implicitly, though we intentionally shape them by consciously altering our perceptions of events. For example, we may develop a fear of spiders as a child. As an adult encountering a spider, we can remind ourselves that it is unlikely to attack us and with practice handle them with ease (at least the non-venomous varieties). Bruner's descriptions of enactive representations align closely with System 1.

> If you have tried to coach somebody at tennis or skiing or to teach a child to ride a bike, you will have been struck by the wordlessness and the diagrammatic impotence of the teaching process. (p. 10)

Enactive representations require more than memorization of following a process and are difficult to master without practice. Bruner (1966c) primarily described enactive representations within the domain of physical activity. Dual process theory allows us to comfortably extend enactive representations to any mental ability that is automatic, unconscious, and forms implicitly. Enactive representations provide the 'gut feeling' (i.e., intuition) for a subject.

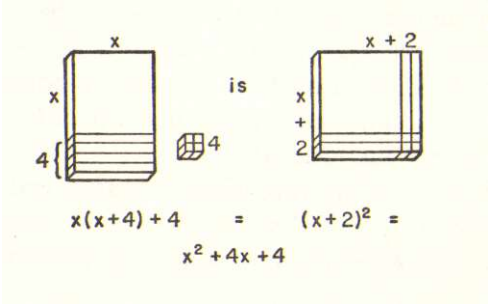

Figure 6.1. Using blocks to teach quadratic equations (Bruner, 1966c, p. 62)

Bruner suggested that forming enactive representations is a critical step in learning. In his research, he saw that when left to the symbols alone, learners fumbled with even basic conceptual understanding.

> What was so striking in the performance of the children was their inability to represent things to themselves in a way that transcended immediate perceptual grasp. (p. 65)

Instruction alone did little to guide learning. Bruner suggested conceptual understanding requires

> the building of a mediating representational structure that transcends such immediate imagery, that renders a sequence of acts and image unitary and simultaneous. (p. 65)

He proposed that learning is more effective when it starts with concrete examples and builds generalizations. The example in Figure 6.1 shows how a set of blocks provides a tangible example of quadratic equations. A child can reorder the blocks to represent the same equation in various ways. The image on the left shows a chunk of (x blocks wide by $x + 4$ blocks tall) plus four more blocks. The set of blocks are reordered to ($x + 2$ blocks wide) by ($x + 2$ blocks tall). Bruner offered students a tangible, modifiable experience from which to understand the abstract rules of algebra. To become proficient at algebra, a student must master the symbols of manipulating variables, but Bruner believed that abstractions are most powerful when formed out of the generalized experience provided within enactive representations.

6.3.2.2 Iconic Representations

An iconic representation serves several purposes, the most important of which is generalizing experience and combining experience with formally acquired knowledge. Bruner (1966a) summarized the iconic representation as "a set of summary images or graphics that stand for a concept without defining it fully" (p. 44). Iconic representations are not perfect 'textbook' concepts. They offer general ideas that emerge from personal experience. Unfortunately, authors too often claim any picture acts as an iconic representation, but Bruner was quite specific on what makes imagery iconic.

> Iconic representation summarizes events by the selective organization of percepts and of images, by the spatial, temporal, and qualitative structures of the perceptual field and their transformed images. Images "stand for" perceptual events in the close but conventionally selective way that a picture stands for the object pictured. (Bruner, 1964, p. 2)

A teacher cannot simply provide imagery to a leaner and state it satisfies the need for the iconic representations. Imagery must "stand for" a concept that a person is forming from experience. Iconic representations help to move away from action alone and begin manipulating ideas.

> Once a schema becomes abstracted from a particular act and becomes related to serial acts in a one-many relation it can become the basis for action-free imagery. (Bruner, 1966a, p. 19)

Enactive representations are an implicit response based on experience. Iconic representations allow a person to change that response to accommodate new demands.

Bruner offered precious few examples of iconic imagery but did offer one useful example based on his research. He described an experiment where a child sees a rack of 42 pegs set in 7 rows and 6 columns of pegs with a ring on one of the pegs (see Figure 6.2). The researcher asked the child to place a ring on a second rack to match the first. Children at any age can easily place their ring so long as the racks face the same direction. Simply turning the two racks to different orientations changes the task entirely. Placing a ring within the same orientation is a matter of perceptual matching, an enactive task. By changing orientation, the child must create an intermediary mental representation (iconic) to find the proper place on the grid. As an adult, you may simplify the task to simply remembering the row/column of the target peg, then determining the new orientation of the starting point. Where the child can complete the enactive task (the same orientation) perceptually, the iconic task (shifted orientation) requires conscious consideration of the change, and many struggle do so at first.

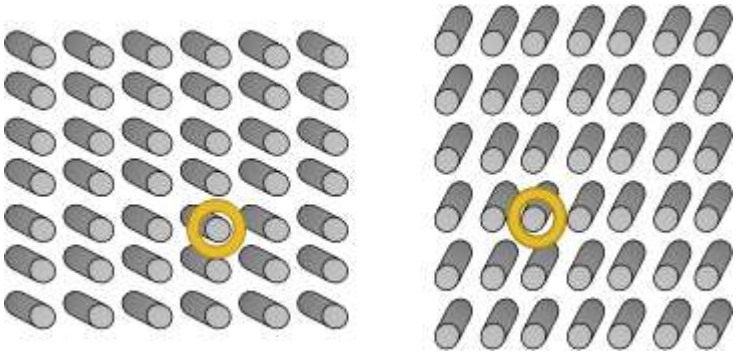

Figure 6.2. Bruner's peg task

The peg task demonstrates a difference in the level of abstraction required to solve even simple problems. As a perceptual task, 42 pegs in the same orientation are relatively easy to match but is difficult to translate without the aid of counting or some other representation. Simply mimicry is not enough to reliable complete the task when the orientation shifts, a conscious strategy is helpful. Where most adults have seen similar tasks before (e.g., finding your car in a parking garage where every floor looks the same), children may not have a strategy easily at hand. When the task shifts from enactive to iconic, they are left to derive a novel solution, not merely retain a row/column value in short-term memory and then compute the proper placement. Iconic

representations are not merely a simpler perceptual image of something, but also the ability to manipulate that image in a meaningful way.

Many novice programmers struggle in a similar shift in 'orientation' within coding examples. Since most simple loops count upward, say from one to ten, new programmers begin to associate a loop as a thing that starts low and goes high. They become experts at reproducing this pattern of code without much consideration of the details of the looping constructs. When the test rolls around, the clever instructor presents a problem that loops 'backward' from the high number to the low instead to test the students' conceptual understanding of loops[31]. Some students are unfazed by this trick, yet many fall prey to the 'backward' loop and miss the question – a seemingly clear indicator that they do not know loops! Like the children seeking the proper peg for their ring, past perception dominates their reasoning. Until they encounter a 'backward' loop, they did not need to consider the specific details of the looping construct; their enactive representation served them well. Iconic representations are a measure of 'advanced thinking', but Bruner suggests form best through varied experience.

The crux of iconic representation is that they are individualized. Bruner did not merely show students the diagram from Figure 6.1 to show a picture of quadratic equations. Students needed a physical device to manipulate and form their own iconic representations. As Du Boulay et al. (1981) noted, programmers need to see inside the black box that is a computer to understand its workings. Bruner may have agreed that such knowledge is essential in building the iconic representations needed to learn to program. The challenge in computing is creating environments that are truly interactive where learners can manipulate and see the results. Merely providing intermediate notations is insufficient. Section 5.2.2 offered an example of a *forced intermediate symbolic notation* in the form of the trace table. Lister et al.'s (2004) example showed the student preferred their own doodles to the provided (and more logical) imagery. In time, a learner may come to adopt such standard documentation styles, but not until they have acquired sufficient symbolic representations.

[31] For the moment, ignore the fact that the instructor probably lectured the students on the components of a loop and the rules of iteration, both forward and backward. Such instruction is rooted in symbolic representations. This method of instruction is possible, but Bruner says less optimal as will be discussed later.

6.3.2.3 Symbolic Representations

Symbolic representations store knowledge from external sources, especially those that come with new systems of symbols. The most common symbolic representation is language, though symbolic representations hold any system of symbols, like the notations used in mathematics, physics, or programming. Even images can contain symbolic content, as Bruner (1966b) clarified since images "can be infused with symbolic functioning" (p. 30). When knowledge comes from external sources, it begins as symbolic until internalized into other forms of representations.

> In general, when we speak of the grammar of a language we mean the set of rules that will generate any or all permissible utterances in that language and none that are impermissible. When a person speaks a language, he "knows" these rules in some fashion, though he cannot (like a linguist) recite them to you. (p. 33)

Symbolic representations are paradoxical in that they can develop quite implicitly. Children do not learn to speak from lectures but from the experience of listening to others and receiving subtle feedback. Some aspects of 'knowing', though seemingly symbolic, are implicit, though Bruner suggested the creation of symbolic representations opens the door to the conscious application of advanced logic.

> But the critical point of symbolic representation is that these powerful productive rules of grammar are linked to the semantic function as well–to the "real world"; that is to say, having translated or encoded a set of events into a rule-bound symbolic system, a human being is then able to transform that representation into an altered version that may but does not necessarily correspond to some potential set of events. It is in this form of effective productivity that makes symbolic representations such a powerful tool for thinking or problem solving: the range it permits for experimental alteration of the environment without having, so to speak, to raise a finger by the way of trial and error or to picture anything in the minds eye by imagery. (p. 37)

Bruner implied that symbols enable hypothetical reasoning without the need to take action or construct images. Language can transform reality by simply presenting a fanciful idea without considering the plausibility of such statements. He used the example of a child asking, "What if there were never any apples?" (p. 37), which indeed is a simple language construct with profound logical consideration.

Symbolic representations present a dichotomy that Bruner explored by invoking both Piaget and Vygotsky. Invoking Piaget, he noted that children are generally unaware of the semantic implications of their words and thus do not often use words in creative ways outside their everyday implications. Our language generally reflects our range of experience, so such fanciful "apples" questions are more likely to be spurious than deeply considered. He seems to describe this aspect of language in terms of System 1. At the same time, Bruner invoked Vygotsky's inner speech, noting how difficult it is to observe and analyze. Bruner rejected Vygotsky's notion of internalized speech, instead preferring the idea that language helps to organize our thoughts.

> I tend to think of symbolic activity of some basic or primitive type that finds its first and fullest expression in language, then in tool-using, and finally in the organizing of experience. (p. 44)

Learning a symbolic representation does not unlock logical potential but instead influences how people organize experiences. Bruner might say that learning to code does not make one a logical thinker, but rather provides a framework to organize experience logically, at least using programming languages.

Building useful symbolic representations requires not only recognizing symbols but also blending the symbols with other representations of knowledge. Bruner pointed out that children reach syntactical maturity by the age of five yet cannot deconstruct their sentences into their grammatical parts. The words convey intent and meaning with little connection to the implications of the chosen words. Language is as much habitual as carefully chosen (e.g., System 2 often System 1's word choice rather than directly forming sentences).

> One is thus led to believe that, in order for the child to use language as an instrument of thought, he must first bring the world of experience under control of principles of organization that are in some degree isomorphic with the structural principles of syntax. Without special training in the symbolic *representation of experience*, the child grows to adulthood still depending in large measure on the enactive and ikonic modes of representing and organizing the world, no matter what language he speaks. (p. 47, emphasis is Bruner's)

Bruner, like Vygotsky, reinforced the need to blend knowledge and experience lest a leaner exhibit empty verbalism, speaking without comprehending. The shallow understanding of symbolic notations is not limited to language. Perkins (2010) told the story of physics students who masterfully compute the speed of an object falling from a tower, yet forget the same process applies

when falling into a hole. Mastering a symbolic notation does not guarantee its application to all relevant circumstances. Without symbolic representations, people may never transcend their experiences. Generations of people watched the sky and concluded the Earth is flat and circled by the Sun. Without a symbolic representation to restructure experience, we are unable to move beyond perception. While instructors dedicate much of education to the expeditious creation of symbolic representations, Bruner's tri-part model helps to organize the best use and formation of all three types of representations.

6.3.2.4 A system of representations

Bruner's three representations are highly interconnected, and knowledge of the same topic may transcend all three representations or 'move between' them. Bruner suggested a preferred, possibly optimal, but hardly exclusive sequence to forming representations.

> It is true that the usual course of intellectual development moves from enactive through iconic to symbolic representation of the world, it is likely that an optimum sequence will progress in the same direction. Obviously, this is a conservative doctrine. For when the learner has a well-developed symbolic system, it may be possible to bypass the first two stages. But one does so with the risk that the learner may not possess the imagery to fall back on when his symbolic transformations fail to achieve a goal in problem solving. (Bruner, 1966c, p. 49)

In childhood, we learn first through observation and action and form enactive representations. As we gain experience, we create iconic representations that reconcile various experiences with each other and with newly formed symbolic representations. When we reach school, formal education often introduces information in symbolic form without associated experiences. A student can learn geography to some degree without visiting each location, but the best part of chemistry may be the lab experiments for grounding the concepts with experience. Du Boulay (1986; 1981) noted that programming students are often taught languages before understanding the inner workings of computers. Bruner suggested that instruction about the system of symbols (e.g., language syntax and semantics) is possible but risks developing students who are weak at solving problems, an issue well documented amongst programming students (See Chapter 2). Bruner's representations offer a useful model for describing the different representations of knowledge that a programmer must develop and how these representations of knowledge work together in problem solving.

6.3.3 Applying Bruner's theories

There is no evidence that Bruner ever discussed dual process theory, but his works are full of similar observations. In "The Process of Education", Bruner (1976b) dedicated an entire chapter to "Intuitive and Analytical Thinking" (p. 55). He discussed the value placed on intuition in many fields suggesting, "the intuitive thinker may even invent or discover problems that the analyst would not" (p. 58). Sounding quite the dual process theorist, he predicted that many problems were either unsolvable or would take much longer by analytical means and perhaps shadowing enactive representations he discussed the importance of intuition to learning.

> For, as we have seen, it may be of the first importance to establish an intuitive understanding of materials before we expose our students to more traditional and formal methods of deduction and proof. (p. 59)

Much of Bruner's work focused on learning mathematics, finding ways to create intuition through hands-on activities. In "Studies in Cognitive Growth", Bruner (1966c) noted, "when children give wrong answers it is not so often that they are wrong as that they are answering another question, and the job is to find out what question they are in fact answering" (p. 4). In processing the disconnect between language and action, Bruner's observation mirrors that of Kahneman (2011), "If a satisfactory answer to a hard question is not found quickly, System 1 will find a related question that is easier and will answer it" (p. 97). I believe that Bruner saw the same quirks in cognition that led to dual process theory, and as such, his theoretical constructs relate to those of dual process theory as follows.

Premise 1: Enactive representations are the memories used by System 1
Premise 2: Symbolic representations describe factual knowledge used by System 2.
Premise 3: Iconic representations model the individual's use of facts and experience, particularly during problem solving
Conclusion: Jerome Bruner's representations provide a useful language for describing the mental representations used within the two Systems and how they interact during cognition, particularly for solving complex problems.

6.3.3.1 Representations versus Systems

Bruner's representations may not directly align with dual process theory, but the representations and two Systems may provide different models of the same mechanisms of cognition. Bruner's obviously has three components, while dual process theory has two. More

importantly, they capture different perspectives on cognition yet describe significant overlap. Enactive representations are the easiest to deal with as they seem to fall entirely within System 1– they are implicit, govern action, and represent the sum of experience, but not episodic memories.

> Enactive representation is based, it seems, upon a learning of response and forms of habituation. (Bruner, 1966c, p. 11)

Enactive representations seem to describe the same phenomena captured within System 1. Symbolic representations are more challenging to dissect. Bruner described the acquisition of syntax and semantics in children as implicit and unconscious (fitting into System 1), yet also says symbolic representations enable higher forms of conscious thought (a hallmark of System 2). Likewise, iconic representations seem to offer both implicit and explicit benefits that seem to span the two systems, so each requires more analysis to place in the context of dual process theory properly.

One aspect that Bruner described of iconic representations is the metacognitive function of recognizing patterns in experience. Section 4.2.2.1 described Berry and Broadbent's (1988) study where participants were better at learning to control a sugar factory implicitly, and that experience alone did little to improve their conscious understanding of the rules. Pattern matching alone seems to be a function of enactive representations/System 1 but Bruner described a metacognitive aspect to the formation of iconic representations that seems to transcend the two Systems. If iconic representations support active deliberation, thus must reside within System 2. Can iconic representations learn directly from enactive representations rooted in experiences? Bruner (1966a) seems to have leaned towards the conscious discovery of abstractions in using phrases such as "represent the world to himself" (p. 21). He also focused much of his research around tracking the perceptual attention of a child (a task that requires the conscious attention of System 2). He noted, when discussing iconic representations that

> Affective and motivational factors affect imagery and perceptual organization strikingly, particularly when impoverished stimulus materials is used and linguistic categorization rendered ambiguous. (Bruner, 1976b)

Dual process theory includes little discussion on affect and motivation, yet Bruner noted their impact on iconic processing. By noting that 'non-cognitive' factors impact perception *and imagery* (iconic), he seems to be reinforcing the conscious nature of iconic representations

implying they are an aspect of System 2[32]. Before finalizing the intersection of iconic representations and dual process theory, Bruner's model presents one more wrinkle.

While Bruner (1976b) occasionally discussed intuition directly, he admitted the need to clarify its nature. Perhaps he saw intuition as playing some role in his representations, but he did not say so explicitly. A scant few years after introducing his three representations, he dedicated a chapter to the importance of intuition. It seems that any iconic or symbolic representation that is frequently invoked could become automated. Piaget noted that schemas build by combining smaller actions into larger ones (e.g., grabbing a racket combined with swinging it translates into playing tennis), but Bruner never indicated a lifecycle for his representations. Iconic representations may fill a temporary need for conscious cognition and are either forgotten in time or if encountered repeatedly become enactive representations. Some iconic representations may originate to aid System 2 manage knowledge yet gradually become a System 1 process if used regularly.

Bruner's work contains one more moment of synchronicity (or coincidence) with dual process theory to consider. Vygotsky proposed the concept of inner speech as the driver of logical thinking, but Bruner disagreed. Bruner proposed iconic representations and imagery as the inner model of conscious reasoning. In rejecting inner speech, Bruner separated language and logic, and neuroscience may offer some support to his assertion. One study analyzed fMRI scans and determined that people have a hard time thinking purely in language (Amit, Hoeflin, Hamzah, & Fedorenko, 2017). They concluded that "people tend to generate visual images of what they think about verbally" (p. 29). Bruner's description of imagery might be not just a useful metaphor, but an apt description of how our mind considers concepts and language.

The evidence from Bruner's writing hints that his representations, if not aligned with dual process theory, do not conflict with it. TAMP revisits Bruner's representations not only through the lens of dual process theory but also using innovations from neuroscience about the nature of memory to understand the differences between each representation better and their alignment with dual process theory. Bruner's model provides a solid foundation of concepts from which to

[32] Kahneman used the example that providing students with blurry test papers made it less likely to make mistakes (Frederick, 2005). The blurry pages presumedly caused the students to engage System 2 more than the students who received clearly legible papers and reduced the number of errors they made.

reconsider how people think and learn as well as a well-conceived model for describing complex aspects of cognition that combine action and logic.

6.3.4 Next steps

Even simple examples of using Bruner's representations in programming leaves a lot to unpack about how we learn and teach programming.

- If enactive experiences are difficult to create and even then, may not transfer, how do we create these vital representations?
- What is the use/value of 'simplified' representations such as pseudocode, flowcharts, or activities/languages that reduce the need to master syntax?
- The examples thus far have focused on the notional machine level of code, but what about higher-level algorithms and design patterns?
- How do Bruner's theories change the pedagogical approach to instruction?
- How can we better measure progress in learners?

Hopefully, this list is just the start of bigger questions that Bruner's ideas, as well as those of Piaget and Vygotsky, inspired. The goal of reviewing the learning theories was to expand not only how dual process theory challenges traditional assumptions of learning, but also to see how to further the methods currently in use.

7. THE THEORY OF APPLIED MIND OF PROGRAMMING

7.1 An Advance Organizer for building TAMP

TAMP provides researchers and educators with new perspectives on computing education grounded in the well-established literature on cognition and learning. As a 'theory of mind' TAMP creates a model of cognitive function that provides disciplinary educators with the same guidance that a notional machine provides programmers. A notional machine portrays the inner workings of a computer. TAMP builds upon existing theory to do the same for cognition. Instructors cannot reasonably cover every possible feature and use of a computer language in a single notional machine. Likewise, this dissertation introduced a small slice of what TAMP might become. This chapter tackles a relatively narrow slice of programming cognition, but Section 9.1 highlights some of the possible future content of TAMP. Before jumping into the construction and justification for TAMP, it is worth taking a moment to summarize what this dissertation has stabled and where it is going (see Figure 7.1) as an advance organizer (Ausubel, 1960) for the question "What is TAMP?"

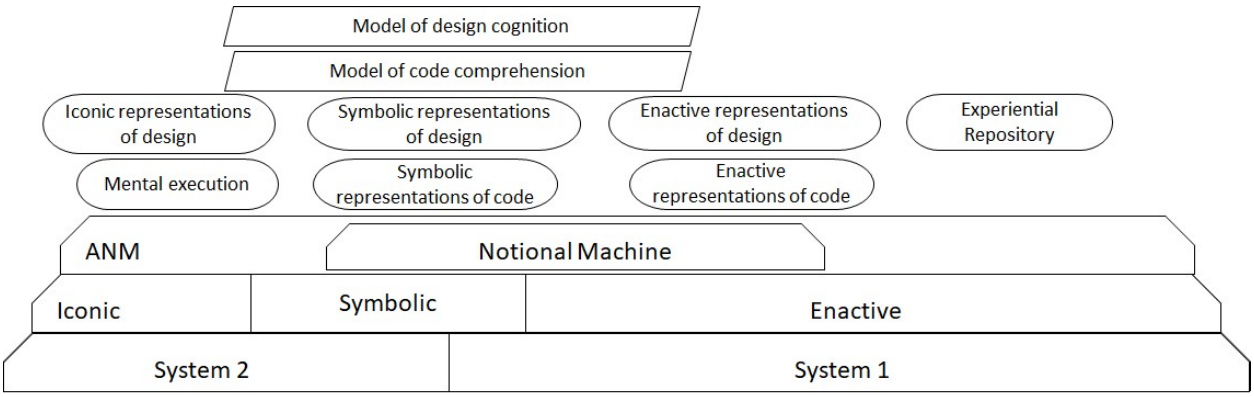

Figure 7.1 A model of TAMP's scope within this dissertation.

The inception of TAMP begins with dual process theory and its new perspective on how people think. Chapter 4 argued that dual process theory provides a powerful model that might explain many elusive aspects of expert cognition (see Section 2.3.3). Chapter 5 demonstrated dual process theory's power in reinterpreting influential computing education research by finding additional insights and explaining areas that may have been previously unclear. Dual process theory provides a powerful abstraction that lives between the models of traditional learning theory

and the detailed findings from neuroscience. The last half-century has seen major advances in our understanding of the brain, which has dramatic impacts on theories of learning[33] but often describes neural mechanics at too low of a level to be useful in the classroom. Dual process theory provides disciplinary educators with accessible constructs (System 1 and 2) for understanding the cognition of decision making. Grounding TAMP's 'theory of mind' in dual process theory provides a bridge between traditional learning theories and the evolving model of cognition emerging from the cognitive sciences.

While dual process theory provides vital insights to cognition, its focus is not learning. Understanding how experts think is useful but TAMP also seeks to provide guidance towards educating novices. Chapter 6 revisited a century of research into educational psychology by revisiting three of the most influential theorist in Piaget, Vygotsky, and Bruner. When viewed through the lens of dual process theory, these theorists foreshadowed the role of tacit knowledge and skills that dual process theory describes within System 1. TAMP utilizes Jerome Bruner's representations as a model of knowledge that, of the available learning theories, best aligns with dual process theory. Bruner's three representations seem to encapsulate many aspects of dual process theory. His enactive representations describe the same phenomena modeled within System 1. The split between symbolic and iconic representation provides a model to describe the interplay between the two Systems of dual process theory and, as Section 7.3 will consider, possibly models the differences between episodic and semantic memories. TAMP refines Bruner's representations with new insights from neuroscience to operationalize his constructs within computing education.

The heart of this dissertation's contribution to TAMP is the Applied Notional Machine (ANM). While many computing education researchers stress the importance for students to create mental models of the notional machine, there is little guidance as to what those models look like. The ANM looks to define the mental models that programmers form in response to exposure to a notional machine. The ANM is, in essence, a 'cognitive notional machine' for computing educators to use in understanding students' mental models of the functional aspects of programming. The ANM places the knowledge portrayed by a traditional notional machine within Bruner's representations. At a high level, programmers acquire facts about programming (symbolic), tacit knowledge and skills (enactive), and information used in problem-solving

[33] This chapter uses the biology of memory and learning to support TAMP's suggested new conventions that help to clarify Bruner's representations!

(iconic). Section 7.5 defines the ANM as a new theoretical construct within computing education. The ANM looks to serve as a framework for modeling and measuring learning by building upon the insights from Bruner, dual process theory, and the neuroscience of prospection. The chapter then builds upon the ANM to propose specific mental constructs of programming knowledge. These new constructs lead to two propositions: *how programmers read and comprehend code* and *how they use knowledge and experience when designing*. These two propositions demonstrate the ANM in action and provide the theoretical building blocks that Chapter 8 uses in revisiting existing studies on how programming students struggle to design and write code from scratch.

TAMP organizes the bodies of knowledge from dual process theory, neuroscience, and learning theory that culminates in a set of new disciplinary constructs and propositions within computing education. Chapters 3-6 established that existing theory offered useful insights to computing education and was complementary to a cohesive model of cognition and learning. This chapter builds upon these theories to provide a modern view of Bruner's representation as applied to computing education. By the end of Chapter 8, this dissertation provides researchers and educators with a new set of internally validated constructs from which to conduct additional research. Chapter 9 then considers additional aspects of a full 'theory of mind' that were out of scope for this dissertation, as well as future empirical research to add to the evidence provided via rhetorical argument and reinterpretation.

7.2 Defining TAMP

Rene Descartes, among others, may have accidentally confounded the work of generations of educators by promoting the supremacy of human reasoning and overlooking the vital role of intuition[34]. Dual process theory does not dispute human rationality but suggests a different internal process by which people come to decisions. For example, by revisiting influential research in computing education, TAMP offers an alternative, if not richer, explanation of phenomena such as fragile knowledge (T. Lowe, 2019) or movers/stoppers (D. N. Perkins et al., 1986). Even when educators and researchers know about dual process theory – I have met dozens of educators over the past year who have read Kahneman's (2011) bestseller – they may not see how to put its tenets into practice within computational education. This chapter attempts to demonstrate how dual

[34] And emotions, but 'non-cognitive' factors fell out of the scope of this dissertation.

process theory impacts programming cognition yet doing so requires illustrating the interplay between intuition and reason, for which Bruner's representations may offer guidance.

In revisiting educational and developmental theory, I try to establish the foundations behind many of the modern pedagogical approaches and blend traditional models of learning with dual process theory. Piaget, Vygotsky, and Bruner stand the test of time because of the quality of their research and insights, and their theories are enhanced and possibly improved when considered through the lens of dual process theory. Where dual process theory describes how people think, learning theory focuses on the acquisition of knowledge and skills, making a natural pairing. Piaget described how we learn from actions and their influence on conscious reasoning. Vygotsky considered how social interactions influence our learning and thinking. Bruner combined these two aspects of learning in his mental representations. TAMP not only seeks to refine existing learning theory with dual process theory and findings from neuroscience but also to help transfer abstract ideas about learning and cognition into programming education practice.

7.2.1 Scope

This chapter builds upon dual process theory and Bruner's representations to propose a model of how the mind stores and uses knowledge when programming. Early in my process, I imagined TAMP as a model of a programmer's mental activity during each task of the development process. Such a list might include,

- Reading code
- Tracing code
- Writing code (from a guide)
- Analyzing needs (writing requirements)
- Design (but not writing code)
- Creating tests
- Debugging
- Refactoring (rewriting code to improve the design)

The challenge with modeling every single activity on such an exhaustive list is one of variation across individuals. While I believe there are common memory structures (as encapsulated with Bruner's representations) and mechanisms (System 1 and 2), it is unlikely that all experts follow

exactly the same mental processes. As we will see in Section 8.1, expert designers share characteristics in their thinking, but they still follow individualized processes.

If educational researchers cannot proscribe universal pathways to becoming a programmer, is the secret to training novices personalized curricula? Individualized learning is a laudable goal, yet current attempts are too often cumbersome and perhaps ineffective. Piaget and Vygotsky each advocated that educators could help students by identifying their needs and customizing their education. Teaching at scale, however, makes such a prospect daunting. Perhaps one day, intelligent educational systems can diagnose a learner's troubles and guide them. I stand with Bruner (1966c) who warned educators to "prevent oneself from becoming a perennial source of information" and promote the student's "ability to take over the role of being his own corrector" (p. 70). The heart of constructivist educational philosophy states that individuals build knowledge through experience, rather than instruction. Educators help most when they present students with the right experiences at the right times, but even better when they teach students to seek educative experiences. Rather than providing a handbook for diagnosing student struggles, TAMP defines the concepts and practices needed by competent programmers and suggests the types of experiences useful in building such knowledge.

Rather than attempting an exact roadmap for how every programmer thinks, TAMP seeks to model the types of memories and the mechanisms of cognition that support programming behaviors. Within this chapter, I will occasionally use my professional experiences as examples of the theoretical constructs of TAMP in action. I do not expect they match yours, or they are even prototypical, but they attempt to illustrate the concepts being discussed. These experiences may resonate or vary drastically with other programmers; the goal is not to model how intuition drives all programmers. These examples demonstrate intuition in action as a vital part of cognition in addition to less story-driven empirical data. Chapter 8 revisits empirical evidence for the purpose of validation. My goal at this stage of defining TAMP is not to capture every mental activity of programming. Instead, I aim to refine existing ideas about how programmers think using TAMP's newly proposed theoretical constructs.

In modeling design thinking, TAMP captures how knowledge about and skills in programming transfer to solving problems. Design tasks encompass the "Whole Game" of programming, a phrase Perkins (2010) coined to describe one of his principles of teaching. Playing the whole game gets to the heart of using knowledge to find and solve problems. Perkins believed

his schoolwork in math and artificial intelligence transformed him into a strong problem solver but left him undertrained as a *problem finder*.

> Problem finding concerns figuring out what the problems are in the first place. It also involves coming to good formulations of problems that make them approachable. Often it also involves redefining a problem halfway through trying to solve it, out of the suspicion that one may not be working on quite the right problem. (p. 26)

Perkins' description of problem finding seems similar to the early stages of programming – clearly defining the problem, deriving promising solutions, and iterating when they prove unsatisfactory. The computing education literature from Chapter 2 seems to reflect the dichotomy Perkins described during his time as a student. Students know a lot *about* programming, but too many remain unable to apply that knowledge to solving complex problems. TAMP seeks to connect the skills we teach knowledge to the ways that experts seem to apply that knowledge.

7.2.2 A look back at theory building

Theorists often write at length about their ideas, yet in a way that was far from methodical, and at times, feels quite underdefined. Piaget, Vygotsky, and Bruner primarily employed rhetorical arguments that compared their observations with contemporaries (often each other) while building their theories. They laced their discussions with loosely described empirical studies, yet often their new propositions remained ambiguous without explicitly defining the underlying constructs. Piaget never seemed to pinpoint when the mind moves from assimilating to accommodating. Vygotsky neglected to provide measures for the zone of proximal development. Bruner often described but provided few examples of iconic representations. Despite the disconnect between a proposition and its measurable observation their work has proved significant in educational research. Piaget, Vygotsky, and Bruner may not have seen their writings as defining new theories, but since that is the aim of this dissertation, it would be at best shortsighted and, at worst hypocritical to ignore my critiques of their approach. Before defining TAMP, it seems useful to revisit the theoretical building blocks discussed in Chapter 3.

The methodology of Chapter 3 does not set out to 'prove' a theory so much as document its foundation, assumptions, arguments, and content. Having reached this point in my argument, you might agree that theory building often leverages intuitive leaps followed by reasoned argument.

Formal theory (as opposed to the substantive theory from Section 3.1) rarely forms solely out of data since its very nature requires defining abstractions that theorists capture from the generalizations they see in data. The same cognitive models from dual process theory I am applying to programmers seem to apply to theorists! The process of theory-building should be transparent and the outcomes actionable. The resulting theory should explain the relationships between what we measure and things we cannot directly observe. The process defined in Chapter 3 promotes those connections by establishing the building blocks of a well-defined theory.

- Concepts – related to perceptions
- Constructs – abstractions build upon concepts
- Propositions – formal assertions grounded in constructs

Researchers and practitioners can use theory more consistently and effectively when propositions utilize well-defined constructs that tie to established concepts.

Table 7.1. Theoretical propositions from the major theories

Theory	Key Proposition(s)
Piaget	Children progress through four stages of cognitive development that measure their increasing ability to work with abstractions. Learning occurs through the process of assimilation and accommodation
Vygotsky	Reasoning matures through the internalization of inner speech. Assessments tend to measure only the actual level of development, but a more knowledgeable other can measure the extent of the ZPD explaining why some people progress faster than others despite showing equal abilities
Bruner	Experience forms enactive representations which may develop into iconic representation through conscious generalizations Symbolic representations provide external direction for redefining experience based on shared knowledge People can learn from symbolic representations, but without experience, such learning risks hampering problem-solving abilities
Dual Process Theory	Cognition includes two mechanisms of reasoning that support each other in thinking and learning System 1 is fast to act, takes little effort, and is slow to learn System 2 encompasses conscious deliberations, is effortful, and primary handles novel tasks

In many ways, these three building blocks of theory mirror Bruner's representations. Concepts (and enactive representations) model things we directly observe. Constructs (and iconic representations) combine observations into generalizations, perhaps with narrow applicability. Propositions (and symbolic representations) share knowledge with others. Bruner suggested that people are best at applying knowledge when they integrate all three types of representations. In the same sense, a theory is most effective when its users can align theoretical propositions with their underlying construct and concepts, as Silver (1983) suggested. It seems incumbent on the theorist to explicitly elucidate their propositions with these key building blocks. The foundation of TAMP identifies, occasionally refines, and then leverages time-tested building blocks from existing theories. Classical theorists may not have explicitly described concepts, constructs, and propositions, but they are often identifiable.

Table 7.1 captures my summary of key propositions from each theory as they relate to TAMP as a summary of Chapters 4 and 6. Table 7.2 highlights important constructs from the major theories underlying TAMP. TAMP benefits from the decades of research into these constructs and their underlying concepts. Using familiar concepts also provides a bridge into existing computational education literature grounded in Piaget and Vygotsky's theories.

Table 7.2. Identifiable constructs from the existing theories

Theory	Key Constructs
Piaget	Operational Thinking, Schema, Décalages
Vygotsky	Inner Speech, Zone of Proximal Development
Bruner	Enactive, Iconic, and Symbolic Representation
Dual Process Theory	System 1 and System 2

Accepting a theory does not require agreement on every proposition, so much as understanding the theorist's viewpoint and when their underlying constructs apply. Piaget, Vygotsky, and Bruner agreed on and utilized may of each other's propositions. Their disagreements on a few propositions may even have stemmed more from how they defined concepts. For example, Piaget believed the increase in *operational thinking* (construct) leads to the fading of egocentric speech marking the start of socialized speech with others. Vygotsky thought egocentric speech faded into *inner speech* (construct) because of socialization with others. Piaget and Vygotsky never reconciled the nature of egocentric speech (concept), leading to

opposing propositions on the role of language in cognition. The separation in distance and language between the two, not to mentions Vygotsky's untimely death, meant their writings never overlapped, and they never reconciled potential differences. Perhaps had they had time to reconcile the cognition behind speech (the concepts tied to these constructs), they may have reconciled other differences in their theories.

Modern theorists might reconcile some conflicts between theories by redressing the underlying constructs and concepts. Bruner, having studied both Piaget and Vygotsky, suggested that speech and thought are not always the same. We can now reconsider egocentric speech in light of dual process theory. People may mutter aloud without being aware they are doing so because System 1 automates a good portion of our language abilities. In many ways, operational thinking and inner speech describe the same mental processes (System 2). However, the competing interpretations of the observable concept (egocentric speech) as being the same construct (System 1 versus System 2 'speaking') led to seemingly conflicting propositions. Unfortunately, without clear definitions of the concepts and constructs, we are left to speculate at the differences in Piaget's and Vygotsky's observations. In building TAMP, I intend to provide greater transparency to the construction of my propositions. Over time, few, or none of the propositions I make in this first iteration of TAMP may survive. Perhaps, though, the constructs and related concepts will assist in the development of better computing education theory.

7.3 Neuroscience, theory, and struggling programmers

Neuroscientists and cognitive scientists start from different places to investigate how we think. Neuroscientists provide an increasingly refined look at the brain's mechanics but primarily study simple behaviors in participants (often animals). Simple tasks like memorizing lists of data, however, are too removed from complex decision making to provide grander insights into creative thinking. Cognitive scientists study more complex behaviors like decision making, but their interpretations are sometimes inconsistent with the brain's mechanics, like Piaget's assertion that formal operational thinking, once acquired, is a level of development that can transcend any mental activity. TAMP has the advantage of using insights from both fields to triangulate its new theoretical constructs.

This section starts with the presumption that while our brain is complex and multipart, biology probably shares common mechanisms of thinking and learning. For example, a different region of the brain may be dominant when learning to read than when learning to count, but the mechanics of learning remain consistent across content. Neuroscientists describe two distinct types of memory, declarative and nondeclarative, that mirror the descriptions of System 2 and System 1. They further break down declarative memories into semantic (facts) and episodic (recalling events). Bridging a few gaps between learning theory and the underlying neuroscience may explain why students are both promising learners yet fail to apply their learning to complex tasks like programming. This section looks to connect Bruner's representations to the brain's memory structures and, as a result connect his constructs to underlying concepts from neuroscience.

7.3.1 Declarative and nondeclarative memory

Bruner, dual process theory, and neuroscience each distinguish knowledge into at least two different categories. Bruner separated how we implicitly learn through action (enactive) from that which we retain through conscious effort (iconic, and with a few caveats, symbolic). Dual process theory similarly separates knowledge into implicit action within System 1 and explicit facts within System 2. Neuroscience also seems to confirm this divide at the lowest level within declarative and nondeclarative memory systems.

> What we now call nondeclarative memory includes a large family of different memory abilities sharing one feature in particular. In each case, memory is reflected in performance – how we do something. This kind of memory includes various motor and perceptual skills, habits, and emotional learning, as well as elementary reflexive forms of learning such as habituation, sensitization and classical and operant conditioning. Thus, nondeclarative memory typically involves knowledge that is reflexive rather than reflective in nature. (Squire & Kandel, 2003, p. 24)

Squire and Kandel's description of nondeclarative memory seems to align the functioning of System 1 and enactive representations. They even mentioned Bruner's influence on framing the difference in declarative and nondeclarative memories.

> Some years later, the psychologist Jerome Bruner, one of the fathers of the cognitive revolution, called '*knowing how*' a memory without record. (p. 14, emphasis added)

Squire and Kandel's invocation of Bruner helps to establish a link between the phenomena modeled in nondeclarative memories, System 1, and enactive representations. Nondeclarative memories allow us to pedal away even after years without riding a bike. They prompt the carpenter to square a deck by measuring 3, 4, and 5 feet around a triangle, without needing to recall the Pythagorean Theorem. Our conscious mind neither has nor needs awareness of the information stored within nondeclarative memories to take advantage of trained behaviors. It seems that Bruner noted the implicit behaviors that neuroscientists eventually cataloged into the mechanics of tacit memories, and dual process theory operationalized into a model of cognition.

Our conscious reasoning relies on declarative memories to recall facts, details, and stories. Squire and Kandel continued to invoke Bruner in describing declarative memory, which "he called '*knowing that*' a memory with record" (p. 15, emphasis added). Declarative memories form comparatively quickly, sometimes instantly, yet may not last long unless used frequently. Declarative memory is not the same as short-term memory. Short-term memories last a few seconds or possibly a few minutes if the thinker maintains concentration. The brain retains declarative memories for days, weeks, months or possibly longer so long as they remain in use. Declarative memories, however, will fade in detail or entirely if left unused. Students who seem to thrive in one class only to forget what they learned by the next class were likely successfully cramming knowledge into declarative memory only to see it fade when unused. System 2 excels at incorporating new information (which some are better at than others). Our brain will forget information that goes unused in both types of memory, but declarative especially.

The mechanics of memory suggest that learners must repeatedly practice over time if they wish to retain knowledge, but such practice must also account for both *knowing how* and *knowing that*. Memorizing facts and processes (*knowing that*) does not always transfer to *knowing how*. At the same time, if instructors only provide activities that emphasize skills (nondeclarative memories) students may learn *how* does not translate to conscious *knowing that*.

> Moreover, even though you successfully perform the tasks encoded in nondeclarative memory, the encoded information will not enter your consciousness. Once stored in nondeclarative memory, *this* unconscious never becomes conscious. (Squire & Kandel, 2003, p. 24 Emphasis theirs)

Our brains can perform complex mental activities without ever learning explicit rules (Section 4.2.2). Young children learn to speak grammatically correct (if not perfectly) long before their

first (or any) formal English lesson. Even when a teacher starts by teaching rules, declarative memories of a rule may fade as nondeclarative memories form and take over processing the rule. Squire and Kandel essentially are saying that as *knowing how* emerges during formal education, *knowing that* may fade without reinforcement.

Programming students typically acquire the rules of syntax and semantics first to support early efforts to trace or write code. Squire and Kandel imply they may forget the details of these rules as their mental model of the notional machine matures[35]. For example, programming students may know how to prevent compiler errors by adding exception handling logic, yet forget the concepts involved in exceptions. I have seen students do exactly this to make their code compile, inadvertently suppress vital failures in their code, and wonder why its logic does not work as expected. These students procedurally add error handling logic because the compiler requires it, but do not consider its ramifications, despite dedicated instruction on the handling of exceptions. They seem to fail to integrate information about exceptions with the procedure of handling errors. Instructors should plan their curriculum to revisit activities that encourage both declarative and nondeclarative memories if they want students who both *know how* and *know that*.

7.3.1.1 *Knowing how* and *knowing that* in computing education

Educational activities in computing often focus on either *knowing how* or *knowing that*. For instance, studies into tracing (as described in Section 5.2.2) investigated *knowing how* a programming language works. To measure *knowing that* educators use "explain code in plain English" tasks (Bayman & Mayer, 1983; Lister et al., 2009; Lopez et al., 2008; Whalley et al., 2006). Where tracing requires the mental execution of code, a programmer can do this implicitly without considering the code's purpose. Explaining purpose typically requires reflection on how the various language constructs, algorithms, and data structures integrate to solve a problem (at least if the code does not fall into a familiar pattern). The two activities seem similar but may not leverage the same types of memories to complete. Lister, Fidge, and Teague (2009) investigated the relationship between a student's competence in tracing, explaining, and writing code to see if

[35] Section 5.2.2 suggested that novices memorize rules to cope with early tracing tasks, and Section 7.4.2.2 provides an example of a case where many experts - including myself - might forget a rule that seems like a fair test question. The segmentation of memory further confirms Section 5.2.1's assertion that fragile knowledge may result from the many ways our minds capture the same information.

a hierarchy, and thus order, exists for developing these skills. They saw statistically significant relationships between students' abilities in each area, yet,

> Our data does not support the idea of a strict hierarchy; where the ability to trace iterative code would precede any ability to explain code, and where the development of both tracing and explaining would precede any ability to write code. (p. 164)

Lister et al. find that tracing, explaining, and writing code develop independently in students. Some could trace code, yet not explain it while others could explain code yet struggled to trace it (Lister et al. speculated that they were shrewd guessers). One student even wrote code more effectively than either tracing or explaining. A Cartesian viewpoint – that people must recall information about a subject orderly (i.e., all, or nothing) – makes their findings quite confusing. How can students explain things they cannot trace or vice versa if they use the same stores of knowledge? In Bruner's terms, many educators may conflate *knowing how* without *knowing that*. The division of *knowing* described in each of the three models – Bruner, dual process theory, and neuroscience – perfectly explains why students can independently function across these three tasks, but what does it take to build the required knowledge equally?

A group of computing education researchers (Margulieux, Catrambone, & Guzdial, 2013; Margulieux, Morrison, & Decker, 2019; Morrison, Decker, & Margulieux, 2016) have studied the cognitive load theory-based *subgoal labeling* within computing education. Subgoal labeling activities are created to explicitly identify the purpose of code and the choices made to solve the problem. While CLT recommends managing the load on short-term memory, subgoal labels seem to help by focusing attention on evaluating the design.

> Subgoal learning explicitly teaches students the subgoals, or functional pieces, of a problem-solving procedure. For example, to solve a problem with a loop, students must define and initialize variables, so defining and initializing variables is a subgoal of solving a problem with a loop. (Margulieux et al., 2019, p. 548)

When instructors add (or ask students to add) subgoals, student attention moves from syntax and execution to intent. Margulieux et al.'s subgoal example described the purpose of variables within a loop, which shifts the student's attention from the mechanics of the loop construct to the designer's choice to use a loop to solve the problem. Identifying subgoals forces the novice to tap into *knowing that* memories that may go unused while tracing. Subgoal labeling activities help

new programmers to attend to different aspects of the problem, thus exercise different types of memories that are equally important in programming.

Subgoal labeling seems to present a focused pedagogy that fits in well with the two types of knowledge each of the foundational theories of TAMP describe. Ironically, it seems that without tasks like subgoal labeling, instructors expect students to acquire knowledge about the purpose of algorithms *implicitly*; a type of knowledge that seems to be more useful in conscious (explicit) reasoning. Equally ironic, the *explicit* instruction on the rules of coding constructs does little to support the implicit (nondeclarative) knowledge required to trace code quickly and effectively. The diversity of pedagogical interventions in computing education hints at a tacit understanding that educators must build various types of knowledge in their students. Research like Lister et al.'s shows that these stores of information may not automatically lead to success in all aspects of programming. Bruner hinted, though, that the formation of iconic representations is critical towards problem-solving.

7.3.1.2 Declarative memory and iconic representations

A programmer designs and writes code to solve a problem, which Bruner suggests is easiest when they have a combination of *knowing how* and *knowing that*. Squire and Kandel's (2003) description of declarative memory suggests synergy with Bruner's iconic representations.

> Declarative memory is well adapted for forming conjunctions (or associations) between two arbitrarily different stimuli. (p. 99)

Iconic representations blend experiences (stimuli) while generalizing from multiple experiences (forming conjunctions). When designing, programmers need to reverse the process of explaining code, starting with a purpose, and then creating code. Iconic representations, rooted in declarative memory, offer the flexibility to restructure memories of the beneficial uses of coding constructs to fit the current need presented by a problem.

> declarative memory is designed to represent objects and events in the external world and the relationships between them. A key feature of declarative memory is that the resulting representation is flexible. (p. 99)

Bruner's iconic representations seem to model at least some aspect of declarative memories, given their flexible integration of various facts and experiences.

Squire and Kandel noted that some declarative memories form quickly (e.g., pairing a face and a name of a memorable person) where others require time and practice. Likewise, expert programmers often suggest solutions with little deliberation to some problems yet require time to consider others. Iconic representations provide a model for the fast and slow activation of memories. It is easier to solve familiar problems because our mind merely connects new information to past solutions (think Piaget's description of an expert's schema). When we encounter a novel problem, it takes longer to think of a reasonable solution because we must invent one.

Think about how much easier it is to remember copious details about your loved ones (e.g., their favorite color, birthday, last dental appointment) than it is to attach a name to a new face. The mind must pair two new and unrelated facts, as opposed to adding one new fact to well-learned information. The more we use the information, the stronger the neural pathways become, and the same information activates in many ways. A new face and a new name begin as unrelated symbolic representations if they make it beyond short-term memory. People use many tricks to help promote transitioning new information into memory, the simplest of which is repeating a person's name frequently. The same is likely true when a programmer encounters a problem unlike any they have previously seen. They can derive a promising solution, but it will require focus and attention. Experts learn – and innovate – faster since they can connect bits of new information to their past exploits rather than being forced to create and connect all new memories.

Iconic representations are more than mere repositories of facts available for problem-solving. Problem-solvers need to organize facts in some meaningful way. Programmers imagine designs by organizing facts about the problem domain within the capabilities of the programming language. Programmers must then imagine how the combination of language constructs and databases and third-party code libraries will behave when executed – facts in action. Squire and Kandel distinguished two types of declarative memories that seem to relate to this distinction; semantic memories describe "memory for facts" where episodic memory capture "memory for particular times and places" (p. 106). These two types of memories describe very different experiences. For example, it is one thing to remember how someone describes a roller coaster ride; it is entirely different to remember riding on one. Both are conscious experiences, but episodic memories seem to contain an implicit construction that transcends our consciousness. Our mind

consciously recalls episodic memories, but as more than a series of facts. We will see in the next section they are constructed in interesting ways.

Semantic memory is a bit paradoxical to model within dual process theory. We use facts in our conscious deliberations yet acquire and retrieve many of these facts implicitly (i.e., we forget where we acquired them). Squire and Kandel hint that semantic knowledge may form without effort by System 2.

> Semantic knowledge is thought to accumulate in cortical storage sites simply as a consequence of experience (p. 106)

Semantic knowledge is only useful to our conscious deliberations (System 2), yet we acquire facts tacitly (System 1)? The same tacit acquisition seems to be true of symbolic representations. For example, toddlers learn to speak without consciously memorizing words. Language impacts how we think, perhaps not to the extent Vygotsky suggested, even when most language 'instruction' comes as informal corrections (e.g., "it's taken not tooken"). Vygotsky noted the importance of culture, as expressed in language, on our development, yet culture is rarely explicitly described. Similarly, Piaget's notion of the schema also seems to capture the implicit acquisition of facts. He reasoned that experts remember more because of their superior ability to restructure their schema logically, yet do experts consciously decide where best to 'store' a fact? Bruner's iconic representation provides a simpler model: people with a large base of experience (enactive) acquire facts (symbolic) easier because they are easier to associate with existing ideas (iconic). Bruner's set of representations may not help describe how we form memories, but how we recall them.

Episodic memories associate various facts with additional details about times and places to form stories. Squire and Kandel described research on the differences in brain activity when people recall episodic versus semantic memories. Episodic memories activate the regions that store semantic memories but also the medial temporal lobe (MTL). The MTL includes the hippocampus – spatial memories and is critical to forming long term memories – and the amygdala, which manages emotions. The hippocampus and amygdala connect and blend memories throughout the brain and span cognitive and 'non-cognitive' activity. Our brain does not catalog episodic memories in a special area of the brain, like shelves of video recordings. Rather the MTL serves a vital role in constructing episodic memories from the various regions where memories of facts, sensation, and emotions reside – literally constructing knowledge. Educators do not seem to consider a significant difference between semantic and episodic memories, but they may hold

the key to understanding problem-solving. Bruner's iconic representation seems to describe a similar if not the same phenomenon in describing the construction of knowledge.

7.3.2 Prospection: the expert's secret weapon?

Researchers have frequently referred to the *undefinable something* that experts take years to develop and is often chalked up to intuition (see Section 2.3.3). We may never capture exactly how every expert think but the theories of cognition and learning, as well as findings reported by neuroscientists, may help make shrewd educated guesses. Neuroscientists describe *prospection* as a model for how people plan `by manipulating episodic and semantic memories. Understanding the mechanics of prospection may provide insights into how programmers think when they plan their code while designing.

7.3.2.1 The differences in expert and novice thinking

We often marvel at an expert's uncanny ability to string together seemingly unconnected ideas to imagine creative solutions to pressing problems. It is not just that experts are faster and more accurate than the rest of us, but they seem to see deeper into problems and quickly leap to conclusions. Fix, Wiedenbeck, and Scholtz (1993) studied the different mental representations that novices and experts form when reading and analyzing code. They gave a group of 'novices' (just completed their first course) and 'experts' (with an average of 7 years of professional experience) fifteen minutes to review a small program (~130 lines of code) before taking the code back and asking questions about its structure. They predicted that the expert's superior ability to make mental representations would make them better at answering a variety of questions about the program.

> We propose that an expert's mental representation exhibits five abstract characteristics, which are generally absent in novice representations
> 1. It is hierarchical and multi-layered;
> 2. It contains explicit mappings between the layers;
> 3. It is founded on the recognition of basic patterns;
> 4. It is well connected internally;
> 5. It is well grounded in the program text. (p. 74)

Fix, Wiedenbeck, and Scholtz extracted these properties of expert memories from computing education literature. The questions they asked their participants specifically queried each of these areas, as described in Table 7.3.

Table 7.3 Expert and novice performance on various remembering tasks from (Wiedenbeck, Fix, & Scholtz, 1993)

Characteristic	Task	Novices %	Experts %
Hierarchical	Recall the names of procedures	41	61
	Recall the network of procedure calls	***49***	***86***
Explicit mapping	Goal of the entire program	79	88
	The goal of individual procedures	***40***	***74***
	Methods of individual procedures	***21***	***73***
Patterns	Explain simple code segments	93	100
	Explain complex code segments	***43***	***93***
Well connected	Recall important variables	56	69
	Recall the flow of data through variables	***20***	***51***
Well grounded	Physical location of code constructs	91	93
	Name the program units given an outline	***74***	***98***
	Name the procedure in which a variable occurs	***17***	***56***

Bold *shows a statistically significant difference*

Novice and experts performed roughly the same on tasks that only required memorization or basic analysis. Experts outstripped novices – rather dramatically at times – on tasks that required either additional knowledge or merely deeper analysis of the code. Remember, each participant answered the questions from memory. Experts not only acquired more information while reading the code, but they also formed robust mental representations of code they read (not executed) for *only* 15 minutes. Fix, Wiedenbeck, and Scholtz created a methodology with the power to dissect the mental representations of programmers at various levels, which TAMP can explain even further.

Of the five characteristics Fix, Wiedenbeck, and Scholtz proposed to differentiate novices and experts, the third that is probably the most telling. Pattern recognition in experts is driven by nondeclarative memory and System 1 (remember the discussion of Chess masters in Sections 2.3.3 and 4.2.1.1). TAMP suggests that experts are not consciously searching for patterns, but like chess masters, they spring to mind and reduce the demands on short-term memory. Experts formed richer iconic representations of the code's design because their enactive representations (built over many years) helped to spot patterns in the code. By contrast, most everything in the code was new

to the novices, and thus, they remembered less – primarily surface details. TAMP suggests patterns are not ***a*** characteristic, rather ***the*** characteristic that distinguishes expertise. Computing education literature has its own example of Chase and Simon's study (1973).

> Soloway and Ehrlich (1984) presented evidence that experienced programmers' comprehension was disrupted by programs that were written in an unplan-like way, thus supporting the idea that plan recognition must occur in comprehension. (Wiedenbeck et al., 1993, p. 75)

Fix et al. added to Soloway and Ehrlich's (1984) previous study that itself reinforced the similarities between chess masters and experienced programmers. When code follows familiar patterns, Soloway and Ehrlich's (1984) noted the experts performed better than when the code was nonconforming to coding standards. The experts in Fix et al.'s study similarly recalled significantly more information about the code on questions that utilized patterns (e.g., the structure of the code) but about the same as novices on unstructured or mere memorization tasks (e.g., the names or physical location of things in the code). Their study reinforces the connection between the way programmers think and the neuroscience that underlies dual process theory. If the mental mechanics of programming indeed align those described in neuroscience, perhaps neuroscience can also hint at why novices struggle when they have yet developed the nondeclarative memories that drive such tasks as pattern matching within code.

7.3.2.2 Understanding breakdowns in complex cognitive systems

One strategy that neuroscientists sometimes use to understand the areas of the brain that support specific mental activities (e.g., vision, language, emotions) is to compare uninjured people's behavior and abilities to those who have suffered injuries or sickness in different regions of their brain. If a person suddenly loses the ability to form new long-term memories, as we saw with Henry Molaison in Section 4.1, the damaged area of the brain likely plays some role in that mental activity. Seeking expert programmers that suffered brain lesions seems like a slow and rather sad way of conducting such a study. An analogous approach is using reports from neuroscience on how the loss of brain function impacts reasoning and compare with the deficiencies that novices seem to exhibit compared to experts.

Premise 1: Stories about individuals who lost knowledge or abilities due to brain lesions may indicate the functioning of that region of the brain.
Premise 2: If a region of the brain is untrained, a novice might exhibit a similar lack of knowledge or ability compared to an expert as to those who suffered lesions to that area of the brain compared to a person with an undamaged brain.
Conclusion: We can theorize the type of the missing knowledge an untrained or undertrained people lacks based on the types of struggles they exhibit.

This section explores this proposition by considering how the brain processes language and what happens when key language centers of the brain suffer damage or go untrained.

Exploring when our language abilities go wrong

Speaking, writing, or using signs to express language utilizes several areas of our brain beyond our conscious reasoning. Most language production is implicit, and as such, damage to one of the affected areas can have startling effects. For instance, when damage occurs in Broca's area of the brain, people understand but can no longer produce coherent language through speech, writing, or even sign language. For example, one such sufferer described the plot of Disney's Cinderella.

> Cinderella… poor… um 'dopted her…scrubbed floor, um, tidy…poor, um…'dopted…Si-sisters and mother… ball. Ball, prince um, shoe. (Eagleman & Downar, 2016, p. 340)

The speaker clearly knew the story of Cinderella but seemed unable to organize their words into proper sentences. The participants described the essence of the story, but with no sense of grammar. Broca's area aids in the transformation of ideas into proper sentences, but not comprehension so sufferers can understand others (except perhaps when the message includes complex grammar) and are aware of their own difficulties in communicating. Broca patients are often frustrated by their sudden impairment.

Another area of the brain also impacts language, but in a very different way. Patients suffering damage to Wernicke's area struggle to comprehend language. They cannot understand spoken or read, even if they are speaking, as demonstrated in Figure 7.2.

