Evaluating and Improving Domain-Specific Programming Education: A Case Study with Cal Poly Chemistry Courses

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

William Fuchs

June 2022

COMMITTEE MEMBERSHIP

TITLE: Evaluating and Improving Domain-Specific Programming Education: A Case Study with Cal Poly Chemistry Courses

AUTHOR: William Fuchs

DATE SUBMITTED: June 2022

COMMITTEE CHAIR: Dr. Ayaan Kazerouni
Professor of Computer Science

COMMITTEE MEMBER: Dr. Zoë Wood
Professor of Computer Science

COMMITTEE MEMBER: Dr. Ashley McDonald
Professor of Chemistry

ABSTRACT

Evaluating and Improving Domain-Specific Programming Education: A Case Study with Cal Poly Chemistry Courses

William Fuchs

Programming is a key skill in many domains outside computer science. When used judiciously, programming can empower people to accomplish what might be impossible or difficult with traditional methods. Unfortunately, students, especially non-CS majors, frequently have trouble while learning to program. This work reports on the challenges and opportunities faced by Physical Chemistry (PChem) students at Cal Poly, SLO as they learn to program in MATLAB. We assessed the PChem students through a multiple-choice concept inventory, as well as through "think-aloud" interviews. Additionally, we examined the students' perceptions of and attitudes towards programming. We found that PChem students are adept at applying programming to a subset of problems, but their knowledge is fragile; like many intro CS students, they struggle to transfer their knowledge to different contexts and often express misconceptions about programming. However, they differ in that the PChem students are first and foremost Chemistry students, and so struggle to recognize appropriate applications of programming without scaffolding. Further, many students do not perceive themselves as competent general- purpose programmers. These factors combine to discourage students from applying programming to novel problems, even though it may be greatly beneficial to them. We leveraged this data to create a workshop with the goal of helping PChem students recognize their programming knowledge as a tool that they can apply to various contexts. This thesis presents a framework for addressing challenges and providing opportunities in domain-specific CS education.

ACKNOWLEDGMENTS

Thanks to:

- Ayaan Kazerouni, for everything that he taught me, all of the thoughtful edits, and, most of all, inspiring me with his passion for education
- The wonderful CS Professors at Cal Poly for teaching me how to learn
- Sierra Brill, for her patience, kindness, and brilliant ideas
- Kurt & Stacy Fuchs, for encouraging curiosity and listening to all of my ideas
- And so many more wonderful friends for all the good times and support

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

Chapter 1

INTRODUCTION

Programming is a key component of research in domains outside of computer science. For example, the field of Bioinformatics utilizes computer science to drive advances in biology. In order for researchers to succeed in their fields, it is important that they are able to program. Purely computing professionals cannot write the software for them, because they do not have the necessary background knowledge. Similarly, researchers do not need to be trained in the full breadth of computer science because much of the field will be irrelevant to them. There are exceptions to this; some niche fields will require researchers to be experts in computer science as well as their other domain. However, for most researchers, programming is simply a supporting skill that enables them to be more productive. These researchers who program are referred to as *end-user programmers*, and it is estimated that for every professional software engineer there are at least four end-user programmers [24]. End-user programmers are computationally literate - they can express themselves through the medium of computing, which allows them to think about their domain in new, interesting ways [11].

As programming continues to be used in other sciences, more students will learn programming in the context of their domains. These courses will focus on programming to different degrees, and may or may not be taught by CS faculty. In domain-specific programming courses, students are able to directly see the applications of programming to their field and focus on the most relevant programming constructs for their field. This means that they do not have to transfer programming knowledge into their domain, which is widely regarded as a large challenge to students [11]. How-

ever, it is also possible that students will not learn much programming through an integrated curriculum [15]. The Physical Chemistry curriculum at Cal Poly offers an excellent opportunity to observe the challenges and opportunities presented by a domain-specific programming course.

Cal Poly currently teaches MATLAB programming to Chemistry students in the Physical Chemistry (PChem) series of courses [19]. This is a 5-course series that is required for all Chemistry and Biochemistry majors at Cal Poly. The programming component of the series was introduced in 2013 with the intent to deepen students' Chemistry knowledge through