> **Examiner:** is your name Brown?
> **Patient:** Oh mistress triangland while listen you walking well things things this for for thee.

> **Examiner:** Okay, just say "yes" or "no." Okay, is your name Brown?
> **Patient:** What it is here, then let me see. I just don't know. No I'm not going to an eat sigh no.

Figure 7.2. An example of Wernicke's aphasia, from (Eagleman & Downar, 2016, p. 342)

Whereas Broca patient chooses appropriate, if ungrammatical, words, the Wernicke patient utters somewhat grammatical nonsense. Damage to Wernicke's area corrupts the meaning of language from any source (e.g., spoke, written, signed), including oneself. Unable to comprehend their own communications, they are unaware of their malady, if perhaps a bit confused at times that people do not understand their words.

The afflictions of Broca and Wernicke patients tell us much about how the brain segments memory and skill. The consumption and production of language occur in very separate regions of the brain also, counter to Vygotsky's theory, separated from the rest of our reasoning. The struggles of novice programmers may follow similar patterns. Some students write extensive code that has no hope of compiling (they are unaware they are producing syntactic nonsense). Other write compiled and even runnable code that bears little resembles the desired goal (it works but does not solve the problem). Still others are so aware of their struggles with syntax they produce nothing at all. Our brain compartmentalizes even seemingly related skills that may not function as desired until the entire network is working properly.

Since the brains of novice programmers are presumedly undertrained rather than damaged, it would be helpful to see the impact of the lack of education and experience. We can see the impact of an undertrained language center by revisiting the impact of the tragic isolation and under-stimulation that Genie (see Section 4.2.2.1) and other such 'feral children' suffered.

> The fact that Genie is able to understand all these questions shows a more developed cognitive ability than is found in children whose grammars are more highly developed, but whose cognitive age is below hers. (Curtiss et al., 1974, p. 541)

Like Broca's patients, Genie could understand questions, but could not reply in grammatical statements. It seems without the years of practicing speech with others, her ability to speak mirrored that of those with brain lesions in that critical area. Comparing Genie to younger children who received the interactions she was so cruelly denied shows they were able to use grammar better even when they were not as adept at other cognitive tasks. Genie's sad case shows that a

lack of training can have the same impacts that we see in people with brain damage, not to mention the importance of practice in some aspects of learning.

After years of patient training, Genie's language improved (Curtiss et al., 1974), but the delay in acquiring language only underscores the importance of System 1 in many critical tasks we may otherwise consider part of conscious reasoning. No amount of intellect, or patience, or determination can overcome some deficits within System 1. Experts (and for some basic tasks, everyone) do things faster and need to dedicate less effort and attention because of their rich store of enactive representations. How can unconscious, nondeclarative memories help in simple tasks like remembering code or complex tasks like designing new code?

The various theoretical building blocks discussed so far hint at the role of tacit knowledge in complex cognition, yet individually may not provide a complete picture. Dual process theory describes the role of priming in providing System 2 with relevant memories to use in problem solving. Bruner's representations offer a model for how the mind organizes knowledge during problem-solving, but the mechanics of iconic representations remain somewhat abstract. One somewhat obscure theory from neuroscience, ***prospection***, may provide a model to describe the mechanics behind iconic representations and the various types of memory involved in planning.

7.3.2.3 Understanding prospection

Neuroscientists tell us our episodic memories are not only recreated each time we recall the past, but the same mechanism applies when we think of future events. When reminiscing, our mind assembles facts, places, and, most importantly, the timeline of the events into a story of the past. Tulving (2005) proposed that the same mechanics apply to imagine potential futures. Prospection, or "remembering the future" (Eagleman & Downar, 2016, p. 282), informs our plans based on our past experiences. Semantic and episodic memories, being declarative, share many traits: quickly adding new information, consciously recall information and conscious reasoning but,

> It is important to note that neither semantic nor episodic memory as defined here depends on language or any other symbol system for its operations, although both systems in humans can greatly benefit from language. (Tulving, 2005, p. 12)

Tulving supports Vygotsky's assertion that language improves how our brain operates but perhaps not as Vygotsky intended. We do not 'need' language to form memories, but it helps in revisiting them. Our mind uses language as a conduit for sharing information, but not as 'inner speech' to mediate our memory. Bruner's separation of language (symbolic) and action (semantic) offers a better model for describing expertise. In programming, we can see the same disconnect between language and action in teaching students to trace, explain, and write code. Tracing requires little or no planning (prospection), as the programmer can navigate the code one line at a time. Explaining requires enough prospection to connect the translate the semantic meaning of the code's language constructs with imagined execution (or in making sense of observed executions of the code). Designing/writing code requires not just a connection between purpose and the appropriate language constructs, but the mental simulation of the constructs in action to propose/predict the desired effect. It seems a deficiency in any area of programming knowledge results in an inability to produce novel solutions in code.

Benedict Du Boulay (1986) proposed a division between syntax and execution in learning programming languages quite some time ago, as discussed in Chapter 2. He listed developing a notional machine separate from learning the syntax and semantics of a language, saying, "The semantics may be viewed as an elaboration of the properties and behavior of the notional machine" (p. 57). Du Boulay seems to agree that enactive representations support the formation of symbolic representations better than the other way around. His 1986 article is a bit of a departure from the introduction of the notional machine five years prior, where he suggested that lecturing about the language was a way to initiate the notional machine. Learning syntax and semantics alone do not form the notional machine, rather they grow in parallel, perhaps supporting each other. Building a notional machine is only part of becoming a programmer.

If you remember the discussion from Section 2.1.2, Du Boulay's list did not stop with the notional machine; he also added the need to learn patterns and pragmatics. Pragmatics consisted of the tools and processes of writing code, but more important to this discussion is the idea of "standard structures, cliches or plans that can be used to achieve small-scale goals" (p. 58). Episodic memories do not merely organize facts but include a structure to the story. When the story has a familiar structure – follows a familiar pattern – then it is easier to remember as we saw with chess masters and expert programmers. Patterns seem to be echoes of past code projected

into future designs, and if the mechanics of prospection apply to design as to planning, form the foundation of problem-solving.

An example of prospection gone awry

To understand the value of prospection, it is helpful to understand what it looks like when it is lost. Tulving (2005) told the story of a young man, K.C., who received a traumatic head injury in a motorcycle accident. K.C.'s cognition was generally unimpaired except for his episodic memories.

> His thinking is clear, his intelligence is normal; language is normal, he can read and write; he has no problem recognizing objects and naming them; his imagery is normal (he can close his eyes and give an accurate visual description of the CN Tower, Toronto's famous landmark); his knowledge of mathematics, history, geography, and other school subjects is about the same as that of others at his educational level; he can define and tell the difference between stalagmites and stalactites; he knows that 007 and James Bond are one and the same person; he can play the organ, chess, and various card games; his social manners are exemplary; and he possesses a quiet sense of humor. (p. 23)

K.C.'s accident left him unable to form episodic memories. He could tell you facts about his past (semantic memories) – when he was born or recognize the house he grew up in – but could not tell a single story (episodic memory) that occurred within that home. The damage only seemed to impact his episodic recall until realizing he could no longer use facts and general reasoning to make plans.

> When he is asked to describe the state of his mind when he thinks about his future, whether the next 15 minutes or the next year, he again says that it is "blank." Indeed, when asked to compare the two kinds of blankness, one of the past and the other of the future, he says that they are "the same kind of blankness" (p. 26)

Trying to remember past or plan future events evoked the same feeling for K.C. He could function normally except for any task requiring forethought.

> He probably also could, if necessary, walk to the supermarket, and (if he has written down what he needs, if he has not forgotten that he has the list in his pocket, and if he has not forgotten to take money with him), he could fill the basket and walk back home. (p. 29)

K.C. could function in the moment and conduct procedural tasks but could not place himself in a future scenario and plan accordingly. Prospection relies not just on facts, but on ordering and traversing facts in a useful manner.

Like Henry Molaison, K.C. suffered damage to his hippocampal region. Looking at other patients, a group of researchers systematically tested how hippocampal damage impacts the brain's ability to imagine new experiences (Hassabis, Kumaran, Vann, & Maguire, 2007). They compared five amnesiac patients with ten control subjects asking them to describe in rich details exotic yet familiar locations, such as a sandy beach or busy museum. Four of the five amnesiac participants[36] described their imagined locales with little detail beyond semantic cues.

> As for seeing I can't really, apart from just sky. I can hear the sound of seagulls and the sea… um… I can feel the grains of sand between my fingers… um… I can hear on of the ship's hooters (p. 1727)

The content that this amnesiac participant includes is like that of the control group but without traversing the imagined world. Both groups describe the sights and sounds from a beach, but the example from the control group actively navigates through those experiences. Hassabis et al. confirmed that the amnesiac participant's deficiency was not due to any short-term memory loss; they did not forget the task as is common in hippocampal lesions. They remained focused without reminders of their goal. The example above shows that they seem to remember relevant details but could not construct a meaningful story[37].

Tulving (2005) makes it clear that episodic memories (in people and the animal kingdom) are not essential to survival, and creatures can even thrive using semantic memories alone.

> Semantic memory also allows an individual to construct possible future worlds, but since it is lacking autonoetic[38] capability, it would not allow the individual to mentally travel into his own personal future. (p. 19)

[36] The fifth patient scored well on the exercise, seemingly because of the comparatively limited damage to their hippocampus. This individual still retained the ability to produce some semantic memories and thus seemed to retain the ability to imagine rich experiences.

[37] Remember this story when considering the novice programmers in Section 8.2 who write code with no meaningful purpose or overall design. They have semantic memories about coding but struggle to connect these facts to produce a solution to a seemingly familiar problem.

[38] Autonoetic at a minimum refers to a person's awareness of their existence within time. Tulving (2005) said that only humans are autonoetic and thus can make rich future plans.

K.C. did not require episodic memories to go about his daily tasks, perform mathematical operations, or even create strategies in chess or card games. What he and other people with similarly afflicted brains cannot do is create or *visit* imaginary worlds.

> There is no need whatsoever for any temporal marker to be attached to and retained about the learned facts of the world. The facts can be put to good use by the owner of a semantic memory system regardless of whether episodic memory is functional or not. (p. 19)

People use facts when considering the here and now, but the facts alone are insufficient for planning. The amnesiacs in Hassabis et al.'s (2007) study started to form an imagined world but struggled to connect the facts across time. It seems that rational thinking alone is not enough for planning. Performing an immediate mental task is different from planning for the same task in the future – ironically, the heart of a programmer's work, automating future tasks. If people like K.C. can function 'normally' in the present, what do they lose after damage to the hippocampus?

The hippocampus, prospection, and iconic representations

Animal testing can help shed light on the role of the hippocampus in learning. Squire (1984) researched animals' ability to transfer knowledge between tasks.

> Subsequent transfer tests then demonstrate that normal animals and the animals with damage to the hippocampal system have acquired different kinds of knowledge. The normal animals have acquired a flexible representation that can be expressed in new ways. (p. 237)

Animals who acquire both declarative and nondeclarative memories become more flexible problem-solvers than those whose damaged hippocampus prevents the formation of declarative memories. Squire noted that animals form nondeclarative memories slower when they cannot form declarative memories. System 2 enhances the development of System 1, presumedly through conscious feedback based on quickly learned corrections[39]. Squire's work with animals reinforces the importance of combining Bruner's representations with dual process theory. Bruner modeled this 'missing link' as iconic representations that blend symbolic (semantic) and enactive (nondeclarative) knowledge. To answer the question posed at the end of the last paragraph – what

[39] As an example, I might learn to throw a frisbee with enough trial and error but remembering the advice my coach gave me speeds up the corrections I need to make in my body positioning and movements.

is lost when the hippocampus is damaged may be the ability to navigate iconic representations through imaginary scenarios.

Prospection may relate to how programmers design software. Prospection's role in design is compelling but entirely theoretical at this point. I find prospection useful and promising for two reasons. First, stories like K.C.'s seem very similar to the struggles of advanced novices. K.C displayed procedural abilities equal to most people yet struggled to organize these skills in preparing for future events. Some novice programmers show promise in all areas of learning to program up to the point where they must write original code. Chapter 8 will introduce examples where some students perform well when properly supported, while others seem to crumble without support. Second, even if the model is merely analogous (does not use the precise regions of the hippocampus), the mechanics of prospection may still apply to how another region of the brain assembles and navigates knowledge for creative planning. TAMP does not seek or need an exact map of the brain regions involved in programming, so long as the theoretical constructs align with the mechanics of the brain's functioning. If TAMP captures the general way expert thinks and the mechanics that describe such thinking, it may be possible to pair the 'symptoms' of novice struggles with the types of mental representations that they need to develop and the types of activity that would be the most beneficial.

7.4 Contextualizing Bruner's representations within TAMP

Bruner's representations and dual process theory describe the mind at different levels of abstraction but seem to agree on the underlying mechanics. Enactive representations model the memories used by implicit and automatic System 1. System 2 calls upon iconic representations that blend reflections on experiences and integrate with formal learning. Symbolic representations are enigmatic since they reside within each System at different points in time given practice. A person first encounters *symbolic knowledge* from an authoritative source outside their personal experience. Symbolic knowledge is the information that comes from external sources (e.g., books, lectures, etc.) that forms the basis of symbolic representations. Each person acquires and retains symbolic knowledge quite differently from the next. Bruner and Vygotsky noted that people acquire symbolic systems more quickly when they already understand the concepts. Section 7.3.1 noted that declarative memories fade when replaced with nondeclarative skills if the facts are no

longer useful. While skill alone may be sufficient for some tasks (e.g., tracing), the model of prospection as a guide for design thinking suggests that retaining some declarative memory may be essential to planning. Bruner suggested that his iconic representations modeled such mental problem-solving, yet his distinctions between iconic and symbolic representations are not always clear.

This section looks to provide refined and perhaps enhanced definitions of Bruner's representations grounded in dual process theory and relevant findings from neuroscience. At this point, it may be fair to classify TAMP as a neo-Brunerian theory. I intend to align with the spirit and description of Bruner's representations rather than imagine new constructs. I prefer to reuse – repurpose, refine, enhance, or whatever word seems most apt – as Bruner's constructs provide a strong body of literature from which TAMP can expound. If my ideas prove incompatible with Bruner's (but valid), future authors should assign new names, but I prefer to reconcile nuances and discrepancies rather than avoid them for now. Table 7.4 offers a summary of how this section revisits Bruner's representations and ties each to concepts from neuroscience and dual process theory.

Table 7.4. Overview of how TAMP revisits Bruner's representations

Representation	TAMP 'differences'
Enactive	TAMP changes nothing from Bruner's original definition of enactive representations. It reinforces Bruner's description by tying enactive representations to the neuroscience of nondeclarative memories and the characteristics of System 1.
Iconic	TAMP seeks to clarify 'iconic as imagery' to reinforce the important role iconic representations play in learning and problem-solving. Bruner hinted at but never clarified why iconic representations are essential. TAMP ties Bruner's main role for iconic representations – the intermediary between other representations – to the neuroscience concepts of episodic memory and prospection. Prospection, the brain's ability to manipulate memories to plan for the future, uses the same mechanism as recollection, only to plan rather than remember past events. Iconic representations within TAMP still serve the roles Bruner described. However, by adding prospection, researchers might discover new ways of measuring different types of learning and how they contribute to problem-solving.
Symbolic	TAMP directly tackles the dual nature of symbolic representations: 1) understanding a new system of symbols and 2) acquiring the knowledge those symbols offer. TAMP distinguishes the liminal stages of acquiring mastery over a system of symbols. At first, a person may work to understand a new system of symbols by forming symbolic representations (i.e., facts about syntax and grammar). Given time, this knowledge becomes implicit within enactive representation, and working within a system of symbols becomes automatic (which is the primary pathway by which children acquire language). To truly master a system of symbols requires a person to develop enactive representations for reading and some tasks producing symbols. Bruner's notion of symbolic representations are essential since they model 'external' information that people socially acquire. What remains beyond a system of symbols is a set of facts that seem to align with the concept of semantic memories from neuroscience. TAMP uses Bruner's split between iconic and symbolic representations to provide a model of cognition and learning that encapsulates neuroscience's division between semantic and episodic memories. Adding the mechanics of memory reinforces and helps to explain the transient nature of knowledge between representations.

7.4.1 TAMP and Bruner's representations

This section revisits Bruner's representations to refine their meaning and scope using dual process theory and insights from neuroscience. The following three propositions, in logic book

form, encapsulate the expanded definition of enactive, iconic, and symbolic as well as providing an advanced organizer for the rest of this section.

Enactive Representations
Premise 1: The descriptions of enactive representations, System 1 processes, and nondeclarative memory all match at a level of abstraction useful for TAMP *Premise 2*: Enactive representations describe specific content (e.g., a habit, skill, perception) stored in nondeclarative memory *Premise 3*: Enactive representations are the memories formed in and used by System 1 ***Conclusion***: Enactive representations, System 1, and nondeclarative memory are all synonymous within TAMP

Symbolic Representations
Premise 1: Symbolic representation model data with 'truth' from some other source and sometimes a new set of symbols with which to share knowledge. *Premise 2*: Symbolic systems contain knowledge that eventually fits into both semantic (e.g., $\pi = 3.1459$) and nondeclarative memories (e.g., basic addition). *Premise 3*: Symbolic representations only contain 'static' knowledge, such as facts or the steps in a procedure. ***Conclusion***: Acquiring symbolic representations includes nondeclarative memories (e.g., grammar, processing of symbols), and semantic memories. Complete semantic representations (i.e., masterful learning) spans both System 1 and System 2.

Iconic Representations
Premise 1: Bruner described the conscious uses of iconic representations thus they model declarative memories *Premise 2*: Iconic representations support other types of learning: generalizing experiences and integrating experience with facts *Premise 3*: Since iconic representations blend experiences and symbolic facts, they model episodic or some equivalent structure *Premise 4*: Bruner places iconic representations as key to problem-solving, requiring the mental manipulation of concepts in novel ways ***Conclusion***: Iconic representations model the mechanisms of recall described in prospection, making them essential for many types of problem-solving.

7.4.2 The duality of symbolic representations

Chapter 4 suggested that symbolic representations are a special type of learning rather than a third mechanism of cognition missed in dual process theory. On the one hand, distilling symbolic

representations into only System 2 processes would provide a clean mapping between Bruner and dual process theory: System 1 = enactive and System 2 = iconic + symbolic. On the other hand, eliminating symbolic representations from our lexicon overlooks a crucial distinction between semantic and episodic memories. TAMP proposes a few refinements to symbolic representations based on the transient nature of learning and the duality of our memories. The first refinement is epistemological.

7.4.2.1 *Knowing how* to use symbols and *knowing that* they mean something important

TAMP models the 'fluency' aspect of symbolic representations as transitioning of knowledge from symbolic to enactive representations. The first time a novice sees source code, they likely do not comprehend its meaning or possibly get the wrong idea what it means (based on what the words mean in English, not the programming language). By recalling instruction and interrogating reference materials, they reason out the meaning of constructs much the same way we might decrypt a secret message with a simple cipher. The process relies on System 2 and growing declarative memories, and many students quickly develop some level of automaticity at the same time[40]. In short order, a novice ideally has enough practice to develop the intuitive feel for code that Boulay called the notional machine. By the end of their first year or two of coding, a programmer should develop strong enactive representations of code that unlock automaticity in reading, tracing, and even writing code. The important distinction in this process: *novices use different cognitive processes for the same programming tasks over time*. TAMP suggests not only why but how this transition might manifest itself in their performance.

The evidence of this transition appears regularly in anecdotal stories and empirical research. The simplest example is when a student answers conceptual questions well on the first exam only to forget the answers on the final exam. Their forgetting can partially be attributed to the nature of declarative memory (unused knowledge fades), but all the more confusing when the programming student forgets the semantics of a construct they use perfectly in their coding. Perkins and Martin (1985) likely observed a similar transition in some types of fragile knowledge.

[40] Even up to the point of writing this, I am struggling if it is more accurate to consider this mix of facts and action (the student is presumedly tracing, writing, or otherwise working with code in action) as an iconic representation, but I believe that keeping the name of symbolic for this stage helps to differentiate learning about the language and learning of design. The mental process is probably more fairly considered iconic, per the definition given in Section 7.4.3, but for convenience/clarity sake I think it is worth preserving the idea of a symbolic representation.

Their protocol had researchers start with a prompt that helped a good percentage of students with little more than context clues. When the researcher's hint instill action, it triggered a dormant enactive representation that had not recognized its relevance to that problem. Students who required a hint lacked enactive representations, but the hint activated the associated symbolic representations and allowed System 2 to tackle the problem. These examples provide early evidence not only of Bruner's representations in action but their connection to neuroscience principles and computing education.

If the automaticity aspects of symbolic representations (e.g., syntax, grammar) with practice transition to enactive representations, then symbolic representations are easier to model. TAMP simplifies the role of symbolic representations to simply containing facts (semantic memories). Bruner's model intertwined the processing of symbols with the facts those symbols represent. His enactive representations describe automatic processing, but like many before him, he does not distinguish between the conscious and automatic processing of language that Brocha and Wernicke patients so easily demonstrate. A loss in language processing is not a loss in reasoning or knowledge. A person with Brocha's aphasia is aware of and unable to correct their speech. Wernicke sufferers function normally outside the inability to express their ideas. Not understanding a new system of symbols (e.g., programming syntax) will hamper early learning, but as we see in programming, mastering basic coding skills alone does not make a full programmer. The interesting aspect of symbolic representations, as Bruner described them, is not a new notation so much as their externality. They are knowledge devoid of or in conflict with personal experience.

Imagine sitting down to lunch with your local priest at a café in Frombork, Poland. It is a warm summer afternoon in 1515 after a harsh winter that saw the Thames freeze and terrible flooding in Krakow. Over a glass of wine, your dining partner reveals that the Sun you are enjoying is not, in fact, moving across the sky. The Earth is instead rotating as it circles the Sun. You say, "Nicolaus, am I not to trust my own eyes?" He tells you that he has been observing the heavens, and his observations and calculations tell him it must be so! Father Copernicus is attempting to alter the way your enactive representations describe the world by providing you with the conclusions of his observations and deliberations.

Copernicus did not need a new system of symbols to construct or even to share his idea. The symbols of mathematics helped to derive his evidence, and conventional language is sufficient

to convey the idea of a rotating Earth. What makes the heliocentric view challenging is its dissonance with lived experience. Hundreds of years later, a small group of the population still debates the shape of the Earth and rejects symbolic knowledge over their own experience. Our decision making does not always give equal weight to facts and use them evenly in our deliberations (even when we trust them). A person may remember something but never put it to use in their decision making. For example, knowing the calorie count of your favorite take-out meal does not help unless you choose to order a healthier option. At the same time, certain types of knowledge may not influence decision making even when it should. Is it important that a programmer remembers the number of bytes assigned to various integer and floating-point data types? For the most part, my choice of data types is dictated by habit (and convenience) rather than a careful analysis of the memory and throughput needs. Like most questions of this nature, the answer depends on the circumstances. TAMP suggests that merely remembering such facts may not always improve a fledgling programmer's decision making. Symbolic representations are distinct from other types of knowing because they are 'external' and may not influence reasoning by simply being remembered via decontextualized memorization.

Piaget proposed that we learn through assimilation and accommodation, yet his differentiation between these processes was hazy at best. Assimilation occurs gradually as each new experience refines our mental model of the world, yet the confirmation bias (Kahneman, 2011; Nickerson, 1998) makes us equally likely to ignore outliers and prefer cases that support our perspective. Symbolic knowledge alone, even if remembered, may not change our thinking. Major shifts in reasoning require accommodation, an abrupt shift to our schema that occurs only when our internal model of the world no longer reflects our experience. Symbolic knowledge might offer a reason for our mind to enact such an abrupt shift, but only if the person dedicates System 2 to curbing System 1's automatic response. They must work through the dissonance that Piaget called disequilibrium when Systems 1 and 2 disagree. Assimilation and accommodation are apt descriptions of the learning process but do not account for a learner's "resistance" to new information. Reimagining Piaget's model with the other theoretical constructs within TAMP provides a clearer picture of this transition. Accommodation is most likely to occur when either the individual dedicates to changing their enactive representations to accommodate conflicting symbolic representations, or after enough examples that enactive representations begin to shift tacitly.

7.4.2.2 Transient states of knowledge in learning to program

If it were not enough that experts may possess 'more' and 'different kinds' of knowledge than novices, we now know they also store knowledge differently. The unexpected speed Wiedenbeck (1985) observed in experts comes from their store of enactive representations powering their System 1. Years of practice may also expand the breadth (i.e., transferability) of such skills, but semantic memories might fade if they are not relevant to daily activities. Thus, experts quite literally forget more about a subject than many novices might know. Early in my career I worked extensively on the Ada programming language, even writing papers analyzing and proposing extensions to the language (T. Lowe, 1999a, 1999b). When I first start to learn Java, I had a hard time learning its conventions (especially camel case), yet as I write this, I could not write a simple program without extensive research. I can remember arcane details (semantic memories) of Java because of my roles as an instructor and developer for two decades, but without seeing Ada code I cannot recall even the symbols that indicate a comment. The average practitioner – and even most exceptional ones – have little need for encyclopedic memories that mostly go unused, so our brain focuses on what we see the most.

The ideal state of *knowing* is the combination of *knowing that* and *knowing how* the problem-solving at hand requires. I believe that I, and likely other educators, remember arcane rules about programming languages more readily than others because teaching forces me to deal with rules of syntax and semantics. There are plenty of areas of programming that I have not taught and thus have less ability to explain than I do in performing. This section includes an example of a recent time where I 'forgot' useful programming knowledge for a bit, before considering how knowledge might transition between mental representations as novices learn.

A tricky test question

Programming offers an excellent example of the way knowledge shifts from declarative to nondeclarative memory, the split between symbolic and enactive representations. Section 5.2.2 suggested that inexperienced programmers rely on System 2 when tracing until enactive representations take over the mundane tasks of mental execution. Squire and Kandel (2003) remind us that a side effect of transitioning knowledge to nondeclarative memory is sometimes the fading of unpracticed details. Given that warning, answer the following question.

```
double area;
double volume;
double length = 2;
double PENTAGON = 6.88;
volume = 1 / 4 * PENTAGON * pow(length, 3);
printf("Volume: %.1lf\n", volume);
```

Which of the following is the first line of output generated by the program above?
A) Volume: 13.8
B) Volume: 13.7
C) Volume: 0.0
D) None of the above.

Figure 7.3. A tricky programming question that tests very specific semantic knowledge that is potentially at odds with System 1

The question in Figure 7.3 asks the student to predict the output of the code, which seems to be computing the volume of a pentagon. I encountered this question when a student in my Matlab class asked if I could help them understand why they missed this question in their C class. The answer key said the correct answer was C, which was not my intuitive answer. My attention immediately turned to the expression "`%.1lf`", which I first took as eleven-f, but is actually "1LF". Since Java and Matlab use `f` for float, not `lf` (long float) when printing out floating-point numbers, the unusual syntax caught my attention, but I knew the printf statement should only format and print the value, not force it to be zero[41]. I was a bit stumped. Since the printing is an unlikely culprit, I calculated the volume using the following steps:

```
volume = 1 / 4 * 6.88 * pow(2, 3);      Insert the values
volume = 1 / 4 * 6.88 * 8;              Compute the power*
volume = 6.88 * 2;                      Simplify the easy expression
volume = 13.76;                         Do the hard math
```

** I think I did the pow first because I do not use this regularly, so I had to attend to the unusual operation first and get it out of the way before I had to remember other computations*

The student and I agreed, for the moment, the correct answer should be A or B since the question seems to test knowledge about rounding, but the key said the answer was C. Did you catch my error in tracing the code?

[41] Notice, my System 1 was drawn to the syntax that did not match what I expected. In considering the rules of printing text System 2 guessed that this unfamiliar syntax was not the likely cause of the program printing zero, but this example seems even further proof of System 1's influence in how 'experts' read code.

The instructors designed a problem that requires perfect mastery of data types and the order of operations, which are at odds with mathematical intuition. My System 1 primed System 2 with the skills required for a math problem, so my lazy System 2 simplified the task by reducing the 8 and ¼ to 2. The problem could not be about rounding since the answer key insisted the correct answer was 0.0. My mind thrashed through possible explanations until the "ah-ha" moment struck. Unlike the poor students taking the exam, I had the advantage of knowing the answer and could activate the requisite (dusty) memories. The correct trace of the code, attending to order of precedence should have been

```
volume = 1 / 4 * 6.88 * pow(2, 3);[42]
volume = 0 * 6.88 * pow(2, 3);
```

The semantics of C places division and multiplication at equal precedence and processes the operations from left to right. The 1 and 4 are both integer literals[43], and thus when divided, the result of ¼ is *rounded to zero*. Did my 25+ years of programming knowledge suddenly become fragile?

The pentagon question is particularly tricky as it potentially triggers competing knowledge bases (math versus programming). System 1 offers quick answers to the math problem, made all the more seductive under the pressure of a timed test. A student would need to have operated like the computer, procedurally working through each operation, to have a chance. In many ways, advanced novices have a better shot at correctly answering this question than even experts. Attentive students may have picked up on a hint to study the semantic rules of operator precedence and integer literals in preparing for the test. Students who have yet to develop or still distrust their intuition may consider the problem more carefully. My knowledge was hardly fragile, just obscured by the context and presentation; within a minute, I activated the proper semantic memories (neuroscience) of the C language's semantics (computer science). Cartesian models of cognition (see Section 4.1) place the blame on students for questions that prey on the natural process of learning.

The pentagon question is not exactly unreasonable, but perhaps unfair through the lens of TAMP. The author of the question probably felt that giving away too much would make the

[42] I cannot remember if the function call would happen first, and I don't have a C compiler handy. Perhaps the power call would happen first, but the point is trivial, yet I do want to acknowledge a potential oversight for those who are quite detail oriented. You have a wonderfully trained System 1 or 2.
[43] A literal is a data values typed directly into code (e.g., 1, 3.14, "Hello")

question trivial that for an expert it probably would. I do not believe it would do so for most novices. This question is particularly unfair because of the similarity to computations in math. The problem even fooled me as an 'expert' who has regularly made similar mistakes. More importantly, what is the purpose of testing such specific knowledge? The question was on a 'paper-and-pencil' test, but given a compiler and runtime environment, the mistake would be instantly visible. It may take a few minutes to correct the mistake, but ironically the problem already violates a coding standard – avoid magic numbers (i.e., hard-coded numbers without context). An effort to create a testable idea created a problem that both tricks System 1 and probably does not test knowledge that is important to experts while designing code. Chapter 9 will consider other such pedagogical implications, but for now, it serves as a reminder that during the learning process, knowledge transitions between memory structures as TAMP describes using Bruner's representations.

Changing mental representation while learning basic coding skills

Bruner's representations help us to model the transient nature of programming knowledge as students mature. For example, a novice may follow a learning process similar to Figure 7.4, which annotates the different stages of mental representations.

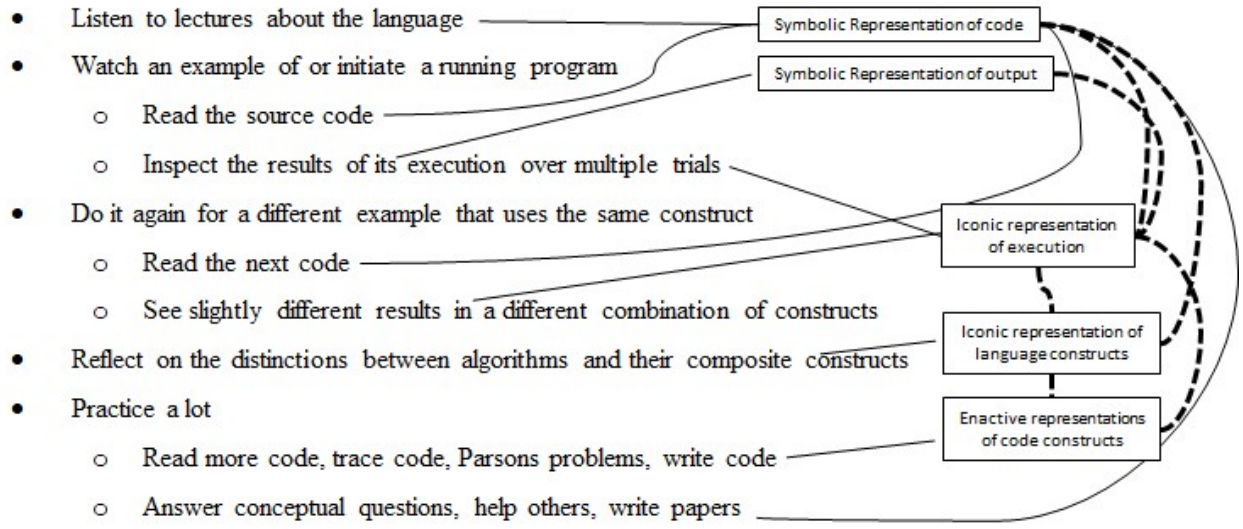

Figure 7.4. The mental representations created when learning a language

The hypothetical learning path in Figure 7.4 uses common pedagogical activities from Chapter 2 as a prototypical curriculum. Students completing each activity would likely form the annotated mental representation. The dotted lines show how different mental representations interact, for the most part, the 'higher' ones helping to form the 'lower' ones.

Traditional lectures, live or recorded, provide students with knowledge (e.g., the rules of syntax and semantics) that form *Symbolic representations of code*. Students retain these facts (e.g., operator precedence, how integers work) in semantic memory. Many instructors include worked examples portraying a computer simulating or an expert tracing the code. The tangible results of the execution form a new symbolic representation– log files, text on a screen, or for tracing, sketches whether ad hoc or using structures like tracing tables. As code executes, students must also acquire *Symbolic representations of output* to make sense of the results[44]. Just like a programming language has a syntax, the output of programs presents some set of symbols for describing the results of program execution. Unlike a programming language, the symbolic output of a programming language might change based on the author of the code, which means that connecting the results of the code with the language constructs can become a dynamic target if the code does not follow consistent standards for how it outputs information.

To understand the nuance of coding constructs, students must learn to distinguish the nuances within the results. In addition to learning the language, they must learn to parse the output presented from a program. When learning the piano, a student learns to associate the pressing of the proper key with hearing the desired sound. The feedback is instantaneous, particularly in familiar songs since our mind easily connects the motion of our fingers and the right or wrong note. Programming languages cannot provide such instantaneous feedback. Programming students must associate the code they wrote a while ago with a flood of random output. Blending the results of action with the original intent requires an additional memory structure, an *Iconic representation of execution*. If a student takes the time to reflect on the code and the execution, they can begin to

[44]. Many instructors use examples grounded domains they assume are familiar to students, hoping to minimize additional explanation. General familiarity with the domain may make it easier to decipher a new symbolic representation, but unless the output is some universal notation, the results are still symbolic. The possible exception to symbolic results is perhaps some intuitive behavior exhibited by graphics, robotics, or some other medium where code commands translate into familiar action. Even a seemingly intuitive simulation may still need further processing by the novice. Unless a program perfectly mimics a familiar experience, it risks breeding misconceptions when the simulation and real-world experience do not match. Instructors are safer in assuming that any execution results are as new and unfamiliar as the programming language itself.

see how a code construct influences the output. Passively watching another do this only engages System 2 so much in evaluating these connections, though, so instructors use tracing problems to ensure active participation.

A student needs more than a single example to illustrate the full extent of any topic, particularly a robust and nuanced programming construct. For example, novices probably need to see ***if*** statements with and without an ***else*** branch to begin to appreciate the full construct, and even then, are unlikely to grasp its full potential. Instructors create various examples designed to highlight aspects of a language construct[45]. Working through multiple examples, students form an *Iconic representation of language constructs* by considering the different ways each algorithm includes the language constructs. For example, one algorithm uses a `for` loop to search for a value in an array, the next algorithm uses the `for` loop to sort values, and a third sorts the values in reverse order. To compare these various uses of the `for` loop, the student must form an iconic representation of each variation of the `for` loop in each algorithm. Active comparison can yield awareness of variances in how the construct works, yet this is not the only way we learn (remember the sugar factor of Section 4.2.2.1). Just as often, students implicitly start *knowing how* the language works since System 1 forms *Enactive representations of code constructs* regardless if we consciously consider algorithms or not. It is at this stage that knowledge starts to divide into unequal enactive and symbolic representations.

Most students are not lazy, but like System 2, they follow the easiest path to the goal. System 2 demands quite a bit more energy than System 1, and evolution prefers to avoid unnecessary expenditure of resources. We do not run everywhere we go because it burns more calories than walking, so neither do we deploy our full brainpower unless the need is urgent. As soon as a student can trace and even write code using System 1, they are less likely to be reflective. When instructors assign students additional tracing practice once enactive representations form, they will likely acquire little additional information beyond the answers unless the student is notably disciplined. The more System 1 engages the less information about the process gets into consciousness. The student will only consider the semantic knowledge that System 1 has automated within an enactive representation if the student deliberately choices to. When

[45] The desire to highlight a specific feature often leads to example code that does not mean anything, but exercises the feature precisely as intended (see the xor example in Section 7.6.2.2). Such examples are helpful in the instant but may present further complications in transferring that knowledge to 'real' programs as discussed later.

instructors include tasks such as subgoal labeling, code reviews, peer programming, or tests of conceptual knowledge, students must reengage System 2 and focus on the other types of knowledge programmers need. Students who neglect metacognitive reflection while learning to code may develop strong tracing skills yet struggle to explain code or write code to solve unfamiliar problems.

Section 7.6 elaborates on the learning process started in Figure 7.4, but for now, it previews how the mental representations describe learning. A programmer needs activities that promote the development of enactive and symbolic representations. Lister, Fidge, and Teague (2009) investigated the relationships between tracing and explaining, as described here, but also writing code. Figure 7.4 demonstrates the path students take in *knowing how* to read, execute, and perhaps even write basic code. It also provides a path taken in *knowing that* an algorithm performs a specific functionality. I believe that writing original code beyond following a familiar example depends upon the third representation that had little attention here, iconic.

7.4.3 Refining the iconic representation

Bruner included the iconic representation as a bridge between formal and informal learning, which in my mind, bridges many of the gaps between the constructivist ways of knowing Piaget and Vygotsky captured. As discussed in Section 6.3.2.2, Bruner's language of 'iconic as imagery' sometimes overshadows his definition of iconic as an intermingling of interaction and experience that becomes critical in advanced problem-solving. The 'reduction' of iconic representations to mere pictures devalues not only the critical mental role iconic representations play but perhaps also the applicability of Bruner's constructs overall. Iconic representations seem to be at the heart of both how experts tackle complex tasks and how people become experts. Experts use iconic representations to assemble and manipulate new information with recalled facts as the mental workspace of problem-solving. Students need iconic representations as an intermediate workspace for remembering and using new ideas as they practice. While Bruner consistently described iconic representations across his works, he sadly did not expand on that description far enough to offer a rich construct with associated concepts. This section looks to refine iconic representations with additional details making the role and use of iconic representations with TAMP more tangible than 'personalized imagery'.

At this point, TAMP may be expanding the scope of or even redefining iconic representations. The hope is to stay within the spirit of Bruner's original descriptions while integrating concepts from neuroscience that seem to support the role of iconic representations as a critical linkage between various types of knowledge.

7.4.3.1 Iconic representations and 'experts'

Bruner described iconic representations as both a place where we integrate knowledge, but also the place where we use memory to solve problems. Bruner's detailed description of iconic representations suggested significantly more nuance than his "iconic as imagery" summary. Iconic imagery distinguishes the hazy space between concrete enactive experiences and associated socially constructed syntax and semantics of symbolic systems. As students are learning, their iconic image probably differs from the symbolic, but by the time they master the concept, the iconic and symbolic are for all intents and purposes aligned. A new coder might substitute English when they are unsure of the proper syntax from the programming language, but an expert programmer seems to think in code. If experts eventually align their iconic imagery to the symbolic, why is forming iconic representations critical in problem-solving?

Iconic representations are useful as a middle ground for forming and manipulating concepts. The next section details the use of iconic representations in learning, but iconic representations also allow flexible thinking beyond instinctive/intuitive behavior. Nondeclarative (enactive) memories provide fast and efficient processing (System 1), but the brains of more evolved creatures use declarative memories (System 2) to alter automatic responses based on new information and changing circumstances. I might drive the same route to work without thinking every day but take a different path when the weather is bad[46]. System 2 needs a place to store recently gathered facts, add in existing knowledge, rearrange plans, and retain such plans, which Bruner identified as iconic. TAMP looks to improve on his vague assertions by applying the 'new' information discussed in Section 7.3 from neuroscience.

[46] Assuming I stay focused! Otherwise, I berate myself when System 1 took me the usual path cluttered with weather related traffic!

Iconic representations and memory structures

Bruner's description of iconic representations seems to align with declarative memory – specifically the mechanics of prospection[47]. The following example to considers the nature of iconic representations. An instructor asks a brand-new programmer to explain the functionality of some code containing a loop. The novice must lookup (short-term memories) or recall (semantic memories) facts about a loop and any additional language constructs to determine its function. In children, Bruner described the perceptual aspects of iconic representations, which would seem to indicate aspects of short-term memory. The ephemeral nature of short-term memories poses a problem for iconic representations. When we turn our attention away from something in our short-term memory for more than a few moments, we risk forgetting it. If iconic representations were subject to forgetfulness Bruner would seemingly have captured it. Short-term memory may act as a scaffold to fill in missing knowledge temporarily, but only in very simple problems. Iconic representations seem to describe more lasting memories.

Iconic representations cannot merely be semantic memories. Semantic memories provide facts without context or even association with a time or place like episodic memories. A skilled programmer can describe the operations of a loop independent of possible algorithms that use loops, where a novice may need to discuss a loop based on an example. Bruner noted that iconic representations form to generalize from enactive or integrate with symbolic representations. Early on, a novice programmer integrates facts about loops and other language constructs to consider its behavior. Many novices need to see code in action to determine its intended purpose (Bednarik & Tukiainen, 2008) and use the execution results to mature an iconic representation of the code's behavior. As discussed more in Section 7.5, Bednarik and Tukianien noted that experts use the code alone to explain its purpose. Iconic representations form to combine new information, some enactive and some symbolic, and derive new meaning. Bednarik and Tukianien's data seem to indicate that with practice, experts can simplify or even skip the need to build certain types of iconic representations. Iconic representations, therefore, do not seem to be merely a collection of facts in semantic memory.

[47] Whether iconic representations model the activity of the medial temporal lobe is not critical to this discussion. It may be that the MTL is where most or all creative problem solving occurs but TAMP is seeking a higher level of abstraction in modeling cognition.

Bruner's later descriptions of the roles of iconic representation seems to relate more to episodic memory and the mechanics of prospection. Prospection/recollection assemble and organize knowledge from various regions of the brain as required for problem-solving. To explain code, a programmer reads and considers the presented code. The novice may build their iconic representation using one or more traces, where the expert has more tools in their box. The expert may see a pattern in the code (e.g., this loop is searching a list for an item) and leap to an answer. They may need to integrate several such intuitive leaps but can do so with little need for tracing (or the tracing is virtually instantaneous thanks to their robust notional machine). Experts either answer more quickly than novices or can handle significantly more complexity because of their experience. As novices repeat exercises in tracing and explain code, among other activities, they may automate activities (form enactive representation) that previously used iconic representations and move closer to becoming an expert.

Aligning iconic representations with prospection not only seems to fit but helps provide a better definition of the representation Bruner called critical to problem-solving. In early descriptions of iconic representations, Bruner evokes the idea of personalized imagery. He noted that iconic representations grow out of enactive representations. Iconic representations help to blend symbolic representations with enactive. Prospection is the reconstruction of personal experience and semantic memories into an imagined world. Some tasks only require semantic memories. Ss Section 7.3.2.3 discussed, K.C. could perform many daily tasks using semantic memory (symbolic representations) but struggled when asked to plan. K.C.'s circumstances seem to mirror what Bruner (1966c) said about iconic representations.

> For when the learner has a well-developed symbolic system, it may be possible to by-pass the {enactive and iconic} stages. But one does so with the risk that the learner may not possess the imagery to fall back on when his symbolic transformations fail to achieve a goal in problem solving. (p. 49)

I use this quotation from Bruner frequently because it seems critical to understanding why novices struggle to write code from scratch. Programmers do not solve problems with code by learning facts about programming languages alone. Symbolic representations can help in some programming activities, but according to Bruner, they may not be enough to solve complex problems.

Prospection is a natural fit for describing design cognition. Designers do not exactly time-travel, yet at the same time, time to the mind is merely navigating a model of the world in some sequence. Software design seems particularly well aligned to a 'time-travel' model as the primary goal of software design is defining a sequential order of rules in achieving a prospective task. Designers do more than merely stringing together rules; they must navigate the flow of data and control through those rules in much the same way as we tell stories. The amnesiac patients from Hassabis et al. 's (2007) study were able to recall, and to some degree, string together facts in their stories, but could not do so in a way that bound these facts into a cohesive narrative. Tulving (2005) pointed out that people (and animals) solve many problems using semantic memories alone, but only certain types of problems. It may be that novices, still undertrained in key areas of programming knowledge, solve problems that demand only semantic memories until they learn enough to apply prospection properly.

Iconic representations in programming

Novices can accomplish many basic programming tasks by following a procedure or mimicking a pattern. For example, I see many students[48] cobble together a seemingly complex program by copying and making small changes to existing code. The following code demonstrates the absolute minimum example for receiving user inputs from a command line in Java.

`Scanner in = new Scanner(System.in);`	The helper to retrieve inputs
`System.out.println("Where are you going?");`	Display a prompt to the user
`String destination = in.nextLine();`	Retrieve the user's response

Retrieving additional inputs is as simple as copying the last two lines and changing the prompt and the variable name. Even this simple task can confound abject novices, but with a little guidance[49], almost anyone can build a simple user interface. Building a user interface demands little forethought since the programmer can incrementally add code and test each step. Slightly more advanced students can even implement basic validation rules for inputs (e.g., ensuring the user

[48] And more than a few professional programmers

[49] If left to their own devices, many struggle to mimic another. Vygotsky noted that mimicry is not trivial. A person cannot mimic work unless it is within their ZPD, so an inability to replicate example code is a useful test of basic proficiency in coding.

provides any input; it is a number; it is greater than zero). Validation requires slightly more advanced logic, and language constructs (e.g., decisions, loops, exception handling), but the process of validating a single field is something that can be described by a procedure. If a task can be completed by following a predefined plan, that plan can be memorized and become a symbolic representation (possibly automated in time). Where students truly begin to struggle is the logic to ensure *all* required inputs are valid and planning how to organize and what to do with the results.

Some seemingly simple tasks are actually quite difficult without a robust iconic representation. For example, not all knowledge about programming easily transfers to new programming languages. Post Y2K, I spent a few years teaching Java to Cobol programmers. Some of these people had decades of programming experience and high-paying jobs, but they struggled with certain aspects of Java, particularly object-oriented concepts. They understood how computers and programming languages worked, how to define and test code, but just like my brand-new programmers, they struggled to define simple Classes[50]. Object-oriented design required a new mix of knowledge for even experienced programmers. Cobol experience did not transfer to the 'new world' of objects, which slightly alters the basic programming paradigms of early programming languages. When focusing on familiar constructs (e.g., decisions, loops), my Cobol programmers were far superior to students learning their first language. However, when it came to object-oriented aspects of Java, many struggled even more.

Experts are ‘better’ at designing because they have an alchemical blend of declarative and nondeclarative memories that allow mental navigation of their proposed design. Brand new programmers seem to struggle to design new solutions when they lack a procedure to follow or example to emulate. Experts hold an advantage over novices because they have internalized a catalog of examples but seem to struggle when the problem falls outside of that catalog. Many of my students over the years expose their experience in coding other languages than what I am teaching by writing code that “looks like” the style of the other language. Maybe the naming conventions, capitalization, or punctuation are slightly different. Even when working in a new language, System 1 has habits that take time to change. My Cobol programmers exhibited the same ‘misalignment’ in their designs when they built structured solutions in an object-oriented

[50] For reader not familiar with object-oriented design, a Class provides a model of a real-world entity within software design. For instance, an Automobile class might store data like make, model, year, and color. Part of object-oriented design is translating real world attributes into variables (also called attributes) within the class.

language. Their intuition influenced their designs significantly more than the rules and best practices (symbolic representations) I was teaching.

Fix, Wiedenbeck, and Scholtz (1993) provided evidence of prospection at work in their tests of expert and novice memories about code (Section 7.3.2.1). Since recollection and prospection use the same mental mechanics, the same advantages experts hold when designing software likely apply when remembering the details of briefly reviewed code. Fix, Wiedenbeck, and Scholtz showed that experts were significantly better at breaking down and remembering details of a design than novices. The expert's advantage was not merely knowing more about programming in general; they even recalled "information readily available in the program, yet novices [did] not extract it" (p. 78). Experts gleaned more information as they read the code and then reconstructed the details. Several of the study's questions relied on decomposing, analyzing, and then reconstruing a mental model of the presented code. The experts did better on this despite concerns that the task might be unnatural.

> A few of them commented that in studying a program they normally had a concrete objective in mind, such as finding a bug or determining the effects of a potential modification. In this case they were on a fishing expedition and, as a result, were not sure where to focus their efforts. (p. 78)

The experts did not know what the questions would ask yet managed to extract and recall more in their iconic representations. Their experience did not foretell the types of question they were expected to answer, they simply gathered the information that seemed pertinent.

The advantage the experts demonstrated with not merely knowing more about the language but a different type of information. On the tasks that merely required semantic memory (e.g., names and places of things) the experts and novices recalled the same amount of information. The experts' prowess over novices in design is unsurprising, yet the sizable advantage even when performing an 'unnatural' task such as memorizing code for a few minutes before answering a few questions. The data that Fix, Wiedenbeck, and Scholtz gathered showed the power of patterns in recollection. Since recollection and prospection share the same underlying mechanics, it stands to reason that patterns are equally important in design.

Prospection arranges implicit (enactive) and explicit (symbolic) knowledge within iconic representations to solve complex problems. Prospection seems to be a process that also transcends Systems 1 and 2. We consciously recall the past or plan for the future, but the systematic errors

that occur in constructing memories are hardly conscious (e.g., confirmation bias or the 'feeling of knowing'). We do not choose to distort the past or overlook future threats. Cognitive biases play a role in both the recollection and planning performed in episodic memory, so prospection provides a useful model of the interplay between the two Systems during creative tasks. Adding the discoveries about episodic memory and proscription to the definition of iconic provides even more definition to these vital but under-defined mental constructs.

Bruner believed that students benefit from developing iconic representations, which becomes easier to understand after considering how experts think. Experts' minds are swirling with facts, intuition, plans, and rules stored across the brain. Bruner's representations capture these abstractions but perhaps fell short in explaining what exactly was special about the iconic, particularly in comparison with symbolic representations. Episodic memory and prospection add to the existing constructs available to TAMP that further define the inner workings of iconic representations. Iconic representations are

- *Constructed* – formed from new and existing knowledge selected by perceptual bias and priming from System 1
- *Easily modified* – the mind can flexibly alter knowledge to imagine unexperienced futures and solve new types of problems
- *Transient* – they will fade if not used repeatedly, at which time they may transition to enactive, symbolic, or persist as iconic if the story matters

Iconic representations are not just useful for experts, though they also enhance learning.

7.4.3.2 Iconic representations in learners

While not every learner needs to become an expert in every subject, instructors may still choose to include pedagogy to build iconic representations since they help in the quality and speed of learning itself. Iconic representations support the development of both *knowing how* and *knowing that*. If you remember from Section 7.3.1, Squire (1984) reported that the hippocampus improves nondeclarative learning. For example, many immigrant children acquire their new language by watching local TV, but the quality and speed improve with formal feedback. Iconic representations are even more important when trying to 'break a habit' to change an enactive representation to align with new symbolic knowledge. Any programmer who adopts a new language must break some habits of syntax to conform to the new language (e.g., do I use `//`, `%`, `#`, `*`, `!`, `REM`, `--` or something else to comment code). Iconic representations help to recognize

when enactive behaviors need altering and help to curb the behavior until System 1 adopts the desired response.

Many educational traditions have noted the risks of teaching procedures for solving one type of problem and expecting people to become flexible problem-solvers. The behaviorist movement showed that instructors could shape behavior, even in humans, using appropriate reinforcements (Skinner, 1965). Reinforcement learning can be effective but often is often shallow. Reinforcement alone leads to connections of stimuli and behavior that Skinner called "superstitious" (p. 85). The same type of learning is possible when forming symbolic representations when learners do not understand the significance of the concepts. For example, people in the middle ages discovered the health benefits of garlic in preventing illness, but the general masses adopted the practice for an entirely different reason. The concept of a microbial infection was too abstract for the science of the day, so many ate garlic to stave off vampires instead (American Society for Microbiology, 2011). In Bruner's terms, a person must connect facts stored in symbolic representations (garlic promotes health) with an unrelated experience (getting sick). Making such a connection typically requires an iconic representation. Images of a fanged nemesis are seemingly easier to conjure than future unseen maladies for the layperson.

Competent programmers need to master knowledge for use in problem-solving, not rely on memorizing procedures that might lead to superstition. Pea (1986) may have documented one form of programming superstition in defining the superbug. If novices believe the computer knows more about their program than they do, it follows that they may come to think they are solving a puzzle rather than creating their program's failures through their missteps. Perkins et al. (1986) noted several cases of students abandoning their code to start the same program from scratch rather than understanding their mistakes. TAMP might suggest these students are not developing am iconic representation of their plan, rather following some symbolic procedure that they hopefully 'get right' on their next attempt. When the resulting code does not work, the novice has no mental representation of what happened to consider in troubleshooting. The way some novices mimic plans is the same way I bake. So long as the recipe is foolproof and I attend to details, my results are acceptable. If any stage goes wrong, I hope to have a tasty if unpresentable mess, as I have no idea of the purpose of the ingredients, the dynamics of temperature and humidity, or any of the

chemistry of baking[51]. For me to become a better baker, or perhaps more importantly for programming students to become strong problem-solvers, requires a strong understanding of the concepts beyond blindly following instructions.

Vygotsky (1962) believed that learning any language required a person first to understand the concepts before they begin to understand their significance in language.

> a concept is more than the sum of certain associative bonds formed by memory, more than mere mental habit; it is a complex and genuine act of thought that cannot be taught by drilling but can be accomplished only when the child's mental development itself has reached the requisite level. At any age, a concept embodied in a word represents an act of generalization. (p. 82-3)

Vygotsky could be describing the role of iconic representations in this passage. Enactive representations ("mere mental habit") alone are not enough to form a concept. Iconic representations form out of a "genuine act of thought" that requires some level of "mental development."[52] Bruner and Vygotsky agreed that mastering a language requires a mix of habits (enactive), concepts (iconic), and words (symbolic). Computing educators are stuck with the challenge – how can a student form concepts about a language without connecting symbols and their resulting action? Vygotsky's warning suggests that neither learning syntax and semantics nor a strong notional machine alone is enough to help novices understand programming concepts.

When instructors focus on piecemeal education, students never progress past mimicry to independently perform complex tasks. Vygotsky (1978) noted that in Montessori schools, children learned to write proficiently, but only when the instructor provided the content of the message. He feared that merely learning the mechanical skill of writing "will not be manifest in [a child's] writing and [the child's] budding personality will not grow" (p. 117). A child copying down messages exercises manual dexterity, spelling, and punctuation, but not in the composition of original ideas. Children go through the act of writing and may even stretch to mimicking new messages as their teachers demonstrated, but this is a far departure from creative storytelling. Similar stories abound in computing education literature, and Chapter 8 will revisit three such studies in detail.

---

[51] I am doing my best to learn from cooking shows, but they tend to be heavy on drama or polite British laypeople rather than lessons on food science.

[52] This passage by Vygotsky also seems to support the supposition that System 1 forms the core of mental development. If mental development is grounded in "mental habit" formed by "drilling", he seems to be describing the way System 1 learns.

7.4.3.3 The revised iconic representation

The iconic representation is the most confounding theoretical construct. I believe it is the most critical in defining expert thinking and the largest gap in programming pedagogy, yet it is also the hardest to influence and measure. Iconic representations by their nature are individualized. Bruner described iconic representations as imagery, yet it seems clear that iconic imagery, like all revolutionary art, makes more sense to the artist than the audience. We each can overlay art with our own meanings, but our interpretation is equally iconic, blending our experience with the symbols of another. Instructors can nurture a student's iconic space, but imposing imagery (e.g., flowcharts, diagrams, trace tables) cannot replace the need of students to create and, in turn, blend experience with programming content. Over time, a learner will use code, diagrams, or other such symbols inherently[53], but until such a time, it is challenging to 'measure' the quality of iconic representations. Imagine being asked to interpret the "word salad" produced by those with Wernicke's aphasia (Figure 7.2). Even if a machine could learn to map between the random words a Wernicke patient utters and their intended meaning, the algorithm would work only for that person. It may, likewise, be impossible for instructors to pinpoint faults in a student's iconic representation. So, what is there to do about the theoretical construct that is iconic?

Within the scope of TAMP, I am refining the definition of an iconic representation starting with Bruner's core ideas. Bruner's evocation of imagery as iconic is useful, but misleading. I believe that when Bruner referred to imagery, he meant more in the sense of art than diagramming. The imagery of art typically requires interpretation by the observer, rather than conveying a clear meaning. Iconic representations are in the space of poetry and improv, not that of textbooks and diagrams (which the creators intend to be symbolic). In programming, iconic representations are, like the notional machine, abstractions of the design – a plan for or summary of what the code does. Iconic representations deal in patterns, often inspired by experience as governed by the concepts of prospection (see Section 7.3.2). They are slightly hazy, but less so, the closer the current problem is to past examples. For programmers, the process of coding and even more so, debugging, refines an iconic representation. Sometimes the debugging process begins with overcoming the iconic representation's understanding of how code should be working with the actual execution of what is happening. Iconic representations are inevitably short-lived compared

[53] I believe I think in UML and Java code, but perhaps my System 1 is just so automatic at translating them I no longer notice any difference.

to other representations of knowledge, as are unused episodic memories. If used frequently enough, the shadow of past designs will live on as inspiration for future designs (enactive), but most of the details will fade with time.

Theoretical constructs are 'better' when they link to measurable concepts. Constructs without concepts are difficult to quantify as they have no clear measure. As it stands, measurements of iconic representations are binary: a person who shows alignment in problem-solving is mature or immature otherwise. It seems difficult to create a middle ground when progress may be equally individualized. It may not be important to 'measure' the state of the iconic reasoning if researchers and educators can measure the impact of pedagogical interventions as a proxy. For example, if educators can identify what types of tasks, like coding user interfaces, require only memorizing procedures and which require true manipulation of concepts, they can sequence questions to promote different types of learning. Rather than, for example, dissecting the exact biology and chemistry of weight loss, it may be useful in the meantime to find habits and activities that best promote a healthy lifecycle. The first task of researchers may be to identify such useful pedagogical activities.

Instructors can create better curricula to grow iconic representations, but students can also benefit from understanding the learning process. Bruner (1966c) advocated one of the best things an instructor can do is get out of the way.

> Perhaps the greatest problem one has in an experiment of this sort is to keep out of the way, to prevent oneself from becoming a perennial source of information, interfering with the child's ability to take over the role of being his own corrector. (p. 70)

Students may benefit from understanding the nature of their struggles and have some idea what how expertise looks in action. Often students feel betrayed when they succeed on every homework problem only to fail miserably on tests that suddenly shift to 'transfer problems' that require *iconic manipulation*[54]. Iconic manipulation is the conscious restructuring of knowledge in memory. Bruner described people using iconic representations to generalize from experience, for example, but TAMP suggests that the same process might be used in problem-solving. Iconic manipulation is required when System 1 is unable to offer direct priming to aid in problem-solving. The problem

[54] There are many reasons and types of questions that can throw students for a loop, but questions that demand iconic manipulation are perhaps more prone to expert blind spots, since experts spot patterns implicitly where students may rely on procedure.

statement may not have prompted any enactive representation or, possibly worse, triggered an inappropriate one. Without useful priming, System 2 must consciously attempt to recall useful information and organize that knowledge within an iconic representation in a 'brute force' manner. System 2 must determine what primed knowledge is not useful, and hopefully trigger useful information that was not primed. An expert might consider this process 'brainstorming' and while perhaps demanding, it can be productive because of their extensive System 1 automation. Novices may find this process overwhelming and possibly even futile if they cannot remember enough information or cannot associate what they know as applicable to the problem.

I regret that despite the preponderance of data suggesting the presence and value of iconic representations thus far and still forthcoming in this chapter, I still have no concrete concepts to suggest are useful for measuring iconic representations. My proposals linking prospection to iconic representations feel supported, yet still lack direct measures. To some degree, the inability to directly describe iconic representations is frustrating, yet the iconic representation construct may simply rely on measures of enactive and symbolic representations. For example, the 'memorization' task used by Fix, Wiedenbeck, and Scholtz (1993) could measure the ability of a programmer to detect patterns in the design, and thus form richer iconic representations than novices who have not formed such patterns. Fix et al.'s test, combined with a think-aloud protocol, could measures the influence of System 1 (automatic pattern matching) and System 2 (conscious deliberations) but may not say much about how that knowledge translates to new designs, only the reconstruction of memories. In the meantime, the refined definition of iconic representations in this section, hopefully, inspires further study into this type of thinking, and the concepts and associated measures may emerge in time.

7.5 The Applied Notional Machine as a core construct of TAMP

The notional machine provides a powerful construct modeling the core knowledge of programming – the language in action. Computing educators often use the notional machine as a buzzword for knowledge that a programmer, and possibly anyone who wants to work with computers, must acquire. Under a model of cognition that assumes our mind universally applies what we learn, a mental model of the notional machine may seem sufficient as an abstract concept. The challenge of making practical use of the notional machine as a pedagogical tool is the inability

to align the various uses (e.g., tracing, explaining, or writing code to start) with the types of knowledge and ways of knowing identified within dual process theory and Bruner's representations. Accepting dual process theory and Bruner's model of knowledge demands a notional machine that accounts for these views. Rather than abandoning the notional machine, TAMP seeks to expand the initial literature and research into the Applied Notional Machine.

7.5.1 The notional machine as a theoretical construct

The notional machine provides computing education researchers and educators with a useful description for a mental model of a programming language. Du Boulay et al. (1981) first introduced the notional machine as a guide to creating to pedagogy, yet despite the many ways literature references the notional machine through the years, its impact seems rather limited. Du Boulay et al. provided a highly descriptive name, but it seems too abstract to provide meaningful direction to research and pedagogy as it stands. First and foremost, how does a programmer exhibit a strong notional machine? Reading? Tracing? Explaining? Writing? The notional machine seemingly should contribute to all of these and more, yet the notional machine seems to be more than the sum of these parts. From a theoretical standpoint, the notional machine is a theoretical construct without clearly identified theoretical concepts.

7.5.1.1 Managing the inflating definition of the notional machine

Premise 1: A programmer must implicitly read, write, and execute code
Premise 2: A programmer must know the syntax and semantics of the programming language and its runtime environment
Premise 3: The notional machine from one language supports the formation of notional machines of functionally similar parts in other languages
Premise 4: The notional machine supports design activities, but indirectly
Conclusion: The notional machine must include enactive and symbolic representations that contribute to iconic representations of design.

The unclear boundaries of the notional machine have led to an expansion of its role and value over the years, without a clear link as to why. For example, Sorva (2013) summarized the notional machine:

> —is an *idealized* abstraction of computer hardware and other aspects of the runtime environment of programs;
> —serves the purpose of *understanding* what happens during program execution;
> —is associated with one or more programming paradigms or languages, and possibly with a particular programming environment;
> —enables the semantics of program code written in those *paradigms* or languages (or subsets thereof) to be described;
> —gives a particular perspective to the execution of programs; and
> —correctly reflects *what programs do* when executed. (p. 8:3, emphasis added)

Sorva's description includes several italicized words to highlights possible implications of otherwise semantic knowledge. Sorva used the word *understanding* rather than predicting, tracing, or other functional aspects of running code. It implies the notional machine must help comprehension – explaining – as well as execution. Understanding, a function of System 2, requires a different type of support from System 1 than mere automation of execution, at least if it is to act as the intuition noted in the literature (see Section 2.3.2). The authors who wrote about intuition did not describe experts gradually coming to understand the code by considering the output of the execution; experts leap to seemingly instant understanding of simple and sometimes more complex designs. Such intuitive leaps do not seem to be the province of any mental model of the notional machine as traditionally defined. If the notional machine is foundational to so many aspects of programming, a more refined and measurable notional machine might make it easier to apply in research and the classroom.

Researchers have attempted to measure a programmer's notional machine using several approaches. Ma et al. (2011) used a multiple-choice test to measure the consistency of students' mental modes. Several studies used tracing as a measure of the notional machine (Cunningham et al., 2017; Lister et al., 2004). Lopez et al. (2008), followed by Lister et al. (2009), measured tracing, explaining, and writing code without finding a clear hierarchy to these skills[55]. When students are reasonable tracers, yet cannot write code, what does this say about their notional machine? When they write code, but cannot explain it? More often than not, authors invoke the notional machine without any connection to their methods (e.g., data to collect and analyze to describe student learning)[56]. TAMP suggests that conceptual knowledge and tracing skills barely scratch the surface of the breadth of *knowing that* or *knowing how* required to program, and even

[55] They never explicitly mention the notional machine, but it seems reasonable to include their findings
[56] I won't call out any authors by citing them, but you know who you are! 😉

mastering these two does not mean their skills will transfer when needed. As it stands, the notional machine seems to serve more as a useful goal than a working theoretical construct.

7.5.1.2 A basic redefinition for the traditional elements of the notional machine

The notional machine, as described in the literature, seems to contain knowledge that falls under symbolic and enactive representations within the mind of an experienced programmer. In introducing the notional machine, Du Boulay et al. (1981) emphasized the need to 'see' within the black box that is a computer to understand its operation. He, and others noted in Section 2.1.2, implied that a programmer must eventually develop a conscious understanding of the rules of the language as well as automating the processes required to mentally predicting code execution. Figure 7.5 splits the notional machine by distributing syntax and semantics to a symbolic representation and their operational execution to an enactive representation. The shift to Bruner's representation distinguishes the way a novice must learn information and reinforces the multiple ways of 'knowing'. It is not enough to remember the rules, as programmers must use them in practice fluidly. As many computing education researchers noted, students need to develop an intuition to match their knowledge, which can be captured by redefining the notional machine as part symbolic and part enactive.

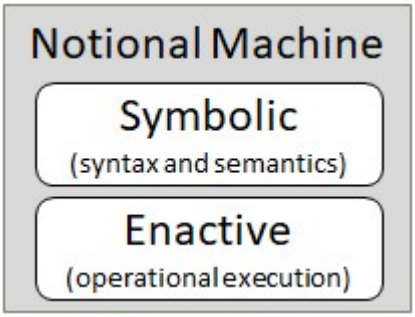

Figure 7.5. The notional machine in terms of Bruner

Measuring symbolic aspects of the notional machine

The easiest, and perhaps least useful, aspect of the notional machine is acquiring facts about the language (e.g., rules of syntax/semantics). Conceptual knowledge tests often rely on multiple-choice questions to explore how well the students remember semantic details, often disguised within problems like we saw in Section 7.4.2.2 (the pentagon). Educators should use conceptual knowledge tests to assess *knowing that* questions about details of the programming language, environment, tools, or process, but with the caveat that they do lead many students to skills in writing code. We have seen many examples where proficiency in tracing and explaining do not

predict success in writing code, and these at least put knowledge into action. Merely knowing about programming concepts does not assure students will successfully write code later.

For those who insist upon or have no choice in using such conceptual knowledge tests, TAMP offers further advice. Because experts possess broader experience, questions whose answers seem obvious are often not to your students or even your teaching assistants. Since our first impression of a question comes from System 1, a student can easily take a 'distracting' answer (to use Lister et al.'s term) and overlook seemingly obvious rules. We have seen examples of this from Perkins (2010) – the tower and hole problems – and in the analysis of the fill-in-the-blank problem from Lister et al. (2004). Instructors often fear to 'give away' a question by showing an example that is too similar, but TAMP suggest that without varied practice, System 1 may not activate the appropriate information. Ironically, timed tests may be a better determination of System 1 maturity, but only if a.) the students understand that the test seeks to challenges their automaticity, b.) they are given ample practice problems, and c.) the test questions are reflective of the training[57]. The next section considers more on timed tests as a test of enactive representations.

Measuring enactive aspects of the notional machine

Enactive representations of mental abilities are particularly hard to measure as System 2 can often compensate when System 1 is undertrained. It can be difficult to distinguish between a talented and disciplined performance versus a well-trained and automatic one by results alone. If you want to test an archer, you have them shoot many arrows in a narrow span of time. An amateur may get lucky on a few shots, but over time, the practice put in by a professional will show. For some tasks, consistent performance is a good sign of strong abilities. Testing mental abilities is a bit tougher. Instructors often add tracing questions to assess practical, rather than memorized, abilities that conceptual tests assess, but tracing examples do not always test enactive maturity in the way expected. The pentagon question from Section 7.4.3.2 shows that good students (and 'experts' like me) can fall into a trap because of their familiarity with similar problems and overlooking a rather mundane detail[58]. Questions like the pentagon question essentially risks

[57] Ironically, b and c are also requirements of training an artificial neural network!

[58] If graduates of your program really need to mentally execute code with perfect order of operations, the question is fair. Otherwise, it falls very short of my "solve in 30-seconds with a computer" rule.

making "Type I" errors, penalizing students who are otherwise on the right track in automating their programming skills. The bigger problem with tracing questions are "Type II" errors, students who seem on track, but are not.

If a student's final answer does not establish the quality of their enactive representations, then educators need some other measure. The first impulse might be to use timed tests as a means of ensuring students are operating primarily within System 1. Timed tests certainly distinguish students who have achieved automaticity but come with other concerns. For example, I took an introductory robotics course that required extensive use of vector algebra and trigonometric identities. In the first week or two, my System 2 spent most of its effort deciphering all I had forgotten about the symbols and rules of trigonometry. I managed to automate enough of the rules to survive the initial test[59] but struggled through the entire course to automate the required algebraic manipulations involving vectors. It was not that I did not put in the time; I spent 8 hours practicing a single homework problem spanning 16 pages of algebraic manipulations. I suffered from both a lack of recent experience (I would have thrived 20 years go) and too few examples to use for practice. I feel as though I understood the concepts well, though. I started every problem and only failed to complete ones due to time limitations. My saving grace was work ethic, a desire to be there, and the awareness that *I was not in a position that was reasonable to succeed in the assigned task*. My educational background perfectly qualified my participation in the class, but my skillset was too rusty, and my System 1 too decayed to perform the tasks as presented. The timing proved that my System 1 was not fit to complete vector algebra at the level expected for the class, *which had nothing to do with the professional application of robotics*.

Timed tests in programming are more reasonable, as we expect students to be fast and effective programmers, but they ignore other factors. Programming students are often in their first year of college. Some are not sure whether they belong. Most, I propose, believe that knowledge is knowledge, and not performing well on a timed test reflects their inherent ability, not their work ethic in practicing. Some students suffer from test anxiety, must juggle other academic and non-academic priorities, and may not enter class with the same System 1 advantages (ignoring inherent abilities) as others. Timed tests, particularly when only offered a single time, risk sending the wrong message. They can be useful when instructors clearly tell students they are expected to

[59] I probably remember none of it now, more than a year later.

automate specific behaviors, have ample opportunities to practice, and have a growth mindset to learning. I fully intend to include such assessments as part of the class but done in a primarily formative way, allowing the student numerous attempts and most of the term to eventually complete. Timing is one way to measure enactive abilities, but not one with much feedback for intermediate growth.

Section 5.2.2 proposed that the exemplar student whose work Lister et al. (2004) featured was still operating primarily from System 2, also suggesting that sketching/doodling offers an interesting measure of the maturity of System 1. I suggested that when the student used redundant sketches, it was a sign that their System 2 needed extra support navigating through various levels of unautomated language constructs. Moreover, once System 1 matures, their need to sketch will diminish, possibly to zero. Instructors from several disciplines sometimes ask their students to "show their work" and may even take away points unless they do, since showing one's work is a sign of mental maturity (and a way to track mistakes). TAMP suggests that neglecting to show work is caused by System 2's laziness, and a powerful and well-trained System 1.

When teachers require students to show their work, they are placing a burden on any student who primarily relies on System 1. When System 1 provides answer, there is no work to show. System 2 does not know how System 1 answered. To show work requires the student to ignore the answer they 'know' is correct (System 1) and recall the procedure that may have at least partially faded from memory. Ironically, identifying errors in the documented work will only correct errors if the student retrains System 1 (with sufficient practice on similar problems) or System 2 regularly intervenes to correct System 1. When students push back against showing their work they do so because countermanding System 1 is effortful and may induce an emotional response (Naccache et al., 2005). While it may be helpful when instructors can identify student mistakes, correcting the mistakes is not as simple as providing such feedback.

Tracing is just one of many types of enactive knowledge programmers must mature. Strong programmers can write code producing relatively few syntax errors. When their code results in a compiler error, they use enactive representations of error messages to redirect their next actions towards quick fixes (e.g., add a semi-colon, set an uninitialized variable in the else statement). When their code results in a runtime error, System 1 suggests where to look first (e.g., array index out of bounds means check the bounds of the loop). Many studies have captured student actions

at the compiler, runtime, or even keystroke level. Such data is not useful for summative assessment but can provide formative feedback on progress, particularly to researchers.

7.5.1.3 The state of the notional machine

At this stage in TAMP's creation, I feel confident suggesting the theoretical constructs and hinting at useful concepts that might provide insights to forming mental representations of notional machines. The notional machine as traditionally defined is very useful in teaching coding but may be confusing in modeling other aspects of programming, particularly design. While symbolic and enactive representations aid in creativity and understanding, it is the iconic representation that drives such activities. To fully understand how programmers read and write code, it is helpful to include iconic representations.

7.5.2 The Applied Notional Machine

The Applied Notional Machine (ANM) further expands the 'refined' notional machine using iconic representations to capture a programmer's mental models that captures design[60]. As traditionally defined, the notional machine models the language and its execution yet does not address design. Sorva's (2013) list (see 7.5.1.1) implies that the notional machine contributes to algorithmic comprehension by stating it "correctly reflects what programs do when executed" (p. 8:3). Sorva, like many others, asserts that programmers deduce the purpose of code using the notional machine's repository of semantic rules or mental execution. Building on the refined model of the iconic representation, TAMP suggests a third possibility.

Computing education literature provides several examples of novices and experts explaining code, as discussed throughout this chapter and Chapter 2. Bednarik and Tukiainen (2008) presented code samples to students and experts within a visualization tool that simulated code execution. The students used the tool quite significantly to aid in comprehending the code, where the experts generally used it a single time, and one not at all. The students needed to see the code in action to make sense of its purpose, while the experts could determine its purpose from

[60] The concept behind the ANM, separating design and language, could apply without the revamped notional machine using Bruner's representations. Sorva (2013) seems to be adding some of these elements to the notional machine already, but such additions seem to clutter rather than clarify the notional machine as a construct. It is reasonable to adopt the ANM without adopting the full scope of Bruner's representations.

reading alone. The novices relied on executing code, where the experts seemed to watch the visualization to confirm their guess. Are experts better at remembering the rules? Wiedenbeck (1985) saw they were slightly better and faster, but later Fix, Wiedenbeck, and Scholtz (1993) asserted that one major difference was that "knowledge of recurring patterns may be deficient among novices and need to be built up through study and practice" (p. 78). A full mental model of how programmers think requires a theoretical construct describing the resulting abstraction, algorithm, or other higher-level plans that translate into design. Iconic representations are perfect vessels for describing the way our mind constructs and manipulates knowledge.

7.5.2.1 Defining the Applied Notional Machine (ANM)

The Applied Notional Machine (ANM) wraps the existing notional machine within its context, design. As described in Section 7.3.2.1 and 2.2.2.2, an expert easily and consciously distinguishes design from code. When presented with a code sample, they quickly form an iconic mental image to remember the details of the algorithm[61]. To the novice, investigating the details of a code sample may be like finding a specific zebra in a meandering herd – it is difficult to do when all zebras look alike. Their first experience with code relies on the actions they take and the results they perceive. The astute student may be able to use System 2 to integrate examples and details from the lecture, but this learning strategy likely works only on simple examples for easily observable rules. Fix, Wiedenbeck, and Scholtz (1993) described the difficulties novices had in processing, much less remembering the structure of design out of code. The iconic representation adds a construct to represent design knowledge apart from the notional machine and offers insights into why experts are better at using this knowledge than novices.

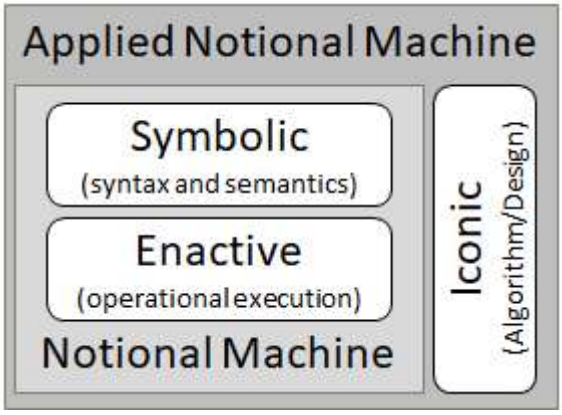

Figure 7.6. The Applied Notional Machine

[61] To be clear, at this stage the algorithm/design are merely as represented in written code for a specific problem. The notion of algorithmic/design patterns is another important mental representation, but the ANM is not intended as a construct to capture the nuance of design at this stage. Briefly, it is worth saying that a design pattern begins as an iconic representation capturing frequently used approaches to solving problems.

The Applied Notional Machine models the representations that form in a programmer's mind. Figure 7.6 shows the iconic representation as a holder of the algorithm/design apart from the notional machine. An expert's mature notional machine gives them conscious access to the rules (symbolic) and automatic execution of basic code constructs (enactive) in support of their mental work with algorithms and design. The iconic representation holds abstractions about the purpose and goals of code (data and functionality) beginning as rough approximations and maturing over time as the programmer works with the code.

The ANM offers a look at expert thinking that explains why experts are not infallible when tracing code or other simple tasks. Iconic representations are not exact models of the world, rather the gist of what we perceived. Reading code into memory line-by-line is no less difficult than memorizing a book, so the natural tendency is to summarize. Even in writing code, an expert thinks more in phrases than exact words, as can be evidenced in Youngs' (1974) study mentioned in Chapter 2. He compared the mistakes made by experts and novices, noticing they averaged the same number of mistakes on their first runs.

> It is surprising that experience is not evident from the number of first pass bugs. Since differences in experience level are not obvious from the number of errors on the first run, it is natural to ask what happens after the first run. (p. 366)

Experts made a similar number of errors because while they are less likely to make beginner errors, they are still building the program from generalities. After the first execution, the feedback from the computer starkly portrays each disconnect between intent and the produced code, which the experts were better at correcting. The expert's mature notional machine means the source of errors stems more often in erroneous logic than a misuse of the construct. The novice must consider if the source of the error is their plan or mistyped code, where and expert System 1 will likely spot typos or misapplied pattern instantly once the computer highlights the error. The more experience a programmer has, or the more familiar the problem, the more refined the initial iconic representation will match the resulting code and execution. The main advantage the expert holds it the ability to focus on the design since they have greater confidence in their understanding of the language.

7.5.2.2 Using the ANM to interpret error messages

New programmers hit their first major obstacle when trying to understand and correct errors produced by the tools (e.g., compiler, interpreter, runtime). Even when merely mimicking an example, novices often mistype code leading to compiler errors, or once those are corrected, runtime errors, long before they need to tackle functional bugs.

> One of the many challenges novice programmers face from the start are notoriously cryptic compiler error messages, and there is published evidence on these difficulties since at least as early as 1965. (Denny, Becker, Craig, Wilson, & Banaszkiewicz, 2019)

Denny et al. noted that one branch of research seeks to simplify error messages, so it seems that modeling how programmers process errors might provide insights into not only how novices struggle but perhaps how to head off such struggles (or if that is even possible). Error messages are an interesting special case in learning. The tools provide the programmer with immediate feedback on their mistakes, but that feedback is sometimes incomprehensible. The underlying theoretical constructs of TAMP may illuminate the cognitive quandary novices experience in trying to decipher error messages. Figure 7.7 proposes a process that programmers might use to reconcile error messages, beginning when a programmer encounters an error[62].

[62] It may be easier think in terms of compiler errors when reading Figure 7.7, but the same process applies to runtime errors. The algorithm does not influence compiler errors removing the specter of functional issues, test cases, and other aspects.

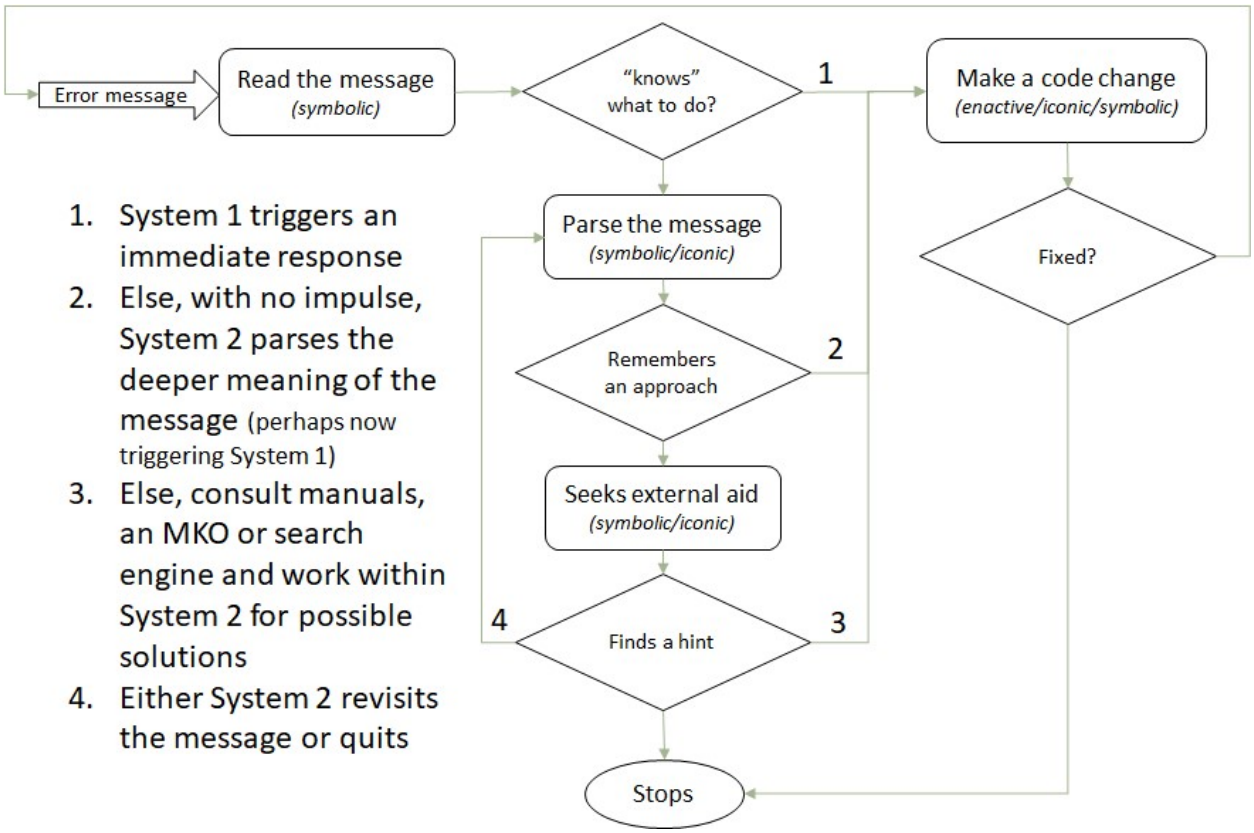

Figure 7.7. A programmer's mental process for reconciling error messages

Under TAMP, a programmer has three sources of knowledge for fixing an error. When error messages are immediately familiar (*Path 1*), System 1 prompts the required fix, if not completes the fix without diverting attention (e.g., fixing a missing semi-colons). Experienced programmers often intuitively fix errors, sometimes without considering the content of the error message or its significance. As Skinner (1965) proposed, sometimes we associate stimuli to action with no comprehension forming "superstitious" behaviors. For example, some of my struggling students will restart their code from scratch, performing the exact steps they took the first time, hoping that this time the compiler error does not appear. Their problem-solving approach is the equivalent of the cliché "have you tried turning it off and on again?" Experts are hardly better, as I am often at a loss to explain compiler error messages. I know exactly what to do, but I am not always sure, even as an instructor, how to decipher their wording. *Path 1* suggests an important new category of enactive representations formed around *strategies for identifying and repairing errors*.

If the error is new to the programmer or System 1 offers no hints, the content of the message becomes important. The programmer might study the error message to see the nature of the error, which itself might trigger *Path 2* if some part of the error message sparks an enactive representation (e.g., the word "exception" reminds the programmer to add an exception handler). Without further inspiration, System 2 parses the error message and seeks guidance in symbolic representations (e.g., what do I know about exceptions?), which again might spark an "Aha!" moment (System 1, based on some other association) or System 2 chooses a potential fix based on the remembered syntax/semantic rules. If they cannot remember anything helpful, the programmer can seek external sources of information[63].

When a novice is unable to recall helpful corrections, programmers turn to external sources of information. New sources of information help programmers (*Path 3*) by either

- Something in the information source triggers System 1 to try something else
- Something in the materials activates some to-this-point inactive symbolic representation that System 2 integrates into its problem solving
- System 2 integrates new knowledge from the source into existing knowledge, forming an iconic representation to juggle the sources of information

That fix may come when the new stimulus sparks another "aha" moment from System 1 or when the new information suggests a potential fix[64]. If the programmer is not finding what they need, they might need to refine their search (*Path 4*) or give up (becoming a *stopper)*. Experts are less likely to stop since within their prowess lies a secret – they are also better at seeking and processing resources that may help.

Experts are also much better at retrieving information than novices. Experts know the best resources, how to navigate those resources, keywords that narrow results in search engines, and a sense of how much to trust resulting sources. How did they build such skills? Experience! For example, when I was learning the newly released Ada 95 language, the only available resource was a several-inch-thick reference manual. Language designers write language reference manuals

[63] Some novices may stop at this point or may even have stopped when inspiration fails to guide their action. These lines are not shown to manage the complexity of the diagram.

[64] The new information is iconic since the programmer may or may not remember what they discover in their research. I can't say how many times I have looked up some bit of trivia about programming only to forget the solution when I encounter the error again. System 1 provides that déjà vu feeling of familiarity but not the answer without practice. The quicker the fix the more likely an iconic representation with the new information will fade.

for programmers making tools more than programmers using the language. To support my team, I needed to become a student of language design, but thankfully search engines have displaced the language reference manual for most problems. These days I am teaching Matlab, so when I hit an error, I copy the error message into a search engine. The results are rarely a conceptual explanation of the mistake, more likely a similar error on a discussion board with a reply containing instruction of what to change[65]. The strongest programmers I have worked with rarely read reference manuals; they code through a combination of examples and searching errors.

As tech-savvy as students have become, many are unaware of the basic triage strategy: Google the error message. Even when they think of searching, they are often unsure of what to search on or what to make of the results. Unless the search results jump directly to the required answer, the novice must enact some level of interpretation. Search results present another set of symbols a novice must integrate into their mental model of the problem. Experts develop a filter for which content is promising versus what is superfluous. To some degree, they know what they are looking for. By this stage of searching for fixes to errors, novices may no longer be confident they initially chose the correct strategy. If they are unsure of the goal, made a mistake, don't understand what the error message means, found information but still struggle to understand the 'fix', it should be little wonder that new programmers are likely to stop.

7.5.2.3 Revisiting Movers and Stoppers under the ANM

By watching young programmers, Perkins et al. (1986) captured not just their mistakes, but their process and coding behaviors. As introduced in Section 5.2.1, the researchers noticed that when students encountered adversity, they tended to keep going (movers) or quit (stoppers). Stoppers draw a great deal of interest – they can't learn if they stop – but Perkins et al. noted that students who 'move' to a new solution before understanding the current problem (extreme movers) struggled to make progress as well.

> If stoppers illustrate the powerful influence of negative affect on students learning to program, certain movers in a different way show such an influence

[65] The helpful online resources seem to break into *knowing how* (support forums) and *knowing that* (reference manuals)! The popularity and usefulness of online resources reflects the way programmers think about error messages. It is quicker and easier to search for similar errors than to scan the reference manual, just like the brain saves energy by listing to System 1 priming rather than using System 2 to search for specific facts.

> too. An extreme mover is also, in his own way, disengaging from the problem. (p. 42)

The stoppers and extreme movers seemed to have a great deal in common. Perkins at al. hint at 'non-cognitive' factors for these students. Movers seem immune to failure, where stoppers are often described as frustrated. Computing literature describes programming as instilling fear in student (Eckerdal et al., 2007; Kort, Reilly, & Picard, 2001; Shneiderman, 1977) for various reasons but TAMP suggest that some of the emotions programmers experience may be the results of cognitive gaps in learning as much as 'non-cognitive' factors.

Extreme movers and stoppers have one cognitive trait in common: they halt[66] an unproductive investigation by System 2. The difference is where they go next. The characteristic that distinguishes movers and stoppers is not random, but their response to adversity. Figure 7.7 describes a likely source of adversity, error messages, and possible paths novices can follow. Novices become stoppers when System 1 remains quiet, the error message is meaningless, and they are unsure how to query their resources[67]. Regular movers (the smaller set of novices in the middle) make progress using their resources and "tinkering" with code. Ignoring the action and looking at their behavior, extreme movers seem just a different type of stopper.

> Instead of dealing with mistakes and the information they might yield, the extreme mover seeks to avoid them by moving on. (p. 44)

Judged on activity alone, extreme movers and stoppers appear at the opposite ends of the spectrum. Evaluated through the lens of TAMP, however, the cognitive activity leading up to this choice seems very similar.

Following Perkins et al.'s story of one student, Tom, helps to show how the names "extreme movers" and "stoppers" might describe the same gaps in knowledge.

> Tom was quick to point out that he had had no programming experience prior to this course, and that he didn't really know what he was doing. He said that the other students around him knew a lot more than he did (p. 43)

Tom's background afforded little intuition for the subject or confidence in his abilities, but Tom at least took a shot. When Tom allegedly "stopped", he did not quit programming but moved on

[66] Sometimes preemptively

[67] Remember, in Perkins et al.'s study a researcher sat with the student all the time available for prompts, hints, or deeper help, yet they still stopped. If novices can stop, ignoring the dedicated expert sitting next to them, how do they manage when they are alone?

to the next problem since he had no new ideas for the current problem. Moving on to the next problem seems less defeatist than pragmatic. Figure 7.7 suggests that stoppers may simply exhaust their enactive representations quicker than extreme movers who seem to have an abundance of intuitive strategies.

> Extreme movers, however, move too fast, trying to repair code in ways that, *with a moment's reflection*, clearly will not work. This approach sometimes leads students to abandon prematurely quite promising ideas because they don't work the first time. Moreover, the extreme mover often *does not appear to draw any lessons* from ideas that do not work. There is no sense of "homing in" on a solution. Indeed, the student may even go round in circles, retrying approaches that have already proven unworkable. (p. 42, emphasis added)

Extreme movers seem to be full of ideas but spend no time planning or analyzing. They seem to be creatures of System 1, activating various symbolic representations without engaging System 2 and forming iconic representations to organize and learn from their attempts. Stoppers quickly become movers when activating the proper symbolic representations. In one example, Tom was about to move to the next problem when the researcher pressed him to analyze his error. With minimal guidance, Tom discerned not only the nature of the error but a successful solution. The researcher was able to trigger the required information in Tom's memory to solve the problem.

The ANM offers an alternative explanation of why students simultaneous 'know' and yet fail to apply knowledge. Tom's first trek through Figure 7.7 prematurely stopped when he did not reach out for help. The researcher helped Tom by re-engaging his System 2 when asking "what he thought the error message meant" (p. 43). Tom may not have considered the content of the error before this question. The error message's strange words, `subscript out of range`, probably meant little to Tom. The error message was not familiar, and Tom did not have the experience to guide action. The researcher's question helped System 2 persist beyond Tom's presumedly missing 'feeling of knowing'. Once reengaged, System 2 could recall the rules of subscripts and ranges in arrays and connect the error message with the problematic doe. Perkins et al. described System 2's influence.

> When pressed for an answer, Tom thought for a little while, and then said that maybe the number in the parentheses needed to be smaller. (p. 43)

Tom's System 2 – taking time to think for a while – combined facts about arrays with the symbolic error message to work out a solution. What stoppers and extreme movers seem to lack is a contributing iconic representation.

Bruner says problem-solving from symbolic representations alone is difficult, as Tom's story shows. We cannot know how Tom's teacher introduced arrays, but I am guessing if he/she used the word `subscript`, Tom had yet to adopt it[68]. Without recognizing the word, Tom may not have recognized the error was associated with his array. He did not independently associate the word `range` or that being out of range has to do with arrays. I believe that Tom never formed a meaningful iconic representation since "he encountered a bug when running a program that he had copied from the text" (p. 43). He did not design the code; he merely was responding to the results[69] and had neither the enactive representation nor the start of an iconic to use in problem-solving. Experts seem to naturally build iconic representations (Fix et al., 1993), where many novices seem unable to do so unless guided.

7.5.2.4 Defining the contents of the ANM

A 'three-part' ANM is a useful but significant simplification. At first, it was enough to introduce a notional machine combining semantic (symbolic) and non-declarative (enactive) memories. Bruner's representations provide an epistemological slant to the notional machine but do not describe the specific content a programmer must know. Educators might be thinking, what about variables, data types, operators, decision structures, loops, recursion or the other topics that seem to trip students' progress. These are logical next steps for research and the classroom, but at this stage of theory construction there are still important intermediate layers within the ANM to capture.

The content of the notional machine

[68] The more common term is index or indices rather than subscript in most languages. It is possible that BASIC instructions used the term subscript more, but Tom described the subscript as "the number in the parentheses" (p. 43) seeming to indicate he had not adopted this specific term.

[69] Just today I helped a First-Year college student who could not understand why "line 43", which was blank, produced an error. He identified "line 43" but overlooked that the error said "line 43" was in a different file. Even worse, runtime errors in Matlab appear with hyperlinks to jump to the erroneous code, but he had yet to learn that feature. He also neglected to engage System 2 to analyze the message, jumping straight to an external resource (me).

The discussion in Section 7.5.2.2 on error messages shows that programmers develop different aspects of knowledge as well. Educators tend to think of their subjects in terms of major functional groups (such as Figure 7.11), which researchers sometimes capture as Concept Inventories (Taylor et al., 2020). Some Concept Inventories focus primarily on core concepts (Wittie, Kurdia, & Huggard, 2017) while others capture both core concepts and how those concepts are used in various tasks (K. Goldman et al., 2008; Herman, Loui, & Zilles, 2010). Concept Inventories help to centralize the knowledge about a subject but TAMP suggests they are not representative of how our mind organizes such knowledge. Neuroscientists seem to indicate our mind organizes knowledge around *its* function areas (e.g., sight, sound, language). System 1 uses such knowledge implicitly, or the medial temporal lobe gathers knowledge for conscious consideration, as discussed in Section 7.3.2. Concept Inventories may be useful in organizing the desired end goal of learning, but perhaps are less informative to the stages of learning.

The ANM does not invalidate or even challenge the traditional ways of describing content, such as Concept Inventories, rather it looks to capture how we organize information within the various types of memory as modeled using Bruner's representations. Bruner's representations segment the same content Concept Inventories capture, slicing the concept into a matrix of their epistemological roots. For instance, a programmer's knowledge of variables contains an enactive representations writing the syntax and properly naming variables, iconic representations that define the purpose of the variable in the algorithm, and symbolic representations that hold facts about the various data types. Figure 7.8 provides high-level categories of the types of knowledge each representation contains within the ANM[70].

[70] In theory, a researcher could create a matrix that combines a concept inventory as rows and Bruner's representations as columns to identify the specific content that is enactive, iconic, and symbolic. While Section 7.6 provides an example for reading and designing, it feels premature to undertake a comprehensive list before additional research confirms and refines the ANM.

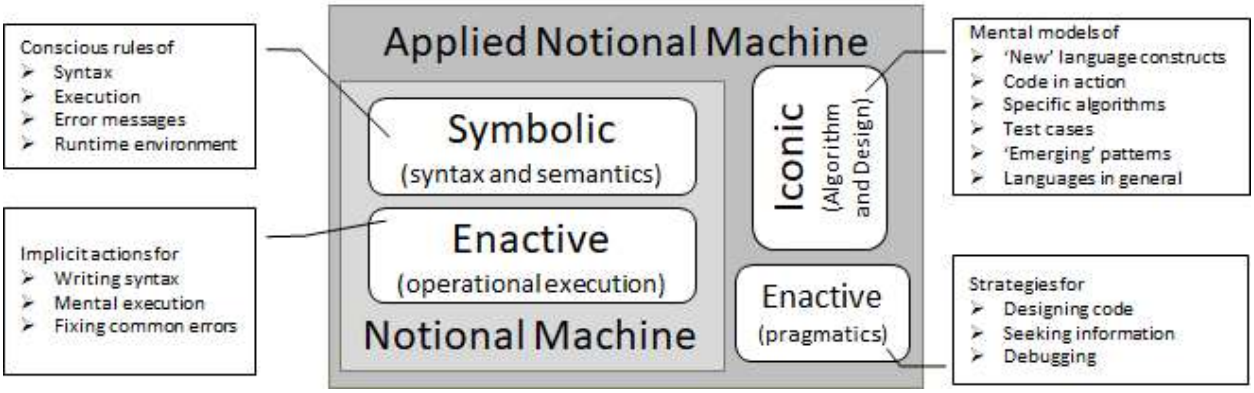

Figure 7.8. The different types of knowledge within the Applied Notional Machine

Du Boulay (1986) split syntactic and semantic knowledge (symbolic) from the notional machine (presumedly, the enactive portion), supporting the same divide in the ANM. The ANM goes further in noting that even syntax and semantic have symbolic and enactive portions. Du Boulay was 'on the right track' in noting that the difference between *knowing how* and *knowing that*, yet that difference may be difficult to perceive without theory to explain why it is important. A recontextualizing notional machine, including enactive and symbolic segments, can encompass many additional types of programming knowledge, like error messages. The last section's discussion of Tom highlighted the importance of familiarity with error messages and their corrections in addition to the rules when coding. Researchers and educators can use the expanded categories of knowledge in Figure 7.8 offers to improve models of learning.

One of the challenges in conducting researches with novices is the inability to measure a programmer's notional machine with any precision[71]. Researchers often measure programming abilities as a black box; students can or cannot trace, explain, or write code, for example. A binary measurement of students can/cannot perform certain tasks may miss the learning that students are achieving. For example, McCracken et al.'s (2001) test to write a calculator suggested that most students were not learning enough. The measures of learning used by Lister et al. (2004) showed that students were learning, but as Lopez et al. (2008) reported, piecemeal progress does not seem to predict inevitable success. The notional machine would seem to be a valuable measure of learning. The notional machine, as a loosely defined theoretical construct, does not include underlying theoretical concepts and thus perhaps has not helped to informed research in this way.

[71] This very problem is why I started TAMP!

Many computing education researchers struggle to identify useful measures of learning and consequently report confounding or conflicting results. The 'best' data seems to come from studies that create new tests for the explicit purpose of research, but every classroom already has a preponderance of data available in the form of course grades. Course grades as a whole (e.g., GPA) are imprecise since students may lose points to absences/tardies or late penalties, and benefit from points unrelated to individual programming skills (e.g., completing surveys, team assignments, extra credit). Students who struggle on tests may show strong homework scores because they had help that overshadows their level of competence, and students who otherwise can program well may test badly due to 'non-cognitive' factors. Studies using course grades should take care to carefully identify which graded activities pinpoint the desired learning outcomes (as an example, see (T. A. Lowe & Brophy, 2017)). Many computing education researchers avoid grades entirely and use student perceptions of pedagogy as a measure of the learning outcomes. Student perceptions are a valuable qualitative contribution to a study[72] , but perceptions of learning may not reliably reflect actual learning. Students may report they learned a great deal because a pedagogy builds a strong 'feeling of knowing' but no demonstrated skill. Students might feel they learned little because the effort required to correct a System 1 error (and the associated Stroop effect (Eagleman & Downar, 2016; Kahneman, 1973)) left them uncomfortable despite learning critical lessons. Any of these measures – targeted grades or student feedback – can contribute to research but will do so better when guided by theory.

The ANM seeks to provide better measures of learning since it builds upon concepts and constructs that already exist in theory. When a researcher seeks to understand the impacts of pedagogical interventions, it is helpful to have an accurate measure that includes intermediate stages of learning. As an analogy, we want to know how much weight a dieter loses, not just measure if they fit into their target size. As Chapter 8 will discuss further, McCracken et al. (2001) attempted to refine their 'all or nothing' measure of success by rating the degree to which students completed their assignments, but the rating system was at best unclear and difficult to replicate. Revisiting the notional machine's definition allows for finer measurements of learning by not only identifying the content, but the type of memories students should form. The ANM goes one step

[72] Single-subject studies often require the collection of such qualitative feedback to ensure the intervention is not just effective, but also ethical and holistically beneficial to participants (Kennedy, 2005).

further by proposing a model of how these granular skills combine to create an integrated set of skills needed for solving problems in programming.

First the ANM helps identify what traditional assessments of programming abilities actually measure. The disconnect between tests of tracing, explaining, and writing (among other skills) does not mean that any of the assessment methods are invalid, but it is a mistake to assume any test measures *the entirety* of the notional machine. Given the epistemology suggested by dual process theory, the full maturity of a programmer's notional machine requires several types of tests.

- Tests of conceptual knowledge typically measure symbolic knowledge[73]
- Tracing can assess enactive knowledge, if measuring the process, not just the results
- Tests that have programmers write code require careful consideration
    - Can the programmer solve a problem using symbolic knowledge alone (e.g., memorize a procedure)?
    - Does the problem follow a pattern that students have seen before?
    - Is the problem requiring creative problem-solving?

Researchers should apply the theory within TAMP to evaluate the nature of the test and the portion of the ANM being measured. TAMP also suggests new ways of analyzing data. For instance, the number of errors novices produce may be less important than how quickly they fix those errors – remember Youngs (1974) reported that experts produced a similar number of errors as novices but fixed them faster. The ANM serves as a theoretical tool for not only dissecting coding behaviors but also reconstructing how each behavior fits into the larger picture of becoming a programmer. One of the limitations of the original notional machine within research was its inability to explain the various ways novices struggle.

Content of the ANM

Sorva (2013) expanded the notional machine beyond a mental model that helps execute code into an engine that aids with comprehension. Ironically, Sorva's expansion of the notional

[73] Testing conceptual knowledge is challenging as even within conceptual change theory there is not a consensus definition of a concept. What I mean here is test that talk *about* concepts, not apply them. Tests such as short-answer or multiple choice. Certainly, practitioners apply concepts in many ways – programmers certainly portray conceptual knowledge when they are building software – but the point of this is to test developing understanding of concepts. In that sense, conceptual knowledge, *knowing that*, more likely falls into symbolic representations, where 'true' conceptual understanding likely requires a combination of all three representations.

machine may exhibit a superbug (Pea, 1986) about the brain. Sorva started with the following definition

> A notional machine is a characterization of the computer in its role as executor of programs in a particular language or a set of related languages. (8:2)

Neither memorizing nor executing programs guarantees comprehension any more than millions of high schoolers memorizing soliloquies from Shakespeare means they can explain the nuance of the bard's words. Sorva suggests that the notional machine should be an explicit objective of teaching, yet teaching the notional machine as traditionally defined only tackles certain types of knowledge. Sorva's expanded the hopes of what the notional machine provides without explaining how. I believe the notional machine helps but is only part of that expansion. Intuition and facts only go so far in problem-solving, hence the ANM, which includes the iconic representations that Bruner believed were critical.

Iconic representations are vital in blending experience with facts to solve complex problems. The iconic representations of the ANM (Figure 7.8) describe transitional information a programmer currently needs (e.g., the algorithm under design, the test case being debugged). Brand new programmers need iconic representations to manage 'new' programming constructs[74]. Before a programmer intuitively understands, for instance, the initialization, advancement, and conditional testing of a `for` loop, their mind needs someplace to store and integrate each of these rules[75]. Iconic representations allow System 2 to solve problems until enactive representations form to speed up and automate programming thinking.

Experts and novices alike need iconic representations to model the assumed behavior of code, either of their own or someone else's design. Iconic representations help in modeling algorithms (or higher-level designs) when programmers are debugging, modifying, or constructing new code[76]. When novices try to explain code by interpreting execution results, they need iconic representations to blend language knowledge with observations about each successive run.

---

[74] Or at least new to them.

[75] Iconic representations are 'one level higher' than short-term memories. If a student remembers little about programming constructs, they may need to look up information and juggle facts in short-term memory. Students at this stage are still forming rudimentary knowledge and thus likely have little support from iconic representations, since they also have few symbolic and probably no enactive representations.

[76] Some novices may use iconic representations in tracing as well, but most do not seem to do so. Tracing uses short-term memory primarily and does not demand understanding or even remembering more than a few lines at a time, thus making iconic representations optional.

Likewise, comparing the use of language constructs across various algorithms helps distinguish a construct's functionality from its role in that implementation. Novices use iconic representation to consciously identify patterns (across languages, design, algorithmic, testing, or otherwise). If a pattern appears frequently enough (e.g., several problems require searching for an item in a list) System 1 will implicitly identify such patterns as enactive representations, as portrayed in the *Enactive (pragmatics)* portion of the ANM in Figure 7.8. These pragmatics represent aspects of programming that are critical in expect programmers but may be implicit in traditional curriculum. They are not aspects of the programming language or the tools, but strategies, patterns, and other such tacit knowledge.

Iconic representations provide the workbench for forging mental models of design, existing or newly created. As discussed in Section 7.5.2, experience changes the way programmers use iconic representations. Experts tend to derive meaning directly by reading code where novices often need to see code running (generalizing from 'code in action' iconic representations). Some of the experts interviewed by Fix, Wiedenbeck, and Scholtz (1993) complained that they lacked "a concrete objective in mind, such as finding a bug or determining the effects of a potential modification" (p. 78), hinting that many experts focus their iconic representations with some particular goal. Experts likely create multiple iconic representations of the same design with a different purpose. After all, they create multiple diagrams to describe different aspects of design (e.g., class diagrams, activity diagrams, state diagrams, and multiple copies of each for different use cases). Educators and researchers must apply the constructs of the ANM to describe their circumstances.

Beyond coding, the ANM includes other mental representations critical to programming, such as those needed for testing. Test cases pair inputs with expected output to ensure the desired behavior. Anyone involved in testing software uses iconic representations to connect facts about the requirements, specifics of the test scenario, and knowledge of the design to fully understand a test case. A programmer also needs to understand the connection between a test case and their code. Executing a test case might only require symbolic knowledge, much like many tracing tasks, but truly understanding the purpose of and building test cases requires juggling many types of knowledge. Iconic representations are vital to many types of programming knowledge but none more so than debugging. When tests fail, the programmer must now juggle what they planned to happen (the code they wrote or how they understand existing code), with the planned behavior

elicited by the test case, with the results of execution. Debuggers often must recheck every assumption, implicit or explicit, and validate every source of knowledge. Experts debug more effectively than novices, who sometimes struggle to debug at all. The ANM offers a model of the experience and skills that, yet again, experts possess, and novices struggle to replicate.

Many of the advantages that experts demonstrate come from 'automated' iconic representations. Each time a programmer searches an error message, explains, modifies, or debugs an algorithm, or creates a test case, they build strategies for similar attempts in the future. Often these strategies become implicit, called upon when needed without conscious consideration, shown in Figure 7.8 as *pragmatics* (enactive). These are the cases when experts can't explain how they do what they do. An expert's repository of patterns helps to explain by just reading code or jump directly from a problem statement to writing code. You can see evidence of the implicit nature of such patterns when a programmer changes language. Beyond the differences in style between two languages (e.g., `firstName` versus `first_name`), the preferred constructs and even algorithmic approach may vary between languages. For example, I typically forget that Matlab allows complex operations within matrices without requiring loops. Pragmatics drive many of the decisions programmers make and explain why two experts with the same training produce very different results.

The ANM provides a set of theoretical constructs that researchers and educators can use for describing how programmers think and learn. Bruner's representations provide a useful abstraction that combines the brain's memory systems (nondeclarative, semantic, and episodic) and mechanisms of cognition (System 1 and 2). So far, this chapter has revisited these ideas and combined them into the theory that is TAMP, but I would be a hypocrite if I left you with a 'set of rules' alone. The rest of this section provides two examples of how TAMP supports building theoretical frameworks that model cognition.

7.6 Applying the ANM

TAMP provides theoretical constructs designed to describe the ways programmers think, not intended to exclude possible relevance into other disciplines, but the validation data primarily comes out of computing education studies. Before returning to 'hard' studies of novice programmers, it is worth a minor departure into literacy. Applying TAMP briefly to an example

outside computing may demonstrate the use of these theoretical constructs. As we dive deeper into specific applications of TAMP, it gets easier to lose the forest for the trees over concerns about my technical description or bias towards/against a specific study. The classification of programming skills into enactive, iconic, and symbolic representations seems to hold value even if you disagree with the way I describe their application to some language construct or design practice. This section starts by modeling the web of skills involved in traditional literacy. Additionally, I believe that literacy education is a powerful analog, if not a literal example, for how novices may come to adopt programming languages, so an example about literacy may provide insights that translate back to programming.

After applying TAMP to literacy, this section compares the way experts and novices read code within the theoretical constructs of the ANM. Learning to read code is a common entry point for novices yet looking at the way experts read code shows how large a gap novices face in learning to even understand an existing program. From there, I jump to the most integrated skill of the programmer, design. When I say design, I mean to encompass the entire programming processes starting with a problem statement to the point the programmer feels they have completed their program. Modeling design cognition briefly touches on a slew of intermediary programming abilities but does not look to capture every aspect of programming. This work only provides a prototype of TAMP's potential, not a canonical model of how programmers think[77]. Section 9.1 will explain further why I feel it important not to deliver a strongly stated model of cognition, but instead the foundations for flexible theoretical constructs that might advance research and teaching practices instead.

7.6.1 A web of skills – The example of literacy

The Applied Notional Machine provides a language for dissecting the mental activities of programming, but an analogous example may help demonstrate how TAMP models cognition. Most people are familiar with the process of learning to read from their own experience or better yet in helping a child. Since Piaget, Vygotsky, and Bruner each discussed literacy education, Chapter 6 also offers several examples of children learning to read and write for comparison.

[77] The "Theory of Applied Mind of Programming" may oversell the current state a bit, but a more confident and comprehensive model is a goal once empirical research starts to emerge to prove some of the assertions made here.

While my descriptions of literacy pedagogy will occasionally be woefully naïve, the intent is to focus on the cognitive aspects of developing literacy as a programming neutral introduction to TAMP. By avoiding programming dogma, I hope to provide an example that demonstrates the theory without requiring complete agreement on issues within computing education. Literacy also provides a topic that heavily relies on new symbolic representations where students must both learn mechanical (reading and writing) and creative (composition) use of the symbols.

A person typically learns to read after becoming fluent in their native language. To some degree, literacy focuses on adopting just the symbolic pieces of a symbolic representation, not necessarily new ideas. Eventually, becoming literate opens a person to new worlds of independent thoughts and ideas, but the ubiquitous books of childhood like "Brown Bear, Brown Bear" (Martin & Carle, 1996) or "Hop on Pop" (Seuss, 1963) hardly expand a reader's horizons. The writing in such books chooses familiar words and concepts to encourage connections between familiar ideas and the written word. Books for early readers intentionally build upon existing knowledge, allowing the reader to focus solely on the system of symbols. Figure 7.9 presents a network of skills and their associated representations of early literacy.

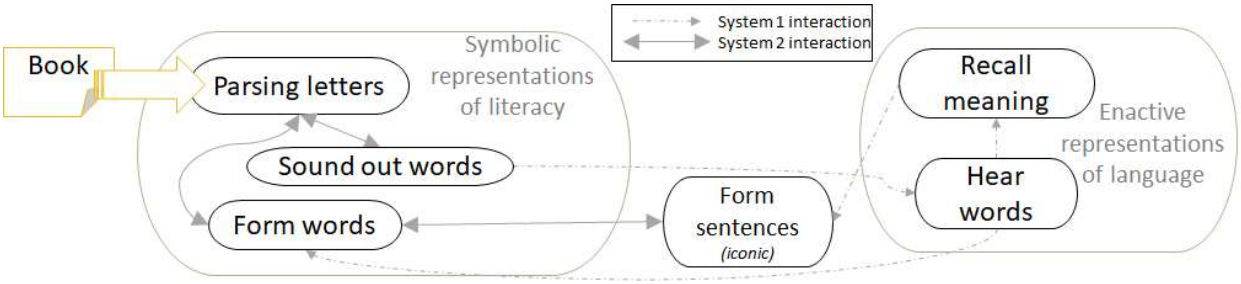

Figure 7.9. Using TAMP's constructs to describe the web of reading skills

A quick note before discussing Figure 7.9:
This section includes the analysis of several mental activities using the constructs of the ANM. The dissection of skills (in this case, learning to read) follows the example of Kurt Fischer's neo-Piagetian theory (Fischer & Bidell, 2007; Morra et al., 2008). Fischer described cognition as a network of interrelated skills that develop at different rates. Fischer's web of skills opens the possibility that Piaget's décalages occur due to unequally developed skills rather than the improper transfer of general cognitive abilities. The breakdown of skills is focused on mental activities, though, not the context of the domain.

TAMP further contextualizes Fischer's web of skills by using Bruner's representations (enactive, iconic, and symbolic) as labeled within the individual or a group of skills. Diagrams like Figure 7.9 also use different styles of arrows to identify when information is exchanged via System 1 or 2. System 1 only delivers information or perform an unconscious action, where System 2 can exchange knowledge between enactive or symbolic representations.

Generally speaking, it is easier to create separate diagrams for novices (e.g., Figure 7.9) and experts (e.g., Figure 7.10). The bad news is in the difficulty of creating a single diagram because experts may have mental structures novices lack and stop using structures novices rely upon. The good news is by drawing two diagrams it is easier to identify the types of knowledge novices must develop and equally importantly, the 'way of knowing'.

A person can leverage existing knowledge, and better yet automated knowledge, when learning to read. When a reader knows the words used in a book, their trained System 1 can support and self-correct System 2 as it goes about reading.

1. Beginning readers need to recognize letters and their associated sounds. System 2 must attend to each letter and remember its sound until practiced enough for System 1 to do so automatically[78].
2. Recognizing letters helps to sound out words. Readers often vocalize unfamiliar words, attempting alternative pronunciations until recognition floods in from System 1. When System 1 hears a familiar word, it provides System 2 with its meaning and the thrill, or relief, of recognition[79]. Instant recognition is one of the goals of some branches of phonic education, where students are taught rules of letter combinations to help sound out words (Ehri, Nunes, Stahl, & Willows, 2001).

[78] Growing automaticity is the likely reason nearly every early elementary classroom is decorated in the alphabet. Merely recognizing the letters, though, does not cover all the phonetic combinations of letters required for reading, though, and English is notorious for varying the phonetic rules of spelling!
[79] Once again, System 1 will mature to automatically recognize frequent words like it once did for letters.

3. Each word adds to the growing meaning of the sentence at hand. Each word recognized in the sentence provides context for the next[80]. The context again helps System 1 predict candidates for the next word and further enhances the recognition process when combined with step 2. A long sentence risks overloading short-term memory and forgetting the context before recognizing the next word[81].

Early reading leans heavily on System 2, which is visible in the pauses, considerations, and focused effort required of even adults learning to read. Memorizing the alphabet does not make a person a ubiquitously perfect reader. Ehri et al. (2001) reported that systematic phonics instruction helps children read better than non-systematic or other approaches. What we consider 'true' literacy occurs when System 1 takes over the task, and we read effortlessly, but reaching that point requires a combination of System 2 and support from other related System 1 skills. Figure 7.9 presents the network of skills for a novice reader, but the representations look different in proficient readers.

The minds of proficient readers have automated and combined rudimentary skills. They read most words implicitly and only barely slow down or struggle with unfamiliar words. The feedback loop for speaking words aloud internalizes such that they can mentally sound out words. Once freed of the need to support every minuscule step of reading, System 2 can turn its attention to comprehension. Figure 7.10 provides a web of skills for a proficient reader.

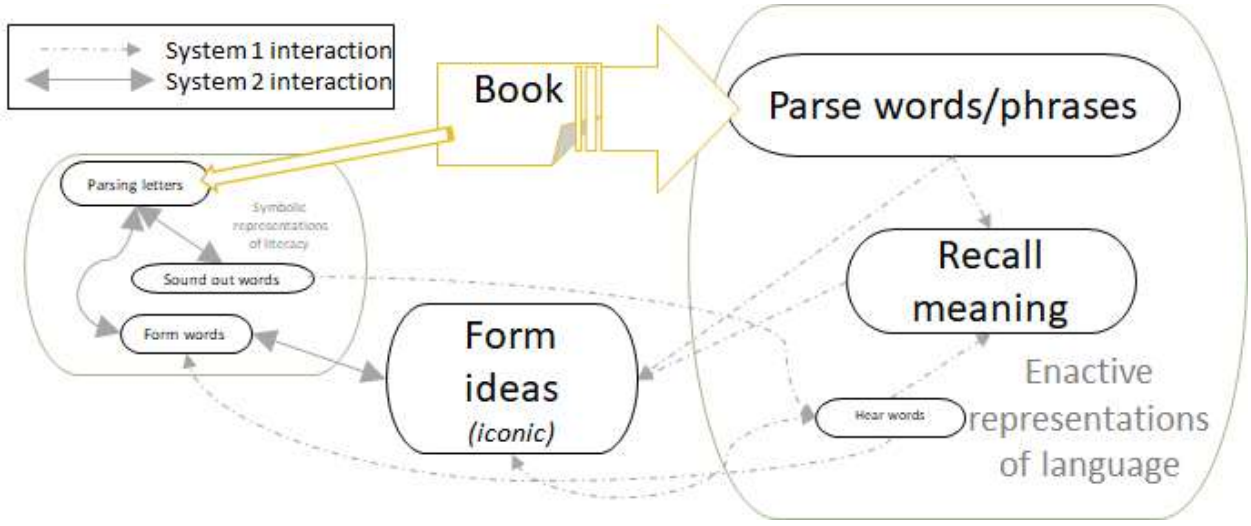

Figure 7.10. A breakdown of the skills of a literate person

[80] Context clues are also a strategy taught by phonics education (Ehri et al., 2001).
[81] This is the reason children books use easy words and are so short! Each time they struggle to recognize the next word they risk forgetting one or more of the previously read words.

With practice, a proficient reader automates the mechanical parts of reading. System 1 drives the reading process using a wide array of enactive representations that recognize familiar words and even phrases directly from the page. Some readers practice until they achieve daunting speed and comprehension, training their minds to process larger chucks or entire sentences. The main challenge when speed reading is comprehension. System 1 will continue to scan words with or without System 2's attention. I am a very fast reader, but my automaticity does not always yield comprehension. If I read something compelling, get tired, or bored my eyes continue to the bottom of a page or even the next page before I realize I don't know what the author said. My System 1 did not stop reading; my System 2 stopped listening. I know my System 1 kept working because when I reread the passage – this time, doubling down on attending to the page –words and phrases are familiar but not their content. Like reading to a child, System 1 (mom or dad) keeps going long after System 2 (the distracted child) has turned its attention elsewhere. Connecting literacy to programming, novices who become proficient at tracing (largely System 1) may not attend to the details of the algorithm. Activities that promote automaticity within the notional machine may not yield equal benefits in promoting comprehension.

The mind's transition from illiterate to literate is not just about adding new knowledge but adding the knowledge in the *right way*. The layouts of Figure 7.9 and Figure 7.10 highlight the diminishing of some representations and the emergence of others. Learning to read is a rite of passage for many people that highlights the differences in novice and expert cognition. Advanced readers still have the option to 'sound out' words, but the need to do so diminishes with the internalization of other skills. In programming, experts occasionally use paper and pencil to trace code, but only need to do so for extremely complicated code[82]. Where beginning readers struggle to parse a word, advanced readers comprehend several at once. Expert programmers comprehend the purpose of code with a glance that many novices struggle to explain code even with the help of visualizations. Experts are not merely better at basic skills; their mind use entirely different cognitive mechanisms than novices.

The example of literacy offers an interesting parallel for learning to program and perhaps puts into context how challenging learning to program can be. System 1 helps new readers predict upcoming words from context or recognize the sounds of words when sounded out. However,

[82] And unusual circumstances, because log files are much more reliable.

System 1 often misleads programmers because of interactions with smart devices or who anticipate the behaviors of code constructs based on the meaning of the repurposed word from English (see Section 2.2.2.1). Like reading, developing automaticity for the system of symbols is a small fraction of the benefits and requirements of reading or programming. The parallels between literacy and programming literacy seem both analogous and possibly literal with further research. For now, this section's goal was to present examples of how TAMP models and contrasts the thinking of experts and novices to demonstrate the gaps novices face in becoming experts.

7.6.2 Programming 'literacy'

One of the first skills a new programmer learns is to read code. New programmers are presumedly literate in their native language (though may or may not know English, which most programming languages use for their keywords), and the number of words they must learn within the 'vocabulary' of a programming language is generally quite limited. Where most spoken languages contain hundreds of thousands of words, programming languages have a few dozen keywords, and novices only typically need another few dozen build-in operations (e.g., operators, input/output, advanced math functions, graphics commands). Foreign language students probably memorize more vocabulary words in the first few weeks than a programmer uses in their entire career. What makes programming challenging is not memorization of syntax or even semantics (symbolic), it is the lack of underlying experience (enactive) about the concepts at work.

Even considering the complexity involved in computing, the major concepts covered in early programming classes are quite limited compared with the range of subjects covered in literacy (e.g., colloquialisms, cultural allusions, business terms, scientific terms). Like authors of children's literature, programming instructors typically try to use familiar problems[83] that allow students to focus on language concepts. Figure 7.11 provides a very rough example of the types of concepts and relationships that often drives programming pedagogy.

The complexity in Figure 7.11 is not in the number of concepts, nor in any single concept, but in their interconnectivity, particularly when lacking any connection to experiences. Like spoken languages, a learner can only go so far by memorizing word meanings. Communication requires a sense of grammar and nuance to word choice. Likewise, teaching programming

[83] They hope at least that students are familiar with, but many times are not, as we will see in Chapter 8.

concepts independent of each other risks building skills in novices that are disconnected from each other. The organization of Figure 7.11 mirrors the language manuals (and traditional textbooks), yet we do not learn a foreign language from a dictionary. Section 7.3.2.2 showed that the brain processes language using several specialized regions that do exactly not match any logical content structure[84]. Children do not study the structure of language until much later in schooling, long after developing basic competency in using language. Why do we teach programming so much differently than we promote literacy?

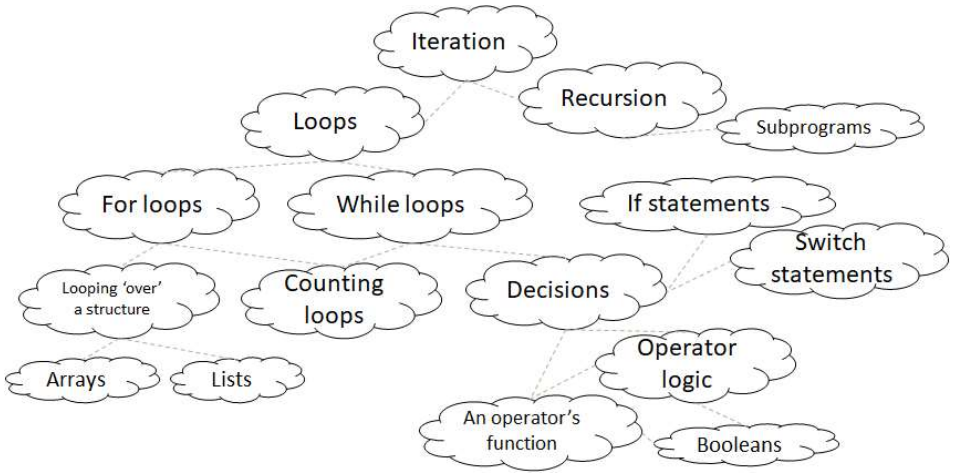

Figure 7.11. A network of skills devoid of the mechanics of cognition

When it comes to the act of learning to program, experts may indeed have forgotten much of the process they took to become a programmer. Squire and Kandel (2003) told us that the formation of nondeclarative memory often comes at the cost of forgetting. As System 1 matures, we forget why we structure knowledge as we do, or just as often, System 2 creates structures that 'feel right' but are disconnected from System 1 operations. Organizing materials according to the inherent conceptual structure, like Figure 7.11, provides a logical organization *for experienced programmers*. Instructors want to help students, so they devise logical dependencies and ensure students learn their lessons in that logical order.

> How can a novice understand a decision if they don't understand operators? Oh, and variables. But to understand operators and variables, you must understand

[84] Our brain does not have verb, noun, and adjective regions per se. Some research shows that people with aphasia may have a tougher time with verbs, but not because a specific 'repository of verbs' is damaged, but because verbs require complex relationships and actions (Alyahya, Halai, Conroy, & Lambon Ralph, 2018).

> data types. And to test any of this code, you have to understand the main method/function/procedure, so the first activity should be "Hello World"

Breaking content down into orderly concepts feels logical, and after all, most programming students are academically strong and often high school graduates or older. By this point, they are clearly operational thinkers, as Piaget suggested! It seems respectful to teach them like professionals, yet such an attitude forgets what it was like to think like a novice. Novices struggle even to read a programming language because their minds process information differently than experts. Modeling how experts think in comparison to novices helps to explain why traditional modes of instruction may miss many students.

7.6.2.1 Experts and reading code

An expert's System 1 is so powerful that they often can't distinguish exactly how their mind works. When researchers ask experts what they were thinking (i.e., think-aloud protocols), their answer is constructed in episodic memory. Since the work of System 1 never reaches conscious thought, experts often say they don't know how they know. When experts explain their thinking, it could be accurate, or it could be System 2 manufacturing a story to explain System 1's reaction (Kahneman, 2011). Unless an expert is aware of dual process theory or one of its counterparts, they may not distinguish *knowing how* from *knowing that*, and System 2 is happy to construct a seeming logical (but possibly untrue) story. As discussed in Section 2.3.3, intuition has been an underlying theme in much of computing education research, but using TAMP, we can begin to identify how by inferring mental activity using what we know about the brain. Figure 7.12 suggests how experts may use intuition when reading code.

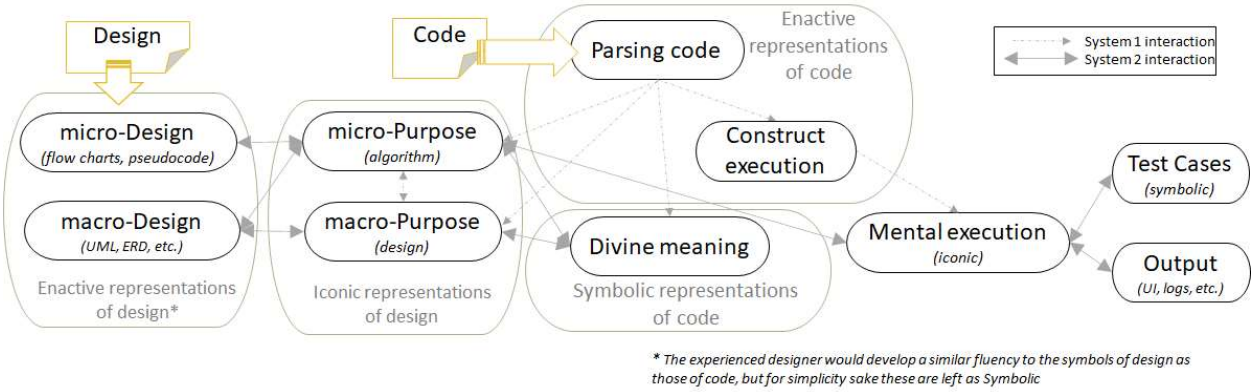

Figure 7.12. How experts read code under the ANM

Experts rely on enactive representations to provide instant recognition allowing them to focus on the task at hand[85]. Figure 7.12 includes two main pathways (understanding the structure/purpose or mental execution) that support the several plausible reasons experts are reading code (e.g., debugging, planning changes, reviews). As Fix, Wiedenbeck, and Scholtz (1993) noted, experts tend to approach reading code with a purpose (the reason some were confused when asked to study code with the only purpose of remembering). That purpose directs System 1 in gathering the desired knowledge using enactive representations that relate to similar experiences.

Like a strong reader, experienced programmers seem to understand code more profoundly than novices reading the same page. A glance might provide the framework of an algorithm, design, or even execution results from code. When experts encounter something unfamiliar, they can turn to a deep repository of symbolic knowledge, yet do so in a targeted way, since System 1 activates the most promising facts for System 2's consideration. Even when they must seek more information on different aspects of programming, experienced programmers hold an advantage over novices. The expert's System 1 helps to ignore unrelated facts and direct their search to the most promising sources and relevant results.

7.6.2.2 Novices and reading code

Ironically, novices probably read code using the exact methods experts expect, toiling from concept to concept, rule to rule, their System 2 trying to hammer out meaning. Unlike new readers, new programmers have little, no, or worse, misleading intuition for what to expect from the code. Where a new reader can sound out words and gain helpful feedback from System 1, novice programmers have no internal source of reliable feedback. The feedback they receive is primarily symbolic (e.g., log files, error messages, tracing tables), adding further burden to System 2 to process the meaning of the output and connect the results with the code. In summary, for many first-time programmers, they are asked to

1. Look at mysterious symbols that are both recognizable and foreign

[85] This is why experts are often annoyed by badly formatted code. When code fails to conform to standards, System 2 must engage more frequently. By following standards, at least when the code is indeed routine, later readers can spend less effort when reading. By the same token, when something needs to stand out, the best way to ensure people attend to special cases is to intentionally break standards. Standards are not always helpful!

2. Connect these 'magic words' with unseen consequences
3. Mentally recreate the steps of an algorithm precisely, where any tiny miscalculation can be disastrous
4. And along the way make sense of what it all means

Much like the experts, the purpose of reading code is not always clear to the novice. Example problems sometimes are familiar, but just as often they are baffling (e.g., What exactly does the formula, `A | B & ~(xor(A, B)) == B`, mean?). Du Boulay (1986) noted the importance of showing novices programming's "orientation" (i.e., the value it provides). People learn to read for entertainment and knowledge, but tracing and explaining code seems to serve little purpose beyond learning syntax and semantics[86]. Learning to read or program are each cognitively demanding, yet even the most illiterate person has innate advantages and motivators that may not be as clear to novice programmers. It is interesting to compare the mental representations of a novice programmer (Figure 7.13) not only to those of an expert (Figure 7.12) but to that of a novice reader (Figure 7.9).

As within the mind of a new reader, System 2 must recognize each programming language construct (word) and how the algorithm emerges from the surrounding constructs (phrases/sentences). Unlike a new reader, System 1 does nothing to validate the resulting structure. Instructors mostly ask novices to read code either for tracing or explaining its purpose[87]. Unlike experts, novices must rely on System 2 until System 1 is mature enough to take over and are probably better served double-checking System 1 for quite some time. Tracing at least limits the scope of what must be remembered down to a few lines of code. When the problem is familiar, novices can use sketching to take notes on the values of variables to alleviate cognitive load. Explaining code may not have such advantages. As Fix, Wiedenbeck, and Scholtz (1993) reported, novices are similar to experts in explaining simple patterns of code that require little to remember, but show significant differences explain complicated patterns. With fledgling symbolic and

[86] Not to say these are not important, but they hardly feel productive for new students. As I think about it, I never saw or considered using tracing exercises with experienced coders learning a second language. Either I missed out on a useful pedagogy or these tasks are only good for first-time programmers?

[87] It is worth noting, many instructors provide worked examples or the equivalent to students, essentially acting as a More Knowledgeable Other (MKO). I believe that watching an MKO perform the tasks helps to form enactive representations, but not nearly a much as actively participating. Thus, examples help to establish familiarity that might primes the appropriate symbolic memories when needed, but true enactive representations require interactive experience.

insignificant enactive representations, novices seem to struggle to form the vital iconic representations required to understand, much less manipulate complex code. Experts do not merely know more than experts; they call upon mental mechanisms and different ways of knowing than novices.

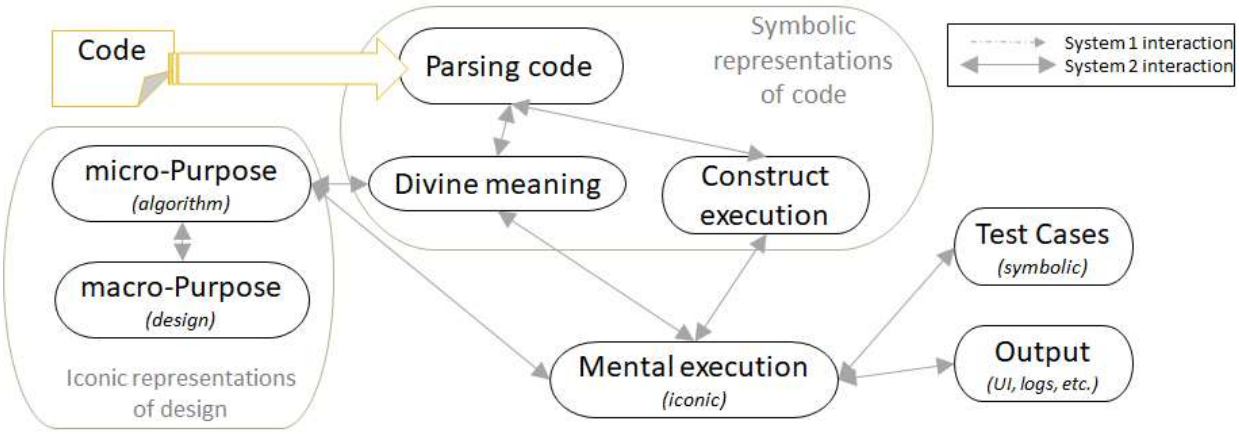

Figure 7.13. The representations a novice requires to comprehend code.

7.6.2.3 Comparing how novices and experts read code

While the models of expert and novice cognition during reading are inspired by TAMP, computing education offers several studies that support such comparisons (Bednarik & Tukiainen, 2006; Brooks, 1983; Fix et al., 1993; Floyd, Santander, & Weimer, 2017; Wiedenbeck, 1985). In one such study, Bednarik and Tukiainen (2006) used eye-tracking technology to compare the visual focus of experts and novice during a code comprehension task. Participants evaluated three short code segments within a code visualization tool that could sketch each step of the code's execution. As Figure 7.12 predicts, experts' eyes focused on the code, and only after reading the code turned to the visualization.

> In our experiment, more experienced participants read the code first, created a model and hypotheses, and then confirmed their hypotheses by (usually) only one run of the animation. (p. 130)

Experts glean most of the useful information they need from System 1 and fill iconic representations as needed. The visualization, as the authors suggest, serves as System 2's confirmation of a job well done.

Bednarik and Tukiainen's novices took a radically different approach according to the eye-tracking data. Novices "did not read the code at the beginning, but instead animated the program

several times and let the tool to visually explain the execution" (p. 130). Given the option, the novices in this study avoided engaging with the source code until after they developed an intuitive feeling for what the program did from the visualization tool. For the novice, watching the visualization is easier than engaging with the unfamiliar code. For the expert, reading the code is not only familiar and easy but probably more so than deciphering the unfamiliar visualization tool[88]. Each group of participants seemed to embrace the tools that required the least of System 2 – novices built iconic representations by watching the execution to help understand the code, where the experts' System 1 jumpstarted iconic representations, which helped experts validate their model and make it easier to understand the visualization.

Unfortunately, Bednarik and Tukiainen's protocol did not verify the participants' comprehension. We know neither the quality or when in the sequence, each group finalized their model of the code's purpose[89], only where they focused their eyes and for how long. TAMP would predict the same outcomes that Fix, Wiedenbeck, and Scholtz (Fix et al., 1993) reported with novices only remembering surface details in comparison to experts. The visualization tool may actually overly focus on algorithmic methods and overload a novice's ability to generalize structure design details that better explain a code's purpose. Experts begin to populate iconic representations of design as they read, where novices would need to consciously turn System 2 away from line-by-line traces to determine the purpose of each subprogram and their duties and sequence within the full program. Since novices spend most of their time considering low-level language constructs, they also have yet to develop the habits of mind to consider questions of design.

7.6.2.4 Design documents and reading code

When design documents exist, experts likely benefit from reading them more than novices. Figure 7.12 includes an additional source of information, the design documentation, which is not included in Figure 7.13. I debated whether to include design as a source of information for experts, deciding to include it as it may spark further research in this area. Based on theory and anecdotal

[88] Bednarik and Tukiainen noted that the experts had little experience with the tool, but seemingly no less experience than the novices. Novices still chose to engage with the tool. I would guess this is because all things being equal, at least the tool was dynamic where only System 2 can decipher static code.

[89] The study also used graduate students as 'experienced' programmers. Not to impugn the expertise of graduate students, but it would be interesting to study full-time industry programmers whose primary duty is programming.

experience, I would guess that most programmers only selectively read design documents. I have heard (and said) that design documents are not to be trusted when the source code is available since they are usually poorly maintained, and it is just as easy to read the code. TAMP contends that since experts look to minimize the efforts of System 2, reading code not only offers the most accurate model but removes the task of reconciling iconic representations of the design and code. If the design says one thing and the code another, the code is obviously right, so why spend the time to determine if the design is accurate at all[90]?

One subtler point TAMP suggests is experience makes designers more comfortable in reading/producing design documentation. Since I spent years teaching object-oriented analysis and design, I feel equally fluent in UML and source code, but I can produce diagrams in 15 minutes that my peers claim would take them hours. Novices, on the other hand, likely have significantly less experience dealing with the alternative symbolic representations of design. While UML, for example, is less syntactically finnicky than source code, students not only spend less time building UML than they do code, and the feedback on the quality of design is less tangible. Designs do not provide any direct feedback – they neither execute or give error messages – so student must rely entirely on feedback from others or build the proposed design in code. Teaching design may feel simpler because of its iconic nature (use of summary pictures), but standardized design notations are still symbolic representations. As Bruner noted, it is easier to learn symbolic representations when the learner already understands the concepts the symbols represent. In my experience, novices eschew design (and even comments) and prefer coding as to minimize the number of different representations they must manage. The rule of thumb about design might be *programmers choose to document whatever minimizes System 2's workload by relying on the task System 1 performs best*.

If experts are occasionally reluctant to engage in design documents, novices probably find little use in them when reading code. Unless they can read design documents/diagrams fluently, design documents add yet another symbolic notation that novices must reconcile. Cognitive load theory (Plass et al., 2010) warns against presenting students with multi-modal data in the same learning experience, as discussed in Section 2.2.3. TAMP explains this overloading as an additional burden to System 2, requiring the mind to juggle new symbolic representations within

[90] This is another argument why many programmers prefer not to document long-term designs.

existing iconic representations. Until novices are confident in code, design, execution results, tracing tables, or whatever other sources of information they may have about a program, they may waste effort in determining if these various sources of knowledge agree, and if not, which is in error. TAMP suggests that teaching novices design notations may not make it easier to learn to program. While design notations typically use imagery, they are not iconic since the generalization comes externally rather than from the novice's experience. Learning design as an easier-to-learn symbolic system does not circumvent the need to create – and may distract from – creating interconnected enactive, iconic, and symbolic representations.

7.6.2.5 A brief look at how to improve pedagogy

Novices cannot become expert programmers merely by collecting knowledge. Even looking at the simple task of reading code, novices (Figure 7.13) do not become experts (Figure 7.12) through a linear path. Thinking like an expert requires building many enactive representations of code through deliberate and varied practice, as well as the habits of mind within Figure 7.8's pragmatics. Chapter 9 considers the pedagogical ramifications of TAMP in more detail, but before leaving the example of reading code, it is worth considering a few of the common pedagogies that novices see in these early stages of learning to program.

Asking novices to trace code is an easier method of maturing the notional machine than the network of skills required to write code. The most common tracing task presents a novice with inputs to a sample of code, asking them to determine the output. Unless specific technology drives the tracing tasks, the novice does not receive definitive feedback until the end of the process[91]. Waiting until the end means any misstep is up to several minutes (hour or days, if relying on grading) in the past. Even then, unless the tool/grader identifies the misstep, the student can only identify their mistake *by tracing their incorrect tracing* to find the error in their thinking. They must essentially ***debug their notional machine***. If they kept meticulous notes, they might be able to revisit each step, or more likely trace a second time and compare the two attempts. The novice might find a silly mistake on the second (or third, or fourth) trace, yet such mistakes are not

[91] The novice would need to be intimately familiar with the process realized in code to gain the same benefit as sounding out words. If the novice must split their attention between predicting each intermediate step of the process and the code that automates the process, they risk overloading System 2 or confounding their tracing with what "the code should be doing" according to their understanding of the problem.

misconceptions merely human errors that computers would not make[92]. Without the help of a more knowledgeable other, novices essentially must backward propagate from the correct answer until they find their error – the mirror image of the process they are learning. Backward propagation is quick within a GPU-driven machine learning algorithm (and System 1) but quite effortful on traditional processors (and System 2).

More likely than working backward, students will repeatedly start over from scratch or just move to the next problem, as Perkins et al. (1986) observed. Starting over is more natural than reverse-engineering a mistake as System 2 can restart its ritual, varying its execution (much like a decision tree) in the hope it eventually leads to the correct answer[93]. Even if the novice identifies their error in this manner, it may not help mature System 1, *which does not learn explicitly*. System 1 is just as likely to pick up bad habits from all the failed attempts as it is to recognize the proper approach from the single successful instance – possibly explaining why novices often continue to make the same mistakes even after learning the correction.

In dissecting tracing, I am not looking to call its efficacy into question, so much as explain why some students do not seem to learn by tracing. I do believe there is a better option than tracing, as introduced in Chapter 2, but I am not saying that tracing exercises are either harmful or futile, merely limited. Reading instructors assign short, simple, familiar books not solely because they tie to experience, but because they lend to deliberate practice. Programming instructors should consider similar heuristics in choosing tracing tasks. Creating complex examples that engage and challenge students may accidentally encourage either 'the wrong practice' or shortcutting to the 'right' answer by any means possible, thereby skipping the practice entirely.

Tracing is not the only highly useful pedagogy with potential drawbacks[94]. Through the lens of TAMP, many pedagogical innovations serve to scaffold a novice's fledgling System 1. Parson's problems (Section 2.3.1) divide code into meaningful 'chunks' that the novice order into an algorithm. Parsons problems straddle the line between "explain in plain English" and tracing tasks. They encourage considering the purpose of chunks of code, yet unless carefully designed are solvable from context clues, a basic understanding of semantics, or even trial-and-error rather

[92] The point of tracing is not to pit human mental execution against that of the computer, but to promote familiarity with the language. Simple mistakes only matter when the programmer writes them, and then the important test is can they debug such mistakes.

[93] Assuming the novices understands the points in the code they have a choice in execution!

[94] Instructors must often select less-than-ideal pedagogy. Knowing its strengths and weakness helps to emphasize what is good and avoid the pieces that are miseducative, to use Dewey's (1938) term.

than demanding comprehension. If too simple, Parson's problems become trivial. If too involved, they could either be fruitlessly difficult or allow students to ignore the algorithm and focus on the few syntactically legal combinations. It seems Parson's problems must fall into the 'goldilocks zone' of being just right and require instructors to take care when creating.

Subgoal labeling (see Section 7.3.1.1) challenges novices to add purposeful labels to worked examples to emphasize the purpose of code. Subgoal labeling lessons typically start by watching an experienced programmer complete some task while adding subgoal labels as they work or reviewing code just to add labels. Adding subgoal labels draws attention to elements of design, and particularly for advanced novices, provides a reason to attend to a demonstration that otherwise covers basic concepts they already understand (possibly countering some of the expert reversal effect). True novices may not have the System 2 bandwidth to attend to both the basics and the design elements, though. A major limitation of subgoal videos is the students are inherently passive. Assigning labeling tasks results in similar problems where the feedback comes from an external source and may be either temporally detached or only useful after further analysis.

Learning to read code is significantly tougher than memorizing syntax and semantics, in many ways, more akin to learning a first language than a second. Students who fully embrace metacognitive reflection while completing any of these pedagogical interventions will likely excel but possessing such study habits likely makes a strong student under any pedagogy. The goal of TAMP is not to draw any pedagogy into question, but to emphasize when and how teaching practices help students as well as identifying gaps in what each pedagogical intervention promotes (i.e., types of knowledge from Figure 7.8). Only further research, as described in Chapter 9, can definitively show where TAMP is accurate and how students benefit from any teaching practice.

7.6.3 Applying the ANM - Design

Reading code is a starting point, but beyond that, it is not clear that instructors can create an 'optimal' path to teach programming. Is iteration or recursion a 'more natural' way of applying logic? It depends on the language, problem, and background of the student. Should we teach arrays before loops? Putting arrays first would provide better examples for using loops, but how then do we demonstrate the proper way to use arrays? Such questions miss the inherent interconnectivity of the subject, and more importantly, the way we think. Humans rarely construct

knowledge, outside formal education, in such orderly concepts. Most of what we learn outside the classroom starts from messy interconnected ideas that we eventually learn to disassemble. Did you learn to play baseball or hopscotch by sitting down to a structured lecture on the rules[95]? The piecemeal approach has its merits but also fails an embarrassingly large number of students.

Computing education is full of stories where students show an understanding of individual concepts yet fail to pull them together to design, write, test, and debug programs.

> Careful retesting revealed that syntactical knowledge was not the problem. Students who did well on multiple choice tests could successfully "hand trace" syntactically correct code, spot syntax errors in incorrect code, and successfully implement a detailed algorithm given in simple English. The difficulty was the inability to form the algorithm in the first place. (Beaubouef & Mason, 2005, p. 104)

Beaubouef and Mason told a sad tale where, despite demonstrating learning, student are not seemingly translating what they learn to make it past the early stages of learning to code.

> At our university, there are over four hundred declared majors in Computer Science. Each semester, however, only about fifteen to twenty students graduate in this field. (p. 103)

TAMP has many potential uses, many of which are refining how we currently teach and research programming practices, but it also has the potential to rethink how we train new programmers.

Revolutionary ideas still need some level of grounding in evidence. Rather than proselytize a new curriculum based on anecdotal faith, I sought to understand how people learn, leading to how people think, that in turn led to a gap in the theory that drives how we measure learning in new programmers. Two years and four-hundred-odd pages later, I have a few answers, many more hypotheses, lots of data from the literature, but no definitive proof. I know what I would do if I were to start a programming class from scratch, but I am not ready to pronounce it superior or even replicable. The missing link, which anyone who teaches software engineering should know, is defining the desired end goal. Despite all of the literature reviewed in Section 2.1 describing what we want programmers to learn after their first year or so, I am not sure we have a high-level (and certainly not a low-level) understanding of what that means to be a programmer. Rather than using a distinct list of skills, assessing when a person becomes a programmer is like United States

[95] I am still not clear about the rules of hopscotch, though I am an encyclopedia of baseball rules and I never played in a single formal league or read the rules until I volunteered as an umpire a few years ago.

Supreme Court Justice Potter Stewart's famous phrase, "I know it when I see it" (Lattman, 2007). So long as companies are glad to hire our graduates, we may not be failing as a profession, but we may be failing capable students, but we may not understand how to promote these unnamable skills.

The purpose of this section is to create a strawman description of the mental structures required to be a competent programmer, specifically design[96]. Software design is not just a vexing aspect of programming for many students, but conveniently it also models most mental activities programmers use in other aspects of solving problems. This section starts by revisiting the essential role iconic representations play in design and establishing a scope a definition for what I mean by software design. Intuition plays a strong and generally intangible role in design but TAMP provides the tools for modeling the role of intuition in expert thinking.

7.6.3.1 The importance of iconic representations in design

Strong designers need a robust and integrated skillset. Bruner (1966c) claimed that *knowing about* a subject is not always enough to solve complex problems, so building strong iconic representations is critical.

> For when the learner has a well-developed symbolic system, it may be possible to by-pass the {enactive and iconic} stages. But one does so with the risk that the learner may not possess the imagery to fall back on when his symbolic transformations fail to achieve a goal in problem solving. (p. 49)

I keep returning to this Bruner quote because it seems to get at the heart of what confounds novice programmers. They have a wealth of information but stored in a way that makes it inaccessible when most needed, and it is not their fault. Students, being human, are victims of our shared cognitive architecture that prefers to preserve the hard work only when required (i.e., System 2 is lazy). Students may take the path of least resistance because they are young and foolish, or because System 2 is generally lazy. Bruner suggests it is up to instructors to build curricula that encourage all three types of representations.

[96] By defining the end goal of the ideal programming skillset, it is possible to work backward to rethink the best path in teaching programmers. A new curriculum would likely preserve of many current and past practices, but perhaps not in their current or past form. In choosing design, I am not looking to describe any specific degree program or full skillset, merely a person who can take a problem and code a solution, as will be detailed more in a bit.

Bruner seemed to indicate students who learn facts they cannot connect to experience are less flexible thinkers. His discussion after the quote above turns to the instructor's choices within the order of instruction.

> Exploration of alternatives will necessarily be affected by the sequence in which materials will be learned become available to learners. When the learner should be encouraged to explore alternatives widely and when he should be encouraged to concentrate on the implications of a single alternative hypothesis is an empirical question (p. 49)

The way an instructor chooses to present information influences how students later make decisions. Bruner's pedagogical discussion mirrors that of Perkins et al.'s (1986) stoppers and extreme movers. How do we encourage stoppers to consider alternate options rather than giving up on a single failed attempt? Can we convince an extreme mover to slow down and explore a problem more thoroughly? Designers need to decide when to evaluate and when to move on from a failing design. Bruner does not provide clear-cut guidance but does place the iconic representation at the heart of such reasoning. If iconic representations are critical to flexible problem-solving, and flexible problem-solving is key to design, it follows that iconic representations are key to design. This section explores what roles iconic representations play in the cognition of programmers while designing.

7.6.3.2 What is design

Before diving into the mental representations of design, it is important to capture what I mean when saying design. The traditional software development lifecycle (SDLC) includes phases such as requirements, *design*, code, and test, occurring either as sequential activities (a.k.a. the waterfall approach) or with some iteration between the stages. Through the eyes of the SDLC, design is an isolated and distinct activity that happens somewhere after defining the problem and before writing code, possibly incrementally. Some programmers hold the extreme view: *programmers shall write no code until completing the design*[97]. Completing design prior to coding seems to have stemmed from manufacturing where the cost of raw materials and production time are sizable investments

[97] On one such project, a manager chastened my choice to reverse-engineer UML diagrams from code rather than using the slow click and drag interface. Writing an incomplete framework of code to produce diagrams more quickly was "cheating the process" in their eyes – certainly, an extreme view, but one that follows the waterfall process quite literally. It seems very few companies follow this approach these days, but the waterfall ideas still linger.

compared to the cost of mental planning but made their way into early software processes. In time, software engineers proposed new methodologies, such as Extreme programming or Agile, that embrace incremental design. Processes like Agile do not eliminate the design phase but allow the design to emerge incrementally, and some projects choose limited (or no) formal documentation. Even iterative projects still hold a limited view of design compared to what I am proposing.

The traditional view of design is unhelpful in modeling cognition since design choices are omnipresent and never-ending for programmers. While programmers make major decisions in the timeframe dedicated to 'design', they continue to make equally impactful, if smaller decisions until the product is delivered. Rather than hashing out whether the cognitive activity of design extends beyond a specific time period, I am embracing the description of design proposed by Socha and Walter (2006). They argue that software design is different from design in other fields, and the true artifact from software design is the delivered source code.

Socha and Walter stress the need for flexible thinking and responsive engineering when building software products. Software developers are not even designing a digital product so much as they are defining a *complex adaptive system* (CAS) that includes software but also encompasses some human process(es). Unlike physical systems, they suggest code has essentially no production or delivery costs, allowing for nearly instant feedback from users in a way that may take months or years in other products.

> As current development systems make it easy to start generating code (or is it design?), the software developer tends to jump in and start producing something without incorporating any explicit design methodology. Furthermore, the value of an up-front design is less compelling if CAS theory implies that we cannot understand what the design will produce until we execute the design. (p. 543)

The tools of programming allow us to skip traditional levels of abstraction (e.g., sketches) since such abstractions are less useful in determining the inevitable quality of the product. Since part of the measurement of the quality of a software product is the interactions between humans and the system, it is difficult to predict before finishing code. The sociotechnical feedback is critical to the long-term design than that of the predefined structure of code. Socha and Walter certainly extend design well beyond planning the efficiency of memory and throughput or even writing code that is easily maintained. Design, by their definition, encompasses all aspects from problem definition through verification and validation.

Even if you disagree with Socha and Walter, the expanded view of design is helpful when talking about how programmers think. Especially if you advocate that programmers should conceive much or all their intent before starting to code, it is vital to promote design thinking apart from coding. The design of software must consider not only the requirements and underlying technology, but the design's testability, maintainability, and ease of troubleshooting. Design always must account for the full SDLC lifecycle. Regardless of how and when a programmer completes their design, the mental constructs I am proposing remain the same.

7.6.3.3 The role of intuition in design

System 1's contribution to programming is not limited to automating the notional machine but plays a vital role in all aspects of thinking, including design. Remembering the alternative name for System 1, The Autonomous Set of Systems (TASS), helps to frame design as a sequence of many types of tacit knowledge. Before capturing the specific types of enactive representations (i.e., TASS processes) within the mind of a trained programmer, it is helpful to remember how we form and use intuition.

Different types of intuition

The way expert programmers think is far from homogeneous, and the body of knowledge that is computing is far from monolithic. Fix, Wiedenbeck, and Scholtz (1993) noted that experts "are not distinguished from a novice along a single dimension or just a couple of dimensions" (p. 78). Expert thinking uses different sources of knowledge and analysis techniques than novices, but instructors typically do not teach these sources explicitly.

> The results suggest that a number of skills contribute to the formation of the mental representation, for example, skill at *recognizing basic recurring patterns*, skill at *understanding the particular structure* inherent in a program text, skill at *recognizing the links* tying the separate program modules together, etc. (p. 78, emphasis added)

The specific skills Fix, Wiedenbeck, and Scholtz pointed out in their summary all seem to be behaviors of System 1. Pattern recognition, perceptually grouping text, intuitively associating links are not only attributable to System 1, but the speed at which experts perform these tasks hint at automaticity. Wiedenbeck (1985) demonstrated the superior speed that experts demonstrate in

the similar task of recognizing syntax errors (identifying non-conforming patterns) in an earlier study. Fix, Wiedenbeck, and Scholtz (1993) suggested that experts had an easier time with some questions because they know more about programming, thought TAMP suggests much of their additional knowledge was actually nondeclarative and contributed to their perceptual advantages as well. Several of the questions that novices struggled to answer were "based on information readily available in the program, yet novices do not extract it," presumedly because novices "are using a different program comprehension strategy than experts" (p. 78). TAMP suggests that experts are not using an explicit strategy of comprehension, rather they are implicitly better at filtering and organizing the information they are perceiving.

Chase and Simon (1973) captured the perceptual advantages chess experts hold over novices (see Section 4.2.1.1). Chess pieces, like coding constructs, have symbolic and semantic meaning. Each piece represents its allowed moves – the 'castle'/rook can move any number of spaces vertically or horizontally where the 'horse'/knight moves in its funny L-shaped motion. New chess players must internalize the allowed motion of each piece, yet knowing the legal moves does little to suggest fruitful strategies any more than memorizing `if-else` semantics does not ensure problem-solving with code. Playing a strong game of chess requires knowledge beyond the rules of the game, it requires developing a sense of strategy.

Strategy in chess revolves around creating advantageous board positions. My favorite tactic was always positioning knights to simultaneously imperil two valuable pieces creating what is commonly called a 'fork' – the opponent must choose which pieces to lose. Finding and avoiding forks requires novices to carefully analyze the board seeking/avoiding the spaces a knight could introduce a fork. As a youth, I spent hours poring over books on chess puzzles that presented such opportunities/threats, such as shown in Figure 7.14. The typical chess puzzle shows a mid-game board position, declaring the desired outcome in some number of moves (e.g., checkmate in 3 moves). At first, I poured over each piece and the possible moves (also having to imagine the opponent's potential responses) using System 2 to construct potential board positions (iconic representations that rely on my enactive knowledge of how pieces move). In time, the puzzles became easier as I began to recognize the most desirable/threatening board positions. Novices learn quicker using chess puzzles over just playing games from the start because the puzzles narrow attention to finding strategic positioning. Nearly half a century ago, Chase and Simon (1973) captured the main advantage of chess masters, a library of familiar board positions, which

we can now say resides within System 1. Not only does System 1 recognize configurations of pieces, but it offers a *desirability measure* that helps direct strategy. A glance provides a chess master with a sense of threats and opportunities. The experts in Fix, Wiedenbeck, and Scholtz's study also seemed better at identifying and remembering patterns they saw in code.

Figure 7.14 An example of three steps in a chess puzzle, including possible positions of a knight

If intuition is a vital source of strategy in chess, it stands to reason that experts in other fields garner similar benefits from intuition. System 1 seems to provide experts with two distinct advantages: procedural automation and perceptual filtering/pattern recognition. System 1 internalizes the rules of the game (e.g., chess moves or semantics of language constructs) within enactive representations, sometimes overlooking or forgetting the conscious rules along the way. Such procedural automation offers programmers speed and accuracy within their notional machine and tasks that require mental execution (e.g., tracing, some aspects of debugging). Design, in the sense described in Section 7.6.3.2, requires programmers to go much further than selecting individual lines of code, they need strategies. Experts seem to call upon a repository of patterns and strategies that novices struggle to recreate within System 2 alone. One of the major gaps between novices and experts that pedagogy must tackle is how to encourage the formation of the types of knowledge to support strategic thinking.

Building intuitive strategies

Within Bruner's three representations lies a hint at how inspiration and logic combine in a designer's mind. TAMP's refined definition of the mechanics of iconic representations offers explanations of how intuition influences and supports reasoning during creative planning. The

challenge to educators is capturing how such knowledge forms. System 1 internalize procedures and automates how we communicate with others. Take, for example, the implicit acquisition of grammar. Compare the following two statements.

> rectangular green old French silver lovely little whittling knife (mine)
> *versus*
> lovely little old rectangular green French silver whittling knife (M. Anderson, 2016)

Was one line easier to read/comprehend than the other? Did you know that a grammatical rule governs the ordering of adjectives[98]? Do you think of the rule any time you choose multiple adjectives? Common advice to aspiring writers is to read more. In reading more, we train our System 1 to recognize patterns, and as such, become more likely to produce similar patterns in our writing.

Merely reading about grammatical rules only changes habits when System 2 has the bandwidth to monitor the parameters of such rules. I can attest to this personally, as the sections of this work that needed the most editing were also the most creative. It was easier to easier avoid or at least catch my habitual mistakes when writing sections that simply summarized familiar ideas that required little planning. System 2 can use symbolic representations to curb our behavior (enactive), but only when our planning (iconic) is not all-consuming. Similarly, expert programmers, having automated much more than novices, can learn faster, as is commonly noted when programmers learn multiple languages. Expert coders who are also educators may underestimate the learning curve novices face since they are more likely to remember learning a second or third language, that was much easier than learning their first.

If System 1 drives how we translate ideas into words, it seems reasonable that mechanisms govern our creative expression in coding. The adage "think before you speak" seems to be an apt description of how we communicate. We often choose words implicitly and only recognize just before (a pause), during (a stutter or typo), or after (a restatement or edit) does System 2 step in to alter what we communicate. System 2 monitors rather than drives many of our expressions[99]. It seems much of our decision making happens in a similar manner. Damasio (2006) suggested that

[98] According to Anderson (2016) the explicit order is "opinion-size-age-shape-color-origin-material-purpose"
[99] Remember the discussion of Wernicke patients in Section 7.3.2.2 who thought they were communicating could not understand themselves or others.

our unconscious mind preselects good ideas (and discards bad ones[100]). It seems Damasio's preselection is a byproduct of System 1 jumpstarting System 2 by priming important associated memories to the current train of thought/perception.

Designs (and memories) rarely spring to mind fully formed but rather are constructed. We may rehearse a future conversation using memories of similar conversations (e.g., what will my committee ask me in my defense? Let me think about what other audiences have asked me!). I believe, as discussed in Section 7.3.2, the mechanisms of prospection are useful in describing the mental activity in design, and enactive representations are critical. Not only does System 1 enhance the mental simulation of prospective designs, but as described in Section 7.6.3.4, it stores patterns of design that inspire designs. Why do some excellent programmers choose loops while others choose recursion? While they may have wonderful justifications for their choices, the answer probably is just as much habit. They may have had a professor or manager or programming language who encouraged/demanded one over the other. Why did my coworkers insist that debit cards were impossible to integrate (section 4.2.1.2)? I believe it was too much time working on the same types of problems that influenced their thinking (they understood perfectly once the inspiration hit). TAMP suggests that intuition (System 1) may be equally critical as strong analytical and planning skills in creating strong designers, but identifying the type of intuition, the specific repositories of enactive knowledge, that is important in preparing novices.

7.6.3.4 Mental representations of design

The constructs within TAMP describe expert knowledge and thinking as more than a mysterious black box. So far, important propositions include:

- Iconic representations capture the 'here and now' of conscious design reasoning
- Enactive representations describe tacit knowledge acquired through experience
- Symbolic representations incorporate 'new' information from external sources
- System 1 explains the mechanics of inspiration and fast processing

[100] It seems more reasonable that the 'quality' of thoughts is irrelevant to System 1. Like an artificial neural network, the inputs (perceptions) results in an output (priming thoughts) that at best come with some strength of confidence (feeling of knowing). It seems unlikely we are subconsciously cycling through options and weighing them, but rather the slow training of System 1 has done that already. Multiple options that come to mind do so when our System 2 seeks alternatives because it is dissatisfied with the first prime, and in doing so changes the inputs to result in a different idea (or not).

- System 2 drives deliberation and makes decisions in novel circumstances

Given these five constructs and their value propositions, the next stage in building TAMP is looking at the cognitive activities of programming. It is important to note that I will not tackle programming at its lowest level at this time. Dissecting the pieces of language constructs (e.g., `if`, `else`, and `else if`) does little to illuminate the mental representations of experts like drawing a diagram aids little in tying your shoes. Most adept shoe-tyers only consider the act of tying their shoes (writing code), checking if their shoe is tied well (reading code), and dealing with unfortunate knots (debugging). They have little need for 'knot theory' so long as they have a way to keep their shoes on their feet. Rather than dictate the "right way" to teach each concept, I will consider the abstract categories of knowledge and thinking that goes into design[101].

Figure 7.15 describes the mental representations at such a level of abstraction used by an experienced programmer while designing.

[101] I am not diminishing the need to find ways to teach novices basic concepts, merely that they are unrelated to design cognition. Without enactive representations of code, novices have little chance of tackling complex design tasks anyway, so they are a precursor skill, but far from the only one.

Table 7.5 provides a brief definition and identifiable traits of the elements from Figure 7.15.

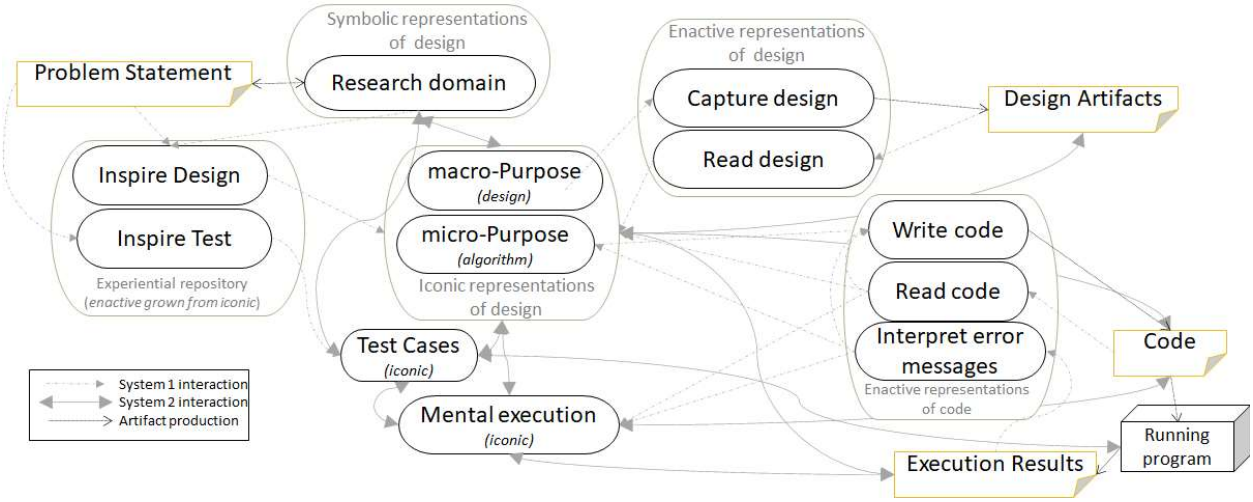

Figure 7.15. The mental representations of design

Table 7.5. Descriptions of the elements from Figure 7.15

Diagram element	Description
Problem Statement	Any source of details about the current programming task such as written documents or communications directly with stakeholders.
Symbolic representation of design	Any new facts, processes, rules, or other information that is new to the programmer from an external source. Such knowledge initially resides in semantic memory and would quickly fade if left unused.
Research domain	Seeking further knowledge about the domain typically by forming initial questions and follow-ups. The act of forming questions may inspire updates through System 1 or 2.
Experiential repository	The collective set of enactive representations that remember larger programming patterns. For example, common algorithms, architectural patterns, or testing approaches that transcend individual projects.
Inspire Design	An enactive repository of patterns of design formed primarily through repeated exposure of similar iconic representations of design. System 1 primes System 2 on future problem statements with these patterns. Note: Building this repository may require active consideration of code, more than simply reading code. Whether merely coding an algorithm (someone else provided) without metacognitively considering the pattern provides sufficient growth is unclear.
Inspire Test	Similar to *Inspire Design* patterns, enactive representations form inspiring patterns of testing, the same in every way but content.
Iconic representations of design	The pattern of episodic memories formed by the mechanics of prospection (rather than recollection). These are mid-term memories that may only last the duration of the project and would fade over time. Themes across projects or profound strategies may make it into the experiential repository.
macro-Purpose	The mental model of the overarching solution, dealing in abstractions that group *micro-Purpose* design choices into higher functionality. The model most often originates as a vague 'outline' proposed by the enactive repository but is refined through System 2 deliberations and System 1 feedback when completing other mental activities.

Continued on next page

Table 7.5 - continued from previous page

Diagram element	Description
micro-Purpose	The mental model of algorithms relating directly to lines of code and their purpose. The rules of creating and refining *micro-* and *macro-Purpose* iconic representations are the same. Note: An expert's System 1 has matured to the point they read functionality and purpose from code simultaneously in most instances. Novices may read code in such a manner that they construct functional details without divining any of the purpose. In novices, the micro-Purpose most likely is accompanied by a functional model of the code built upon instances of *Mental execution*. For unusual or novel code, experts may use this same capability, but so infrequently, it is not worth adding to an already busy diagram.
Test Cases	A mental model of the plans for testing, similar in all ways to *micro-* and *macro-Purpose* representations except for content. Note: It may be that testing planning is a separate variant from test cases, but again it is not portrayed here for simplicity's sake.
Mental execution	The storage repository of the mental prediction of code execution as performed by the notional machine, storing variables, and perhaps including presumed intent. Updating this mental model seems to be a key aspect of debugging, as the programmer attempts to reconcile their assumed code execution with the actual *Execution Results*.
Enactive representations of design	The enactive representations that support the reading and 'writing' of design notations. For example, fluency with flowcharts or UML would depend on these constructs.
Read design	An acquired familiarity allowing System 1 to generate (or update) *iconic representations of design* from merely looking over *Design Artifacts*. A programmer must acquire each alternative symbolic form of design through repetition, otherwise reading design adds to the burden of System 2. Note: Experienced programmers can learn new design notations faster since they 'know what they are looking for' managing the burden placed on System 2. Novices may undergo significant burden when lacking *enactive representations of design*.
Capture design	The automated mental habits and muscle memories for transforming *iconic representations of design* into *Design Artifacts*. Each symbolic variant of design *and the skills of its creation* must be individually practiced and learned to form these enactive representations. For instance, writing pseudocode does not build flowcharting abilities, *and* drawing flowcharts by hand does not generate fluency in using a computer drawing tool.

Continued on next page

Table 7.5 - continued from previous page

Diagram element	Description
Design Artifacts	The various ways intermediate symbolic representations of design can be captured in software, whether using text, diagrams, or other shared notations. Note: These are only describing shared notations. A rough sketch by an expert or novice that is of their own devising is a literal incarnation of an iconic representation. *Enactive representations of design* are only required when interacting with someone else's design format.
Enactive representations of code	The enactive representations that support automaticity in each programming language.
Read code	Like *Read design*, System 1's ability to transform code into meaning instantly and effortlessly. Experts can focus on functional implications at the same time as they derive purpose from code, but novices typically begin with the former.
Write code	The automatic transition from intent to the creation of code, but through the trained habits including the tools. Thus, a programmer's robustness at writing code may vary depending on their toolset (e.g., IDE's code generation abilities).
Interpret error messages	A repository of actions, a possibly but not certainly explanations, associated with the occurrence of error messages. For instance, a compiler error may drive System 1 to add a semi-colon. Some errors may drive action without ever processing the contents of the message itself, or without the programmer understanding why the change matters, only that it needed changing.
Code	The whole point of this entire exercise, the files that contain the precious lines of code.
Running program	The runtime environment and its associated inputs/outputs and behaviors. A programmer may need to understand very little about the running program's environment or a great deal depending on the nature of the problem, the language, and the level of debugging required.
Execution results	Any perceivable way a programmer can see the next effect of the code in action. Execution results may be active, like user interfaces or debugging, or entirely passive like updates to files or databases. Note: Dual process theory suggests most of the review of execution results may occur in System 1, as lazy System 2 presumes the *Mental execution* matches the actual results. Like *Interpret error messages*, a developer may build enactive representations recognizing unexpected results, but forming these is likely context specific. Experts may instead form habits of mind to double-check their assumptions, making them better debuggers.

The mental activity in the early stages of 'traditional' design

Dual process theory indicates that most of what we perceive comes to us filtered through System 1. Imagine you are an expert reading the problem statement from a new project. A question that pops to mind – System 1 triggered that question. A nagging feeling tells you that something is missing – System 1 is not seeing something it expects. You spot the perfect solution and are itching to start coding – System 1 has seen this problem before, and System 2 is ready to act. Design researchers have noted that expert designers generally leap to and refine an approach rather than deliberating over various possibilities (Adams, Turns, & Atman, 2003; Atman, Chimka, Bursic, & Nachtmann, 1999). Long before System 2 processes the full implications of a problem statement, System 1 is spotting relevant information, finding holes in familiar patterns, and invoking strategies that might help solve the problem. The *Experiential repository* from Figure 7.15 'preloads' an *Iconic representation of design* with a plan based on experience. The model of prospection suggests that System 1 activates semantic memories of past designs that System 2, in turn, navigates possibly prompting further research of the problem. Each new source of information adds to the *Symbolic design* or may spark further inspiration. TAMP suggests that early design cognition is less deliberative than reactive to a flood of inspiration coming out of System 1.

Once a problem moves beyond abstract patterns, System 2 works to acquire details of the problem at hand. *Iconic representations of design* capture semantic memories of the solution-under-construction. The more familiar a programmer is with the domain, technology, implementation strategy, the less need they will have in documenting their design. Programmers who are newer to certain elements may need support in offloading details to either design documents or, in some cases, directly to code. In many situations, programmers are expected to produce design documents either as a part of the development process or for posterity's sake. Many programmers become fluent in design notations (e.g., UML, flowcharts). Figure 7.15 presumes that most expert programmers possess some *enactive representations of design*, allowing the programmer to read and create design notations with little effort. While some believe the value of formal design notation may be overhead, TAMP suggests they can add value when strategically used.

One of the reasons to teach formal design notations is to speed up and improve the quality of communications between people. Design notations, like language, attach certain significance

to its symbols. When each party understands the implications of the design symbols, it improves the quality of communication. Before mastering a symbolic representation of design, each 'speaker' delivers their iconic imagery in its native form. The 'listener' then interprets that image through their own, rather than the socially constructed interpretation. Without a shared 'language' of design, two collaborators are essentially playing Pictionary. One person attempts to share an abstract idea through a picture the other must interpret. Just like Pictionary, the communication would be instantaneous using words, but the task is difficult (and presumedly entertaining) by taking away the shared system of symbols. Design notations encapsulate nuanced significance into their system of symbols (e.g., a diamond means a decision or an open triangle as an arrowhead means inheritance)[102].

When a programmer implicitly masters a symbolic system, they blend iconic thought and imagery with the socially 'approved' set of symbols. Their *Iconic representations of design* translate seamlessly to the page because their early – organically created – imagery has integrated with or adopted that of the design notation (e.g., I now "think" in UML when designing). When two parties each internalize symbolic notations, their communications are less hampered with noisy communications as they spend less time and effort sharing ideas. Even when working independently on complex projects, the designer can refresh their iconic memories from the page. As mentioned earlier, it is just as common for programmers to skip design and jump into coding, which has many cognitive benefits over design, as TAMP can explain.

Cognitive reasons for design versus direct coding

Explaining a programmer's tendency to skip formal design presents an early test as to how well TAMP explains their cognition. Developers have probably rebelled against the "waterfall model" for software development[103] nearly as long as it has been prescribed.

> The problems with[in] one phase [of the waterfall model] are never solved completely during that phase and in fact many problems regarding a particular phase arise after the phase is signed off, this result in badly structured system. (Balaji, 2012, p. 27)

[102] It is the nuance that also makes learning design notations difficult. If the novice does not understand the nuance of the concept (e.g., inheritance's function and value in design), they are likely to miss its significance in the notation.
[103] Write all the requirements before designing before coding, etc.

It is challenging, if not impossible, to perfect each stage of a project without feedback from later stages. As stakeholders learn more about what is possible, they see valuable additions to the planned functionality. Building some of the code provides valuable insights to programmers on improvements to the design (see Section7.6.3.2). The open-ended potential of software makes predicting any aspect of a project challenging, so "several methodologies recognized the critical role of iterations in creating effective designs" (Thummadi, Shiv, & Lyytinen, 2011, p. 68). Software engineering accommodates for linear and cyclical models of design, yet the relationship between 'design' (the planning stage) and code states is still sometimes contentious.

Conventional wisdom tells new programmers to dedicate time to planning their code before writing. Would-be mentors evoke the carpentry adage "measure twice, cut once" to remind novices of the value of planning, lest waste time in producing an inferior product. Some literature suggests that without creating preliminary designs, programmers risk making mistakes (Segue Technologies, 2013) or adding to technical debt (Montesi, 2015). Buna (2018) created a list warning new programmers against "writing code without planning" just before telling them it is equally bad when "planning too much before writing code." Adam Morse (2017) offered the sardonic tweet,

> In carpentry you measure twice and cut once. In software development you never measure and make cuts until you run out of time.

Presumedly, Morse was critiquing software engineering's lack of discipline and ability to predict quality. The consensus seems to be that some preplanning is required, yet how much is the right amount seems debatable.

Moving from anecdotal advice to empirical studies, Ji and Sedano (2011) compared the productively of 50 student project teams working under either a waterfall or Extreme Programming (a variant of an Agile) process.

> Waterfall teams spent more time creating high ceremony documents where as Extreme Programming teams spent more time writing code and documenting their design in their code. Surprisingly, the amount of code and features completed were roughly the same for both methods suggesting that on a three month project with three to four developers it doesn't matter the method used. (p. 486)

The process imposed on the student teams' designs did not seem to influence their overall outcomes. Ji and Sedano do not report on quality measures (e.g., bugs, adherence to standards, non-functional

aspect of maintenance); left unreported, it may be fair to assume they were similar, as any difference would have been an interesting finding[104]. From a productivity standpoint, the students produced similar functionality that presumedly worked in equal measure, but it is interesting to note that neither jumping to code nor attending to preliminary design seemed to change the scope of projects[105]. The amount of formal design did not seem to increase either the bugs, or did it decrease the productivity in relatively novice teams. Why is it some programmers insist that design is a distinct precursor step to producing quality code, yet many others show equal productivity with minimal or no formal design?

The perceived value of design may have a lot to do with individual preference that TAMP might explain, starting with the very nature of cognition. Kahneman (2011) described System 2 as lazy, always happy to allow System 1 to drive where possible. When a programmer has strong *Enactive representations of design*, the task of documenting design is less cumbersome, and they may be less resistant. The more System 2 must intercede when documenting the design, the less likely a programmer will find designing helpful. System 2's laziness may play into capturing design in another pragmatic way. How willing are you to start a job you know you may have to redo? A designer who is highly confident in their designs – either because they deserve to be, or because they are oblivious to the design's holes – may be more willing to document their presumedly solid plan than a designer who sees potential flaws. Programmers may choose to jump directly into code to shortcut the arduous deliberative process of comparing design options entirely in one's head. Jumping to code offers improved feedback and may take the same effort as invoking the notional machine to predict the outcome of a specific design.

Preliminary design documents offer limited avenues of feedback to a programmer when they are building new systems. Jumping to code offers programmer feedback from the compiler (e.g., does the library contain my presumed feature), the runtime (e.g., does my algorithm produce my expected results), the coding process itself (e.g., did my decomposition work out as expected), not to mention the potential for actual user testing and feedback. By comparison, design documents do little to help a programmer help themselves. Occasionally the process of

[104] It would be interesting to review the quality of the resulting design but TAMP might predict that the quality would be similar for reasons about to be explored.

[105] Again, the authors do not report projects becoming 'bigger or smaller' deliverables, nor the amount of time students put into the projects. There is an adage in industry that a deliverable always takes a long as its estimate, meaning that even if a product could be delivered faster the team tends to continue to iterate until they run out of time. Thus, the agile teams may have completed the same amount in less time or as reported, did the same amount.

documenting a design will prompt a new insight (System 1) into a flaw (e.g., "Oh, I need an index in this database table!"). A disciplined programmer can engage System 2 to review their design for defects or even mentally simulate their designs planned execution. That developer needs to be particularly disciplined, though, to overcome the confirmation bias and other such systemic biases in our thinking (Kahneman, 2011). Dual process theory suggests such a task is at least demanding, if not futile.

When a programmer reviews their own design, they must 'tune out' the same experience System 1 suggested to create the design. System 2 must consider if every assertion, abstraction, and assumption hold and do so independently from the same enactive representations that hold the assertions, abstractions, or assumptions. For example, if System 1 assured me when I created a design that my database automatically encrypts passwords, I must recognize and double-check that and every other assumption. Not only is such a review mentally taxing, but likely uncomfortable. The Stroop effect (Eagleman & Downar, 2016; Kahneman, 1973) – which seems to occur in dual process theory terms, when System 1 and System 2 are at odds – produces an emotional response making the self-reviews strenuous *and uncomfortable*. Documenting design provides a helpful medium for supporting the memory and focus of System 2 but may not provide an individual with insight into the quality of the design.

Seeking input from others pay provide a "second set of eyes," but their feedback may come from intuition as much as reasoning. I have been in many reviews where a different perspective offered valuable insights and many more where they were just another opinion. Purposefully deliberating through the logic of a proposed solution is hard work! Design documents intentionally describe abstractions of the inevitable code, or why not just write the code. Such abstractions may highlight certain design flaws (e.g., excessive coupling) yet hide others (e.g., lack of reuse). Pressman (2010) described design reviews where reviewers are "asked to read [the design] looking for errors, omissions, or ambiguity" (p. 220). Heuristics often guide the assessment of design, which, by their nature, are subjective and often intuitive. Design processes attempt to prompt

reviewers by providing checklists[106], but in my experience, design reviews tend to dwell on functional implementations[107] over the non-functional questions of code quality.

Assuming reviewers can provide useful feedback, such feedback is still only reflective of the programmer's produced symbolic, not their iconic representation. The design may not fully represent the coder's tacit knowledge that will inform the final implementation. If design originates in System 1, even the designer is blind to its rationale. Designers often overlook, cannot explain, or fabricate stories justifying an unknowable decision (even if it is a good rationale). In a nutshell, design documents tell only part of the story at best. The documentation itself is filtered through the programmer's fluency with the symbolic medium. It describes abstractions that may or may not have common meanings. The feedback may or may not yield the desired changes within the designer. I am not advocating that reviews are pointless, so much that they are limited to certain types of feedback in proportion to the skills of the participants and the time dedicated to the creation process.

My analysis of design through the lens of TAMP should not undercut the value of design or dissuade its use. Rather it should serve as a guide to embracing the effective aspects of design and possibly letting go of traditional yet probably mythical benefits. Perhaps embracing iterative approaches, dealing with upfront uncertainty yields better understanding, can yield similar benefits. In discussing the conflict that is design-versus-code, I hoped to capture reasons on both sides. Advocates of design can point to specific benefits but may be overstating others. Naysayers might be right about design's limited benefits, yet also may be avoiding a task in which they are undertrained. In the spirit of Socha and Walter (2004), Figure 7.15 does not stop with the design but continues to discuss the implementation, testing, and even debugging of code. It may be impossible to train a programmer without equally addressing all these aspects of cognition.

[106] Another important, but perhaps tangential point, checklists probably do more to stimulate System 1 than encourage deeper System 2 analysis. The checklist may focus attention on certain types of errors, but detecting such errors still often relies on System 1 experiences. The type of design novice programmers focus on often is limited to basic algorithm design which benefit significantly less from design representations.

[107] Which are just as easily corrected through testing. Yes, catching a functional error at the design phase provides earlier feedback, but the same impact is achieved through Agile sprints with less total effort as design reviews are expensive in time.

Designing code – the whole game

The mental representations of design are important through the process of developing code. Even when a programmer dedicates the time to fully fleshing out their design plan, their resulting mental model must guide their actions when writing, testing, and debugging code. The act of writing, testing, and debugging exposes flaws in the *Iconic representation of design*, as well as suggesting improvements. Perhaps the most challenging aspect of debugging is seeing past the flaws in our own mental model. Many a coder has a story about the bug that hid behind a rock-solid assumption that turned out to be wrong. As an example, let me share a humbling experience I had when my wife announced our brand-new vacuum cleaner was not working.

My debugging process started when my wife announced that our brand-new vacuum cleaner was not turning on. I was mildly against purchasing a new vacuum as my 15-year-old Kirby had served me well, but the Kirby's bulky frame made it heavy to lug around, so I accepted a less-costly model that would meet our daily needs. I presumed it came with a shoddy connection in the wiring since it was "clearly an inferior brand." Within twenty minutes, I disassembled the unit, carefully cataloging its design in my head as I worked until I exposed the switch in question. Using my trusty voltmeter, I checked if power was reaching the switch, the first in the sequence of electronic parts from the cord. My years as a software developer taught me to double-check every assumption. I plugged the vacuum in and found no voltage. I checked the cord for any cuts or kinks but being new every part was pristine.

The problem, as it turned out, did not lie in the vacuum at all. Hearing the vacuum would not turn on, my System 1 blamed the brand-new device I did really want to buy (fear of buyer's remorse influencing my logic). As a trained electrical engineer, my System 1 suggested a detailed iconic representation of the inner workings of such machines and a plan to find and repair a loose connection or demand a replacement for a faulty part. As I disassembled each piece, my System 2 added to the mental model of the circuit diagram, enriching my confidence in how well I understood the engineering of the vacuum. It was not until I hit a data point that undeniably put my mental model at odds with the physical could I see past the simple assumption I had made – the vacuum did not fail to turn on, it suddenly stopped working because it tripped the circuit breaker in the kitchen. The vacuum was fine; the power went out.

Expert programmers seemingly are much better at forming and manipulating mental models than novices. My iconic representation of the vacuum formed quickly, calling on semantic

memories of rules of circuits and voltages and adding new symbolic knowledge as I took the machine apart. I even manipulated the model to create a methodical sequence of testing, which 20 minutes later (plus another 20 to put it back together again) resulted in a successful repair after I reset the circuit breaker. My analytical skills had served me well, but if I were an electrician, not a software engineer trained as an electrical engineer, I probably would have thought to check the power supply first. Our mental models guide our work as programmers, just the same. Whether you agree with Socha and Walter, it is hard to argue that a program's design, especially how we mentally model that design, is subject to change until the programmer delivers the code.

My experience in coding, and it seems in some of the literature discussed earlier in this chapter, indicates that intuition guides experts as much as reasoning in design. A designer never really starts tabula rasa because their experience suggests a starting point. Novices who start without any inspiration seem to make little or no progress at all (think stoppers). Just as often, though, programmers start within an existing system. We saw in Section 7.6.2.1 that even here experts use their *Experiential repository* and skills in reading code developed within *Enactive representations of code* to make intuitive leaps about what the code does and extract details that novices miss. Whether we are creating new designs or constructing mental models from existing designs, the memory stores described in Figure 7.15 are the same. What varies is how our mind 'time-travels' through them.

By framing design cognition using the rules of prospection, TAMP explains differences between experts, but also why novices struggle to design. Recollection (prospections partner) is more than the direct retrieval of facts; stress, personal biases, or other factors can systematically alter episodic recall (Harari, 1995). Since they share the same mechanisms, then planning would also vary based on not only personal factors, but perhaps the designer's state of mind. Programmers often create radically different designs despite using identical toolsets for the same problem[108]. While Table 7.6 goes through a detailed example of how intuition might drive experts to different solutions, empirical evidence of how design varies lies within the application of academic honesty policies.

Many instructors and schools use tools to compare student coding assignments and assess if they may be copying their answers rather than producing their own code. At one point, Stanford

[108] Consider that the core instruction set of a processor has changed little, particularly relative to the virtual machines common in modern software, yet Computer Scientists keep generating new programming languages. Either each new language comes out of a new set of priorities (experiences) of the designer, or we should stop teaching language design as a course that encourages Computer Science students to recreate the wheel!

University saw an average of 37% of all academic honesty issues from the CS department, with some years, the number exceeding half (Roberts, 2002). If the semantics of programming constructs were the primary driver of design, student submissions would be so similar as to make academic honesty nearly impossible to detect. One tool used to detect plagiarism assumes that "programs of a few hundred lines, anything over 50% mutual overlap is a near-certain indication of plagiarism" (Bowyer & Hall, 1999, p. 2). Unlike plagiarism tools for essays, code tools use underlying programming constructs so that merely changing the names of variables cannot fool the tool. Since most early programming assignments use few constructs, have highly constrained expectations, and are less than a "few hundred lines," it seems that students create solutions that are highly unique using logic other than merely translating from well-memorized rules. Design is not purely rational and draws from many different types of knowledge *and experience*.

An example of designers in action

Despite all the complexity within Figure 7.15, it remains a simplification of everything stored within the mind of a programmer. The *Inspire Design* representations within the *Experiential repository* could be split into technical (e.g., exemplar uses of constructs, general algorithms) and domain (e.g., the rules of banking, insurance, avionics) knowledge. As discussed in Section 7.6.3.3, programmers take their design inspiration not only from the language and their experience in algorithms but also from the paradigms of the industry in which they work. What a designer chooses is not just influenced by examples of success, but also by counterexamples of times they struggled or failed. In the early 2000s, I worked with a manager who insisted that configurable software would never work – he suffered through a miserable project when computers of that day were as powerful as modern calculators. He battled against my design until his concerns proved moot when my prototype was successfully performance tested[109]. He held such a strong bias, he willfully overlooked advances in computing (see Moore's law) or even waiting for the evidence before forming his final opinion. He preferred 'safer' design options that took significantly more time and effort to implement and maintain. To be fair, my insistence was equally driven by intuition - I didn't want to build a bunch of repetitive, boring code! The

[109] His prior experience saw early Java web application struggle to manage a few dozen concurrent users. I forget the exact numbers, but the performance test handled several hundred or more concurrent users – on my work laptop.

difference between our initial choices was not reasoned, but intuitive based on his past struggles versus my, perhaps naïve, success.

Eckerdal and Berglund (2005) reported that students believed they needed to think differently as a programmer. From the outside, watching a programmer work might seem foreign and mysterious, but is that any less so than a masterful artist, lawyer, teacher, or parent? TAMP suggests that programmers do not change into different ways of thinking (i.e., algorithms) but rather simply have different memories to draw from (i.e., internal state variables). Figure 7.15 defines the different mental repositories that each programmer could draw from when solving a programming problem. To demonstrate how the same cognitive mechanisms can result in different results, Table 7.6 compares three programmers – two experienced programmers with different backgrounds and an advanced, but struggling novice[110]. One of the experienced programmer's primary experience has been in writing database scripts (i.e., the DB Expert), where the other's experience is more in Java (i.e., the Java Expert). The three programmers receive the task:

> Compute the total amount due to each provider for today's claims. For instance, if Dr. Smith submitted three claims for $50, $25, and $100, the result would show $175 for Dr. Smith among all the other providers that day.

I have created the scenarios in Table 7.6 based on prototypical developers I have worked with and taught solving different problems. The point is to illustrate the mental structures from Figure 7.15, not imply a literal outcome.

[110] Our advanced novice has competency in much of the language mechanics but has little design experience. They could, given a flowchart, readily construct and test a solution.

Table 7.6. Working through the cognitive mechanisms of a sample design process

Step	Cognitive mechanism
Consider the *Problem Statement*	
Both	System 1 reads the problem statement using native language processes
Experts	While reading, System 1 primes System 2 with an impression of the problem and 'initializes' *iconic representations of design* with hints towards design approaches.
Novice	System 1 may offer rough connections but probably at the level of language constructs (e.g., use a loop) rather than algorithms or design.
Consider the design	
DB Expert	System 1 suggests retrieving claim data using a "group by" query[111] into a database. If the expert is familiar with the existing system design, the details of the query (table and column names) spring to mind, else they research these details.
Java Expert	System 1 suggests organizing claims in a Map structure[112]. If new to the system, they look up how to retrieve the claims data, else their System 1 already knows how.
Novice	System 1 hinted that a loop might be involved, but first, the novice needs to research how to get access to the claims. Are they passed in as a parameter? Does the user input them on a screen? What do I remember about loading things from files? Even if the novice is taking a database class, their experience may trigger lessons from their coding classes, rather than the intended database practice.
Research domain	
Experts	Lookup domain details or double-check their intuitive design as needed.
Novice	With so many gaps in the domain and technical understanding, it may be difficult to know where to start. Are the details of a claim most important, or should I start capturing the technical details for retrieving the data? They may start writing code to capture what they 'know' without a master plan or hint at a master plan to fill in with the details they will look up in a minute.
Document the findings of the query	
	The next step varies depending on the programmer's 'feeling of knowing'
Experts	o Feel <u>supremely confident</u> in knowing they chosen the 'right' algorithm, they either sketch out a quick design or jump into code. When coding starts, they quickly breeze from one step to the next, tackling issues as they arise. o <u>Confident</u> about the approach, but unsure about the details, they document some aspects yet want to validate others. Jumping into code, they write basic code for any easy prerequisite tasks (e.g., loading the data) focusing on the unknowns (e.g., the exact format of the data, the system performance). Once settling any concerns, they complete their design and/or fill in the parts of the solution they skipped over in their prototype.

Continued on next page

[111] For those not familiar, the group by statement, as the name implies, would return a list of claims ordered by the provider (e.g., the first five claims are for Dr. Smith, the next 3 are for Dr. Shah, and so on).

[112] Maps allow quick sorting of data under a key. In this case, using the provider as the key would quickly sort the claims by the provider making the rest of the algorithm quite simple.

Table 7.6 – continued from previous page

Step	Cognitive mechanism
Novice	○ They feel confident about a piece of the whole solution, so they write code for that part of their design hoping to tackle other areas once they finish, such as ▪ A loop to cycle through the claims ▪ A query to load the claims ▪ A data structure to organize claims by the provider They write code but introduce compiler errors they can't fix, so they attempt to code the next ideas. That code ends up with several more unreconciled compiler errors before they try again until running out of time. ○ They are unsure about the problem or solution, but they know they need to load the data. They recognize they need to make a query, but don't remember how to do so right now. They turn instead to the loop they know must be there before realizing they are not sure how to start any of these pieces. After some time, they have a few ideas but have not started to write any code.

The primary experience our experts bring influences their different but equally viable approaches. The scenario in Table 7.6 is fictionalized, but not imaginary. I was the 'Java expert' a decade ago, joining a team of coders who spent most of their time working in databases. The story of the novice summarizes observations in the literature, Perkins et al. (1986) and stories seen across the three studies discussed in the next chapter. The three alternative paths towards design demonstrate how the mental constructs of Figure 7.15 influence design. It seems rare that a programmer mentally weighs each option – would it be more efficient to let the database order my claims or to manage that in the distributed client – much less writing the code to test each. More likely, experience drives their preference.

If intuition drives expert thinking during design, then how do novices compensate (or in Table 7.6, not) when lacking experience? As discussed in Chapter 2, some computing education researchers talk about a bimodality in students, where some thrive while many more struggle. TAMP suggests that thriving is much less common because fewer novices will have repositories of experience to draw from. It also suggests that bimodality may be overstated and more of a problem in the question than the students.

> when [students] give wrong answers it is not so often that they are wrong as they are answering another question, and the job is to find out what question they are in fact answering (Bruner, 1966c, p. 4)

Bruner did not know about System 1 but could see that students answered with something in their realm of experience; it just may not have matched the question. Experienced programmers also encounter gaps in their knowledge and use System 2 to seek information but are better at adding

to their iconic representations since they can add new facts to already strongly associated knowledge (enactive and symbolic).

Human memory is much like data stored within a database. Information is most useful when it is easily associated and searchable. Database designers go to great lengths to ensure data in one table is accurately associated with data in another table. Without well-planned connections, the system could spend all its resources searching between tables for potential associations, without confidence that any results are unique or accurate (e.g., I found an address for this person, but is it the same person? Is it their only address?). Instructors have become very effective at filling student's heads with important information, but it seems novice programmers have difficulty in accessing the right knowledge and putting it to work when needed. Experts have had time to create associations between problems and code-based solutions, that may be possible to foster in novices, but perhaps not formally. System 1 implicitly acts as the vital link between abstract knowledge but only can do so when given time and varied experiences.

Before leaping to applications of TAMP, though, the next step is to offer empirical evidence of the assertions of this chapter. While much of this chapter leverages my anecdotal stories and observations to explain TAMP, the underlying theoretical constructs are rooted in existing theory supported by data. To test if my assertions stand up, I revisit several studies of design and programming in the next chapter, offering an alternative analysis using the theoretical constructs of TAMP.

8. VALIDATING TAMP - REVISITING EMPIRICAL STUDIES OF DESIGN AND PROGRAMMING

Computing and Engineering Education researchers have long studied the process of design. While my focus is on how novices learn to program, two related studies (Atman et al., 2007, 1999) from Engineering Education are useful in evaluating the model of cognition for designers at various levels of expertise. These studies help to establish the general plausibility of TAMP's model of design cognition, even if they are not centered around aspects of computation. Returning to programming, this chapter next revisits the pivotal study by McCracken et al. (2001) that influenced much of the last two decades of computing education research. They reported that instructors expected far too much from their students, particularly in their ability to apply coding knowledge to design, but never really explained why. Since McCracken et al.'s work was so influential to the formation of TAMP, this chapter finishes by reinterpreting a follow-up study on writing code (McCartney, Boustedt, Eckerdal, Sanders, & Zander, 2013) as a form of discriminant sampling. Juxtaposing the difference between these studies offers not only an interesting look at how novices approach software design but also at the rich explanation TAMP offers in considering computational education research.

8.1 Comparing the design activities of experts and novices – The playground activity

Two studies (Atman et al., 2007, 1999) offer empirical data capturing different behaviors amongst engineers at various levels of expertise. By observing how novices, advanced novices, and professional engineers tackle the design of a playground (meeting a set of criteria), the researchers captured differences in the way each group thinks on average and characteristics that separate the more successful designers. While these studies investigate general design without any programming, the engineering design process they described is quite similar to the software development lifecycle taught to most programming students and stands as at least an analog if not the same type of problem-solving.

8.1.1.1 The early stages of design expertise

The playground task offers an interesting testbed for capturing pure design processes as it largely eliminates advantages of domain experience and even technical prowess. It is possible that some disciplines may provide advanced knowledge that aid in playground design, but it seems unlikely to help in the type of criteria the researchers measured. More likely, all participants entered with merely personal experience with playground rather than having specific benefits from formal learning. The main difference between the studied groups was time and experience rather than 'knowing more' about the challenge. The playground task, therefore, seems to have created a test that measures the habits of mind that participants used in following an engineering design process[113], and thus design thinking, in problem-solving.

In the first of these studies, the researchers compared the design behaviors of seniors and freshmen engineering students (Atman et al., 1999). Among many findings, they reported,

> We expected the seniors to have more transitions between design steps than the freshmen. We found that seniors did have both a higher number of transitions between design steps and a higher number of transitions per minute. Transition behavior was related to playground quality for both groups, although the relationship was stronger for the freshmen. (p. 150)

Atman et al. predicted that the seniors, as experienced engineers, would iterate between the steps of the engineering design process (unlike the waterfall model) more often than freshmen. They saw that the more the participants' thinking jumped between the mental activity of design, the better the results, particularly for freshmen. Seniors averaged more time overall working on their playground designs and, during that process, asked for more information across a broader set of categories than freshmen. Freshman, particularly the less successful ones, tended to lock in on each task and stick with it for longer durations.

One of the study's goals was suggesting additions to the engineering curriculum to improve design thinking.

> These results suggest that students *need experiences* to encourage them to iterate through all the steps in the design process, develop multiple alternatives and gather information. (p. 152, emphasis added)

[113] The researchers defined their engineering design process to include problem definition, gather information, generate ideas, modeling, feasibility, evaluation, decision, and communication.

Not only, as we will see, does the data reported by Atman et al. align with TAMP's model of cognition, but their conclusion also aligns with TAMP's call for the need to develop intuition though experience. Atman et al. suggested that students need activities that expand their experience and TAMP explains why.

The frequent cycling between gathering, planning, doing, and evaluating seems to match the cognitive model of Figure 7.15. Experts frequently cycle between finding, using, and validating information because System 1 frequently directs System 2 to follow what experience suggests is the most promising next step. Atman et al. included charts of behavior that showed the freshmen and low scoring seniors were more likely to spend long blocks working through the same activities uninterrupted. TAMP suggests the plodding deliberations occur when System 2 must focus on the current train of thought, whereas System 1 enables the rapid cycling between modeling, gathering information, generating ideas and the other steps of the engineering design process by automating the work within these steps as well as inspiring the need to jump to another task. While it is possible that experts jump between tasks following reasoned thinking alone, it seems that such leaps might overload short-term memory. System 1 as a driver provides a simpler explanation for the almost frenetic hopping between activities while explaining the intuitive advantages that experts seem to hold over novices.

Seniors spent more time on every step of design except for two, *generating alternatives* and *defining the problem*. Atman et al. predicted that experienced seniors would spend more time producing and weighing alternatives, but in the end, they did so less frequently than the inexperienced freshmen. Similarly, Atman et al. found that "those subjects who spent a large proportion of their time defining the problem did not produce quality designs" (p. 142). It would seem that those who could define the problem quickly ended up with better-rated solutions. If designing playgrounds mirrors the cognition suggested by TAMP, the high-performing seniors' *Experiential Repository* inspired their design, and they may not have needed to consider many alternatives. The three or more years of schooling that seniors had over the freshman meant their System 1 had more experiences from which to draw. The seniors' *Experiential Repository* allowed them to both process the problem statement and generate the foundation of their design faster than the freshmen. TAMP explains why Atman et al.'s prediction about seniors taking more time on alternatives went astray. System 1 was not just visible in helping seniors tackle certain tasks

quickly but may show its presence when seniors spend more time constructing their prototype playground.

The seniors spent nearly twice as much time constructing a prototype of their playground than the freshmen. The behavior of seniors may mirror TAMP's reasoning why experienced programmers prefer to 'jump to code' (Section 7.6.3.2). The playground problem is a bit different than code. Even a basic coded application results in a working solution, while the playground prototype was, at best, a physical representation of the plan that still required imagination to operationalize. A programmer can test the execution of their design after writing code. The prototypes still required the designer to imagine how children would play. The extra mental effort (and limited realistic feedback) might explain why seniors spent more time on their prototypes, yet both groups spent relatively little time. The seniors averaged 5.6 minutes (5%) versus the freshmen's 3 minutes (3.1%) to create their prototype despite taking anywhere from 45 minutes to more than 2 hours to complete the full project. The realization of the plan was far from a pressing concern of both groups based on the relative effort compared to other aspects of the process, yet the seniors still dedicated more time to this phase. The fact that seniors spent *more* time on the prototype seems to indicate they found it valuable in a way that the freshmen did not. TAMP suggests that part of the reason might be that seniors not only seek the feedback that a prototype gives, but they probably have more resources available in their System 2 to simulate the prototype in action. Their more experienced System 1 seems to not only speed up certain activities around making complex decisions but also to free up cognitive resources that allowed the seniors to glean more from their prototype and perhaps justified dedicating more time to working with it.

8.1.1.2 Expanding design habits to professionals and programming

Comparing seniors and freshmen bookends the influence of formal education on training engineers, but hardly captures the breadth of experience of professional engineers. A later replication of the playground study (Atman et al., 2007) added industry professionals as a comparison group.

> There is a strong result in the literature that experts do not typically consider a number of alternative solutions and choose among them – rather they tend to choose one major idea and make modifications to that idea or consider a small number of ideas. (p. 374)

Design literature seems to reinforce the earlier report that seniors spent less time considering alternatives[114]. In their first study, Atman et al. (1999) expected seniors would discuss more playground objects (e.g., swings, slides, etc.) than freshmen, yet they averaged about the same. Given this data, the second team of researchers (Atman et al., 2007) hypothesized that the playground objects equated to the number of design alternatives.

> Because we considered the number of objects to be a surrogate for the number of alternative solutions, we predicted that the experts would consider fewer objects than the students. However, we found the exact opposite in our data. The experts worked with almost twice as many objects while designing than the students did. (p. 374)

I believe the misattribution of objects as design alternatives is the same false equivalency that language constructs drive algorithmic design. The function of a slide or swing set does not influence the overall planning of a park any more than the semantics of a `for` loop or `if` statement drives the contents of an algorithm. TAMP suggests that System 1's *experiential repository* inspired design choices rather than System 2 processing a set of rules for playground design or consider the function of playground equipment. Atman et al.'s professional engineers each formed an *Iconic representation of playground design* based on their experiences with playgrounds and values as a designer (e.g., cost, aesthetics, function). Given a general layout, they could then explore objects to see which fit their vision. For example, my inspiration for a busy park would be managing the flow of activity. I want kids to move from one park apparatus to the next throughout the park rather than bottlenecking in certain areas. In doing so, I might consider and place each of the slides, swings, and other such equipment to see how it facilitates or blocks my desired flow. I am not considering alternative designs, so much as taking advantage of my master plan (inspired by System 1). The priority and heuristics that System 1 suggests leave a designer free to consider the best implementation choices for that plan.

As in computing education research (D. N. Perkins et al., 1986), Atman et al. (1999) found stoppers amongst their participants. In building a playground, none of the participants needed to master any diagramming technique or 'programming language'; they could speak and draw as they wished. The *only* restriction on their final design was to use locally available materials. It seems

[114] And reinforces TAMP's assertion that System 1 inspires design that experts then refine using System 2!

First-Year students should be familiar with playgrounds and their standard building materials enough to produce even a subpar design, yet

> Apparently, some of the freshman subjects seemed to 'get stuck' defining the problem and did not progress further into the design process steps. (p. 150)

Atman et al. do not elaborate further on the resulting design or process of the 'stuck' students yet TAMP suggests why early design can be so difficult. Freshmen designers seem to become 'stuck' when the problem statement does not evoke an intuitive solution. With no inspiration from System 1, their iconic representations are left bare. Prospection suggests that planning leverages memory. System 1 might trigger the required memories when reading the problem statement or at some later point during System 2 deliberations (similar to Figure 7.7). Without some initial memories in place, the future is as blank as K.C.'s and his damaged hippocampus (Section 7.3.2.3).

On reflection, perhaps it should be of little surprise that lacking intuition about playgrounds novices would be *more likely* to stop since the problem has no symbolic rules to guide action. TAMP suggests that many programmers make progress because they memorize procedures for solving recognizable problems rather than using creative problem-solving skills. Programmers at least can add a `for` loop or `if` statement to fill the space and seek inspiration. While it might seem obvious that everyone has played on a playground, using a playground does not mean a person has considered their creation. A child might design a playground by whimsy, randomly placing playground objects perfectly happy to ignore the design task's criteria. The freshmen designers who stopped were likely hyperaware of the success criteria and froze. The effort required to decipher the rules without the support of inspiration (System 1) overwhelmed their ability to make any meaningful progress. Sime and Arblaster (1977) noted that beginning programmers were unsure "what the computer will do if they get the program wrong" (p. 207). Sime and Arblaster's participants did not understand what the computer would do with a badly coded conditional structure and it caused them to be "unsettled" (p. 207). It stands to reason that fearing to make a mistake, some novices prefer inaction.

8.1.1.3 How designer think about the design process

In addition to analyzing the participants at work, Atman et al. (1999) captured reflections on their experience with designing the playground. At the end of the experiment, Atman et al.

showed the students a "prescribed design process" and asked them to reflect on their approach. The freshmen tended to agree the presented process seemed correct, despite not following the process as described.

> One freshman said of her design process, 'I did it backwards', referring to the fact that the design was accomplished before the necessary resources and information were gathered. (p. 148).

This participant seemed to think she needed to follow the steps in a specific order. Her System 1 seemingly jumped to the 'design work' rather than explicitly following the early phases of the engineering design process. Another student claimed, "that she went back and forth between the steps of the process" (p. 148). It seems that her thought patterns were desirable rather than problematic as they reflected those of successful designers. The freshmen seemed to believe the engineering design process was correct and their behaviors had yet to conform to its strictures, but the seniors were less accommodating.

The Seniors' comments took a very different tenor, noting the proposed engineering design process was complicated and difficult to follow.

> In general, the subjects felt that this process was not followed step by step, rather that many steps were skipped or completed in their heads, for example a subject said, 'it's all repetitive'. Other comments on the design process included 'this is the ideal if you have time', 'limits creativity', and 'some of this stuff is common sense'. There were seniors who agreed that this is a 'good methodology' and that this process is a 'good general out- line'. (p. 149)

Each of these statements seems to point to the implicit nature of design thinking. A 'waterfall' process may only be possible when a designer is working with a familiar problem and knows what information to gather at the start – something only an expert with a relevant *Experiential Repository* can likely accomplish. Atman et al. implied that the more natural and desired state is jumping back and forth, which seems to follow System 1 impulses. Some freshmen did cycle between design steps[115] but were not confident enough to disagree with the process provided by the researchers. The seniors, on the other hand, held little back in coming to the defense of their thinking. Even when recognized the value of the steps, many rejected the presented ordering and the strictures imposed by such a process. It seems unlikely that seniors are consciously choosing

[115] It is not entirely clear how well the participants knew the steps of the engineering design process before the study or perhaps encountered the steps during or even after the playground activity.

to cycle between the prescribed steps of the engineering design process when they rebel against the 'natural' ordering.

8.1.1.4 Takeaway from the playground

Atman led to studies that show the trajectory of how designers think from early in their formal education through professional practice. Participants within each group show tendencies to follow intuition in their design process. Lower performing designers tend to jump to modeling and spend most of their time working towards a solution. Designers with higher scoring solutions tended to cycle between mental activities, gathering more information both from external sources and assessments of their own work. TAMP suggests that experience not only lends speed to individual tasks but also prompts moments of inspiration that lead to the rapid cycling observed in seniors and professional designers. Revisiting these two studies provides a challenge that largely removes domain knowledge and technical expertise from how experts think. It demonstrates that other habits of mind (outside of learning a programming language) might contribute to the seemingly bimodal performance distribution when students are asked to write code.

8.2 The rise and fall of coding assessments –McCracken et al. (2001)

McCracken et al. (2001) gathered an international group of researchers and created an assessment of basic coding skills that also served as a baseline of what instructors expect students to learn after their first programming course(s). They evaluated new programmers' ability to take a problem from concept to solution by having participants construct a calculator, including a basic command-line interface. A calculator was presumedly familiar and tested the students' grasp on basic operations, mathematical expressions, looping and decisions, and fundamental data structures. The test included three variants of input notation for the calculator: Reverse Polish Notation (RPN)[116], infix notation[117], and infix with parentheses for precedence. The team believed that the RPN notation was the easiest variant[118] with the next two variants progressively more difficult. Most participants only completed one variant of the calculator, but

[116] Reverse Polish Notation (a.k.a. post fix) places operators after numbers, so "2 2 +" is equivalent to "2 + 2" in infix notation. More involved examples become complex quickly as 15 / (7 - (2 + 2)) would be written as "15 7 2 2 + - /"
[117] The 'typical' mathematic notation of 2 + 2
[118] The RPN calculator can leverage the data structure known as a stack, which reduces the amount of code required.

the final report noted that each variant proved difficult, with students performing lower on the RPN than the infix calculator. Beyond missing their guess on which problem was most difficult, McCracken et al. profoundly overestimated the abilities of novices, starting with the assumptions about the 'simple' nature of the problem.

8.2.1 The reason calculators are not all that familiar

The major finding from McCracken et al. (2001) may be how abysmally the research team underestimated the difficulty of the problem for their students. To measure performance, the researchers rated the students' code across several categories totaling a possible 110 points, including

- *Execution* (30 points) – writing code that compiled or ran without runtime errors, even if it accomplished nothing else
- *Style* (10 points) – writing code that followed standard conventions of the language
- *Verification* (60 points) – implemented desired functionality by passing test cases
- *Validation* (10 points) – conformed to the RPN or infix input format

The average score across all schools and all countries was a mere ***22.9***, meaning that a significant number of students did not manage to write code that either fully compiled or ran without errors. Perhaps the abysmally low score was not all bad?

> Though the scores are uniformly low, as a percentage of possible scores, students did best on the execution component (implying that, overall, they wrote programs that compiled and ran) and the style component (implying that the source code looked good). The lowest component scores were on the verification and validation components. (p. 129)

McCracken et al. presented data that is difficult to reconcile between the stated statistics and point totals[119], but students seemingly accumulated just half the points on *Style*[120], and roughly 3% of possible *Verification* and *Validation* points. Reverse-engineering their numbers, if just half the students managed just 25/30 for their execution score, the remaining students' average drops to

[119] They broke their scores into four categories (McCracken et al., 2001, p. 130 Table 3), but the sub-scores do not add up to the average they reported. If we presume the reported *Execution* scores are a typo and the summary is correct in stating the execution was the highest category, we can add the missing points to *Execution* which makes the average execution score 16.6 out of 30 or 55% of possible points.

[120] Style points were optional as not all schools taught or assessed style

less than 15/110! By any standard, a good percentage of the students in this study struggled to produce anything that resembled a working calculator. McCracken et al. reported a "bi-modal" (p. 129) distribution to their data, but it seems like the numbers indicate it was far from an even split of participants. A narrow few may have scored well, but overall, most students struggled.

8.2.1.1 Programming performance

Even when students wrote running code, they struggled to meet all the demands of the problem challenging the assertion of bimodality. McCracken et al.'s charts show only 20% of the students scored above-average on the RPN calculator, where only 40% exceeded the average for the infix calculator. Those who scored better did not distinctly cluster at the top, their scores spread through the possible range with the 'best' RPN scores reaching 80/110 and the infix nearing 100. Many more students definitively struggled, but to say that students either thrived or thrashed does not seem evident in the data; most students struggled to build a reasonable facsimile of a calculator, some just struggled to more profoundly.

Diving into the categories of code and the scores makes the results even more disturbing. If the *Execution* category held 30 points for merely writing compiling code that runs, a wide majority of students may or may not have scored even half of these points[121]. Far too many students could not even produce code that compiled and ran without error. It is not clear if a student needed a perfect execution score in order to pass any test cases (e.g., their code must compile, but what if it created runtime errors only with certain inputs?). If a perfect score in execution was required, only one in nine students who attempted the RPN problem or one in three students who tackled the infix problem had a shot of passing the simplest of tests. While some of the criteria provided by McCracken et al. were unclear, other elements seemed fairly straightforward; for example, the following *Validation* item was worth 10 points.

> The program should terminate correctly (i.e., entering the quit command should terminate the program). (p. 138)

Compared to other requirements (e.g., "The program should react properly to erroneous inputs" (p. 138)), the quit command is clear, requires just a few lines of code, and could easily be one of the first features implemented. A student could theoretically earn a minimum of 40 points by

[121] See footnote 119

writing a program that did nothing but quit (50 if they did so with perfect style)! The *average* score was less than half of that. Later literature has since confirmed that students are learning during their courses (see Chapter 2), and they are not strategically earning points in the most rational manner.

McCracken et al. saw the stoppers just as Atman et al. (1999) and Perkins et al. (1986), which seems surprising. Atman et al. presented a task that required little expertise, but also provided no obvious training. Perkins et al. worked with students who had some training, but they were pre-college learners and not enrolled in a specific course. McCracken et al. worked with students within a collegiate level course focused on the skills tested yet given more than an hour to work on their calculator, many *failed to turn in **any** code* and many more provided code without any meaningful plan or approach.

8.2.1.2 Design performance

Given a problem that 'should be' familiar (the calculator), the best efforts of too many students failed to produce working code or even promising solutions given more time. Since the scores on the overall evaluation were so low, McCracken et al. created an alternative measure that did not require working code. A subgroup of the researchers conducted an additional qualitative analysis, the "Degree of Closeness (DoC) score, a five-point scale that rates how close a student's program is to being a working solution" (p. 129). A perfect score of 5 means the program either worked or would have if the student had more time. Each lower score represented something "farther away" from the goal[122]. McCracken et al. reported the DoC per school, but Table 8.1 summarizes the scores of all students. The majority of students struggled to code a working solution, but the DoC score indicates that they even struggled to conceive of a promising approach. Being 'familiar' with a calculator only aided 16% of the students to even come close to a promising design (5 or 4). The researchers categorized the rest (84%) as "Close but far away" (p. 139) or

[122] McCracken et al. reported the average DoC was 2.3, but this statistic is essentially meaningless beyond saying that most of the students were a long way off from a useful solution. My concern with statistical analysis of the DoC lies in is interpretive nature – the difference between a score of 5 and 4 cannot be proportional to the difference between a score of 4 and 3. Even with perfect interrater reliability, the vague descriptions not rooted in the problem structure make this measure non-replicable. That being said, their analysis of these categories seems trustworthy enough given the small group of researchers analyzing the data, so while non-replicable, the general approach is reasonable enough to trust their qualitative interpretations.

worse. More than a third of all students were "Not even close" (p. 139), scoring DoC 1. The DoC analysis provides another fascinating look at student accomplishments or lack thereof.

Table 8.1. The DoC spread from McCracken et al. (2001)

	Degree of Closeness (n = 217)				
	5	4	3	2	1
Percentage (%)	6	10	25	25	35
Count	12	22	54	54	75

Where the general evaluation score (the 110 points) captures the student's full skillset, the DoC score focuses primarily on their design. The researchers overlooked any errors in the code looking instead for a kernel of promising design. They overlooked runtime and even compiler errors to see if the student showed either a potential high-level design or elements within their low-level design that might help solve the problem. Those students scoring 4 or 5 seemed to have a strong notion of how to solve the problem and only lacked time. Those scoring 2 or 3 understood the problem but were a long way from turning their basic plan into code. Very few students (16%) combined coding knowledge and design sense in a way they might have finished the project. Half of the students produced some semblance of a plan but seemingly did not get far in blending language skills with their design. The DoC 1 students (35%) neither lacked even a basic sense of the problem. McCracken et al. further analyzed the most struggling students (DoC 1) by categorizing their designs into three types.

- Type 1 – wrote no code
- Type 2 – wrote code, but without any larger plan that would be fruitful
- Type 3 – Had a plan that did not translate into working code, that either
    - Type A – held a promising approach
    - Type B – used an approach not likely to succeed

McCracken et al. interviewed a subset of students and compared the high scoring students (DoC 4 or 5) and the plurality that scored DoC 1. The next section revisits their analysis through the lens of TAMP to explain why these students struggled to make any progress.

8.2.2 Revisiting the lessons of McCracken et al.

McCracken et al. (2001) assumed that students near the end of their first year of learning to program should be able to build a familiar application, a calculator. They believed students should be capable of the following,

1. Abstract the problem from its description
2. Generate sub-problems
3. Transform sub-problems into sub-solutions
4. Re-compose the sub-solutions into a working program
5. Evaluate and iterate (p. 126)

McCracken et al.'s list of expectations for programmers seems to cover the entire design lifecycle described for experts in Figure 7.15[123]. The selected domain for the problem (basic math) may be less demanding than many potential challenges, but the researchers still expected quite a bit of creative problem-solving. If their goal were to confirm either their assumption of what students can do or create an assessment that measures their expected skills, then the study would have been highly disappointing. McCracken et al. provided a set of rich data describing how novices struggle when trying to bring their learned skillset to bear on typical programming tasks. This section uses TAMP to explain why novices fall short, starting with the students who struggled the most.

8.2.2.1 Understanding why students struggled to get started

Many of the lowest-performing students (DoC 1) struggled to translate the problem description for the calculator into any meaningful progress. McCracken et al. split these students further into three Types (1-3) based on their progress. Since Type 1 students produced no code, they could easily be classified as stoppers. It seems unfathomable that after that much time, some students could fail to produce at least some code, so it begs for an explanation. Type 2 and 3 students made some progress, yet it still seems strange. After all, how many college students are unfamiliar with calculators, much less basic math. Why couldn't they produce even the simplest calculation in an hour? TAMP provides tools to peel away not just our expert blind spots to see the complexity of the problem but also our Cartesian bias that if we just think logically, we can at least make progress.

[123] If in a highly procedural way of thinking. I am not sure expects consciously design in terms of "generating sub-problems" and "re-composing" them, but from a distance this may be what the process looks like?

Stoppers

McCracken et al. (2001) included interview data across a subset of students, including insights into some of the more struggling students. Type 1 students, those who produced no code, complained about several factors seemingly unrelated to the calculator problem.

> They blamed the amount of time available to solve the problem, their unfamiliarity with the computers in the lab, their lack of Java knowledge, and other external factors. None of the Type 1 students mentioned factors related to the process of solving the exercise. (p. 131)

On the surface, it would be easy to dismiss these complaints as excuses from students who did not study or were simply unfit. Consider, though, that most of these students did not receive a grade on their calculator[124], and the study does not report financial compensation to incentivize students. For most, there was no penalty in submitting blank work. Why would the ungraded students choose to linger in an unfamiliar computer lab for an hour, produce nothing, and not complain that the problem was unreasonable? It seems easier to blame the test as being unfair rather than to blame themselves or external conditions. Perhaps they did not think the calculator was strange or difficult, merely that they could not find a place to start their work and blamed true, but unrelated factors.

TAMP's model suggests that Type 1 students did not start because they had no inspiration where to begin. Knowing how to use a calculator does not mean you understand how to build one. Type 1 students likely had little to no *Experiential repository* to inspire even fledgling *Iconic representations of design*. It is very notable that the students did not complain about the calculator being too difficult. Kahneman (1993) proposed that people remember experiences based on either the biggest extreme or what happens at the end. They could have complained about the strange RPN notation, the difficulty in validating user input, the confusing data structures, but instead, they complained about seeming trivialities. Consider the Type 1 students' complaints. Many took the test on 'unfamiliar' computers[125]. They complained about the room being cold. They considered their knowledge of the language to be insufficient. Their complaints seem to imply

[124] We do not know exactly how many Type 1 students the researchers found. Of the 217 submissions in the study, 75 were ranked at DoC 1, and of these only 15 came from a school who graded their work. We do not know of these 15 how many did not turn in code, and if they were 15 individuals or as few as 5 (this school had students submit all three problems). This school had the lowest percentage of DoC 1 submissions, though.

[125] It would be interesting to know what was unfamiliar. Java as a language is platform independent and even C++ typically has somewhat universal tools. Were the tools strange, the keyboards, the lack of their own references?

that they did not deliberate much on the problem. Without sufficient inspiration from System 1, the rest of their knowledge became inert. Figure 8.1 modifies the mental representations of an expert to note what is likely missing for Type 1 students, resulting in their floundering.

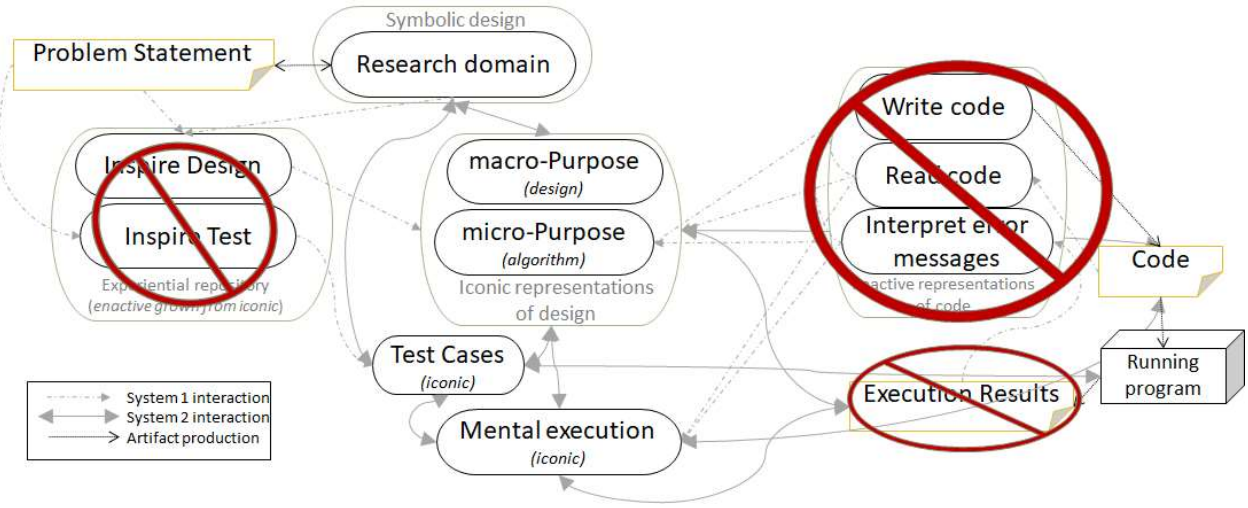

Figure 8.1. Revisiting design representations for Type 1 students from McCracken et al.

Type 1 students likely were missing most or all the essential enactive representations. The specific complaint, "their lack of Java knowledge" (p. 131), hints at their fragile knowledge. There is a lot we cannot know about this student. They could have been failing every aspect of the class, but it would seem the instructor/researcher would qualify such feedback. We cannot know how the instructor provided content, but it seems reasonable to expect the student saw enough prerequisite structures (e.g., basic math, decisions, loops, input/output) to start some coding. Is it easier to believe this student learned nothing, or that they could not activate the needed knowledge when called upon? TAMP suggests that Type 1 students may have survived the course to that point using symbolic knowledge and a combination of mimicry and rote imitation, but they could not tackle the calculator problem since it requires iconic representations grounded in enactive knowledge.

When Type 1 students read the problem statement, System 1 remained quiet on the domain, algorithms, and matters of coding, leaving any progress up to System 2. The problem statement did not trigger any memories about input and output, loops, stacks, or even basic addition and subtraction. Now sitting in an unfamiliar lab, without their books, internet bookmarks, a network of peers, or any of the familiar scaffolds, their System 2 froze. When stoppers faced such a dilemma, they chose to move to the next problem (D. N. Perkins et al., 1986). With only the

calculator problem to solve, Type 1 students moved on to consider environmental problems. Their attention turned to frustration at how little they remembered, annoyance at the strange computer, discomfort with the temperature, or other ideas unrelated to a calculator. System 1 seems essential to at least activate associate semantic knowledge with the current need.

To be fair, I am doing a lot of speculation given a few scant bits of data (Type 1 students did not write code, yet they also did not complain about the problem). It would be interesting to know how well the Type 1 students performed on other aspects of their coursework. Were they failing? Could they trace code? Could they explain code? Could they write code if given some a design (e.g., code this flowchart)? Research such as Lopez et al. (2008) suggests that some of the Type 1 students might be able to do any or all of the tasks above and still struggle to write code like the calculator problem. TAMP explains such anomalies using the different types of memories we form, as well as the need to associate memories within iconic representations to enable creative design intentionally.

Starters, but far from finishers

The remainder of the students who scored DoC 1 must know more than the Type 1 students, but what exactly allowed them to start yet make very little progress? The Type 2 and 3 students received little overall attention in the report, but what McCracken et al. provided reinforces the model in Figure 7.15. McCracken et al. (2001) summarized Type 2 students saying, "they first did what they knew how to do, deferring the tasks about which they were uncertain, but were then unable to proceed beyond that point" (p. 131). Unlike the Type 1 students who had no starting point, Type 2 students received some hint from System 1. Rather than coming from their *Experiential repository*, Type 2 students likely relied on some *Enactive representations of code* to trigger, which triggered learned processes for assembling some specific piece of code. Perhaps they wrote a basic command-line interface or a module to manage basic math. It could be they built several such models jumping, like extreme movers, to the next task when they could no longer make progress on the current one. Without inspiration from their *Experiential repository*, though, they never seemed to form an *Iconic representation of design* from which to organize their pieces. They had enough experience to remember bits of code, but not enough to creatively assemble an application.

The Type 3 students seemed to be a little more advanced than the Type 2. They "apparently understood what they needed to do and appeared to have a general structure for a solution" (p. 131). The Type 3 student received some inspiration from their *Experiential repository* to provide direction for a solution. The inspiration may have been helpful (Type 3a) or set them off in the wrong direction (Type 3b), but it helped form a rudimentary *Iconic representation of design*. Beyond a general framework, Type 2 students seemed either less competent with code than their Type 3 counterparts, as they seemed to lack the inspiration telling how to translate their overall plans into code. It could be that Type 3 students relied on the resources they no longer had available (e.g., books, examples, peers). They solved problems symbolically, and thus struggled without the external resources that scaffolded their immature System 1.

The inclusion of Type 3b students strengthen the hypothesis that the experiential repository is critical to design. If we consider the playground project, students could work for an hour and receive no feedback on the quality of their design. Even without books and examples, a student can presumedly know when a calculator is not working. Type 3b students, though, continued to work *for an hour* towards an approach that was not likely to work. Even if they did not engage System 2 to review their plan, possibly recognizing and abandoning an incorrect approach, they wrote code, conducted tests, and edited code. It may be beyond my expert blind spot to imagine their work process without seeing the results. I would probably start with a very simple problem, 2 + 2 = 4, and code until this sample would work. The Type 3b student's inability to recognize a poor approach could also be exacerbated by deficiencies in the other half of the *Experiential repository*, *Inspiring test* (see Figure 8.1). Rather than working from a logically simple test case, the Type 3b student may continue to follow the poor framing from the design half of their *Experiential repository*. Since test cases are typically implicit within programming classes, it may be that many students are also unable to think of tests when needed, which is the focus of the third study in this section.

What makes some students successful?

System 1 may be critical in getting programmers started, but System 2 is still the workhorse of problem-solving. McCracken et al. further investigated three students who scored DoC 5 and of the three, only one rated the problem as easy, the other two marking it as difficult and hard.

None of the DoC 1 students thought the problem easy, yet only four out of twenty-five ranked it as impossible, the others also said it was difficult or hard. Unlike the low scoring students who did not mention the problem, the high scoring students talked about their experience solving the problem.

> Many of these explanations illuminated particular aspects of the design phase or particularly challenging sub-problems. Examples of comments made by such students were "Simple errors got the best of me" (problem difficulty rated as difficult), "Could not solve for error case" (problem difficulty rated as hard), and "Implementation is wrong but easy" (problem difficulty rated as easy). (p. 131-132)

Students who scored DoC 5 not only derived a plan and wrote significant amounts of code but could speak to their struggles while working. Rather than external factors, they remembered the work that caused the most deliberation (e.g., frequent simple errors, error cases). It would seem that the students who scored DoC 1 would have experienced very little cognitive load, while those scoring DoC 5 spent the most time engaged in complex reasoning.

The students who scored DoC 5 might simply have been exceptional, yet even assuming individual brilliance seems to support TAMP's model. The nature of the stronger student's advantage can take several forms – inherent abilities (fluid intelligence within System 2), prior learning (enactive representations), or being a better learner (iconic representations, see Section 7.4.3.2). Is inherent ability important to programming success? A stronger System 2 would naturally support the rational thinking required in many aspects of programming. I believe this argument is fairly easy to supplant. To put it kindly, not every strong programmer is universally brilliant, and everyone who is brilliant is not a good programmer. Brilliance may increase the rate of learning, yet remember, only one DoC 5 student considered the problem easy, and they believed they had the implementation wrong. What seems more likely is gifted students learn quickly, giving them more time to work on additional examples, further adding to their seeming intellectual prowess. I believe that providing focused experiences can help 'everyone else' learn to program, but more importantly, instructors should believe in the potential of their students.

A better explanation for high-performing novices is the advantages of prior learning rather than inherent abilities. Pre-college programming experiences provide early access to programming knowledge, but within a few weeks, most college courses cover the expanse of high school courses. Students who arrive with no experience sometimes overtake students who have

taken classes. It is not the facts that make prior learning useful, but the experience, at least when it is relevant and accurate. Students who either received poor teaching or possibly overconfident students and neglect to practice might fall behind peers who are starting without any prior experience. McCracken et al.'s study provides an excellent counter-example of semantic knowledge in programming. I must assume the four instructors believed their students received instruction on all the basic language constructs required to build a calculator, yet 84% of the students were nowhere close to a solution. If the students generally knew enough about the language to succeed and did not, then perhaps knowledge about the language is much less important than knowledge *about the problem*, as we will see in a minute.

A third explanation suggests that the students who excel are better learners than the rest of the students. Answering this argument necessitates defining what is meant by 'better' (i.e., better at memorizing information, answering questions, completing timely homework, creatively solving problems). The nature of McCracken et al.'s study implies that programmers need to be problem solvers. Better learning in that sense would mean better at creating, manipulating, and maturing iconic representations. Building iconic representations requires a combination of experience and symbolic knowledge, so students with more fluid intelligence and prior learning hold an advantage here as well. Iconic representations are not formed merely by being clever or experienced, though; they require conscious reflection on the use of knowledge. Some students may naturally form the required iconic representations, but it is possible for instructors to promote the formation of iconic representations through select pedagogy (see Chapter 9).

TAMP provides a model of cognition that explains how any person can learn to program. Hidden within the rest of the data provided by McCracken et al. is evidence supporting not only TAMP's model of expert programmers when designing (Figure 7.15) but also the analysis above. To fully appreciate the role of the experiential repository in design, it is helpful to understand why the calculator task is not as easy as presumed. McCracken et al., presumedly believed that students could quickly grasp the task of building a calculator, given that calculators are ubiquitous devices with simple mathematical operations. Ironically, Du Boulay et al. (1981) used the example of a calculator to note how even simple devices have deceptively complex inner workings.

> For instance the manuals accompanying certain makes of pocket calculator *make no attempt to explain the reason why given sequences of button presses carry out the given computations*. The user must follow the manual's instructions

> blindly because it is difficult for him to imagine what kind of underlying machine could be inside that demands these particular sequences of presses. During the course of a calculation, he has to *guess the current state of the device* using his recollection of what buttons he has pressed since the device's previous recognizable state (e.g. all registers cleared) because the device gives little or no external indications of its internal state. (p. 238, emphasis added)

Being familiar with using a calculator provides no insights into strategies for building one. The user of the calculator is only concerned with their actions, not the logic within. If the logic of a calculator were not enough, knowing addition and subtraction does not explain Reverse Polish notation when most (all?) pre-college math uses infix notation. McCracken et al. learned a valuable lesson in tacit knowledge when students scored better on the infix problem than the presumed-easier RPN problem. Familiarity is very helpful in problem-solving, but only when accompanied by deeper knowledge. A few students who recognized that stacks might simplify the RPN calculator noted that they were not "good with stacks/queues" (p. 133). Students need deeply associated knowledge that only comes through deliberate practice.

The hidden proof within McCracken et al.'s study came from one ambitious instructor's use of the calculator problem as a formal assessment. The students of School V, as anonymized in the study, scored on average 3-4 times as many points as the other schools. The report offered two reasons why this may be so.

> (1) The School V instructor *had given the students an example to study*, which was *a complete answer to a similar problem*, and (2) All students were required to take the exercise, which was given as an examination. (p. 132, emphasis added)

School V students had both the extrinsic motivator of a grade but, more importantly, had seen a similar problem before taking the test. While receiving a grade may incentivize students to not give up on a test, TAMP suggests the example went further in promoting success. School V held an advantage over the other schools, but it does not seem to be universal across all students. Table 8.2 shows that while School V's averages were much higher than all participants, so are the standard deviations. School V still had a significant number of struggling students (44% received DoC 1 or 2). The instructor at School V provided a notable advantage to their students by providing an example, but that example only helped to a degree.

Table 8.2. General evaluation scores for all schools (McCracken et al., 2001, p. 130)

School	Average	Standard Deviation	Interface type
S (n=73)	14	18.6	RPN
T (n=21)	12	16.3	RPN
U (n=47)	8.9	11.4	Infix
V (n=23)	48.7	25.7	RPN
V (n=30)	47.8	29.1	Infix
V (n=23)	30.9	20.9	Infix with parentheses

TAMP explains why the participants at school V performed significantly better than their peers (or at least did not struggle as badly). The similar example inspired how to tackle the calculator problem. It is important to note that School V students still performed relatively terrible scoring less than half of the available points. Seeing a similar example did not make the problem easy; it merely gave them a place to start. As demonstrated in Fix, Wiedenbeck, and Scholtz's study, novices are unlikely to memorize an entire solution, but they might remember where to start. The students from School V who internalized the example probably already developed a useful set of enactive representations within their *Experiential repository* that took them further than many of the rest, but nobody aced the exercise. Those with little experience might only have recalled surface details of the example within their semantic (symbolic) memory.

School V's students all benefitted from the preceding example, but how much depended on the maturity of their representations from Figure 7.15. The calculator problem required much more than a hint at a solution. Memorized processes could not help anyone solve the entire problem, so the quality of the solution varied based on each individual's maturity. School V's instructor enhanced his/her student's ZPD in several possible ways.

- Students with a weak System 1 received a hint about where to begin and a general structure based on a similar example. The example essentially scaffolded their *Experiential repository* preventing them from being a stopper and reducing the DoC 1 students (20% for School V versus 34-70% in other schools)
- Students with some System 1 support internalized the example and made significantly more progress. Even if their gaps in *Enactive representations of code*, *Inspire Testing*, or other gaps in knowledge did not result in a high general evaluation score, 32% of School V students scored a DoC 4 or 5, compared with 0-11% at other schools.

The fact that not all students equally benefitted at School V implies either a wide variance in study habits or, like TAMP suggests, our mind has several repositories of knowledge that build upon existing knowledge and skills.

If the full dataset from 2001 were available, it would be interesting to compare the maturity of the solutions across schools and students. TAMP might predict that the School V students produced more promising structures (e.g., more likely to include stacks for Problem 1[126]). They probably were better at starting their user interfaces, but this may not have translated to error handling, which is a detailed procedural task that would be difficult to master from just one more example. TAMP suggests that unless the other example was some version of a calculator, the general evaluation scores remained low because School V students did not have any advantage in imagining useful test cases. Perhaps a future study can reanalyze the original data or attempt to reproduce the original. In the meantime, the next study already tested a modified calculator program with compelling results.

8.3 Simplifying the calculator problem – McCartney et al. (2013)

McCartney, Boustedt, Eckerdal, Sanders, and Zander (2013) wanted to address some of the concerns in the original study by McCracken et al. and see if the calculator test could become a useful assessment after a few targeted improvements such as,

- *Simplifying the problem description* – focus on the infix calculator, remove extraneous verbiage from requirements (especially 'implicit' content) and add more examples
- *Provide support code as scaffolding* – Students received a startup framework including a helper to read and parse user inputs (e.g., is the next input a number, operator, or quit command)
- *Allow students to use resources* – Allow students to use books or online references in completing the problem since it "seems reasonable to be able to look up the syntax of programming structures (switch e.g.) that students have litte [sic] experience with" (p. 93)

[126] Note that school V scored slightly higher on problem 1 despite the other schools generally scoring better on problem 2

Their stated goals included removing "troublesome knowledge" from the problem description, limiting cognitive load, and providing a familiar environment. The changes helped to improve student performance but only to a limited degree. TAMP helps explain what helped and why some students continued to struggle.

8.3.1 A marked improvement

The participants within McCartney et al.'s study had several advantages that contributed to their improved performance over the original study. In addition to the changes the researchers made to the problem, the researchers provided more details on their participants' demographics. The participants likely held more experienced than many of those taking McCracken et al.'s test since this was at least their second programming course as well as programming language. The combination of more experienced students taking a 'simpler' test showed marked improvements.

> Eighty-five percent of the students made real progress, but just 20 percent had complete solutions, which indicates that the problem is not too simple for the amount of time and the students' capabilities. (p. 96)

It is difficult to quantify what McCartney et al. mean by "real progress", but it seems to be quite a flip from the prior study where 84% of the participants were "close but far away" or worse. McCartney et al. did not attempt to recreate the vague DoC but did use the same 110-point general evaluation scoring system. Their students averaged 68.2, triple the 22.9 of the earlier study[127], but still far from masterful. Table 8.3 presents a comparison of the performance per category in each study.

Table 8.3. Comparing the categorical performance of students between McCartney et al. (2013) and McCracken et al. (2001)

Category (max)	McCracken et al.	McCartney et al.	Improvement
Execution (30)	7.2*	29.3	*+22.1*
Verification (60)	1.6	25.8	*+24.2*
Validation (10)	0.3	8.1**	*+7.8*
Style (10)	4.6	5.03	*+0.4*

** As noted in the prior section, this number might be low (possibly 16.6)*
*** The study does not specify the Validation score, so this number is computed from the other categories and the reported average*

[127] The average for the infix notation problem was 24.1, the only problem McCartney et al. set.

The students in the second study proved much better at producing useful code with a nearly flawless *Execution* score. It logically follows that when students establish a working program, they are more likely to pass tests, resulting in a leap in *Verification* and *Validation*[128] scores. McCartney et al.'s student passed roughly half of the test cases for basic functionality and a third for advanced features, but interestingly less than a third of the submissions seemed to include the quit feature. The results remove any impression of 'bimodality' and produce scores across a more traditional range of student performance.

McCartney et al. seemingly produce a more achievable assessment than McCracken et al., but do not capture all the reasons why. They attributed student success to

> First, they were working in a familiar context…
> Second, the students' cognitive load was significantly reduced…
> Third, some elements of the [McCracken et al. study] that may have involved troublesome knowledge were eliminated. (p. 96)

McCartney et al. overlooked the advantage their students held in this being their second programming experience and within an additional language. They also only attributed the code they provided as helping to manage cognitive load. By choosing the infix notation, they provided an example that was more familiar to students, but is an unfamiliar problem truly troublesome? Leaving the training wheels on a bicycle avoids troublesome crashes, but then has the rider truly learned to ride a bike? I do not wish to dispute the analysis by McCartney et al. so much as I hope to refine their observations with more detailed explanations. Their analysis is fair at the level of abstraction used but sheds little light on *how to prepare students*, not just assess them fairly.

8.3.2 Helping novices think like experts

The theoretical constructs of TAMP provide the tools needed for explaining the various changes McCartney et al. made. Specifically, their modified testing protocol provided scaffolding for the enactive representations that aid in design and coding, as shown in Figure 8.2.

[128] The higher *Validation* score could have been the result of a more generous scoring rubric. McCracken et al. did not provide much details on how they scored this, so McCartney et al. might have used different criteria. This analysis ignores *Validation* since it is so ill defined, but it is worth noting since the 'extra' points amount to a third of the average score of McCracken et al.'s students.

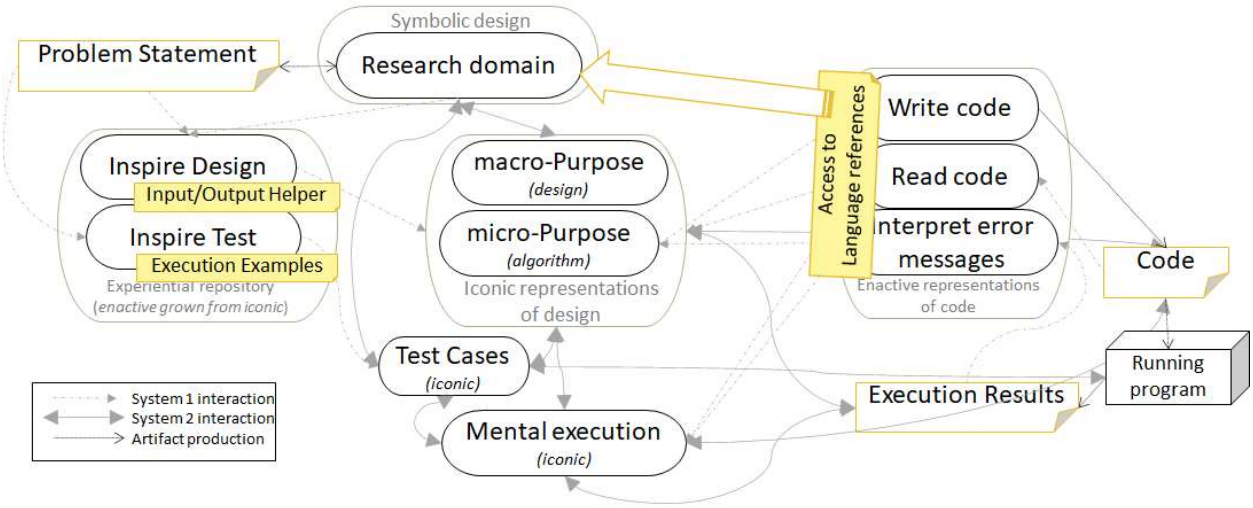

Figure 8.2. The likely mental representation of McCartney et al.'s students

The three main support structures helped make the student's thinking act more like an expert's by providing a supplement for their still-maturing System 1.

- **Input/output helper** – McCartney et al. provided this helper since the class had previously covered input/output in the console. Scaffolding low-level user input would typically just help *Enactive representations of code*, but this helper did much more. First and foremost, it provided a model of the problem. The researchers told students, through the interface, they needed to care about numbers, operators, and quit commands. Students did not need to derive this vital information from the problem description nor plan algorithms to parse text into variables usable by the calculator. In short, the helper code provides much of the information that would otherwise come from the *Experiential repository*, indeed reducing cognitive load, unless the students would have stopped.
- **Execution examples** – Providing extra examples reduced the need to imagine test cases from the problem description alone, providing inspiration that otherwise must come from the student's *Experiential repository*.
- **Language reference materials** – Providing access to language reference materials it removes the need for System 1 to prime System 2 with relevant facts about language structures. When students are unsure how to write code or the meaning of an error, they can fully use System 2 to search for answers.

The aids to students were well designed, supporting the exact areas novices are at a disadvantage because of their lack of experience.

The students from McCartney et al.'s study did not struggle with basic coding in the way that McCracken et al.'s students did. Of the forty students who took the test with the scaffolding, thirty-two produced code that ran without errors and passed at least one of the test cases (80%); a feat that less than 20% of McCracken's students seemed to do[129]. Only one of McCartney et al.'s students (3%) submitted code with compiler errors. TAMP suggests that the new batch of students were probably more successful in writing runnable code due to their prior experience. Coming into their second or more experience in programming, these students would have formed at least a semblance of a notional machine in their prior language (Matlab or Schema). The additional practice, possibly combined with a reflective comparison between the two languages[130], provided richer *Enactive representations of code* than true novices on their first language. The first advantage provided by McCartney et al. was selecting experienced students.

The one student who struggled to produce a working program may be the outlier who proves the importance of experience in programming. The students were given 75 minutes to build their calculator *and had access to reference materials*. Seventy-five minutes would seem plenty of time to at least ensure compiling code, and I expect most experienced programmers are like me in obsessively fixing compiler errors the instant they appear[131]. Whether the 39 other students used reference materials or coded entirely independently, most had enough experience to not only complete compiling but also at least partially working functionality. The outlying student had plenty of time to produce a rudimentary application even if it did little as a calculator. Their erroneous submission implies the not only did not master basic syntax but also did not develop the processes by which to correct their problems using reference. If researchers someday chose to conduct a similar study, it would be interesting to track how students use reference materials in correlation to their results. TAMP suggests that successful students would rarely look at core

[129] I am inferring from the number of students who scored 40 or above. To pass a test case they must have scored most of the execution points, some of the style points and then some of the verification/validation points, so 40 seems a conservative number.

[130] I have considered, not yet strongly, the role in learning multiple languages in maturing the notional machine. I have many thoughts and will write more in the future, but for now I will say that there are risks and benefits for teaching multiple languages simultaneously, but adding a second language after significant instruction in the first seems not to harm students, though it may risk slowing down other areas of maturation.

[131] I am so bothered by underlined red text I also must immediately address any spelling or grammatical errors my editor points out!

syntax/semantic information and rather focus on entirely new information (e.g., some construct they have not used before or error messages they may not have encountered). It may be that the struggling student also found little help from reference manuals as System 1 must drive the syntactic part of writing code to make any substantive progress when the task also demands creativity.

The main advantage McCartney et al. provided was a promising start to the problem. The same experience that helped students write error-free code would have helped in design, but the helper code and additional examples provided a foundation from which students could build. As an analogy, McCartney et al. presented a jigsaw puzzle and provided a picture of the goal and completed the outside edges as a guide. If you have never worked a jigsaw puzzle, completing the outside makes the entire puzzle significantly easier as it provides a starting point and bounds the effort, literally in this case as well. Many children do not intuitively realize that the flat pieces of the puzzle indicate a border, just like it seems many of the programmers in McCracken et al.'s study could not find a reasonable starting point. McCracken et al.'s original problem not only failed to provide a clear border to students; it may have resembled a borderless puzzle that has no straight edges! By supplying the input helper, McCartney et al. directed their students where to start the problem[132] plus strong context clues on how to proceed.

TAMP suggests that the *Experiential repository* acts as the border of a jigsaw puzzle for jumpstarting design cognition. System 1 primes the programmer's *Iconic representation of design* with a promising design approach inspired by their abstracted/amalgamated experiences. Without such inspiration and lacking any scaffolding, many McCracken et al.'s students struggled to find a promising (or any) start. It might seem that novices are unable to apply knowledge of the programming language logically to derive a solution. Experts, however, also seem to start from one big idea and then logically consider their alternatives. Atman et al.'s (2007) asserted that experts pick an approach and iterate from that core idea. The input/output helper scaffolded the design process by modeling key aspects of the problem *in code*. The problem description included example code calling the helper and an example of the output from a running program (Figure 8.3). On top of that, the output of the example demonstrated three possible test cases. McCracken et

[132] Not only is it helpful to attend to the user interface as a framework for the problem, the framework is new and has a lot of text to read as part of the problem statement. The included information, as mentioned helps to outline not only the user interface aspects of the problem but also the structure of the data and provides a model of the application's core actions.

al.'s test required students to imagine the entire solution based on a rough description[133]. McCartney et al. provided most of the analysis work and the start to a high-level design, essentially allowing students to mimic the example and fill in the algorithmic gaps alone.

```
Enter expressions one per line, q to quit
12 * 3.5
Result = 42.0
12 * 12 * 12 ^ .5
Result = 41.569219381653056
5.432 / 2.301 * -6.2
Result = -14.636418948283355
q
All done, have a nice day
```

Figure 8.3. The example provided to students for the calculator program (McCartney et al., 2013, p. 98)

McCartney et al. asked students to perform a fundamentally different skill than McCracken et al. While McCracken et al. probably felt they specified the problem well, it still left students with a gap of where to start. By providing the helper (where to start) and the example flow, students were not required to imagine a high-level design so much as to mimic the provided interface (Figure 8.3). Vygotsky believed that mimicry was a way of determining a student's skills and that students cannot mimic work outside of their ZPD. McCartney et al.'s students did not need a vast or even targeted *Experiential repository* since the helper primed their *Iconic representations of design*, acting as a more knowledgeable other. The example in Figure 8.3 removed the need for already possessing similar experiences (*Experiential repository*) or for planning an imaginative high-level design to solve the human interactions. Instead, the problem became one of realizing the algorithms hinted at by the provided resources.

8.3.3 Helping novices test like experts, to a degree

The refinements provided by McCartney et al. also served to aid the quality of student testing, when directly addressed. The researchers described tests the students passed (or not), that

[133] Presumedly the description was rough, as McCracken et al. did not share the exact description provided to students and McCartney et al. thought it helpful to rewrite the description with examples.

seem to correlate with the examples provided (or not). McCracken et al.'s students averaged 1.6 *Verification* points out of 60[134], meaning most failed to pass a single test. With the provided scaffolding, 32 of the 40 students passed at least one basic computation test (a.k.a. benchmarks). McCartney et al. provided code samples for calling the helper, which helped to get students started. The samples provided little guidance to solving the most basic computation, yet half of the new batch of students correctly implemented all the basic computation tests, and only one in five failed to pass any test (only 16% of McCracken et al.'s students had promising, yet not all working solutions!). The example not only served to guide the design, but it also provided three specific test cases, which by no coincidence are the ones that students were most successful in completing.

The students' success changed dramatically for test cases that lacked an explicit example. As strongly as students performed on the benchmark test cases, they struggled with the rest. McCartney et al.'s (2013) choice of what to trim from the problem description provided an excellent example of how intuition seems do drive design and testing.

> We omitted some requirements that we hoped *did not need to be explicitly mentioned*, such as error-checking. (p. 93, emphasis added)

The researchers assumed that the advanced novices would automatically add logic to handle poor inputs as part of their program yet,

> Only 7 of the 32 programs that could handle any of the benchmarks could handle any of the erroneous inputs – the other 25 either crashed or printed an erroneous result. It should be noted that the task description did not mention dealing with erroneous inputs. (p. 94)

McCartney et al. perhaps exposed an expert blind spot when expecting that students would automatically test for invalid inputs. They perhaps assumed that students would remember the importance of input validation based on prior instruction and perhaps practice problems.

By omitting both a requirement to validate inputs and examples of bad inputs, the researchers relied on the students to remember the importance of validation and how to implement it. The students were allowed resources, so they may have had access to notes, previous examples, or could have searched for examples on the internet. Students did not need to check for any random user input since the helper code already parsed input text. Their only responsibility was to order

[134] Unfortunately, there is no further breakdown. It would be valuable to know the averages for students who wrote working code, as this seeming is a prerequisite to passing any test cases.

numbers and operators properly. They likely did not add such code because they never thought to; they received no inspiration from their *Experiential repository*. The seven students who included input validation could have done so by accident or habit. A few may have accidentally entered an erroneous input (e.g., "2 ++ 2") and then fixed the resulting failure. A few may have had enough experience to include invalid inputs as part of their design or test cases. McCartney et al.'s assumption that students would automatically include error handling in their code seemed overly optimistic in retrospect. Given that the researchers created the helper to compensate for the students' lack of experience in *building user interfaces*, TAMP would suggest they naturally had no experience in validating user inputs, and most would have never thought to do so.

The quit feature offers a paradoxical test case. The problem description included quitting in the general flow of the program (Figure 8.3), yet less than one-third of the students included a quit option. If the students used the example from Figure 8.3 methodically, they would probably have thought to include the quit feature. At the very least, most students did not return to the example after working out the basic computation features to catch the quit option. The missing quit option means they also did not carefully read or return to the description of the helper code, as that also included a command for quitting. Two-thirds of the students entirely overlooked two places where the problem statement mentioned the quit command, plus ignored the simple question, "how do I stop this application?" every time they tested. The lack of quit features in so many calculators implies that novice design and testing processes were far from methodical. As with validation, the students who included quitting most likely were those who did so out of habit or were frustrated by continually having to abruptly exit the program and had the System 2 bandwidth to modify their design. Every student who passed any test (32 of them) *must have seen the need for quitting* but did not add this feature (~19 of them).

The missing quit feature seems to reinforce the role of intuition in programming. The students cannot be unaware of the quit feature (from the example and the practical need to end the program), so this is not even a case of intuitively needed to know that quitting was important. What students lacked was both an intuitive strategy of when and how to quit. Remember, McCartney et al. included the input/output helper because the students had little or no experience in command-line applications. They had not seen many (or any) algorithms that described how to manage user inputs. When left to System 2 to include this basic feature in their design, two in

three students forgot about it, not just at first, but over 75 minutes of coding and testing[135]. I say 'habitual' because the paper fails to mention student questions during the test, so they must have learned to kill failed programs in previous assignments and did so without needing to be instructed. The quit feature both demonstrates the benefit of building an *Experiential repository* in novices – guiding design – but also how easily programmers can overlook obvious needs – giving users a way to exit their calculator – due to habit.

8.3.4 What we can learn about scaffolding

The updated assessment McCartney et al. added well-designed scaffolding to supplement gaps in what students have learned about problem-solving. A remarkable number of McCracken et al.'s students showed little or no promising work. McCartney et al. provided scaffolding – helper code, testable examples, and external resources – that helped students performed better, but only in very specific and explainable ways. The areas left unaddressed showed little or no improvement beyond the aid already provided. The students who continued to struggle helped illustrate the different types of knowledge novices must acquire to become experts. By providing a starting point (the helper code) and a user experience to mimic (Figure 8.3), McCartney et al. created an assessment that better aligned to the maturity of their students. Most of their students could write algorithms given specific guidelines. Very few included additional features that are implicit to experts. Scaffolding might support gaps in student knowledge but does not increase the transfer of knowledge beyond its explicit intent.

McCartney et al. also provided a dataset that highlights the different stages of novice thinking when programming. McCracken et al.'s assessment tested a very large black box – build a calculator using these few rules to guide you. McCartney et al. provided a focused assignment within a much smaller box – using this input/output code, and this example of what the interface should look like, build a calculator. Not only did all but one of their students build working code[136], but they passed many of the basic test cases. The test cases they failed to pass speaks to the

[135] I struggle to determine how they tested their code without this becoming an annoyance. My only guess is they habitual used the 'kill' command from their tool to stop each execution? Their testing process is a microcosm for how easily our System 2 will focus on one problem to the exclusion of others (i.e., "how annoying is it that I can't quit my application easily! Maybe I should fix that?")

[136] Which I think is equally attributable to this being their second class, but the scaffolding helped to avoid locking down System 2 with confusion.

importance of testing in the design process. It seems novices do not design for features they cannot imagine testing. The tests that had examples went well, outside usual hurdles in solving logic, but the ones without examples often went unaddressed. Most students even forgot to complete the quit feature – that appeared in the example and had an explicit command in the helper. The guidance added by McCartney et al. led students to more complete solutions, yet most still struggled to provide robust and error-free functionality beyond the explicit expectations.

8.4 Takeaways from McCracken and McCartney

The assessment created by McCracken et al. (2001) and refined by McCartney et al. (2013), redefined what instructors and researchers might expect from students given the current state of pedagogy. While innumerable uncontrollable factors impact each of the studies, the trend indicates that students perform better when the tasks are scaled and scaffolded appropriately. The hopeful good news: students are learning to code and can do so in the right circumstances. The undeniable bad news: at the end of their initial programming experience, far too many students lack the broad range of the skills required to be an independent programmer. McCracken et al.'s list of skills in 2001 matched that of computational education literature preceding it (see Chapter 2). It seems in the decade that followed the expectations of at least these researchers diminished for programmers reaching the end of their first programming courses. If CS1 does not provide students with the full skillset of a programmer, what course in the curriculum will pick up the gap in design and testing left when removing the scaffolding? TAMP strives to understand and better yet explain the gap, and possibly suggest improved pedagogy.

TAMP suggests that, above all, ***varied and deliberate practice*** is the key ingredient needed to help programmers mature. The three studies seem to show exactly how far conventional computing pedagogy can take an aspiring programmer. Programmers need a combination of facts and experiences to become creative problem-solvers. Computing majors will eventually see enough coursework to promote the various types of intuition that Figure 7.15 describes, but too many programmers do not persist until reaching that point[137]. I believe it is possible to include pedagogy that promotes intuition earlier in the curriculum. A curriculum that is cognizant of the

[137] And non-majors may not encounter another programming opportunity until after their declarative memories degrade.

value of intuition also has the effect of not only reducing cognitive load but also mitigating some of the adverse 'non-cognitive' factors (e.g., imposter syndrome). The high dropout rates of computer science students (Flinders, 2019) suggests that too many leave the discipline long before they could mature System 1 in the ways of programming, thus long before they start to understand and feel confident in the subject. The three main studies of this chapter seem to indicate that even those who succeed seem less ready than desired. These three studies described where students are learning but show how much further they have to go to be strong programmers. TAMP offers an alternative perspective of what 'being a programmer' means that may inspire new understandings and innovations to help students succeed.

9. APPLYING TAMP

This chapter considers the implications of TAMP for two purposes: to illustrate TAMP's efficacy as a guide for researcher and educators (validation of the theory) and offer examples of it in use (training on the use of theory). Readers who wish to move beyond abstract constructs and propositions may benefit from examples of TAMP applied to research and teaching practice. Each example in this chapter seeks to plant a seed for thinking differently about how people think and learn computing. The discussion in this chapter is not exhaustive, nor is it intended to be. I sincerely hope that others will find new and creative uses of TAMP and their work will expose the cognitive biases that I cannot currently see beyond.

The structure of this chapter is much like episodic memory – retrospective and prospective. Within this dissertation, I have revisited numerous studies and pedagogical interventions, some in quite some depth. I used the data from these studies combined with new ideas from theory to reconstruct new interpretations. Like prospection, TAMP can guide future studies and innovations to pedagogy. Before looking at applications of TAMP, a summary of the last three hundred pages of discovery may help understand the scope of what TAMP is and can be.

9.1 What is TAMP?

TAMP models the cognitive activities of programmers to capture the types of knowledge that experts acquire and how they use that knowledge in their thinking. This dissertation validated the plausibility of the included theories as applied to computing education but captures a small slice of what TAMP could become. The contributions this dissertation makes to the theory of computing education include:

- Established the value of dual process theory as an alternate model of cognition
- Proposed Bruner's representations as a model for capturing the various types of knowledge and different 'ways of knowing' programming concepts
- Defined the Applied Notional Machine (ANM) as a computing education-based theoretical construct for how programmers store and apply knowledge
- In revisiting past studies,
    - Offered alternative interpretations of fragile knowledge and explanations of novice performance on tracing tasks
    - Illustrated why novices struggle to complete open-ended coding assignments and how selected scaffolding supports gaps in design knowledge

Before considering the possible uses of TAMP, this section summarizes each of these contributions and the possible future areas into which TAMP can grow.

The foundation of TAMP lies upon the two cognitive mechanisms described by dual process theory: the conscious reasoning of System 2, which relies heavily upon the tacit knowledge and skills of System 1. The intuition and habits of System 1 drive much of our daily activity. System 2's primary function is tackling novel problems, but even here, intuition plays a role in problem-solving. System 1 primes our deliberations within System 2 by providing information it finds relevant from experience and offering support via automated skills. Within computing education, dual process theory further explains such notions as fragile knowledge (Section 5.2.1). Dual process theory suggests, for example, that Perkins and Martin (1985) were able to trigger fragile knowledge by stimulating System 1 in new ways since priming is a major factor in System 2. Similarly, dual process theory helped to explain the presence or absence of sketching during tracing activities. Sketching aids System 2 by helping it manage short-term memories, but the need for this diminishes as new programmers automate the mental execution of language constructs. Furthermore, revisiting Lister et al.'s (2004) analysis (Section 5.2.2) indicated that intuition may drive design decisions, as students chose familiar, yet incorrect answers that are easily disproven after a confirmatory trace. These examples provided initial evidence that dual process theory offers new insights to computing education.

Having validated dual process theory, TAMP established the representations of Jerome Bruner as a model of memory and learning. Bruner defined three types of mental representations, enactive, iconic, and symbolic that TAMP links to different aspects of dual process theory. Enactive representations and System 1 both model the same memory structures and cognitive processing. Enactive representations form implicitly, though we can curb our intuition and habits with deliberate intent, often based upon external sources of knowledge. Bruner modeled our memories of external information within symbolic representations, which TAMP equates to semantic memories, the memories of fact. Symbolic representations sometimes stem from specific systems of symbols, like mathematical formulas or a programming language. TAMP suggests that fluency with new systems of symbols also requires the support of enactive representations. Enactive and symbolic memories do not always form at the same time, though (e.g., we learn to speak grammatically without needing to learn the rules of grammar). These two types of knowledge influence each other but, like dual process theory, have complex interactions.

Bruner's theory models the interaction between implicit and explicit knowledge in his iconic representations. Iconic representations are the bridge between enactive and symbolic representations. TAMP clarifies the nature and role of iconic representations beyond the oversimplification as mere imagery, expanding upon Bruner's hint that they are vital in problem-solving. TAMP suggests that iconic representations resemble episodic memories, which associate various types of knowledge and allow for future planning (i.e., prospection). TAMP blends the mechanics of prospection with iconic representations as an aid in problem-solving. This refined set of Bruner's definitions provides theoretical building blocks for describing the way our mind organizes information and experience and serves as the foundation for describing programming knowledge.

The Applied Notional Machine builds upon Bruner's refined representations to reenvision a longstanding theoretical construct in computing education. The notional machine is a useful construct for describing programming knowledge but provided no model of cognition or learning. The Applied Notional Machine reorganizes the concepts of the notional machine within symbolic, iconic, and enactive representations. Symbolic representations hold a programmer's consciously learned rules of syntax and semantic rules (i.e., *knowing that*), where the automatic ability to read, write, or predict the execution of code (i.e., *knowing how*) forms within enactive representations. These two representations capture the knowledge traditionally provided by instruction and notional machines, but the ANM adds to the mix a mental model of design knowledge within an iconic representation. The iconic portion of the ANM models the use of a programming language to solve problems. It encapsulates several vital categories of knowledge identified in computing education literature that experts acquire with experience (Section 2.3.3) and often automate. Exceptional programmers develop rich mental libraries of design patterns, sample test cases, tips for handling errors, methods of researching information, debugging strategies, and other such pragmatic items outside the typical scope of programming classes. The ANM provides a structure for modeling the various types of knowledge a programmer must learn and, using existing theory, helps define how our mind acquires and uses such information.

The Applied Notional Machine is the most significant contribution from this work and offers a central construct for the evolution of TAMP. Figure 9.1 includes existing and possible future elements of TAMP: concepts, constructs, and propositions. Using dual process theory,

neuroscience research, and Bruner's representations, TAMP defined an 'updated' representation model based on the work of Bruner. The ANM leverages this model as well as measurable observations from computing education that theory suggests. For example, dual process theory suggests that measuring a programmer's speed in complete tasks (e.g., writing syntax, tracing, fixing errors) may indicate the mechanism of cognition at play. Likewise, the presence and nature of sketching, combined with the level of their success, may hint at how a programmer is using System 1 or 2 during tracing. Using the basic building blocks of the ANM, this work proposed models of expert thinking during code comprehension and design (Section 7.6).

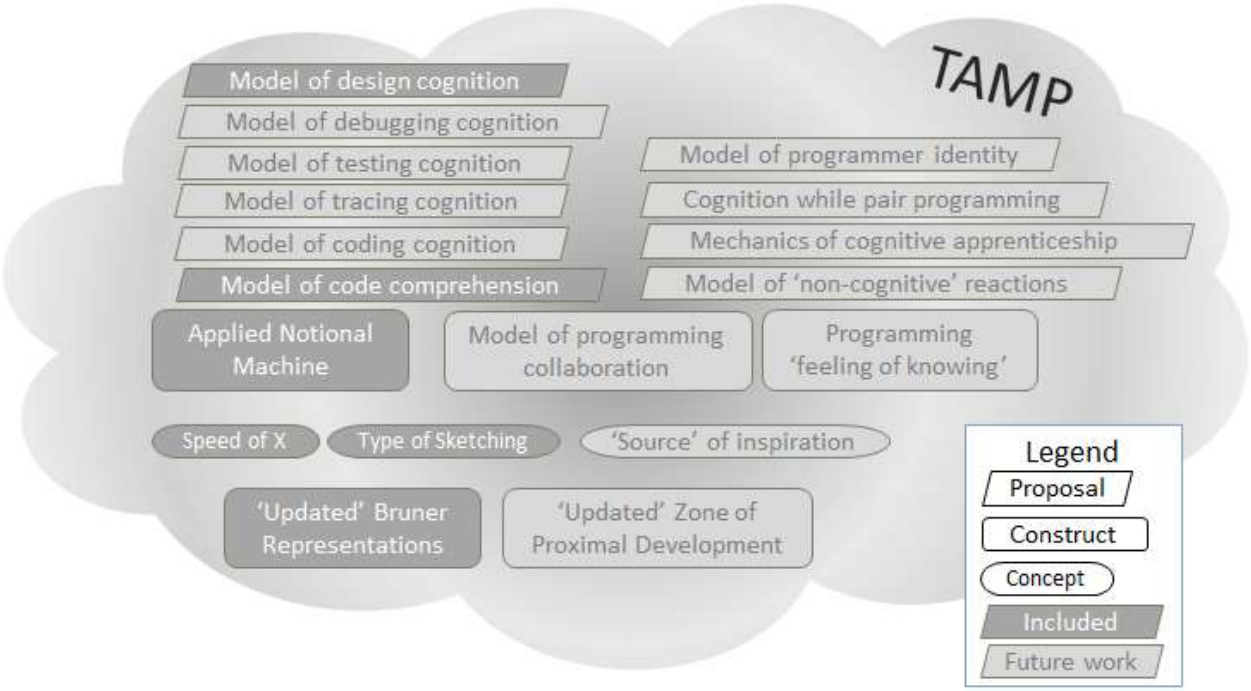

Figure 9.1 The contents of TAMP, current and proposed

TAMP, as a *theory of mind,* should someday move beyond select aspects of expert cognition and include 'non-cognitive' factors and perhaps intermediate stages of learning. The major effort of this dissertation was establishing the underlying theory, its applicability to computing education, and the ANM as an example of a discipline-based theoretical construct. This limited scope is useful for describing expert cognition as captured in the literature and hinting at the nature of novice struggles, particularly where they relate to the functioning of System 1. The limitation of this approach is the inability to dive deeper than the existing data (and time). New sources of data can help theorists add to TAMP on such topics as how 'non-cognitive' factors

influence novice programmers, how programmers work and learn collaboratively, and, hopefully, many other aspects of computing education over time.

This chapter suggests both future research and possible changes to pedagogy, but only through theoretical speculation, and with the gaping hole that the model only covers one aspect of cognition. Including self-confidence, motivations, identities, emotions, and other aspects of cognition adds a richer picture to not just how programmers work, but how they learn. I was slightly crestfallen to admit that my revisit of Vygotsky's Zone of Proximal Development was out of scope for this incarnation of TAMP, as such improved tools could prove useful in the formation of future learning activities. Vygotsky's work might help to define how programmers learn from each other, adding to practices such as pair programming (Cockburn & Williams, 2001) or theories such as cognitive apprenticeship (Brown, Collins, & Duguid, 1989). This dissertation sets the stage, but far from completes my vision of TAMP. The core of TAMP at this point may be the ANM – and the underlying arguments for its formation – with a few propositions that still require further empirical testing. TAMP as a theory will be more effective with continued study of all aspects of cognition and the open-mindedness to add to, change, or remove the elements as proposed in this work.

9.2 Research implications

My original motivation to create TAMP was as a stopgap to fill holes in the research I hoped to conduct. I was taking a course in single-subject design methods, which generally avoids statistical analysis, preferring visibly apparent shifts in behavior/data as an indication of effectiveness (Kennedy, 2005). Single-subject researchers hope to establish causality by presenting irrefutable changes in behavior with and without some intervention. TAMP started as a search through literature to find appropriate measures, which did not seem to be available. The general measures of progress in much of computing education literature are either course grades (broad and imprecise) or student satisfaction (not a direct measure of learning at all!). Even when studies use data other than course results, the measures tend to rely on some level of success (e.g., the number of test cases passed, conceptual questions answered correctly, successful traces). I now believe the lack of specificity of the measurements led many researchers to conclusions of bimodal distributions, rather than a mysterious 'threshold concept' (Eckerdal et al., 2007; Meyer

& Land, 2003; Rountree et al., 2013; Sorva, 2013). I did not know it at the time, but what I was missing was a clear connection between the theoretical construct of "learning to program" and behavioral measures of programmers in action. While TAMP suggests a hierarchy of propositions, constructs, and concepts (primarily borrowed from existing theory), it just *suggests* them until future research offers confirmation, refinement, or refutation.

9.2.1 Lessons to learn from revisiting existing research paradigms under TAMP

9.2.1.1 Design thinking and writing code

It seems the easiest to start where we left off, the studies of design from Chapter 8. Revisiting the studies using TAMP as an alternative theoretical framework provided additional insights into the student outcomes and satisfying explanations for their behaviors.

- McCracken et al. (2001)
    - Students struggled because of a lack of design experience as much as gaps in programming knowledge.
    - The researchers assumed too much transfer of knowledge in students; particularly
        - Knowing how to use a calculator helps in designing its functionality
        - Learning about stacks simplifies the implementation of the unusual RPN
- McCartney et al. (2013)
    - The scaffolding guided the initial design supporting or replacing missing design experiences or transfer knowledge.
    - Researchers assumed novices would implicitly add features that were obvious to experts (e.g., validation) which only happens in rare cases

A common theme from these three reinterpretations is the role of intuition and the divisions in programming knowledge that computing education does not typically capture (e.g., notional machine).

TAMP suggests that experts develop a series of implicit memories (enactive) as well as traditional semantic knowledge about programming. Much of the thinking of experts, and the more successful student in the three studies, is driven by intuition refined by reason in iconic representations. Iconic representations not only help expert thinking but expert learning as well. When programmers form mature iconic representations that help them solve problems, they also

benefit from mature schemas to use Piaget's term. It is not that experts have better memories than novices, but that it is easier to recall facts when they are tightly associated with existing knowledge. The three studies above neglected to include any theory of learning to describe their methodology or findings. While their data is compelling, and their analysis is not entirely off-base, the lack of theory opened the door for misinterpretation and perhaps lowered expectations too far of what novices should learn in their first year.

Recommendation – When a study asks participants top write code, the scope of the coding task should align with the type of knowledge under testing. TAMP suggests that programmers have different repositories of knowledge (e.g., language, design, domain, testing) that may or may not transfer depending on how researchers present the problem. Too little scaffolding and some students will not know where to start. Too much scaffolding and the task may not exercise the types of skills under investigation. Establishing a cognitive model of the study's targeted skillset – possibly using TAMP's constructs – helps to establish relationships between areas of knowledge and skill, and hint at pedagogical interventions.

9.2.1.2 Tracing as a precursor

The 'all or nothing' outcomes seemingly reported by McCracken et al. led Lister et al. (2004) to develop an intermediate measure of programming skills. As described in Section 4.2.2.2, they created a multiple-choice test consisting mainly of tracing and a few fill-in-the-blank code questions. Their study thankfully demonstrated that efforts to teach programming were not in vain. Beyond answering questions about language constructs that might require memorization alone, they proved that most students could read and mentally execute code. However, many still struggled in design-type tasks, as simple as selecting the appropriate line from a choice of 4 options. Using dual process theory as a guide, I proposed that immature/faulty intuition was also to blame for the divide between mental execution and design (see also (T. Lowe, 2019)). My reinterpretation, which did not yet include Bruner's representations, already hinted at different repositories of knowledge that guide tracing and design. The measures Lister et al. used to describe even tracing seemed misleading.

Lister et al. presented a case study on tracing that seemed to imply that strong sketching was a sign of maturity in novices. My interpretation suggests that tracing is a sign of progress, but not maturity. System 1 simply cannot take advantage of sketching as it provides answers without conscious deliberation. When a novice sketches to aid in their mental execution, it is a likely sign they are relying on System 2 for some level of tracing. Sketching provides a tricky measure as

well because it is most likely iconic, specific to that individual. Unless a student has dedicated a great deal of time to learning a specific tracing protocol (symbolic representation), their traces are a hint to their mental process but may not consistently match others. Therefore, mastering tracing in a conventional sense (e.g., using trace tables) may only demonstrate symbolic learning, not the formation of iconic representations vital to problem solving. Tracing may have little value during a formal assessment (particularly formative), yet still can inform research and support student learning.

Researchers and educators might use tracing behaviors as an indicator of the maturity of *Enactive representations of code*. TAMP suggests the amount of sketching that novices exhibit might resemble a normal curve. At first, novices are less likely to sketch as they are not sure what to write down as they lack any notional machine. Sketching grows as students engage System 2 and start to build their own iconic representations of code in action. As their notional machine transitions into enactive representations, their need to sketch ebbs away. The amount of tracing seems to measure the maturity of their notional machine. The exemplar student from Lister et al. (Figure 5.2) showed a lower degree of maturity, as can be seen by the amount of redundant tracing within constructs. The redundant traces within triangles 5 and 6 show the student was not yet amalgamating the automation of individual constructs but needing to double-check their work at each intermediate step. An advanced novice might trace each iteration of a loop, for example, but not the intermediate steps. When novices 'naturally' advance to the use of a tracing table – as opposed to being drilled into using one – they may have reached the penultimate stage before fully automating most mental execution. The sketching patterns of novice might be a useful data point for researchers to measure intermediate learning and for educators to suggest additional practice.

Lister (2016) seems to have based much of his neo-Piagetian theory on tracing as a precursor to other programming skills. Lister, along with Teague (2014), described programmer maturity in terms of Piaget's developmental stages, yet TAMP suggests that a linear progression is too simple of a model. Lister and Teague's model sees programmers progress from struggling to competent tracers to writing code to expert reasoning in code (Section 2.1.3). While their description may be accurate at an abstract level, it neither is supported by some of the underlying research, nor does it provide theory to guide pedagogy. For example, their neo-Piagetian theory offers little explanation for the tenuous correlations between tracing, explaining, and writing code that Lopez et al. (2008) reported. Lopez et al. found some statistical significance countered with

a good number of outliers. Strong students indeed develop a network of skills, but none of these studies link what students do know (concepts/tracing/explaining) to what they are unable to do (design and write code).

Recommendation – Skills such as tracing are an integral part of a programmer's full skillset, particularly around *Enactive representations of code* within the notional machine. Measuring these skills is not merely about successfully predicting the outcomes of tracing code, but how that tracing occurs. Sketching provides a useful indication of maturity but TAMP suggests sketching will fade as System 1 matures. Researchers might find useful measures by qualitatively coding the presence, context, and quality of sketching during traces. Likewise, further research seems valuable in the relationships between explaining, tracing, and writing code using the concept of an *Iconic representation of design* as an explanatory factor. TAMP may offer insights that link the various types of knowledge used in programming skills.

9.2.2 Future research

The reinterpretations studies completed within this work offer strong evidence for the internal validity of the propositions offered by TAMP. While I was certainly aware of a few of the studies early in the process of building TAMP, some of the most compelling studies I did not discover or engage in until much later in the process. I encountered the two articles on the same study led by David Perkins (D. Perkins & Martin, 1985; D. N. Perkins et al., 1986) as well as McCracken et al. (2001) quite early in my preparations. I only found McCartney et al. (2013) and Fix, Wiedenbeck, and Scholtz (1993) when writing Chapter 8, reserving the discussions of these two studies as actual discriminant sampling[138]. The process defined in Chapter 3 remains valid, in so far as any few sets of data validate a theory in its infancy. The advantage of using existing data means that the methodology remains uninfluenced by the hypotheses. However, the disadvantage remains that much of the data is second-hand and can only validate the aspects of TAMP included in their data. The final stage of building TAMP will extend beyond this dissertation but can be proposed by considering future research that may come from the current definition of TAMP.

[138] If you do not recall the concept of discriminant sampling, it involves revisiting some portion of the data after constructing a theory. See Section 3.2.3 for more details.

9.2.2.1 Revisiting prior studies, with a twist of TAMP

One avenue of research to help in confirming TAMP's propositions is revisiting, some of the reinterpreted studies to collect missing data or otherwise confirm similar predictions. TAMP has the benefit not just of a new theory of mind, but also hindsight to see what data would have been helpful in further validating theoretical assertions. The next several pages highlight a few of the most influential studies within this work and tweaks to the methodology or analysis within each. I am not looking to offer an exact methodology, but merely advice or suggestions of what may be interesting twists on replication or similar research and their relationship to TAMP. By no means does the mean any study needs to select TAMP as a driving theoretical framework[139], but using the suggested tweaks might lead to data or findings that not only support TAMP but aid in explaining how experts and novices think and learn.

[139] Though it would be nice, and I would be glad to assist!

Fragile knowledge and Movers/Stoppers *– Section 5.2.1*
(D. Perkins & Martin, 1985; D. N. Perkins et al., 1986)

Methodology

The researchers created a highly effective protocol that provided detailed descriptions of novice struggles and successes. The prompt/hint/provide protocol was particularly effective at evoking inert knowledge (new triggers for System 1), and the commitment to one-on-one interactions between researcher and participant allows for maximum control and follow-up. If replicated today, it would be interesting to use screen-capture and video to garner even more information from the novice's programming process.

One advantage of replicating the study would be more control and understanding of the pedagogy and its relation to the assessment used in the study. The original articles provided few details on how students learned to program. Since TAMP suggests that examples are essential to building an *Experiential repository* of design ideas, the study should include coding assignments that are both familiar and those that require transfer. The two categories could help differentiate when and why novices become movers/stopper, and when they need further prompting or more involved hints.

Analysis

A new set of data allows for a robust exploration of the role of intuition in novice thinking. Performing a new analysis might allow for confirmation of TAMPS's theoretical constructs that separate coding language knowledge from design knowledge. By controlling both the pedagogy and assessment, the analysis can determine the importance of frequency in recollection. For instance, things that show up frequently and with variety in the pedagogy should require fewer prompts and hints than things that mentioned only once and particularly without any examples.

Adding video and screen capture of the student at work to the protocol would allow for comparative qualitative analysis across students and within successive work sessions. The scope of this study allows, within reason, a wide array of investigations since the researcher can solicit the participant to think aloud and compare their commentary to the actions they take while working.

Tracing as a pedagogy *– Section 5.2.2*
(Cunningham et al., 2017; Lister et al., 2004; Lopez et al., 2008; Xie et al., 2018)

Methodology

Tracing studies is one of the richest areas for reinterpretation data, but a few tweaks could reinforce the analysis already conducted.

The 'standard' methodology for tracing is asking participants to answer multiple-choice questions that require some level of tracing, and perhaps other questions of conceptual knowledge or completing missing code. Some improvements in more recent studies included capturing formal training on sketching techniques (which students tended to ignore) and even timing information on how quickly students complete each task.

A new study on tracing may seek to investigate the iconic versus symbolic relationship of sketching by comparing formal techniques of sketching and whether they impact design, not tracing abilities (e.g., are they the 'right kind' of iconic representations, or merely encouraging symbolic/procedural learning). Another variation might be to test tracing longitudinally (a few weeks at most should do) to see how sketching ebbs and flows and the notation changes over time. If sketching while tracing aligns with TAMP's model, sketching should mirror egocentric speech (Ginsburg & Opper, 1988; Piaget, 1995; Vygotsky, 1986) and internalize with growing expertise.

Analysis

As described earlier, a tracing study should consider sketching as a continuum, not a fixed target. Further analysis might confirm the proposition that certain types of questions are not answered by tracing, but instead by *Experiential repositories* of design.

One possible variation within the study could be to require or ban the use of notes/sketches during tracing on alternating problems. Encouraging or discouraging sketching, possibly including a think-aloud protocol to capture the participant's feelings and thoughts while sketching, might hint at the cognition at play (e.g., Is the sketch annoying?, Does it slow down the work? Is the student frustrated they cannot sketch?). By requiring sketching, the researchers can elicit a range of tracing outcomes – from rough, partial, incorrect scrawls, robust use of step-by-step tracing strategies, and minimalistic, impatient, and perfectly accurate notes.

Design and coding assessments – *Chapter* 8
(McCartney et al., 2013; McCracken, 2001)

Methodology

The point of a generalized assessment is to compare students across different pedagogical traditions to ensure they share a standard set of skills. The nuisance variable across the three studies from Chapter 8 seems to be an unclear picture if the different schools within the studies are even remotely equivalent. While it was clear that there is room to improve in each of the studies, it is not clear how much of the differences might transfer between school. While someday I would hope to attempt a study like this, I believe TAMP is still years away from suggesting a universal assessment. For others who may have such a pending study, I will offer a few insights for what I would do given the current state of TAMP.

Researchers should baseline the skillset across multiple tests, possibly including conceptual knowledge, tracing, explaining, and other such necessary skills. Creating such an inventory of skills across diverse populations helps measure both the impact of demographic differences across participants as well as confirming other research on the role (or lack therein) of other skills in designing and writing code.

As noted before, the researchers should establish a cognitive model of the expected skills a student should demonstrate, and which ones are intentionally scaffolded.

Analysis

One of the limitations of such extensive studies is the all-or-nothing nature of the data. It is difficult to track each student's progress and workflow, but perhaps the study could use automation to capture progress at different intervals (e.g., every 15 minutes) to see how students progress at different rates. Capturing timed snapshots would also lend to the analysis of how/when students change their minds or abandon work and come back to it, hinting at the role of intuition versus deliberation.

It also seems that large coding tasks might benefit from multiple data points of the same cohort to track how their thinking changes, how they respond to problems of various difficulty, or reduced levels of scaffolding. It seems as well that detailed analysis of each subgroup within the participating schools in comparison to the others, as well as the total population, may provide insights into localized phenomenon due to pedagogical choices, prior experience, or other potential nuisance variables.

The greatest challenge in such studies is telling a useful story with the numerous uncontrolled variables. The original McCracken et al. study seemed to indicate they were attempting to validate a shared assessment [140]. While large-scale, multi-national, multi-institutional studies are compelling; they also are resource-intensive and difficult to control.

[140] An impression seemingly shared by McCartney et al. (2013)

9.2.3 Studies to validate specific propositions within TAMP

Beyond new twists on existing studies, TAMP proposes a few new ideas out of theory alone. These ideas are less critical toward practical computing education research and practice but may have thought-provoking ramifications to models of thinking or education in general. I am purposefully only providing basic descriptions of these ideas since they are less practical for most readers and not a vital part of TAMP's validity overall.

9.2.3.1 The expanded definition of iconic representations

In formulating TAMP, I expanded quite a bit on Bruner's representations, none more so than the iconic. While I feel I stayed within the spirit of Bruner's description of the iconic space, I also combine concepts from neuroscience, prospection specifically, into the mechanics and purpose of iconic representations. Much as Houde et al. (2015; 2011) and Siegmund et al. (2014) used fMRI to consider how more complex tasks map to different regions of the brain, it would be interesting to see the role of the hippocampus and other brain structures in more complicated programming tasks like design. Research using fMRI has many restrictions and is expensive. Participants often must remain entirely still, and tasks must fit into short time windows. It is questionable if knowing the topology of the brain during design would even add to the model[141]. The neuroscience of programming may be years or decades off, and in the meantime, it seems less useful than other more practical studies.

Researchers may uncover the role of iconic representations in design might using more conventional means. For example, a study might separate participants into groups that will inevitably take the same design and coding assessment, but each group receives a different preliminary intervention. The control group, for instance, might receive a lecture on the use of stacks. A second group might receive a demonstration of using stacks to solve a similar problem. A third group might receive several examples of solving problems using various data structures over several days, occasionally repeating each. The assessment would come after a week of other forms of instruction to allow knowledge to move out of recent memory, but perhaps a fourth group receives some prompt just before the test. The analysis would compare each group's performance

[141] Perhaps discovering that the hippocampus is not very active during design could kill my hypothesis, but that seems unlikely as designers need to remember things after all.

on a task, like the RPN calculator, that requires stacks. Which groups think to use stacks? Which groups can remember enough to implement code successfully?

The nature of the iconic representation hints that students with more practice and time to construct enactive representations, and those with recent reminders to guide in place of System 1 should perform better than those who saw information one time or merely learned semantic knowledge. If possible, the study could track the time students spend on each phase of their project, much as Atman et al. (1999) did in tracking engineering design. TAMP would predict that students exposed to similar designs would jump to a promising design more quickly (e.g., they would choose stacks right away), and thus progress to working on the detailed design elements and coding. Educational researchers from other disciplines could also consider the role of iconic representations within their fields. While computing education demands a higher level of problem-solving than many fields (Socha & Walter, 2006), it does not have a monopoly on challenging problems.

9.3 Pedagogical Implications

While my studies have created a deep passion for research, my heart began and remained in the classroom, making educational research a wonderful crossover. Researchers should know the value of theory, but educators can find equal benefits when theory (supported by research) informs every aspect of their classroom practice. I believe it is not sufficient for instructors merely to adopt a theory-based curriculum. While a well-constructed curriculum is better than an anecdotal amalgamation of tradition, the most effective teaching needs to understand learning in every interaction. Particularly in post-secondary computing education, the best curriculum adapts to the identity of the school and degree program. As stated earlier, one of the reasons I am not proposing TAMP as a canonical theoretical framework or programming curriculum is because I cannot know the needs of every programming student. Even the mental model from Section 7.6.3.4 does not represent every possible type of programming expert, but merely the typical Software Engineer/Computer Scientist that best aligns with my experiences. In the age of specialization (e.g., Human-Computer Interface developers, System Administrators, Database Administrators, Data Engineers), technology-adjunct workers (e.g., Computing Educators, Business Analyst, Data Scientists) as well as non-computing professionals who need to

occasionally program (e.g., Engineers, Mathematicians, Scientists, Economists), it is up to each researcher or educator to transmogrify TAMP to fit their needs. What TAMP can do is expand upon existing and propose new pedagogy.

A Note on Granularity

The pedagogical descriptions within this work balance must between being specific enough to illustrate the theory and its applications, yet abstract enough to leave room for interpretation and study. Imagine a study using a TAMP-driven pedagogy that demonstrates overwhelming success; every student thrives, masters the assigned tasks, and becomes an independent learner. Despite seemingly incontrovertible proof, the next application might show lesser benefits or none at all. Perhaps the next students lack some critical prerequisites. Maybe they are less motivated. Any number of nuisance variables could derail the promising pedagogical intervention, but any analysis might miss the point: did the students need that information in that manner? Were they genuinely read to tackle material at the same level? Piaget, Vygotsky, and to a lesser degree, Bruner advocated for the role of an intelligent mentor to guide and customize learning. TAMP does not replace the need to understand and cater to the specific audience of students.

TAMP is a theory, not a pedagogy. TAMP intends to inform, not make choices. Prescribing content, modes, or sequences of instruction or assessment would transition TAMP from the realm of theory into a theoretical framework, if not curriculum. At times it is easier to "just tell me what to do!" Within programming, code is not always reusable, so we create patterns for solving problems or even design principles that guide the problem-solving approach. TAMP sets principles for designing programming education, which in turn may inspire patterns or specific exercises. These are offshoots of TAMP, however, as much as the spiral curriculum is an offshoot of Bruner's core theory. Ignoring the theoretical foundations and tenants that guide instructional design risks creating instructors that are as less capable of improvising in the classroom, just like the novice coders captured in the literature who are unable to design for new types of problems.

9.3.1 Choosing content and assessments

The most critical decisions made in education happen long before a student signs up for a course or even before a school offers a course – the selected content. Even when an instructor inherits a course with a catalog description that was outdated a decade ago, they still choose what topics deserve emphasis and which become footnotes. Wiggins and McTighe (2005) advocated for what they call 'backward design' – curriculum designers begin by identifying the desired results then choose assessment methods before finally considering pedagogy. Identifying the

desired results provides a razor to help prioritize what is critical for students rather than the convenient content and order found in traditional textbooks. Instructors should as,

> What should students know, understand, and be able to do? What is worthy of understanding? What enduring understandings are desired? (p. 17)

Wiggins and McTighe asked much more than what a course should cover, but what will students remember long after the end of a class. They referred to "enduring understandings," which they define as "big ideas, that have lasting value beyond the classroom" (p. 342). TAMP helps to define the epistemology of content choices within Wiggins and McTighe's questions.

Note: The rest of this section changes the phrase "enduring understanding" to *enduring outcomes*. Authors use the word 'understanding' in so many contexts that it can be confusing. Is understanding merely conceptual knowledge or applied? Do I really need to understand why programmers follow a specific coding style? Is adhering to style sufficient? *Enduring outcomes* encompass any type of educational goals across System 1 and 2 or all three of Bruner's representations. While the name is changing, the intent remains the same.

Instructors not only need to define what a student should learn but also their expected proficiency and use for their learning. Students may not remember every topic that a course covers, so an instructor must choose what things are the most important, and what will students need to do with that information? Wiggins and McTighe hint at an epistemological divide between *knowing that* and *knowing how* that helps distinguish between "understand" and "able to do." A programmer should remember the rules of the language and write code, but Wiggins and McTighe expect more from students than a shallow remembering. It is one thing to ask a student to 'be able to program' by the end of the term, but will they retain their newfound skills over the summer? A year later? Well into their professional career? Beyond developing long-lasting knowledge, Wiggins and McTighe expect also expected enduring outcomes to transfer to new situations. A programmer's skills mean little if they cannot code solutions for new types of problems than what they saw in the classroom. Wiggins and McTighe placed high expectations on instructors and students: learning is not enduring unless students can recall knowledge years later and apply it to solve never-before-seen problems.

TAMP suggests that the goal of creating enduring outcomes is much trickier than it seems. Test results may indicate what a student knows today, but that knowledge risks fading over time,

depending on the type of memories formed. Perhaps educators have accepted forgetting too quickly? While forgetting unused details is inevitable, but perhaps redirecting effort towards building System 1 expertise, which is less ephemeral, can provide students with truly enduring outcomes? The procedural expertise of System 1 may last longer than crammed semantic memories within System 2, but System 1 is only as flexible as its training. Wiggins and McTighe explicitly warn against summative testing as a measure of enduring outcomes. They allow for a generous breadth of assessments over time as evidence of student learning, not merely a final project or exam. Ensuring students achieve enduring outcomes does not mean that instructors must write better tests or only use 'authentic' assessments. Each form of assessment adds to a patchwork of feedback, but adding multiple assessments creates a new problem: modeling how various data points contribute to the desired outcomes.

Theories such as TAMP help educators select meaningful assessments and amalgamate their results. It is difficult to predict from a single assessment, just how knowledge will endure. Vygotsky defined the Zone of Proximal Development to capture the difference between what a student demonstrates unaided and what they can do with support. Measuring enduring outcomes seems to be the reverse process. A student likely has the most support (the freshest memories) during the test before they fade if left unused for the rest of the class and beyond. Not only must an instructor find valid tests in the 'here and now' of the course, to promote enduring outcomes, but they must also consider if the learner is building knowledge in the 'right way' to become enduring.

TAMP does not solve the puzzle of making learning enduring, but it helps to understand how a curriculum promotes or risks falling short of the 'right type' of learning. The theoretical constructs of TAMP suggest what enduring knowledge looks like in experts. Experts need enactive representations to capture long-term *knowing how* and iconic representations to make such knowledge flexible. Understanding the memory structures involved in creating lasting learning may help hint at assessments to test the types of knowledge that last. For instance, nondeclarative memories (enactive/System 1) not only fade slower than declarative but also provide faster results. Cunningham et al. (2017) noted that students who sketched while tracing were generally more accurate but slower than their peers. A traditional evaluation might conclude that these students have mastered tracing, but only when students are accurate *and quick* have they

formed skills that are also enduring. Educators can add elements to existing tests, as well as create new assessments if they understand the characteristics, not just the content of enduring knowledge.

The backward design approach presented by Wiggins and McTighe helps instructors to prioritize learning and ensure students are moving towards the desired skills. Defining the enduring outcomes of a course acts as a compass for assessment and pedagogy. TAMP exposes the complexity in attempting to build enduring knowledge – both lasting and flexible. System 2 learns quickly but readily forgets when left unused. System 1 matures through deliberate practice that tight schedules do not easily accommodate. However, when enough enactive representations form, a student can then build the mature iconic representations that are essential to problem-solving. Students cannot acquire most enduring outcomes in a few weeks over a handful of assignments, so educators must create curricula that revisit critical content and vary how it is encountered and used.

Computing education seems to have already begun to shift towards mastering less in the same amount of time. For example Utting et al. (2013) comprised of many of the same original researchers yet the expectations of what students could accomplish are minimalistic compared to McCracken et al. (2001). The shift is not merely generational as the two teams shared five common members, including the two lead authors. While assessment is an important aspect of education, it is not the only means of achieving enduring outcomes, merely measuring them. It may be reasonable to encourage the habits of mind that eventually might yield enduring knowledge by carefully selecting and ordering activities within pedagogy. Chapter 2 presented a variety of creative pedagogical interventions that present information in new ways and activities that test targeted skills (discussed further in Section 2.3). Before considering specific methods of teaching or testing skills, I will offer a few propositions for choosing content and assessments based on the TAMP and its underlying theories.

9.3.1.1 Lowe's Laws of Enduring Outcomes (LLEO)

Every class likely forms some enduring outcome for its subject matter; it may not be the outcome the instructor planned or desired. In all humility, I submit Lowe's Law of Enduring Outcomes: *what sticks with students is what they practice most (even if that was not what you meant for them to learn)*. Students will always form a lasting impression of a topic since System

1 implicitly learns from any repeated activity. They may acquire the instructor's desired *knowing how* or just as easily form a misconception or distaste for the materials. When we encounter routine tasks, our mind prefers the path of least resistance. As Lopez et al. (2008) observed, good tracers do not always become good explainers or writers and vice versa. The goal of tracing is predicting the output and TAMP shows that novices lack the mental support to determine its purpose simultaneously. Students are not any lazier than the rest of humanity, as it is in our nature to minimize the precious resource that is System 2. If instructors want students to develop flexible (i.e., transferrable) knowledge, the curriculum must encourage both repetition and variation. If students need to improve design skills, instructors must create activities that focus on design (e.g., subgoal labeling). When instruction is not explicit about its goals and desired outcomes, students find their own meaning. As an example, McCracken et al.'s (2001) students believed coding is all about learning the language since that is what they spent most of their time struggling. Instructors should be intentional in defining what expertise looks like and share that knowledge with students.

Instructors have several ways of satisfying LLEO. The most obvious is carefully plan instruction to focus on the desired activities. If you want strong tracers, have them trace a lot. If you want well-rounded programmers, they will need to engage in a variety of activities regularly. Students may also benefit when given the opportunity to be metacognitive about their learning rather than blindly working through activities. A fictionalized example of decontextualized skill development appears in the movie The Karate Kid ("The Karate Kid," 1984). In the story, Daniel asks Mr. Miyagi to teach Karate. Mr. Miyagi agrees, then tasking Daniel with several chores, including waxing cars, sanding floors, and painting fences offering occasional hints on how to perform each task properly. Eventually, Daniel rebels against the seeming busywork demanding to know when he will start to learn Karate. Mr. Miyagi asks Daniel to demonstrate the motions learned during each repetitive tasks and attacks Daniel with various punches and kicks, each of which Daniel deflects using the motions learned while at work. Mr. Miyagi set Daniel with a fruitful pedagogy for developing implicit skills, literal muscle memories, required for Karate. Daniel chaffed because he could not see the connection between the pedagogy and his goals. Had Mr. Miyagi made the simple connection between waxing and Karate, not only Daniel's attitude but also the quality of his practice may have improved.

Instructors should consider LLEO when designing courses. What will students spend most of their time doing, and will that contribute to the desired enduring outcomes? As an electrical

engineering major, I spent significantly more time in computer science courses building user interfaces than I did the content for that class. When students spend most of their time struggling with syntax, they naturally infer the important content must be mastering the language. The rest of this section discusses new and alternative pedagogical interventions that help to move beyond the mastery of syntax and semantics by providing scaffolding to tackle other aspects of programming. Scaffolding not only helps students to practice skills they might otherwise be prepared to perform, but it also helps to focus their attention on how their integrated skillset will function. Offering students the full picture of skills, and how each activity contributes to the desired skillset should also instill more flexible knowledge and skills. Instructors can use TAMP to fill in gaps where they are not entirely sure what activities provide, only that they are useful (and perhaps reasons to abandon some traditions).

9.3.1.2 Flood and Wade Curriculum

To help instructors promote flexible problem-solvers, TAMP suggests students must first automate precursor skills and tie skills to key concepts (i.e., build enactive then iconic representations). Training System 1 and then expanding it to support flexible thinking takes much more time than introductory courses usually set aside in the schedule. Often courses attempt to cover one major idea each week on a cattle-drive through the language constructs (e.g., week 1- variables and operators, week 2- decisions, week-3 loops). Slowing down the pace of teaching risks not covering all the syllabus and losing the advanced students to boredom. The ordering of the content feels important. A student needs to understand one construct before having a hope of understanding the next, right? How can a programmer understand loops without a firm grasp on decisions, or operators, or data types? TAMP questions the traditional assumptions of the logical progression of acquiring programming concepts.

The traditional schedule of teaching programming not only disenfranchises many struggling students but spoils the opportunity to create enduring skills. People do not become literate by practicing nouns before adjectives before verbs before adverbs; in fact, they tend to read long before they study grammatical structuring. Reading assignments begin using a limited vocabulary in stories and typically include concrete ideas that are familiar to the reader. They do not create segmented sentences that are devoid of any story. The story not only makes reading

compelling but, as Vygotsky (1962, 1978) described, expands our reasoning beyond merely a new form of communication[142]. I believe that programmers, like young readers, can acquire programming languages more holistically. I perhaps am a mirror image of what Lister (2016) proposed (see Section 2.3.2); rather than advancing phonics, I suggest immersion.

Immersing novices in a programming curriculum means putting them in the middle of big challenges and letting them come to the knowledge they need to solve them. The traditional approaches often attempt to push a solution long before students understand the problem, which is effective for *knowing how*, but perhaps not for promoting flexible problem-solving. The goal of immersion is to introduce the big picture of programming and acquire the supporting skills over time[143]. TAMP suggests that contextualizing knowledge within problems promotes enactive representations. Otherwise random fact, therefore, more likely tied to experience through iconic representations that rely on enactive representations. Traditional programming education epitomizes the 'symbolic-first' approach Bruner described (Section 6.3.3.1), where novices learn primarily through symbolic means. The first exposure to many programming constructs is through lectures and carefully constructed examples with little resemblance to real-world problems. Novice programmers seem to suffer the same consequence Bruner described, an inability to use programming knowledge to solve problems (Chapter 8).

The curricular pattern I propose to realize immersion is *flood and wade*. Rather than presenting an orderly curriculum tied to specific timelines, the flood and wade approach includes concepts that novices are not prepared to understand fully but will come to in time. A flood and wade course would still utilize traditional lectures and activities (including 'phonics' training), but these materials would be secondary rather than driving learning. For example, instead of starting with a "Hello World" problem, a flood and wade course might start with debugging a complex system with a familiar and relatively simple behavior. The first example code may contain loops, decisions, and calculations, but scaffolded in such a way that students can focus on the problem at hand rather than understanding the nuance of the code. Think of it as the opening of a movie. The director does not need to explicitly introduce each character and plot device if they carefully guide the viewer's attention. The viewer picks up on key ideas based on context and adds important

[142] While I would love to expand on the analogous and perhaps in some cases literal connections between learning to program and second-language acquisition, that sadly must defer to a future publication.
[143] One approach for building an immersion curriculum revolves around the pedagogical innovation introduced in Section 9.3.3.2 called Debugging-first.

details to their growing understanding of the story. A flood and wade curriculum floods students with complex, interrelated ideas before allowing them to wade through the details in a controlled manner.

I could, and hopefully will someday write at length about the nuance and details of a curriculum based on flood and wade. For this work, the goal is not to introduce unproven suggestions but to offer examples of how theory can support radical ideas about learning. The flood and wade approach seems risky by conventional wisdom. What if students succumb to the complexity and *flood and drown* instead? A glib answer might be "they already do," but a simple look at how Bruner models learning, as described above, supports at least giving flood and wade an experimentally controlled trial. One objection to flood and wade from instructors might be the cost of spending more time on basic concepts. Instructors may need to scale the number of objectives within a course to allow time and practice. Based on TAMP, I would predict that students who build strong foundations in programming may acquire additional knowledge much quicker. Since flood and wade associates experience and knowledge, students who form mature iconic representations should also become more effective learners. It may be that course can cover the same amount of materials as before, but with the 'advanced' materials compressed into the waning weeks since students once struggling students gain the advantages of their more experienced peers.

9.3.1.3 Save the least for last

I have one last suggestion to propose before leaving content and assessment. Any content introduced in the final quarter of the course's schedule should not introduce any of the enduring outcomes. It is unlikely a few weeks or even a month is sufficient time to make learning enduring. A month may be enough time to automate some behaviors, but probably not enough to also create transferrable skills. The last quarter of a class should probably remain reserved for encouraging transfer. Early programmers in particular, need time to develop enactive representations, see a variety of examples, and learn the ways of tools and processes. The final weeks of introductory programming classes may best serve students by revisiting the core concepts only applied to complex, open-ended problem solving, such as McCracken et al. (2001) initially expected from their participants. It may be reasonable to add new knowledge upon an earlier enduring knowledge

(iconic representations aid learning after all). However, new content also requires time and repetition to become enduring and thus should not be any of the primary objectives of the course.

9.3.2 Revisiting pedagogical innovations in computing education

Chapter 2 introduced several ideas from computing education literature, most grounded in some theoretical tradition or empirical study. The analysis of teaching methods in this section does not seek to set some activities above others or dissuade the use of any technique. There remains a time and place for nearly all things. Section 2.2.2.2 noted that despite the eventual banishment of `goto` statements from most programming languages, the construct once had vocal supporters. If even the pariah that is the `goto` statement can invoke such passion, there is probably a time and place for most teaching practices. TAMP provides tools for discovering the advantages and limitations of any teaching methods. Table 9.1 covers traditional computing education pedagogies and their associated pros and cons along with the types of Bruner's representations they might promote.

Educators have a wide variety of well-documented pedagogical interventions from which to choose. Table 9.1 provides some guidance of when these interventions are helpful and at which point, they may not add to the type of knowledge students need. Students likely need a mix of existing activities, and perhaps new interventions like those introduced in the next section to build the various types of knowledge described in Section 7.6.3.4.

Table 9.1 Pros and Cons of common computing education pedagogies

Intervention	Pros	Cons
Conceptual Knowledge Tests *(symbolic)*	Usually given as multiple-choice, fill-in-the-blank, short-answer, or other formats that allow auto-grading, such tests put little demand on instructors while providing instant feedback to students on their semantic understanding of programming language concepts	They generally test symbolic knowledge, which has few benefits in many early programming tasks. They may misrepresent progress by 1.) giving false confidence to people who only *know that* but cannot apply knowledge or 2.) punishing people who *know how* but did not study 'arcane' details
Parson Problems *(iconic)*	A very achievable exercise that requires little syntactic mastery and focuses on the structure and flow of code. A friendly introduction to early concepts that might highlight design patterns and groups code in 'explainable chunks'	They require examples that facilitate 'chunking' and not obvious to reassemble. There is no guarantee the novice *will* attend to patterns or purpose if they can solve the problem procedurally (e.g., syntax clues). They seem to provide a valuable early tool but for a very narrow window of time
Worked Examples *(enactive/ symbolic)*	When well-constructed, they provide a type of MKO that the novice can control and revisit as they will. Inexperienced novices can build early skills by mimicking the desired skills using the example as a helpful guide to inform their mistakes without the need of a live expert	They can devolve into a passive spectator event that engenders familiarity but not genuine skills. They also require a specific focus for learners at a specific stage. Once the learner moves beyond that stage (internalizes the content into System 1), they will find little value from watching worked examples[144].
Subgoal Labeling *(enactive/ symbolic)*	These shift the focus from procedural coding issues to investigate questions of design and patterns. They seem effective at encouraging the explanation of code but may or may not result in maturing the programmer's *Experiential repository*.	It seems unlikely novices can attend to both procedural (learning to code) and subgoals (reflection on design) in the same worked example on the same viewing. Subgoals provide a reason for later viewings, but students may benefit from separate videos.

Continued on next page

[144] This relates to the expert reversal effect discussed in Cognitive Load Theory (Plass et al., 2010)

Table 9.1 – continued from previous page

Intervention	Pros	Cons
Tracing *(enactive)*	Tracing provides a direct method of measuring and maturing the notional machine. Repeated tracing ensures automation and may add to the *Experiential repository* after most language constructs become automated and attention can turn to the design of algorithms	Tracing, as typically presented, is inherently unnatural and inauthentic. Experts do not trace without the aid of a computer very often. Hence, tracing does little to build familiarity with code during actual execution and the symbolic representations of code output.
Explain in plain English *(enactive/ iconic)*	Explaining returns the focus of code to metacognitive reflection on the purpose of code constructs and the design for achieving specific goals. They likely promote some level of iconic representations and, when carefully planned, can introduce patterns	It is not clear how novices first come to explain code. Experts do so implicitly, but novice often must mentally execute code to find meaning. Do novices need to be proficient tracers and imagine useful inputs (i.e., good testers) before learning to explain?
Writing code–scaffolded (*enactive*)	Scaffolded coding activities limit the decisions novices must make and provide focus on the specific content. Novices learn to deal with syntax, errors, and algorithms in small doses while engaging in an otherwise authentic activity.	Narrow-focused coding does not always translate to reflection on design or yield experimentation. Too often, the activity comes down to 'getting the answer' by the best available procedural technique. Success may not scale up to more complex coding activities.
Writing code–open-ended *(enactive/ iconic)*	Writing opened-ended code demonstrates the full skillset of a programmer, going from concept to implementation. Prepared programmers engage in challenging and fulfilling tasks scaled to specific complexity and accessible domains.	Very few novices seem ready to engage in problems of any complexity, even towards the end of their first class(es) in programming. Coding is particularly complex when requiring the transfer of ideas rather than a variant on a familiar problem.
Pair Programming *(enactive/ iconic/ symbolic)*	Working with a partner seems to increase satisfaction and retention, as well as perhaps success across the pair. When instructors insist on a formal rotation schedule, it also can ensure each partner spends time coding and reflecting on the shared project.	The learning outcomes beyond the assigned projects while paired are debatable. It is unclear if novices can reasonably act as an MKO to other novices and focus on appropriate learning over completing the work.

9.3.3 New pedagogical interventions inspired by TAMP

The expert model of cognition from Figure 7.15 presents the pedagogical challenge of how to help students develop implicit knowledge explicitly. I am not implying that everything a person learns requires explicit pedagogy, merely that identifying when and how a novice acquires the implicit skills that expert demonstrate helps to plan curricula. Instead of frustrating students with tasks that require knowledge they are unlikely to possess[145], instructors can prepare educational scaffolding. For instance, novices must learn to reverse engineer and work within existing systems of code, yet most early programming activities begin with little or no code. Many computing educators create projects working within open source systems to provide authentic projects from the professional world that also forces students to practice how they read, research, and plan work using existing code. This section lays out a few informal pedagogical approaches that promote specific types of knowledge that TAMP identifies in experts that pedagogy often overlooks.

9.3.3.1 Back to Basics

Educators want to treat their students with respect, but sometimes assuming competency can leave otherwise talented people at a disadvantage. The year I turned forty, I joined an Ultimate (frisbee) Grand-Master's club team, with forty being the minimum age. We had a solid group of 20 to 30 players, some of us who had played for decades. Our early practices were occasionally frustrating and unproductive as it took longer to go through basic drills and in correcting fundamental plays due to miscommunications. At the same time, I was coaching a high-school team working with first-time players of the niche sport, so I volunteered to demonstrate some basic drills. Returning to basics shattered the misconception that because we were all grown men, we were not all experienced players at an organized level. Many played for fun, but never formally practiced in a team setting. By returning to basic, we established a baseline of vocabulary and concepts that made the rest of our training efforts drastically more productive. Our early practices floundered from the assumption that everyone 'spoke the same language' and 'knew the main ideas'. Teaching basics felt disrespectful, yet once we committed to doing so, we realized that even experienced players did not always share the same perspective. Establishing basic ideas and

[145] Various examples might include how to use the tools, read and respond to errors, test a system, or debug code. All tasks that are essential for even basic applications but often receive little explicit instruction.

practicing them until routine helped remove some of the tension, promote inclusion, and secured a birth in the National Championships, but that is another story.

Section 7.6.3.4 identified several types of intuitive knowledge that do not seem to map directly to traditional pedagogy, but perhaps should. Sometimes novices need resources, though probably not lectures, to acquire vocabulary and concepts that experts take for granted. For instance, TAMP suggests that experience inspires design, high and low, more than logic, but where do students acquire these patterns? Even new coders often follow examples in writing new code, but they may not recognize the significance of what they are mimicking. They may not even connect the formal language describing their actions to the result. When helping a student, I might say something like "create another variable to store the total sales", expecting them to produce code as follows.

```
double totalSales = 0;
```

Some struggling novices cannot follow through on my seemingly simple request. For these students, I need to guide them through each word, if not character, to make the connections needed to repair their code. These students bring partially working code, so it seems unusual that a student could produce dozens of lines of code without recognizing the significance of each line. Vygotsky noted the same within literacy; people use words without a firm grasp of their meaning. TAMP suggests that the mimicking an example (as helpful as it may be in promoting System 1) may not translate into a novice understanding of the significance of the actions they take.

Instructors need some activity to link coding tasks where students follow a procedure with that procedure's description and purpose. As rudimentary as it may sound, some novices may benefit from a worksheet full of commands like "build a variable that…" or "define a loop that…" perhaps requiring only a single line of code as an answer. Simple activities like these provide variation and repetition in a very low-stakes activity. Students can see several examples of a construct in use and link that pattern of coding with a specific vocabulary. While many students will pick up such knowledge without guidance, many more will benefit from a conscious effort to acquire the new lexicon and achieve small victories. The same approach scales-up to basic algorithmic patterns, then complex design patterns.

Other aspects of programming also benefit from explicit attention. Error messages, for example, are often ignored by formal pedagogy, but learning to recognize and to react to errors

helps students to respond like experts. The same goes for tool usage, debugging strategies, programing style, sketching (while tracing), flowcharts/pseudocode, or any number of pragmatic items that experts take for granted. Instructors might choose a different approach for each topic, as a lecture on error messages would be arduous to create, much less attend. Instructors can instead weave such topics into other pedagogies (e.g., intentionally make mistakes during worked examples) or simply ensure you always practice what you would otherwise preach. If you want students to use meaningful variable names, then ***never*** show examples that use badly named variables like `a`, `b`, or `val`. If students must add comments[146], then every delivered code should contain examples and demonstrations must include comments. Students emulate what they see more than what they are told. Most importantly, it is risky ever to assume anything about programming is intuitive. The ***Unified*** Modeling Language (UML) is named unified because three programmers came up with different representations for the same ideas and agreed to merge their notations into a single system. If experts do not find each other's notations intuitive, why would an untrained novice?

This section reminds us that programming professionals have expert blind spots and how such blind spots might inadvertently impact learners. Some concepts from TAMP are quite easy to teach yet just as easily overlooked. Others are obviously important, but that does not make them easy to teach. TAMP serves as a reminder that much of what experts do is subconscious but no less vital for being intuitive. Eckerdal and Berglund (2005) noted that students believe that experts think differently than they do. Experts do seem to change the meanings of words, pull vital information from seeming gibberish, work at blinding speeds, and ask questions that are, at first, incomprehensible. Experts may not think differently, but without guidance, it may be difficult for novices to connect their experience with the new and strange task of programming. Educators can show respect to novices by sharing ideas that may seem basic, yet establish a bridge of understanding and a path to success.

[146] And as a side note, I am not sure why this is such a pressing issue. Comments are helpful but rarely essential when code is well written. Meaningful naming strategies, proper use of data types, and a preference for straight-forward algorithms makes comments redundant in most cases. A better lesson is to add comments to the unusual. Comments should be the exception when the source code simply cannot reflect the behavior/intent. Commenting rationale should occur because of extraordinary circumstances, not relatively obvious reasons. When everything is commented, it is difficult to pick out the important comments to attend to. (I also want to point out the irony of talking about comments in a footnote, which is essentially the 'comment' section of a paper)

9.3.3.2 Debugging-first

While many novices seem to learn basic coding skills, the highest hurdle they face in becoming a programmer seems to be problem-solving. One consistent theme from computing education literature is that many new programmers are learning concepts and even producing code in controlled circumstances but are unable to translate their learning into building creative solutions. TAMP suggests the gap between experts and novices might revolve around their ability to form mature iconic representations, which in turn seems to rely on the robustness of their Experiential *repository*. Phrasing that another way, System 2 functions better when System 1 primes relevant memories that support prospection. Unfortunately, few early programming pedagogies are successful in promoting the types of experiences or learning for many learners.

Instructors may not expect novices to design from scratch early in their education, but other programming activities also require the formation of the same iconic representations, particularly debugging. Successful debugging, at least for certain types of problems, demands the programmer develop a strong sense of the design. In practicing debugging, a novice gains experience from existing code that they will need in writing new code later, specifically their *Experiential repository*, but also a sense of the types of problems that arise when coding and strategies to find and repair errors. Perkins et al. (1986) noted that many novices jump from one idea to another or stop when they encounter an error. TAMP suggests these students derail because their System 1 is unable to inspire a fruitful next step. As Perkins et al. demonstrated, sometimes, a novice can recover with a little redirection, but only if they can connect the error message with the semantics of the language construct. Debugging-first looks to tackle debugging by building experience through preplanned errors directly. Just like traditional worked examples seek to teach the process of writing code, Debugging-first utilizes the same pedagogy only focused on debugging. Through a series of Debugging-first examples and activities, novices will acquire experience both in handling errors at the language level and identifying problems in the design.

When an error falls outside of their experience, experts are better than novices at 'brute force' methods of analysis as well. The same experience that makes them faster at mentally executing code will make them better at parsing errors and finding bugs using symbolic outputs (e.g., debuggers, log files). Where a novice must attend to the format and search for the content within a program's output, most experts will effortlessly add that knowledge to their iconic *Mental execution* representation (from Figure 7.15), at least when the output is familiar. Experience shows

experts better strategies for debugging, most of which are so tacit that we have yet to see a definitive textbook on debugging. It may be that the best (only?) way to get better at debugging is by doing it.

The Debugging-first" pedagogy provides an additional tool to those already in use by computing educators. It is neither first-and-only nor even first-instead-of other teaching methods, but it does represent a shift in mindset. Lister stated his preference for a bottom-up 'phonics approach' (Section 2.3.2), where Debugging-first proposes a top-down approach that is more expansive and directive than the reduced-syntax approaches of block-based languages. Students do not merely need scaffolding to defer (or hide) complex ideas and intricate symbols (syntax). Such approaches may help System 2 but do little to train System 1. Debugging-first pedagogy looks expose novices to the 'real' complexities of coding while bridging specific gaps between existing pedagogy. Debugging-first promotes the exposure to habits of mind (e.g., design, testing, debugging) that instructors seem to reserve for very late in the curriculum. Since Debugging-first is a top-down approach, instructors should buttress with traditional bottom-up pedagogies. Debugging-first is not a super-pedagogy, all things for all people, but I believe it offers a scaffolded way to engage in the types of thinking that often are difficult to promote and measure.

Gaps in pedagogy that Debugging-first addresses

Literature's 'big three' triumvirate of skills that novices first learn include tracing, explaining, and writing code. Tracing and explaining represent two distinct aspects of reading code, one for execution (*knowing how*) the other for understanding (*knowing that*). Writing code requires both types of knowledge, but as Lopez et al. (2008) reported, writing code seems to require something more than just these two. TAMP suggests that novices who write code must combine the two types of knowledge (using iconic representations) but also acquire other types of information within the *Experiential repository*. While the traditional pedagogies for tracing and explaining enhance certain types of knowing, they cannot provide the types of experiences to support certain types of problem-solving.

Tracing code provides an accessible teaching method for promoting the essential enactive representations within the notional machine. Students can still succeed in tracing code with a tenuous grasp on the syntax (i.e., enough to read but not enough to write). Tracing provides code

and inputs, reducing the load on System 2 to merely focusing on producing the output. Tracing does not require the programmer to understand the purpose of code, though, hence the need to assign students explaining tasks. Tracing reinforces the connection between code and its execution, but little else at first. Figure 9.2 shows the relationships created by the 'big three' skills. Writing tasks demand the translation of a problem statement into code, where tracing tasks provide code seeking its execution results. Explaining tasks work differently for experts and novices. Experts explain the purpose of an algorithm from the code directly (the dotted line), where novices tend to consider the execution results (see Section 7.5.2).

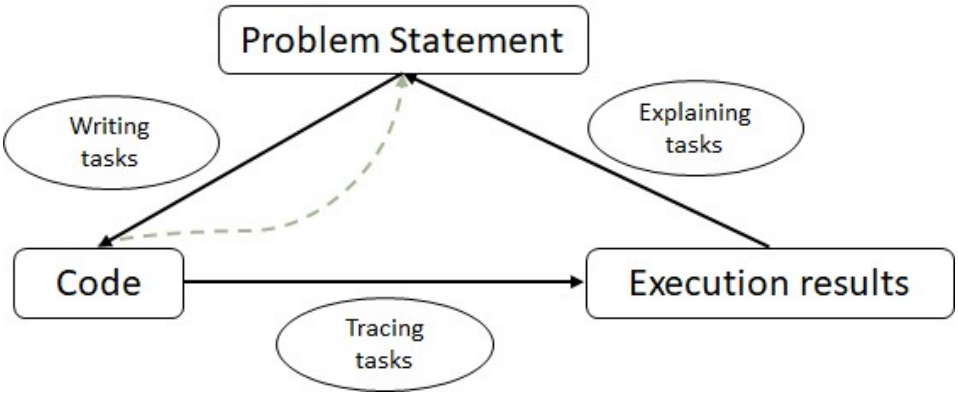

Figure 9.2 The pedagogical relationships between the 'big three' skills, tracing, explaining and writing

At a curricular level, Figure 9.2 looks perfect. Each skill feeds into and supports the next. By teaching tracing, the novice has the skills to find the execution results that support explaining code – which in time experts automate. Descartes or Piaget might suggest that if a person can explain the purpose of code, then they can turn that knowledge around to write code given the purpose. TAMP suggests otherwise since such a leap requires quite a bit of transfer, confounded by the disconnected feedback students receive from these three tasks.

The feedback that students receive from each of the 'big three' tasks has flaws in either the timeliness or robustness of feedback. For instance, if an instructor takes the time to automate tracing exercises, tracing tasks can provide instant feedback (e.g., did the student's trace arrive at the correct result) but with no robustness. It is helpful to the student to know if they were correct or not immediately, but even seeing the correct result of the trace does not necessarily pinpoint the flawed step during their mental execution of the code. The student still must investigate their own flawed thinking to find their mistake(s).

Consider what a novice must do to find their error. At best, the learner used some form of sketching where they can review their work, tracing step-by-step through the original trace and the second attempt with System 2 looking for discrepancies. If the second trace produces a different result on a step, they found the correct answer, but did they discover their problem? If the mistake was made by System 2, then perhaps they can 'correct' their flawed semantic memory. If System 1 was in error, then the 'fix' will take time and practice. It seems novices must debug their thinking about programming before they have learned to debug at all! Most students only really care about 'right' and 'wrong' answers and are unlikely to consider the epistemology of their mistake. More likely, novices will restart tracing until they arrive at the right answer, the learning equivalent of tinkering. Tracing is very useful in promoting the notional machine, but only when the practice is accurate, and it tends to focus only on enactive representations of code.

The feedback provided to students during tasks to explain code is either delayed and detailed or immediate and abstract. Like tracing, the instructor could automate the feedback where the student received an exemplar explanation when they submit their answer, but then the student must decide the quality of their explanation in comparison. It seems unlikely a student who provided a vague description would learn much by analyzing an example alone. Perhaps they will recognize deficiencies in their explanations, yet just as likely, they will not recognize nuances. MKOs (e.g., peers, instructors, T.A.s) provide better feedback, but this likely happens asynchronously. The MKO can only guess where the novice went wrong and thus focuses on flaws in the provided explanation. Even if the MKO knows enough to determine the mental error (something TAMP would help with), the feedback assumes the student will remember the code (much less the result of their mental execution). It seems that the best explaining exercises are 'real-time' rather than graded, giving the richest feedback in the timeliest manner.

Writing code offers the most disconnected feedback of the three. The compiler and runtime system provides immediate feedback (mostly error messages) when writing code, but many novices struggle to connect such feedback with their actions. Compilers cannot point to design flaws either. The quick modes of feedback seem to focus on a limited set of a programmer's skills and still neglects to integrate concepts. Novices must connect the symptom (error message) with the cause (e.g., a mistake in coding, bad design, bad test, misunderstood problem). Nothing from tracing or explaining connects error messages with solutions to such problems. Until a novice gets past basic errors, they have no feedback on if their design and code solve the problem.

Determining if the solution is appropriate is the realm of testing, which none of the 'big three' explicitly covers. Novices will test until their best impression of the problem statement is satisfied but may later find out that their interpretation was poor. Providing novices with automated tests seems to provide a great deal of scaffolding, but it limits the types of problems that novices will tackle. Again, the novice must wait for some of the most important feedback (is the design appropriate for the problems statement) until after an MKO can look at their finished work. By the time they get this feedback, their memories of their working time are fading from memory.

Even if a programmer has mastered tracing, explaining, and writing code when scaffolded, they may still have gaps in testing. Tracing tasks typically provide inputs, so the novice never needs to consider test data. Explaining tasks tend to focus on smaller algorithms with easy to abstract inputs (e.g., a list of numbers). Most importantly, these tasks start with the assumption that the code is flawless. Any input is as good as the next, as finding faults is not part of the expectations. When should a novice programmer start to understand and what pedagogies teach the principles of testing?

The final gap, in not only the 'big three' activities but possibly all typical programming pedagogies, is debugging. It may be that some students are better debuggers as they are naturally stronger students who naturally think reflectively. For those that are less reflective, where do they learn to debug? To put it in TAMP's terms:

- Tracing builds *Enactive representations of code*.
- Explaining tasks link code with its purpose, another enactive representation in the *Experiential repository*[147].
- Explaining code may provide habits of mind that help to break down existing code, more than forming memories that later inspire new designs.

Debugging requires careful comparisons of results and several mental representations, yet our mind sometimes works against us (e.g., the confirmation bias). What experts 'learn' about debugging is practical and intuitive strategies for challenging their expectations. For example, how do I use the debugger to see the successive values of variables? What area of the design

[147] These enactive representations are not shown in Figure 7.15. As shown in Figure 9.2, experts read code which inspires explanations, the equal and opposite of *Inspire design* that 'sees' problem statements and inspires design. These two types of enactive representation may build in parallel, but our mind probably does not create a design pattern for solving problems based on a single explanation formed by reading code.

should I focus on analyzing? What parts can I ignore? So many of these decisions apply general strategies to the details of the problem, so intuition (like in design) seems to play a major role.

When it comes time for novices to write code, the experience they have developed while tracing and explaining help, but not enough for too many. Even if the students have a burgeoning *Experiential repository*, they may not have the habits of mind to realize their design in code. Even the best programmers make small errors that they find and fix when debugging. More than anything, debugging skills determine the success of a programmer[148]. Debugging is also the ultimate expression of programming prospection. Debugging requires a programmer to compare an *Iconic representation of design* with the execution results, the code, the test case, and perhaps even the problem statement. Debugging often means moving past our own confirmation bias. We expect our design solves the problem as intended. We expect the code we wrote to match our intended mental model of design. We expect the program to behave as our design predicted. Peeling apart the actual execution results from the anticipated plan requires mental discipline with the aid of debugging habits of mind.

This section has identified several critical gaps in learning beyond the focus of common programming pedagogies. Tracing supports the development of the notional machine, particularly the enactive portion, and explaining builds certain levels of design knowledge, but even experts seem to seek a greater purpose in such activities (Fix et al., 1993). Neither seems to demonstrate the connection between a problem statement and the resulting design. Most importantly, neither tracing or explaining demands that students practice building or manipulating *Iconic representations of design*. Students need practice in identifying flaws in their mental models and forming habits of mind that help them while debugging. The lack of explicit instruction on testing also seems to limit how well some students perform when writing code.

The greatest advantage of Debugging-first is ensuring that novices experience the integrated mix of skills required to build software. Debugging-first does not replace the need to focus on specific skills, but it offers an integrative programming pedagogy. While athletic coaches run drills that focus on specific fundamentals and band directors split the full ensemble into instrumental groups to focus on specific sections of a song, at some point in each practice session, they reassemble the group to work together. Programming educators have devised many clever

[148] Remember Lister (2016) placed debugging at the formal operational level of thinking, the top of his model.

and effective pedagogical interventions, but few that focus on more than the breadth of integrated programming skills. No single skill seems to be the missing link to producing successful programmers since programmers not only need a variety of skills, but they need a mix of enactive, iconic, and symbolic representations to support them. TAMP suggests that there are some aspects of design work that are not covered by the existing curriculum, and possibly novices only acquire through experience. Debugging-first looks to fill the gap, at least partially, in existing pedagogy to provide students with experiences that focus on skills and knowledge not already covered by other activities.

The precepts of Debugging-first

Debugging-first places novice programmers into the toughest part of the programming process yet provides scaffolding to support their participation. How can novices who know nearly nothing about programming start with one of the most difficult processes? They cannot, but they can go through a similar, scaffolded experience. Drivers education instructors often use a scaffolded driving experience to provide an early driving experience. My high school had a classroom equipped with about twenty simulators, including steering wheels and pedals. Before we jumped into a real car, we 'drove' the simulator[149]. Long before I took this class, my uncle put me behind the wheel in a large empty parking lot. Each of these experiences provided a different form of scaffolding. In the simulator, I could focus on the rules of the road and detect hazards. The artificial vehicle removed the risk of a bad decision turning damaging or lethal. My uncle's chaperoned drive around the parking lot provided familiarity with the vehicle and its controls.

While the driver's education class also included memorizations of vehicular laws and other facts, the truly authentic (but still scaffolded) experiences included the vital perceptual cues that a new driver needs. Students need to use the pedals and steering wheel to acquire the muscle-memory and reduce anxiety. Students equally need to train their senses to watch for signs, listen for horns, feel the road, and respond accordingly. Driving instructors would be negligent if they did not put this all together to ensure the student applies the breaks promptly when seeing a red octagon and then comes to a complete stop before the crosswalk. While System 2 will (literally)

[149] The simulator experience used a reel-to-reel projector and included electronics to monitor our actions. Imagine the potential students have today with electronics!

drive some of these early actions, developing a mature set of driving skills in System 1 produces the safest drivers. While few tasks in programming are as time sensitive as driving, it seems many novices experience similar feelings of anxiety that engendering familiarly with programming may help manage. I think driver's education is an interesting example since putting sixteen-year-olds behind the wheel of a very large, expensive, and potentially dangerous vehicles occurs rather quickly in the driver's education curriculum and seems to be the only truly effective way of learning highly complex, yet intuitive activities.

Largely intellectual endeavors seem to forget the role of implicit learning when symbolic knowledge is readily available. Perhaps this is the bias of experts, who readily learn from books due to their robust System 1[150]. Ironically, it seems that many children learn to read quite implicitly from an early age.

> The single most important activity for building the knowledge required for eventual success in reading is reading aloud to children. This is especially so during the preschool years. The benefits are greatest when the child *is an active participant*, engaging in discussions about stories, learning to identify letters and words, and talking about the meanings of words. (R. C. Anderson & et al., 1985, p. 23 emphasis added)

Parents generally read to kids without first lecturing them on the syntax and semantic of the English language. Somehow the mere act of being read to, even when the child may not be processing the words on the page, helps their later learning. System 1 provides a perfect model for this type of learning. Preschool children are not reading in any meaningful way, but they still can be active participants in forming the habits of mind that make a good reader. Debugging-first seeks likewise to make novices active participants in the mental activities involved in debugging.

Testing actually comes first

Were an instructor to ask novices to ***fix*** defects as an initial activity, it would be ridiculous, but Debugging-first offers a surprisingly accessible starting point for learning about programming. Think of the standard "Hello World" activity, whose only goal is to write a line of text to the screen. In the days before every student held a computer in their pocket, making the screen change in even this small way was thrilling. Even then, "Hello World" demonstrated the bare minimum of

[150] For their area of expertise of course

creating a working program yet does little to portray the significance of programming. What exactly can a student do, having now presented static text on a screen? The procedural task of managing the tools and compiler is important and tricky, but far from compelling outside those rare individuals already compelled to write code. The minefield of potential errors within the basic "Hello World" program can become insurmountable for new programmers since even a miscapitalized letter or misplaced punctuation risks failure. Until students understand that programming is rife with small failures, even small setbacks may shatter a fragile "programmer identity". Debugging-first tackles failure head-on by showing that even experts make mistakes. Debugging-first is a bit of a misnomer since the new programmer's first contact is actually with testing!

Even the most technophobic student can achieve early success through testing. Using programs is so ubiquitous in modern society that starting with testing may seem dull but starting with testing offers several benefits. First, the novice must attend to the details of the problem space before considering code. Testing establishes the "orientation" (Du Boulay, 1986) by portraying the types of problems that programming can solve. Through carefully written instructions or even better a video, an instructor can guide their students through the initial "happy path" testing[151] to demonstrate the program. Testing provides a guided tour of a program's features to establish familiarity before introducing intricate and obscure features/test cases. Before the novice sees a line of code, they have a sense of what the code does.

Introducing testing to students not only sets up the upcoming coding activities but demonstrates testing strategies and patterns. For example, Section 8.3 noted that McCartney et al.'s (2013) students generally forgot to validate user inputs. Debugging-first supports building such habits of mind by demonstrating the need to test inputs early and often. When pedagogy includes input validation is part of the ritual from day one, instructors can spend less time ensuring novices *know that* validation is important since they *know how* to consider invalid inputs as part of their habit. Students who appreciate the need to check for good and bad inputs and do so by habit will likely be less confused and more willing to learn how to write validation logic. A lecture on validation seems decontextualized and abstract by comparison. Validation logic is too difficult

[151] Happy path testing typically refers to the basic functionality when the user provides valid inputs and induces no special cases of the logic. Happy path scenarios offer the core benefits of a software system.

for the newest students and does not get at the essence of debugging, but instructors can seed Debugging-first exercises with purposeful defects.

Before turning to examples of Debugging-first activities, it is important to capture the intrinsically motivating nature of these challenges. Tracing, for all its value in maturing the notional machine, is artificial workflow. I recently helped proctor a test that asked students to trace several problems. At the start of the testing period, I quickly 'took the test' to make sure the problems were fresh in my mind if I were called upon to answer any student questions. I was instantly annoyed by the concentration required for each tracing question[152]. In my decades of coding, the only times I ever trace code is when the output disagrees with my expectations, and I am at a loss to figure out why. ***The only time I trace is when debugging***. It is little wonder that novices eschew sketching or why stronger programmers perform poorly on some tracing questions – they are arduous and prone to mistakes when System 2 does not put forth the required effort[153]. Testing sidesteps the 'academic' nature of tracing tasks and instead presents an authentic puzzle task: fixing bugs.

Debugging tasks

Once a student develops an intuition for the subject domain through testing a program, the instructor can induce a purposefully selected defect to start the foray into debugging. Unlike tracing, debugging task tells the students the desired answer and asks them to find the problem, most likely by tracing. The intuition for how the program should work, plus the familiarity with inputs and expected outputs, provides a different type of scaffolding to tracing tasks. Traditional tracing tasks have no safeguards when System 1 leaps to incorrect (and sometimes confident) solutions. Even as an 'expert', I did not realize my mistakes on the proctored test until the next day. Debugging-first provides the same support as tracing, the same immediate feedback without the need for automation, but encourages the formation of an *Iconic representation of design* since an answer to the trace is not the primary goal.

[152] It was difficult to concentrate in a room with 1800+ people using pencil to capture their answers atop little trays on their lap. And remember, I have quite a strong notional machine with way better automation than most students. It was all the more annoying since it took longer to work out the answers in my head than it would have to type the code into a computer, which is how I would have confirmed my answer as an instructor or professional.
[153] I selected the wrong answer on at least one if not two of the test questions!

Debugging-first places the construction of mental models at the forefront over other skills. It is not only possible but much easier for novices to trace by routine. After all, the computer does not look ahead to determine the purpose of code, so why should the novices when their goal is simply to finish the tracing task at hand? If the question asks for the results of the trace, the best path is to focus their entire mind on accurate mental execution. If your English teacher asks you to read a Shakespearian sonnet aloud, do you focus on the structure and meaning of the words or merely on recitation? It will not matter if you understand the deeper meaning if you stumble through the reading and look the fool.

Adding debugging to the task makes the novice use tracing as a way of gaining meaning, much like an explaining task, yet more demanding. Explaining code, while perhaps more authentic than tracing, requires little precision. Getting the general gist of a program from its output does not mean the novice focuses on nuanced aspects of language constructs or algorithms. Debugging-first combines the precision of tracing with the larger picture of explaining. The novice must reconcile the test case with the resulting execution and identify where the code goes astray from its intended purpose. The best part, the answer is built into the question, and the question naturally draws focus to a specific misconception or highlight language features. Rather than guess at the 'correct result' or 'actual purpose' of the code, Debugging-first provides both. The student still needs to investigate their behavior, but the instructor provided examples of this procedure as part of introductory worked examples on debugging!

Novices engage the greatest breadth of mental representations when debugging, so Debugging-first requires the most scaffolding. Figure 9.3 presents the major mental representations that a programmer might need when debugging a problem. The test case defines the context for errors and demonstrates how code becomes erroneous when it violates the problem statement. As just described, debugging reinforces the formation of mental models of the design, at least at the algorithmic level, if not the program's architecture. It is at this point that debugging and tracing meet, working through the code using the programmer's notional machine. Where tracing

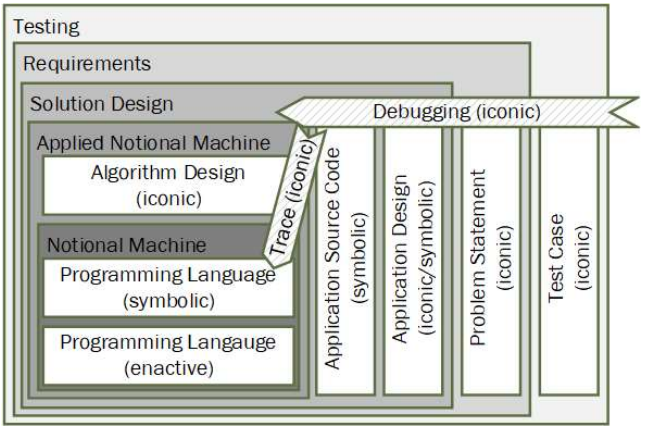

Figure 9.3 The mental representations of Debugging-first, from (A. A. Lowe, 2019)

may or may not encourage conscious consideration of design, debugging demands it. Debugging-first only works when problems are scale to the ability of the learner, as discussed in the next section.

Instructors can use Debugging-first activities as a more effective pedagogy for mitigating misconceptions. While it likely takes more work to build systems rather than individual algorithms, seeding misconceptions within authentic problems provides twofold benefits. First, rather than playing into misconceptions to trick and thus frustrate students by preying on common 'wrong answers', debugging provides experiences that help to reshape System 1. Piaget might say that Debugging-first creates disequilibration (and I would say in a friendlier manner). Where a tracing example that includes a misconception is a trick played by the instructor, a program that exposes a misconception as a defect is a problem to solve. The novice engages System 2 to diagnose the misconception (at least in the program, possibly in their own thinking), and then the repair effort serves to redirect System 1 with new experiences. As noted above, until the test case passes, they know their job is incomplete.

I proposed a Debugging-first example that preys on a misconception involving the order of operations (A. A. Lowe, 2019). Students sometimes forget that code executes in sequence, and the order of operations is important. The misconception appears when they assume that when a program assigns a variable to another variable, rather than sharing values at that moment, the two variables remained synchronized forever. The example in Figure 9.4 exposes this misconception by incorrectly ordering the swapping of two variables. The code should swap the owners of a vehicle but, as written, makes one party the owner of both vehicles. The novice would have a good understanding of the program by completing test cases creating vehicles and owners. By establishing a clear and present issue in the swapping behavior, the Debugging-first task scaffolds the Test Case, Problem Statement, and Application Design from Figure 9.3. For the newest programmers, the Debugging-first task could even narrow the focus down to a few lines of code, scaffolding the need to understand most of the coding constructs. The example in Figure 9.4 can focus on variables, assignments, and the order of operations[154]. Like parents or teachers support

[154] Note that I am liberally mixing traditional assignments (=) with object-oriented getters and setters. This is intentional as I believe the concepts are conceptually interchangeable. Since the equal operator has a preexisting meaning in math, it may be that setters are less likely to invoke this specific bug. I am merely speculating here, and empirical research is needed to confirm my hunch.

preschoolers when reading, Debugging-first places novices within the 'story' of debugging and focuses their attention on the lesson at hand.

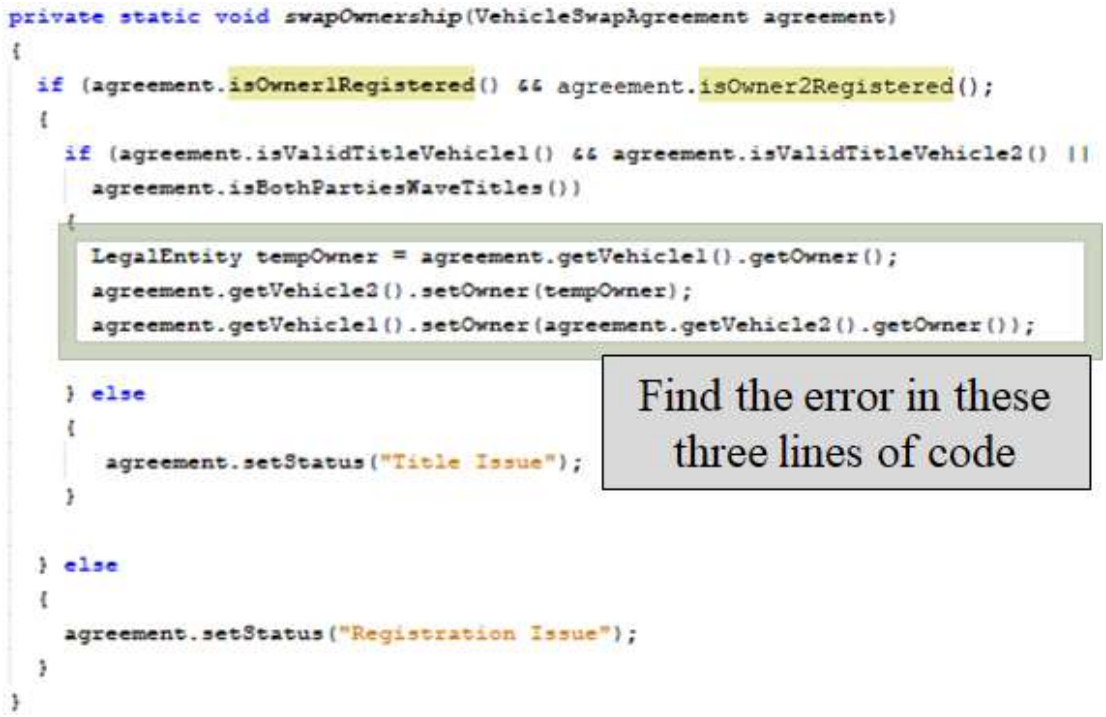

Figure 9.4 An example of a Debugging-first defect modified from (A. A. Lowe, 2019)

Scaffolding students

Like many other early educational activities, the key to Debugging-first is proper scaffolding. The sample code from Figure 9.4 includes more constructs and complicated logic than the traditional examples a novice would encounter until several weeks into their education. The example includes a method, decision logic and nested decisions, Classes and objects, and then hidden within that is the incorrect use of the temporary variable. It feels that giving such an example to students too early would just confuse them more than helping them, right? I believe that theory and experiences from language acquisition literature say otherwise.

Stephen Krashen has worked extensively on language acquisition, including publishing multiple editions of a book on how to teach second language acquisition. While I hope to eventually write in much greater depth about how his work might inform programming pedagogy, for now, one idea stands out.

> Language acquisition does not require extensive use of conscious grammatical rules, and does not require tedious drill. It does not occur overnight, however. Real language acquisition develops slowly, and speaking skills emerge significantly later than listening skills, even when conditions are perfect. The best methods are therefore those that supply "comprehensible input" in low anxiety situations, containing messages that students really want to hear. These methods do not force early production in the second language, but allow students to produce when they are "ready", recognizing that improvement comes from supplying communicative and comprehensible input, and not from forcing and correcting production. (Krashen, 2009, p. 11)

Krashen suggested that instructors teaching a second language should not force students to speak or write in that language for an extended period, perhaps months. Learners need time to absorb and be comfortable with the new language. He suggests that when instructors push too quickly, they risk damaging the student's confidence ('feeling of knowing'), causing them to regress. Most importantly, he seems to describe the implicit acquisition of a language over time yet eschews drills over naturalistic communications. Krashen never talks about dual process theory or Bruner, but I believe that his ideas are supported well by TAMP, though I will not take the time to elaborate further here. His research supports his suggestions within second language acquisition, but much of what he describes mirrors students learning a programming language. How can computing educators apply his advice?

Through the lens of traditional pedagogies and models of cognition, Krashen's advice seems unachievable, but Debugging-first attempts to emulate certain aspects. A foreign language instructor can speak with students, show them videos, or have them read texts in their new language, and allow students to respond in their native tongue. Programming instructors could try the same approach, but programming demands much more than comprehension, and it is unlikely that students will have months to become comfortable with reading code. On the one hand, people start a second language using general conversation, where programmers must learn the language to solve problems. On the other hand, programming languages have a dramatically small vocabulary and vastly simpler grammar than spoken languages. A programmer may not need months if the tasks they are asked to perform build confidence and leverage appropriate scaffolding. Debugging-first offers a pedagogy where students engage deeply with a programming language, yet have little to produce to do so, at least at first.

The first week of Debugging-first may simply revolve around tiny errors that students need merely to identify. Simply identifying the line of code in error is a fairly low-cost attempt and

engages the novice with reading code and considering its meaning. Even for the most grade-conscious student, coming up with the 'right answer' is secondary to the process. Guessing which of the handful of lines is in error is so easily accomplished by trial and error that it seems silly that a student would obsess at finding the correct answer over engaging with the process of understanding the code (as too often happens in other activities).

The earliest activities should focus on building habits of mind and *Enactive representations of code*. Worked examples can guide students through using the debugger and practicing tracing. As Vygotsky suggested, mimicry not only becomes a pedagogical tool but an early warning sign since students who are not putting in the work will not be able even to mimic later steps. Experts should also guide novices through reading log files, dissecting lines of code (e.g., this is the data type; this is the variable name) and connecting such ideas to traditional lectures (which are still important for forming symbolic representations). Once the Debugging-first exercises instill the concepts of tracing, explaining, and habits of mind for strategically accomplishing such tasks, the instructor can trust traditional pedagogies can provide additional practice but the students will have a foundation for integrating new information.

The values of Debugging-first

Debugging-first is an alternative means of promoting the same skills and concepts from other aspects of computing pedagogy. The main difference is perspective; Debugging-first presents concepts as they relate to larger problems, rather than presenting a programming language in an 'orderly' series of progressively more complex ideas. The traditional bottom-up approach first provides the facts about programming (e.g., language syntax and semantics, tools) and builds towards putting those ideas to use. It works for many students, but far fewer than we would hope. Worse, the literature indicates that many students never integrate their fundamental skills to become independent problem-solvers even by the completion of their first programming class(es). TAMP suggests that the issue is not one of better pedagogy, but a different kind. Debugging-first looks to address gaps that the bottom-up pedagogy cannot hope to cover. The problem with a bottom-up approach, like in driving a car, is the need to build intuitive responses to open-ended environments. Intuition is too important, needs all-encompassing experiences, and takes too long in forming to wait until the facts are all securely memorized.

Instructors can introduce authentic experiences but need to do so in a manner that is achievable and promotes the right type of skills. One important goal of Debugging-first is to promote confidence, not to provide another confusing activity. To help guide educators as they attempt Debugging-first, the critical precepts of include:

- *Ensure that students have self-regulating feedback* – Debugging-first starts with testing to provide students a sense of 'right' and 'wrong' about the problem space and instill habits for creating good tests. Debugging ensures that students come to know when they have a good answer, rather than guessing and seeking external validation.
- *Create engaging activities* – Students will persevere longer when the required mental effort directly benefits the outcome. Debugging provides authentic and intrinsically motivating tasks that no one can claim is for learning alone and engages the desired mental representations.
- *Promote habits of mind* – Experts rely on intuition that includes not just domain knowledge but habits that inspire the right action at the right time. Instructors should demonstrate expert habits and put students in situations where they mimic such behaviors to develop the 'different way of thinking' that programmers allegedly exhibit.

Practical considerations of Debugging-first

Debugging-first neither replaces nor diminishes the need for other types of pedagogy. While it may provide a different perspective and different types of knowledge, the approach may have a few downsides. My first concern is the initial burden on instructors to build authentic and robust programs that fulfill all the Debugging-first precepts. In the early stages of programming and perhaps specific advanced topics (e.g., parallel processing), the end-to-end conceptual approach fills gaps in a novice's experience. Once the novice forms the desired habits of mind, becomes familiar with the programming tools, and learns to interpret symbolic forms of output, the need for the robust and contextualized systems may diminish. Programmers still need frequent and targeted practice to build a powerful System 1, but existing pedagogies can supplement the heavy-weight pedagogy of Debugging-first to do so. Focused examples are still important for managing the instructor and student workload, but Debugging-first adds a strategy for integrating

their skillsets and covering some of the gaps that TAMP suggests exists between experts and novices.

Instructors should also weigh if "first" is the best plan. If I were to start a course today, I would introduce Debugging-first activities as a replacement for "Hello World". As an advocate of flipped classrooms, I would also hedge my bet with at-home videos covering more traditional descriptions of programing concepts. I have plenty of faith in the implicit acquisition of knowledge, but literature and theory also suggest that a little help from System 2 improves the speed of System 1 learning. Early introduction of Debugging-first activities also has 'non-cognitive' benefits. Debugging-first embodies Krashen's (2009) advice to "supply 'comprehensible input' in low anxiety situations, containing messages that students really want to hear" (p. 11). If a single pedagogy can instill confidence and enhance learning, it seems a good place to start.

Finally, Debugging-first as an experientially driven pedagogy may compensate for some of the gaps TAMP has yet to uncover. The model in Section 7.6.3.4 likely does not identify a perfect model of expert cognition, merely capturing anecdotal and empirical observations on the differences between experts and novices. While having a perfect model of expert knowledge may be interesting, a reliable system for teaching programming is preferable for instructors and novices alike. Debugging-first places students in situations where they begin to emulate expert-like thinking without already possessing all the skills of an expert and perhaps acquire unidentified types of knowledge. It is no more incumbent that we understand how novices become experts than it is that we understand how machine learning algorithms solve problems. Since expertise, like machine learning, seems to happen within a black box, we may only ever guess at the exact knowledge and the heuristics involved in creative pursuits like design. TAMP's model of expert cognition is useful for describing traditional pedagogies but fostering experiences with timely and meaningful feedback that teach the same lessons seems equally acceptable.

The most practical statement I can make about Debugging-first: *at this point, it is an interesting pedagogy that needs empirical study*. Given the evidence of TAMP and the two years I have spent considering Debugging-first, I am confident enough to test it with students, but not enough to suggest others jump to wholesale adoption. One of the first studies I hope to conduct is observing the impact Debugging-first has on individuals under a single-subject controlled methodology. Large group studies at this stage risk losing the nuance of the pedagogy amongst the statistical averages of the general population. What Debugging-first (and honestly, many of

the computing pedagogies) need is an impact analysis based on individual case studies similar to those by Perkin et al. (1985; 1986). Looking at how individuals struggle or thrive within the different ways of knowing may provide insights into how students receive each teaching method.

Summarizing Debugging-first

Debugging-first has lingered in the back of my mind through most of the creation of TAMP. More than twenty years ago, I attempted to create a lesson to teach debugging. The best I could achieve was a milquetoast list of heuristics and tips that were difficult to explain, so I am sure were impossible to internalize. Having completed TAMP, I am leaning towards the idea that there are just some things that students can only learn by doing. Debugging-first may be revolutionary, a passing fad, or never make it past this page, but its success is not the point[155]. Debugging-first demonstrates how theory can inform innovative pedagogy and can help us reconsider existing teaching practices. TAMP provides guidance not just for creating teaching activities but also in what to look for and how to measure student responses to ensure the 'right types' of learning.

9.4 A final bit of advice from Jerome Bruner

The greatest advantage of creating a theory about thinking and learning is the opportunity to read extensively about the practice of teaching. Like so many professions, teachers are effective in many ways for many reasons. Teaching is much like engineering in that educators have an open-ended problem (make people smarter) with limited resources (primarily time and attention), and you produce better results when using science as a guide. This work has presented many scientific works from a variety of fields that I hope helps to guide your practice. Engineering and teaching may be better when grounded in science, but the results stem from individuals creatively applying their experiences. It is up to you to make of students what you will. In closing, I wanted to share with you some of the inspirational ideas from my favorite educational theorist, Jerome Bruner.

Bruner reminds us that students learn best when education is stimulating and fun. Instructors need to properly prepare students to take full advantage of playfulness and exploration or risk alienating students to their subject.

[155] But I do believe it has potential!

> Curiosity, it has been persuasively argued, is a response to uncertainty and ambiguity. A cut-and-dried routine task provides little exploration; one that is too uncertain may arouse confusion and anxiety, with the effect of reducing exploration. (Bruner, 1966c, p. 43)

According to Bruner, the classroom should be a mix of the routine and the unusual. Instructors walk a fine line between enticing and disenfranchising students in the activities they present. Programming requires a bit of the "cut-and-dried routine" yet also has amazing potential to inspire. The instructor's most critical role may be choosing content and assessments that build not only skills but also confidence and curiosity.

> Yet it seems likely that effective intuitive thinking is fostered by the development of self-confidence and courage in the student. (Bruner, 1976a, p. 65)

Once again, it seems that Bruner spotted the value of intuition in learning and reinforces the connection between cognitive and 'non-cognitive' factors. TAMP suggested that intuition (System 1) links to confidence ('feeling of knowing'), and here Bruner proposes the reverse is true as well. Confidence fosters intuition when System 2 trusts System 1, and intuition fosters confidence.

Students may not become confident when they rely too much on their instructor for feedback and validation. Bruner mentioned multiple times in various works the need for the instructor to step back and encourage students to become self-reliant.

> 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. (Bruner, 1966c, p. 53)

Instructors generally do not intend to make students dependent on external sources of feedback, but the nature of many programming activities provides no other alternative. So long as we provide feedback based on 'correct answers' and guard those answers preciously, we send the wrong message about programming. Programming is not about arriving at specific answers – there rarely exists a single perfect answer – it is about the quality of the process used to derive an answer. Instructors should focus on opportunities for growth via rich feedback in early programming courses, rather than protecting against violations of academic honesty. TAMP does not provide easy answers but offers alternative tools to measure if students are learning the desired lessons

(and retaining those lessons in the right type of memory). Accepting Bruner's challenge to make self-reliant students means rethinking teaching from the ground up.

Bruner stressed the importance of individualization combined with agency in learning. Piaget and Vygotsky each promoted the value of a tutor, or more knowledgeable other, but Bruner suggested that learners need to regulate their learning.

> Earlier we asserted, rather off-handedly, that no single ideal sequence exists for any group of children. The conclusion to be drawn from that assertion is not that it is impossible to put together a curriculum that would satisfy a group of children or a cross-section of children. Rather it is that if a curriculum is to be effective in the classroom it must contain different ways of activating children, different ways of presenting sequences, different opportunities for some children to "skip" parts while others work their way through, different ways of putting things. A curriculum, in short, must contain many tracks leading to the same general goal. (Bruner, 1966c, p. 71)

Bruner suggested that instruction should "must contain different ways of activating", which I like to think of as *educational degeneracy*, in the same sense as to how DNA enables traits in living creatures. The variety of life on our planet relies on the degeneracy of DNA. Our DNA contains redundancies that place traits across multiple genes, so they appear even when part of our genetic code is damaged. Bruner suggested that students also need multiple channels from which to acquire ideas. The notion of *educational degeneracy* suggests that giving learners many ways to encounter a topic will strengthen not only their chance of understanding, but the transferability of that knowledge given the variety of learning opportunities. Educators and researchers should breathe a sigh of relief that we do not need to forge the perfect curriculum but merely create diverse "tracks" that gives students multiple examples, modes, and opportunities to learn. Instructors who use degenerate pedagogies provide options to a diverse body of students as well as allow their desires and creativity to evolve just like living beings. The hardest task educators face is not uncovering any specific secrets or thresholds within a particular discipline but teaching students to think and learn.

Bruner taught many subjects across many institutions across a staggeringly long career as a professor – psychology, education, and even the law – but wrote about the most important value he gave to students.

> I have often thought that I would do more for my students by teaching them to write and think in English than teaching them my own subject. It is not so much

> that I value discourse to others that is right and clear and graceful – be it spoken or written – as that practice in such discourse is the only way of assuring that one says things right and courteously and powerfully *to oneself*. For it is extraordinarily difficult to state foolishness clearly without exposing it for what it is – whether you recognize it yourself or have the favor done you. So let me explore, then, what is involved in the relation between language and thinking, or , better, between writing and thinking. Or perhaps it would be better to speak of how the use of language affects the use of mind. (Bruner, 1966c, p. 102)

Bruner promoted clear thinking and problem-solving by teaching students to express themselves clearly. The same clear thinking seems a vital skill for building in programmers. More than anything else, a programmer's main responsibility is clearly expressing rules and behaviors to an unfeeling, unknowing, and uncaring automaton who will do great or terrible deeds as commanded. To solve complex problems, programmers must be effective learners. Knowing about programming is only the first step; a programmer must also learn the problem space well enough to teach a computer to automate the rules of the domain. Every programmer essentially becomes an itinerant teacher, moving from one problem to the next, creating literally codified lessons for an obedient yet mindless pupil. Programming a computer requires so much more than coding knowledge that it should probably be shocking that so many students succeed in such a short period of instruction. Like Bruner, I believe the most important lesson a programmer can learn is how to learn and validate their learning.

TAMP proposes a model of thinking and learning that likely applies to many disciplines and possibly brains in general but focuses on longstanding questions from computing education literature. This dissertation may have opened more questions than it answered by exposing gaps in traditional views of thinking and learning. TAMP started as a way of answering a simple question "how do I measure if struggling programmers are learning anything?" Pulling at that thread unraveled the epistemological question, "what does it mean to learn programming?" The pursuit of the epistemology of programming spans hundreds of pages and culminates in a theory rather than a simple answer. If you still want 'just' an answer, I challenge you with Bruner's words from this chapter and Paula Silver's from Chapter 3 about the nature of theory. My goal is not to tell you how to research or teach, but to give you the experiences that help you discover what you think the best way to create programming experiences that shape your students.

REFERENCES

5 Personality traits every new programmer should have. (2014). Retrieved November 19, 2018, from https://learntocodewith.me/posts/5-computer-programmer-personality-traits/

Abramovitch, R., Freedman, J. L., & Pliner, P. (1991). Children and money: getting an allowance, credit versus cash, and knowledge of pricing. *Journal of Economic Psychology*, *12*(1), 27–45. https://doi.org/10.1016/0167-4870(91)90042-R

Actis, B. (2017). Italian grandmother learning to use Google home. Retrieved from https://www.youtube.com/watch?v=e2R0NSKtVA0

Adams, R., Turns, J., & Atman, C. (2003). Educating effective engineering designers: The role of reflective practice. *Design Studies*, *24*(3), 275–294. https://doi.org/10.1016/S0142-694X(02)00056-X

Ahadi, A., Lister, R., & Teague, D. (2014). Falling behind early and staying behind when learning to program. *PPIG 2014 - 25th Workshop of the Psychology of Programming Interest Group*, 77–88.

Alyahya, R. S. W., Halai, A. D., Conroy, P., & Lambon Ralph, M. A. (2018). Noun and verb processing in aphasia: Behavioural profiles and neural correlates. *NeuroImage: Clinical*, *18*(January), 215–230. https://doi.org/10.1016/j.nicl.2018.01.023

American Society for Microbiology. (2011, August). Garlic doesn't just repel vampires. Retrieved March 5, 2020, from https://www.sciencedaily.com/releases/2011/08/110815172343.htm

Amit, E., Hoeflin, C., Hamzah, N., & Fedorenko, E. (2017). An asymmetrical relationship between verbal and visual thinking: converging evidence from behavior and fMRI. *Neuroimage*, *152*, 619–627. https://doi.org/10.1016/j.neuroimage.2017.03.029.An

Anderson, M. (2016). Things native English speakers know, but don't know we know. Retrieved January 31, 2020, from https://twitter.com/MattAndersonNYT/status/772002757222002688/photo/1

Anderson, R. C., & et al. (1985). *Becoming a nation of readers: The report on the commission on reading*. Urbana, IL.

Anfara Jr, V. A., & Mertz, N. T. (2014). *Theoretical frameworks in qualitative research*. Sage publications.

AP Score Distributions – AP Students | College Board. (n.d.). Retrieved June 17, 2020, from https://apstudents.collegeboard.org/about-ap-scores/score-distributions

Ariew, R. (1992). Descartes and the tree of knowledge. *Synthese*, *92*(1), 101–116. Retrieved from http://www.jstor.org.ezproxy.lib.purdue.edu/stable/20117041

Atman, C. J., Adams, R. S., Cardella, M. E., Turns, J., Mosborg, S., & Saleem, J. (2007). Engineering design processes: A comparison of students and expert practitioners. *Journal of Engineering Education*, *96*(4), 359–379. https://doi.org/10.1002/j.2168-9830.2007.tb00945.x

Atman, C. J., Chimka, J. R., Bursic, K. M., & Nachtmann, H. L. (1999). A comparison of freshman and senior engineering design processes. *Design Studies*, *20*(2), 131–152. https://doi.org/10.1016/S0142-694X(98)00031-3

Ausubel, D. P. (1960). The use of advance organizers in the learning and retention of meaningful verbal material. *Journal of Educational Psychology*, *51*(5), 267–272. https://doi.org/10.1037/h0046669

Balaji, S. (2012). Waterfall vs v-model vs agile : A comparative study on SDLC. *WATEERFALL Vs V-MODEL Vs AGILE : A COMPARATIVE STUDY ON SDLC*, *2*(1), 26–30.

Bargh, J. (2014). Unconscious impulses and desires impel what we think and do in ways Freud never dreamed of. *Scientific American*, *30*(January), 34–39.

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, *26*(9), 677–679. https://doi.org/10.1145/358172.358408

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students. *ACM SIGCSE Bulletin*, *37*(2), 103. https://doi.org/10.1145/1083431.1083474

Becker, B. A. (2016). A new metric to quantify repeated compiler errors for novice programmers. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* (Vol. 11-13-July, pp. 296–301). https://doi.org/10.1145/2899415.2899463

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., … Prather, J. (2019). *Compiler error messages considered unhelpful: The landscape of text-based programming error message research*. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*. https://doi.org/10.1145/3344429.3372508

Bednarik, R., & Tukiainen, M. (2006). An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications* (pp. 125–132). https://doi.org/10.1145/1117309.1117356

Bednarik, R., & Tukiainen, M. (2008). Temporal eye-tracking data: Evolution of debugging strategies with multiple representations. *Eye Tracking Research and Applications Symposium (ETRA)*, *1*(212), 99–102. https://doi.org/10.1145/1344471.1344497

Ben-Ari, M. (2004). Constructivism in computer science education. *ACM SIGCSE Bulletin*, *30*(1), 257–261. https://doi.org/10.1145/274790.274308

Berges, M. (2015). Object-Oriented programming through the lens of computer science education, (April). https://doi.org/10.13140/RG.2.1.2719.0242

Berry, D. C., & Broadbent, D. E. (1988). On the relationship between task performance and verbal knowledge. *Quarterly Journal of Experimental Psychology, 36A, 209-231 .*, 209–231.

Berry, M., & Kölling, M. (2016). Novis: A notional machine implementation for teaching introductory programming. In *Fourth International Conference on Learning and Teaching in Computing and Engineering (LATICE 2016)* (pp. 54–59). Mumbai, India. Retrieved from http://kar.kent.ac.uk/54393/

Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interactions*, *1*, 133–166.

Bowen, G. A. (2009). Document analysis as a qualitative research method. *Qualitative Research Journal*, *9*(no.2), 27–40. https://doi.org/10.3316/qrj0902027

Bowyer, K. W., & Hall, L. O. (1999). Experience using “MOSS” to detect cheating on programming assignments. *Proceedings - Frontiers in Education Conference*, *3*. https://doi.org/10.1109/fie.1999.840376

Brooks, R. (1975). *A model of human cognitive behavior in writing code for computer programs*. Carnegie-Mellon University. Retrieved from https://apps.dtic.mil/dtic/tr/fulltext/u2/a013582.pdf

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal Man-Machine Studies*, *9*, 737–751.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

Brown, J. S., Collins, A., & Duguid, P. (1989). Situated Cognition and the culture of learning. *Educational Researcher*, *18*(1), 32–42. https://doi.org/10.3102/0013189x018001032

Bruner, J. S. (1964). The course of cognitive growth. *American Psychologist*, *19*(1), 1.

Bruner, J. S. (1966a). On cognitive growth. In *Studies in cognitive growth: A collaboration at the center for cognitive studies* (pp. 1–29). Wiley and Sons.

Bruner, J. S. (1966b). On cognitive growth II. In *Studies in cognitive growth: A collaboration at the center for cognitive studies* (pp. 30–67). Wiley and Sons.

Bruner, J. S. (1966c). *Toward a theory of instruction* (Vol. 59). Harvard University Press.

Bruner, J. S. (1971). Readings in educational psychology. In G. J. Mouly (Ed.) (pp. 359–362). Holt, Rinehart, & Winston.

Bruner, J. S. (1976a). *The process of education*. Cambridge, MA: Harvard University Press. Retrieved from http://ebookcentral.proquest.com/lib/purdue/detail.action?docID=3300117. Created

Bruner, J. S. (1976b). *The process of education*. Cambridge, MA: Harvard University Press. https://doi.org/10.1002/bs.3830090108

Bruner, J. S. (1979a). On learning mathematics. In *On Knowing* (p. 189). Cambridge, MA: Belknap Press of Harvard University Press.

Bruner, J. S. (1979b). The Act Of Discovery. In *1On Knowing* (p. 189). Cambridge, MA: Belknap Press of Harvard University Press.

Bruner, J. S. (1997). Celebrating divergence: Piaget and Vygotsky. *Human Development*, *40*(2), 63–73. https://doi.org/10.1159/000278705

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1956). *A study of thinking*. New York, NY: John Wiley & Sons.

Buna, S. (2018). The mistakes I made As a beginner programmer. Retrieved February 9, 2020, from https://medium.com/edge-coders/the-mistakes-i-made-as-a-beginner-programmer-ac8b3e54c312

Burton, R. A. (2009). *On being certain: Believing you are right even when you're not*. Macmillan.

Case, R. (1996). I. Introduction: Reconceptualizing the nature of children's conceptual structures and their development in middle childhood. *Monographs of the Society for Research in Child Development*, *61*(1–2), 1–26. https://doi.org/10.1111/j.1540-5834.1996.tb00535.x

Caspersen, M. E., & Bennedsen, J. (2007). Instructional design of a programming course: a learning theoretic approach. *Proceedings of the Third International Workshop on Computing Education Research*, 111–122. https://doi.org/10.1145/1288580.1288595

Chabris, C. (2016). Does chess make you smarter? Retrieved June 27, 2020, from https://www.wsj.com/articles/does-chess-make-you-smarter-1479403551

Chabris, C. F., & Simons, D. (2011). *The invisible gorilla: And other ways our intuitions deceive us*. Harmony.

Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, *4*(1), 55–81. https://doi.org/10.1016/0010-0285(73)90004-2

Chetty, J. (2015). Lego© mindstorms: merely a toy or a powerful pedagogical tool for learning computer programming? *Conferences in Research and Practice in Information Technology Series*, *159*(JANUARY 2015), 111–118.

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. *Computer Science Education Research*, 85–100.

Cliburn, D. C. (Hanover C. (2003). Experiences with pair programming at a small college. *Journal of Computing Sciences in Colleges*, *19*(1), 20–29. Retrieved from http://dl.acm.org/citation.cfm?id=948741&CFID=380881129&CFTOKEN=42051081

Cockburn, A., & Williams, L. (2001). The costs and benefits of pair programming. *Extreme Programming Examined*, 223–243. https://doi.org/10.1108/00012530210448235

College Board. (2018). *2018 Total group SAT suite of assessments annual report*.

Creswell, J. W. (1997). *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications.

Creswell, J. W. (2008). *Educational research: Planning, conducting, and evaluating quantitative*. Prentice Hall Upper Saddle River, NJ.

Crk, I., Kluthe, T., & Stefik, A. (2015). Understanding programming expertise. *ACM Transactions on Computer-Human Interaction*, *23*(1), 1–29. https://doi.org/10.1145/2829945

Cunningham, K., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using tracing and sketching to solve programming problems (pp. 164–172). https://doi.org/10.1145/3105726.3106190

Curtiss, S., Fromkin, V., Krashen, S., Ringler, D., & Ringler, M. (1974). The linguistic development of Genie. *Language*, *50*(3), 528–554. Retrieved from https://www.jstor.org/stable/412222

Damasio, A. R. (2006). *Descartes' error*. Random House.

Degregory, L. (2017). The girl in the real world. *Tampa Bay Times*. Retrieved from https://projects.tampabay.com/projects/girl-in-the-window/three-years-later/

Dehnadi, S., & Bornat, R. (2006). The camel has two humps. *Middlesex University, UK*, 1–21. Retrieved from http://mrss.dokoda.jp/r/http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

Denny, P., Becker, B. A., Craig, M., Wilson, G., & Banaszkiewicz, P. (2019). Research this! Questions that computing educators most want computing education researchers to answer. *ICER 2019 - Proceedings of the 2019 ACM Conference on International Computing Education Research*, 259–267. https://doi.org/10.1145/3291279.3339402

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. *ICER'08 - Proceedings of the ACM Workshop on International Computing Education Research*, 113–124. https://doi.org/10.1145/1404520.1404532

Dewey, J. (1938). *Experience & education*. New York, NY: Touchstone.

Dickson, P. E., Brown, N. C. C., & Becker, B. A. (2020). Engage against the machine: Rise of the notional machines as effective pedagogical devices. In *ITiCSE'20* (pp. 159–165). https://doi.org/10.1145/3341525.3387404

Dowling, J. E. (2018). *Understanding the brain: From cells to behavior to cognition*. WW Norton & Company.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, *2*(1), 57–73. https://doi.org/10.2190/3LFX-9RRF-67T8-UVK9

Du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*, *51*(3), 265–277. https://doi.org/10.1006/ijhc.1981.0309

Dubinsky, E., & McDonald, M. A. (2001). APOS: A constructivist theory of learning in undergraduate mathematics education research. In *The teaching and learning of mathematics at university level* (pp. 275–282). Springer.

Dunn, D. (2013). 10 skills computer science students should have. Retrieved November 13, 2018, from http://www.theshorthorn.com/life_and_entertainment/skills-computer-science-students-should-have/article_3a075896-65b3-11e2-aad4-0019bb30f31a.html

Eagleman, D., & Downar, J. (2016). *Brain and behavior a cognitive neuroscience perspective*. New York; Oxford: Oxford University Press.

Eckerdal, A., & Berglund, A. (2005). What does it take to learn "programming thinking"? *Proceedings of the 2005 International Workshop on Computing Education Research - ICER '05*, 135–142. https://doi.org/10.1145/1089786.1089799

Eckerdal, A., McCartney, R., Moström, J. E., Sanders, K., Thomas, L., & Zander, C. (2007). From limen to leumen: Computing students in liminal spaces. *3th International Workshop on Computing Education Research*, 123–132. https://doi.org/10.1145/1288580.1288597

Ehri, L. C., Nunes, S. R., Stahl, S. A., & Willows, D. M. (2001). Systematic phonics instruction helps students learn to read: Evidence from the national reading panel's meta-analysis. *Review of Educational Research*, *71*(3), 393–447. https://doi.org/10.3102/00346543071003393

Ellis, T. (2008). IBM's Deep Blue chess grandmaster chips, *63*(4), 754–761. https://doi.org/10.1227/01.NEU.0000325492.58799.35

Epistemology. (2005). Retrieved August 27, 2019, from https://plato.stanford.edu/entries/epistemology/

Evans, J. S. B. T., & Frankish, K. E. (2009). *In two minds: Dual processes and beyond.* Oxford University Press.

Falkner, K., Vivian, R., & Falkner, N. J. G. (2013). Neo-piagetian forms of reasoning in software development process construction. *Proceedings - 2013 Learning and Teaching in Computing and Engineering, LaTiCE 2013*, 31–38. https://doi.org/10.1109/LaTiCE.2013.23

Feurzeig, W., Papert, S. A., & Lawler, B. (2011). Programming-languages as a conceptual framework for teaching mathematics. *Interactive Learning Environments*, *19*(5), 487–501. https://doi.org/10.1080/10494820903520040

Fincher, S. A., & Robins, A. V. (Eds.). (2019). *The Cambridge handbook of computing education research*. *The Cambridge Handbook of Computing Education Research*. Cambridge University Press. https://doi.org/10.1017/9781108654555

Fischer, K. W., & Bidell, T. R. (2007). *Dynamic development of action and thought*. *Handbook of Child Psychology*. https://doi.org/10.1002/9780470147658.chpsy0107

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93–116. https://doi.org/10.1080/08993400802114508

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*, 74–79. https://doi.org/10.1145/169059.169088

Flinders, K. (2019). Computer science undergraduates most likely to drop out. Retrieved February 13, 2020, from https://www.computerweekly.com/news/252467745/Computer-science-undergraduates-most-likely-to-drop-out

Floyd, B., Santander, T., & Weimer, W. (2017). Decoding the representation of code in the brain: An fMRI study of code review and expertise. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, 175–186. https://doi.org/10.1109/ICSE.2017.24

Frederick, S. (2005). Cognitive reflection and decision making. *Journal of Economic Perspectives*, *19*(4), 25–42. https://doi.org/10.1257/089533005775196732

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernan-Losada, I., Jackova, J., … Thompson, E. (2015). Developing a computer science-specific learning taxonomy, *29*, 341–349. https://doi.org/10.1111/cobi.12410

Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: A preliminary investigation of novice programmers' poroblems. In *Seventh Australasian Computing Education Conference (ACE2005)* (Vol. 42, pp. 173–180). Retrieved from http://crpit.com/confpapers/CRPITV42Garner.pdf

Ginsburg, H. P., & Opper, S. (1988). *Piaget's theory of intellectual development*. Prentice-Hall, Inc.

Glaser, B., & Strauss, A. (1967). *The discovery of grounded theory*. London, UK: Weidenfield & Nicolson.

Goldman, A. I. (1986). *Epistemology and cognition*. Harvard University Press.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *SIGCSE'08 - Proceedings of the 39th ACM Technical Symposium on Computer Science Education*, 256–260. https://doi.org/10.1145/1352135.1352226

Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Applied Psychology*, 93–109. https://doi.org/10.1111/j.2044-8325.1977.tb00363.x

Greenemeier, L. (2017). 20 Years after Deep Blue: How AI has advanced since conquering chess. Retrieved June 27, 2020, from https://www.scientificamerican.com/article/20-years-after-deep-blue-how-ai-has-advanced-since-conquering-chess/

Grover, S. (2014). Balanced designs for deeper learning in an online computer.

Guzdial, M. (2015). Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, *8*(6), 1–165. https://doi.org/10.2200/S00684ED1V01Y201511HCI033

Harari, O. (1995). *The missing link in cognition: Origins of self-reflective consciousness*. *Management Review* (Vol. 84). Retrieved from http://search.proquest.com.ezp-prod1.hul.harvard.edu/docview/206678868?accountid=11311%5Cnhttp://sfx.hul.harvard.edu/hvd?url_ver=Z39.88-2004&rft_val_fmt=info:ofi/fmt:kev:mtx:journal&genre=article&sid=ProQ:ProQ:abiglobal&atitle=The+missing+link+in+performa

Hassabis, D., Kumaran, D., Vann, S. D., & Maguire, E. A. (2007). Patients with hippocampal amnesia cannot imagine new experiences. *Proceedings of the National Academy of Sciences of the United States of America*, *104*(5), 1726–1731. https://doi.org/10.1073/pnas.0610561104

Hatano, G., & Inagaki, K. (1984). Two courses of expertise. *Research and Clinical Center For Child Development Annual Report*, *6*, 27–36. https://doi.org/10.1002/ccd.10470

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *International Journal of Phytoremediation*, *21*(1), 95–122. https://doi.org/10.1076/csed.13.2.95.14202

Head, M. L., Holman, L., Lanfear, R., Kahn, A. T., & Jennions, M. D. (2015). The extent and consequences of p-hacking in science. *PLoS Biology*, *13*(3), 1–15. https://doi.org/10.1371/journal.pbio.1002106

Herman, G. L., Loui, M. C., & Zilles, C. (2010). Creating the digital logic concept inventory. *SIGCSE'10 - Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, (Ci), 102–106. https://doi.org/10.1145/1734263.1734298

Hintikka, J. (1962). Cogito, ergo sum: Inference or performance? *The Philosophical Review*, *71*(1), 3–32.

Horn, J. L., & Cattell, R. B. (1967). Age differences in fluid and crystallized intelligence. *Acta Psychologica*, *26*, 107–129. https://doi.org/10.1016/0001-6918(67)90011-X

Houdé, O., & Borst, G. (2015). Evidence for an inhibitory-control theory of the reasoning brain. *Frontiers in Human Neuroscience*, *9*(March), 1–5. https://doi.org/10.3389/fnhum.2015.00148

Houdé, O., Pineau, A., Leroux, G., Poirel, N., Perchey, G., Lanoë, C., … Mazoyer, B. (2011). Functional magnetic resonance imaging study of Piaget's conservation-of-number task in preschool and school-age children: A neo-Piagetian approach. *Journal of Experimental Child Psychology*, *110*(3), 332–346. https://doi.org/10.1016/j.jecp.2011.04.008

Immordino-Yang, M. H., & Damasio, A. (2007). We feel, therefore we learn: The relevance of affective and social neuroscience to education. *Mind, Brain, and Education*, *1*(1), 3–10. https://doi.org/10.1111/j.1751-228X.2007.00004.x

Inhelder, B., Chipman, H. H., & Zwingmann, C. (1976). *Piaget and his school: a reader in developmental psychology*. Springer.

James, J. (2008). 10 traits to look for when you're hiring a programmer. Retrieved November 13, 2018, from https://www.techrepublic.com/blog/10-things/10-traits-to-look-for-when-youre-hiring-a-programmer/

Ji, F., & Sedano, T. (2011). Comparing extreme programming and waterfall project results. *2011 24th IEEE-CS Conference on Software Engineering Education and Training, CSEE and T 2011 - Proceedings*, 482–486. https://doi.org/10.1109/CSEET.2011.5876129

Joy, M., Sinclair, J., Sun, S., Sitthiworachart, J., & López-González, J. (2009). Categorising computer science education research. *Education and Information Technologies*, *14*(2), 105–126. https://doi.org/10.1007/s10639-008-9078-4

Kahneman, D. (1973). *Attention and effort. Englewood Cliffs, NJ*. https://doi.org/10.2307/1421603

Kahneman, D. (2011). *Thinking, fast and slow*. Macmillan.

Kahneman, D., Fredrickson, B. L., Schreiber, C. A., & Redelmeier, D. A. (1993). When more pain is preferred to less. *Psychological Science*, *4*, 401–405.

Kahneman, D., & Tversky, A. (1973). On the psychology of prediction. *Psychological Review*, *80*(4), 237–251. https://doi.org/10.1037/h0046234

Kalelioglu, F., Gulbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing*, *4*(3), 583–596.

Kandel, E. R. (2009). The biology of memory: A forty-year perspective. *Journal of Neuroscience*, *29*(41), 12748–12756. https://doi.org/10.1523/JNEUROSCI.3958-09.2009

Karni, A., & Sagi, D. (1991). Where practice makes perfect in texture discrimination: Evidence for primary visual cortex plasticity. *Proceedings of the National Academy of Sciences of the United States of America*, *88*(11), 4966–4970. https://doi.org/10.1073/pnas.88.11.4966

Kennedy, C. H. (2005). *Single-case designs for educational research*. Prentice Hall.

Kerlinger, F. N., & Lee, H. B. (2000). *Foundations of behavioral research.* Orlando, FL: Harcourt Inc.

Khalife, J. T. (2006). Threshold for the introduction of programming: providing learners with a simple computer model. *28th International Conference on Information Technology Interfaces*, (September 2006), 71–76. https://doi.org/10.1109/ITI.2006.1708454

Kinnunen, P., & Simon, B. (2010a). Building theory about computing education phenomena. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, 37–42. https://doi.org/10.1145/1930464.1930469

Kinnunen, P., & Simon, B. (2010b). Experiencing programming assignments in CS1 : The emotional toll. *ICER'10*, 77–85. https://doi.org/10.1145/1839594.1839609

Kinnunen, P., & Simon, B. (2010c). Experiencing programming assignments in CS1, 77. https://doi.org/10.1145/1839594.1839609

Kinnunen, P., & Simon, B. (2011). CS majors' self-efficacy perceptions in CS1: Results in light of social cognitive theory. *ICER'11*, 19–26. https://doi.org/10.1145/2016911.2016917

Kinnunen, P., & Simon, B. (2012). My program is ok – am I? Computing freshmen's experiences of doing programming assignments. *Computer Science Education*, *22*(1), 1–28. https://doi.org/10.1080/08993408.2012.655091

Klerer, M. (1984). Experimental study of a two-dimensional language vs Fortran for first-course programmers. *International Journal of Man-Machine Studies*, *20*(5), 445–467. https://doi.org/10.1016/S0020-7373(84)80021-8

Knowlton, B. J., Mangels, J. A., & Squire, L. R. (2016). A neostriatal habit learning system in humans author(s). *American Association for the Advancement of Science Stable*, *273*(5280), 1399–1402. Retrieved from http://www.jstor.org/stable/2891421

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, *16*(1-2 SPEC. ISS.), 41–84. https://doi.org/10.1016/j.jvlc.2004.08.003

Kort, B., Reilly, R., & Picard, R. (2001). An affective model of the interplay between emotions and learning. *Learning*, (January), 1–4.

Krashen, S. D. (2009). *Principles and practice in second language acquisition* (First inte). Pergamon Press Inc.

Kwon, K. (2017). Student's misconception of programming reflected on problem-solving plans. *International Journal of Computer Science Education in Schools*, *1*(4), 14. https://doi.org/10.21585/ijcses.v1i4.19

Lattman, P. (2007, September 27). The origins of Justice Stewart's "I know it when I see it." *The Wall Street Journal*. https://doi.org/https://blogs.wsj.com/law/2007/09/27/the-origins-of-justice-stewarts-i-know-it-when-i-see-it/

Ledoux, J. E., Brown, R., Lau, H., & Mobbs, D. (2017). A higher-order theory of emotional consciousness, 1–10. https://doi.org/10.1073/pnas.1619316114

Legaspi, C. (2014). Agile myth #6: "Agile means no upfront design." Retrieved from https://www.javacodegeeks.com/2014/09/agile-myth-6-agile-means-no-upfront-design.html

Leslie, A. (1987). Pretense and representation: The origins of "Theory of Mind." *American Psychological Association*, *94*(4), 412–426. Retrieved from file://localhost/Volumes/Volume_2/ZoteroLib_LV/library/Papers/2001/Leslie/2001 Leslie-1.pdf

Lishinski, A., Good, J., Sands, P., & Yadav, A. (2016). Methodological rigor and theoretical foundations of CS education research. In *ICER 2016 - Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 161–169). https://doi.org/10.1145/2960310.2960328

Lister, R. (2010). Geek Genes and Bimodal Grades. *ACM Inroads*, *1*(3), 16–17. https://doi.org/10.1198/00031300265

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australian Computing Education Conference* (pp. 9–18).

Lister, R. (2016). Toward a developmental epistemology of computer programming. *Proceedings of the 11th Workshop in Primary and Secondary Computing Education on ZZZ - WiPSCE '16*, 5–16. https://doi.org/10.1145/2978249.2978251

Lister, R., Fidge, C., & Teague, D. (2009). Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE* (pp. 161–165). https://doi.org/10.1145/1562877.1562930

Lister, R., & Leaney, J. (2004). Introductory programming, criterion-referencing, and bloom. *ACM SIGCSE Bulletin*, *35*(1), 143. https://doi.org/10.1145/792548.611954

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., … Sanders, K. (2004). *A multi-national study of reading and tracing skills in novice programmers*. *ACM SIGCSE Bulletin* (Vol. 36). https://doi.org/10.1145/1041624.1041673

Logo history. (2015). Retrieved February 23, 2019, from http://el.media.mit.edu/logo-foundation/what_is_logo/history.html

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceeding of the fourth international workshop on Computing education research - ICER '08*. https://doi.org/10.1145/1404520.1404531

Lowe, A. A. (2019). Debugging: The key to unlocking the mind of a novice programmer? In *Frontiers in Education*. Cincinatti, OH.

Lowe, T. (1999a). Efficiency tradeoffs in Ada: Pinching pennies while losing dollars. *ACM SIGAda Ada Letters*, *19 (3)*, 183–193.

Lowe, T. (1999b). Extending Ada to assist multiprocessor embedded development. *ACM SIGAda Ada Letters*, *19 (3)*, 125–132.

Lowe, T. (2019). Novice struggles in two parts. In *ITiCSE'19*. Aberdeen, Scotland, UK.

Lowe, T. A. (2018). Misconceptions and the notional machine in very young programming learners. In *ASEE Annual Conference Proceedings*.

Lowe, T. A., & Brophy, S. B. (2017). Understanding the impact of strategic team formation in early programming education. In *ASEE Annual Conference Proceedings*. Columbus, OH.

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education*. https://doi.org/10.1080/08993408.2011.554722

MacKay, D. G. (2019). *Remembering: What 50 years of research with famous amnesia patient H.M. can teach us about memory and how It works*. Prometheus Books. Retrieved from https://books.google.com/books?id=MeU4DwAAQBAJ

MacKay, D. G., Burke, D. M., & Stewart, R. (1998). H.M.'s language production eeficits: Implications for relations between memory, semantic binding, and the hippocampal system. *Journal of Memory and Language*, *38*(1), 28–69. https://doi.org/10.1006/jmla.1997.2544

Margulieux, L. E., Catrambone, R., & Guzdial, M. (2013). Subgoal labeled worked examples improve K-12 teacher performance in computer programming training. *Proceedings of the 35th Annual Conference of the Cognitive Science Society*, 978-983.

Margulieux, L. E., Morrison, B. B., & Decker, A. (2019). Design and pilot testing of subgoal labeled worked examples for five core concepts in CS1. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 548–554. https://doi.org/10.1145/3304221.3319756

Markushin, Y. (2013). 10 reasons why most people are not good at chess (and how to fix it). Retrieved August 23, 2019, from https://thechessworld.com/articles/general-information/10-reasons-why-most-people-are-not-good-at-chess-and-how-to-fix-it/

Martin, B., & Carle, E. (1996). *Brown bear, brown bear, what do you see?* Henry Holt and Co.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *ACM Computing Surveys*, *13*(1), 121–141. https://doi.org/10.1145/356835.356841

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K., & Zander, C. (2013). Can first-year students program yet? A study revisited. *ICER 2013 - Proceedings of the 2013 ACM Conference on International Computing Education Research*, 91–98. https://doi.org/10.1145/2493394.2493412

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

McCracken, M. (2001). Evaluation of programs for the ITiCSE working group on programming skill assessment. https://doi.org/10.1017/CBO9781107415324.004

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., … Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125. https://doi.org/10.1145/572139.572181

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE Bulletin*, *34*(1), 38. https://doi.org/10.1145/563517.563353

McLeod, S. (2019). Bruner - Learning theory in education. Retrieved January 1, 2020, from https://www.simplypsychology.org/bruner.html

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. St., & Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education - ITiCSE-WGR '06*, (May 2014), 182. https://doi.org/10.1145/1189215.1189185

Merriam, S. B., & Tisdell, E. J. (2016). *Qualitative research: A guide to design and implemenetation*. John Wiley & Sons.

Meyer, J. H. F., & Land, R. (2003). *Threshold concepts and troublesome knowledge*. Edinburgh: University of Edinburgh. https://doi.org/10.1007/978-3-8348-9837-1

Milner, B., Corkin, S., & Teuber, H.-L. (1968). Futher analysis of the hippocampal amnesic syndrom: 14-year follow-up study of H.M. *Neuropsychologia*, *6*, 215–234. Retrieved from papers://d3db4aee-e34c-4aff-b84c-09e10f8ad2a4/Paper/p653

Montesi, J. (2015). Measure twice, cut once: How to avoid and overcome technical debt. Retrieved February 9, 2020, from https://www.veracode.com/blog/2015/01/measure-twice-cut-once-how-avoid-and-overcome-technical-debt

Moravec, H. (1998). When will computer hardware match the human brain ? *Journal of Evolution and Technology*, *1*(1), 10. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.7883&rep=rep1&type=pdf

Moray, N. (2013). *Mental workload: Its theory and measurement* (Vol. 8). Springer Science & Business Media.

Morra, S., Gobbo, C., Marini, Z., & Sheese, R. (2008). *Cognitive development: Neo-Piagetian perspectives*. New York, NY: Taylor & Francis Group LLC.

Morrison, B. B. (2015). Computer science is different! Educational psychology experiments do not reliably replicate in programming domain, 267–268.

Morrison, B. B. (2017). Dual modality code explanations for novices. *Proceedings of the 2017 ACM Conference on International Computing Education Research - ICER '17*, 226–235. https://doi.org/10.1145/3105726.3106191

Morrison, B. B., Decker, A., & Margulieux, L. E. (2016). Learning loops: A replication study illuminates impact of HS courses. *ICER 2016 - Proceedings of the 2016 ACM Conference on International Computing Education Research*, 221–230. https://doi.org/10.1145/2960310.2960330

Morrison, B. B., Dorn, B., & Guzdial, M. (2014). Measuring cognitive load in introductory CS. *Proceedings of the Tenth Annual Conference on International Computing Education Research - ICER '14*, 131–138. https://doi.org/10.1145/2632320.2632348

Morse, A. (2017). Tweet. Retrieved February 9, 2020, from https://twitter.com/mrmrs_/status/919984596166508544

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *ACM SIGCSE Bulletin*, *39*(3), 151. https://doi.org/10.1145/1269900.1268830

Naccache, L., Dehaene, S., Cohen, L., Habert, M. O., Guichart-Gomez, E., Galanaud, D., & Willer, J. C. (2005). Effortless control: Executive attention and conscious feeling of mental effort are dissociable. *Neuropsychologia*, *43*(9), 1318–1328. https://doi.org/10.1016/j.neuropsychologia.2004.11.024

Nickerson, R. S. (1998). Confirmation bias: A ubiquitous phenomenon in many guises. *Review of General Psychology*, *2*(2), 175–220. https://doi.org/10.1214/aos/1031594735

O'Flaherty, K. (2020). Clearview AI, the company whose database has amassed 3 billion photos, hacked. Retrieved June 28, 2020, from https://www.forbes.com/sites/kateoflahertyuk/2020/02/26/clearview-ai-the-company-whose-database-has-amassed-3-billion-photos-hacked/#7a9fc7987606

Papert, Seymour; (1978). Interim report of the LOGO project in the Brookline public schools: An assessment and documentation of a children's computer laboratory. *Artificial Intelligence Memo*.

Papert, Seymour;, Watt, D., DiSessa, A., & Weir, S. (1979). Final report of the Brookline LOGO project - Part II :: project summary & data analysis.

Papert, Seymour. (1987). Computer criticism vs. technocentric thinking. *Educational Researcher*, *16*(1), 22. https://doi.org/10.2307/1174251

Papert, Seymour. (1988). The conservation of Piaget: The computer as grist. In *Constructivism in the computer age* (pp. 3–14).

Papert, Seymour. (1991). Situating constructionism. In *Constructionism* (pp. 1–11). https://doi.org/10.1111/1467-9752.00269

Pea, R. D. (1983). Logo programming and problem solving. *Conference Paper*, *150*(ir 014 383), 1–10.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Pea, R. D. (1987). The aims of software criticism: Reply to Professor Papert. *Educational Researcher*, *16*(5), 4–8. https://doi.org/10.3102/0013189x016005004

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*, *2*(2), 137–168. https://doi.org/10.1016/0732-118X(84)90018-7

Pellegrino, J. W. (2002). Knowing what students know. *Issues in Science and Technology*, *19*(2), 48–52.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In *Psychology of programming* (pp. 45–62). Elsevier.

Perkins, D., & Martin, F. (1985). *Fragile knowledge and neglected strategies in novice programmers*. Washington, DC. Retrieved from https://files.eric.ed.gov/fulltext/ED295618.pdf

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Perkins, David N. (2010). *Making learning whole: How seven principles of teaching can transform education*. John Wiley & Sons.

Perkins, David N, & Salomon, G. (1992). The science and art of transfer. *If Minds Matter: A Foreword to the Future*, *1*, 201–210.

Piaget, J. (1970). *Genetic epistemology*. (E. Duckworth, Trans.). New York: The Norton Library.

Piaget, J. (1972). Intellectual evolution from adolescence to adulthood. *Human Development*, *15*(1), 1–12.

Piaget, J. (1995). Commentary on Vygotsky's criticisms of language and thought of the child and judgment and reasoning in the child. *New Ideas in Psychology*, *13*(3), 325–340. https://doi.org/10.1016/0732-118X(95)00010-E

Piaget, J., & Cook, M. (1952). *The origins of intelligence in children* (Vol. 8). International Universities Press New York.

Piaget, J., & Inhelder, B. (1967). *The child's conception of space*. New York, NY: W. W. Norton and Company.

Plass, J. L., Moreno, R., & Brünken, R. (2010). *Cognitive load theory*. *Americas* (Vol. 32). Retrieved from www.cambridge.org%5Cnwww.cambridge.org/9780521677585

Premack, D., & Woodruff, G. (1978). Does the chimpanzee have a theory of mind? *The Behavior and Brain Sciences*, *4*, 515–526.

Pressman, R. S. (2010). *Software engineering: a practitioner's approach* (Seventh). Palgrave macmillan.

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education*. https://doi.org/10.1145/3077618

Raymond, E. (2005). The cathedral and the bazaar. *First Monday*, *2*(SPEC), 23–49.

Reber, A. S. (1989). Implicit learning and tacit knowledge. *Journal of Experimental Psychology: General*, *118*(3), 219–235. https://doi.org/10.1037/0096-3445.118.3.219

Roberts, E. (2002). Strategies for promoting academic integrity in CS courses. *Proceedings - Frontiers in Education Conference*, *2*, 14–19. https://doi.org/10.1109/fie.2002.1158209

Robins, A. (2010). *Learning edge momentum: A new account of outcomes in CS1*. *Computer Science Education* (Vol. 20). https://doi.org/10.1080/08993401003612167

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, *13*(2), 137–172. https://doi.org/10.1076/csed.13.2.137.14200

Rountree, J., Robins, A., & Rountree, N. (2013). Elaborating on threshold concepts. *Computer Science Education*, *23*(3), 265–289. https://doi.org/10.1080/08993408.2013.834748

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs. *Proceedings - IEEE 2002 Symposia on Human Centric Computing Languages and Environments, HCC 2002*, 37–39. https://doi.org/10.1109/HCC.2002.1046340

Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, *15*(1), 59–82. https://doi.org/10.1080/08993400500056563

Salleh, N., Mendes, E., & Grundy, J. (2011). Empirical studies of pair programming for CS/SE teaching in higher education: A systematic literature review. *IEEE Transactions on Software Engineering*, *37*(4), 509–525. https://doi.org/10.1109/TSE.2010.59

Segue Technologies. (2013). How does a preliminary design phase improve software development? Retrieved February 9, 2020, from https://www.seguetech.com/preliminary-design-improve-development/

Seuss, D. (1963). *Hop on pop*. Random House Books for Young Readers.

Shneiderman, B. (1977). Teaching programming: A spiral approach to syntax and semantics. *Computers and Education*, *1*(4), 193–197. https://doi.org/10.1016/0360-1315(77)90008-2

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., … Brechmann, A. (2014). Understanding understanding source code with functional magnetic resonance imaging. In *ICSE* (pp. 378–389). https://doi.org/10.1145/2568225.2568252

Silver, P. F. (1983). *Educational administration: Theoretical perspectives on practice and research*. Harpercollins College Div.

Sime, M. E., & Arblaster, A. T. (1977). Structuring the programmer's task. *Journal of Occupational Psychology*, *50*, 205–216.

Simonite, T. (2019). The best algorithms still struggle to recognize black faces. Retrieved June 27, 2020, from https://www.wired.com/story/best-algorithms-struggle-recognize-black-faces-equally/

Simons, D. J., & Chabris, C. F. (1999). Gorillas in our midst: Sustained inattentional blindness for dynamic events. *Perception*, *28*(9), 1059–1074. https://doi.org/10.1068/p281059

Skinner, B. F. (1965). *Science and human behavior*. New York: The Macmillan Company.

Sleeman, D., Putnam, R. T., Baxter, J., & Kuspa, L. (1986). Pascal and high school students: A study of errors. *Journal of Educational Computing Research*, *2*(1), 5–23. https://doi.org/10.2190/2xpp-ltyh-98nq-bu77

Sloyan, T. (2017). 7 top qualities that make a successful software developer. Retrieved November 13, 2018, from https://www.siliconrepublic.com/advice/software-developer-top-qualities-skills

Socha, D. (2004). Aikido and software engineering. *Proceedings of the 2004 ACM Workshop on Interdisciplinary Software Engineering Research - WISER '04*, 4–7. https://doi.org/10.1145/1029997.1030000

Socha, D., & Walter, S. (2006). Is designing software different from designing other things? *International Journal of Engineering Education*, *22*(3), 540–550. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.8109

Soloway, E. (1986). Learning to program = Learning to construct mechanisms. *Communications of the ACM*, *29*(9), 850–858.

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM*, *26*(11), 853–860. https://doi.org/10.1145/182.358436

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, *19*(5), 595–609.

Sorva, J. (2010). Reflections on threshold concepts in computer programming and beyond. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, 21–30. https://doi.org/10.1145/1930464.1930467

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)*, *13*(4), 15.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, *29*(7), 624–632. https://doi.org/10.1145/6138.6145

Squire, L. R. (1984). Nondeclarative memory: Multiple brain systems supporting learning. *Neuroscience*, *4*(3), 232–243. Retrieved from http://www.mitpressjournals.org/doi/abs/10.1162/jocn.1992.4.3.232

Squire, L. R., & Kandel, E. R. (2003). *Memory: From mind to molecules* (Vol. 69). Macmillan.

Stanovich, K. E. (2012). Distinguishing the reflective, algorithmic, and autonomous minds: Is it time for a tri-process theory? *In Two Minds: Dual Processes and Beyond*, 55–88. https://doi.org/10.1093/acprof:oso/9780199230167.003.0003

Stephenson, C., Derbenwick Miller, A., Alvarado, C., Barker, L., Barr, V., Camp, T., … Zweben, S. (2018). *Retention in computer science undergraduate programs in the US*. *ACM*. New York, NY.

Strawhacker, A., & Bers, M. U. (2014). I want my robot to look for food: comparing kindergartner's programming comprehension using tangible, graphic, and hybrid user interfaces. *International Journal of Technology and Design Education*, 293–319. https://doi.org/10.1007/s10798-014-9287-7

Sun, R. (1997). Learning, action and consciousness: A hybrid approach toward modelling consciousness. *Neural Networks*, *10*(7), 1317–1331. https://doi.org/10.1016/S0893-6080(97)00050-6

Sun, R., Merrill, E., & Peterson, T. (2001). *From implicit skills to explicit knowledge* (Vol. 25). Retrieved from papers3://publication/uuid/D3A1E80A-D0BE-40CD-829A-9BC55BEE6382

Sun, R., Slusarz, P., & Terry, C. (2005). The interaction of the explicit and the implicit in skill learning: A dual-process approach. *Psychological Review*, *112*(1), 159–192. https://doi.org/10.1037/0033-295X.112.1.159

Svinicki, M. D. (2004). *Learning and motivation in the postsecondary classroom*. Anker Publishing Company.

Sweller, J. (1989). Cognitive Technology: Some Procedures for Facilitating Learning and Problem Solving in Mathematics and Science. *Journal of Educational Psychology*, *81*(4), 457–466. https://doi.org/10.1037/0022-0663.81.4.457

Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, *4*(4), 295–312. https://doi.org/10.1016/0959-4752(94)90003-5

Szabo, C., Falkner, N., Petersen, A., Bort, H., Cunningham, K., Donaldson, P., … Sheard, J. (2019). Review and use of learning theories within computer science education research: Primer for researchers and practitioners. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* (pp. 89–109). https://doi.org/10.1145/3304221.3325534

Talbot, M. (2012). Critical reasoning for beginners. Retrieved August 11, 2019, from Critical Reasoning for Beginners

Talbot, M. (2014a). Critical reasoning: A romp through the foothills of logic.

Talbot, M. (2014b). *Critical reasoning: A romp through the foothills of logic for complete beginners (Kindle edition)*. Metafore. Retrieved from https://www.amazon.com/Critical-Reasoning-Foothills-Complete-Beginners-ebook/dp/B00MSUX7E6

Taylor, C., Clancy, M., Webb, K. C., Zingaro, D., Lee, C., & Porter, L. (2020). The practical details of building a cs concept inventory. In *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE* (pp. 372–378). https://doi.org/10.1145/3328778.3366903

Teague, D. (2014). Neo-Piagetian theory and the novice programmer. Retrieved from www.ppig.org

TechNation Radio Podcast. (2019). Episode 19-15 Beyond moonshots? There's loonshots! Retrieved August 11, 2019, from https://www.podomatic.com/podcasts/technation/episodes/2019-04-10T03_09_20-07_00

The Karate Kid. (1984). Retrieved March 27, 2020, from https://www.imdb.com/title/tt0087538/?ref_=fn_al_tt_1

The rationalism of Descartes. (2019). Retrieved August 20, 2019, from https://www.britannica.com/topic/Western-philosophy/The-rationalism-of-Descartes

Thomas, L., Ratcliffe, M., & Thomasson, B. (2004). Scaffolding with object diagrams in first year programming classes: Some unexpected results. *SIGCSE Bulletin*, *36*(1), 250–254. https://doi.org/http://doi.acm.org/10.1145/1028174.971390

Thornburg, D. (2013). *From the campfire to the holodeck: Creating engaging and powerful 21st century learning environments*. John Wiley & Sons.

Thummadi, B. V., Shiv, O., & Lyytinen, K. (2011). Enacted routines in agile and waterfall processes. *Proceedings - 2011 Agile Conference, Agile 2011*, 67–76. https://doi.org/10.1109/AGILE.2011.29

Thuné, M., & Eckerdal, A. (2010). Students' conceptions of computer programming. Retrieved from http://www.diva-portal.org/smash/record.jsf?pid=diva2:347073

Tindall-Ford, S., Agostinho, S., & Sweller, J. (Eds.). (2019). *Advances in Cognitive Load Theory: Rethinking Teaching*. Routledge.

Top 10 qualities of information technologists. (2018). Retrieved November 13, 2018, from http://computerschools.com/resources/top-10-qualities-of-information-technologists

Tulving, E. (2005). Episodic memory and autonoesis: uniquely human? In *The missing link in cognition: Origins of self-reflective consciousness*. Oxford University Press.

Turner, J. H. (2003). *The structure of sociologoical theory* (Seventh Ed). Belmont, CA, USA: Wadsworth/Thompson Learning.

Tversky, A., & Kahneman, D. (1983). Extensional versus intuitive reasoning: The conjunction fallacy in probability judgment. *Psychological Review*, *90*(4), 293–315.

Utting, I., Tew, A. E., McCracken, M., Thomas, L., Bouvier, D., Frye, R., … Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports* (pp. 15–31). ACM. https://doi.org/10.1111/cobi.12410

Vainio, V., & Sajaniemi, J. (2007). Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin*, *39*(3), 236. https://doi.org/10.1145/1269900.1268853

Van Toll, T., Lee, R., & Ahlswede, T. (2007). Evaluating the usefulness of pair programming in a classroom setting. *Proceedings - 6th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2007; 1st IEEE/ACIS International Workshop on e-Activity, IWEA 2007*, (Icis), 302–307. https://doi.org/10.1109/ICIS.2007.96

Vygotsky, L. (1962). *Thought and language*. (E. Hanfmann & G. Vakar, Eds.) (Third Pape). Cambridge, MA: M.I.T. Press.

Vygotsky, L. (1978). *Mind in Society*. https://doi.org/10.1016/S0006-3495(96)79572-3

Vygotsky, L. (1986). *Thought and language*. (A. Kozulin, Trans.). Cambridge, MA: MIT Press. https://doi.org/10.1037/11193-000

Waal, F. de. (2016). *Are we smart enough to know how smart animals are?* New York, NY: Norton.

Weintrop, D., & Wilensky, U. (2015). Using commutative assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. *International Computing Education Research Conference (ICER)*, (January), 101–110. https://doi.org/10.1145/2787622.2787721

Werner, L., Ruvalcaba, O., & Denner, J. (2016). Observations of pair programming: variations in collaboration across demographic groups. *Sigcse '16*, 1–6. https://doi.org/10.1145/2839509.2844558

Weyer, S. A., & Cannara, A. B. (1975). *Children learning computer programming: Experiments with languages, curricula, and programmable devices*.

Whalley, J. L., & Kasto, N. (2014). A qualitative think-aloud study of novice programmers' code writing strategies. *Iticse '14*, 279–284. https://doi.org/10.1145/2591708.2591762

Whalley, J. L., Lister, R., Thompson, E., Clear, T., Robbins, P., Ajith Kumar, P. K., & Prasad, C. (2006). An Australasian study of reading and comprehension skills in novice programmers, using the bloom and SOLO taxonomies. *Conferences in Research and Practice in Information Technology Series*, *52*, 243–252.

Wiedenbeck, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, *23*(4), 383–390. https://doi.org/10.1016/S0020-7373(85)80041-9

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*. https://doi.org/10.1006/imms.1993.1084

Wiggins, G., & McTighe, J. (2005). *Understanding by design*. Alexandria, VA: ASCD.

Williams, L., Layman, L., Osborne, J., & Katira, N. (2006). Examining the compatibility of student pair programmers. *Proceedings - AGILE Conference, 2006*, *2006*, 411–420. https://doi.org/10.1109/AGILE.2006.25

Wilson, B. C. (2010). A study of factors promoting success in computer science Including gender differences. *Computer Science Education*, *12*(1–2), 141–164. https://doi.org/10.1076/csed.12.1.141.8211

Wittie, L., Kurdia, A., & Huggard, M. (2017). Developing a concept inventory for computer science 2. *Proceedings - Frontiers in Education Conference, FIE*, *2017-Octob*, 1–4. https://doi.org/10.1109/FIE.2017.8190459

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x

Xie, B., Nelson, G. L., & Ko, A. J. (2018). An explicit strategy to scaffold novice program tracing. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18*, 344–349. https://doi.org/10.1145/3159450.3159527

Youngs, E. A. (1974). Human errors in programming. *International Journal of Man-Machine Studies*, *6*(3), 361–376. https://doi.org/10.1016/S0020-7373(74)80027-1

Zajonc, R. B., & Rajecki, D. W. (1969). Exposure and affect: A field experiemnt. *Psychonomic Science*, *17*(4), 216–217.

Zetterberg, H. L. (1966). *On theory and verification in sociology*.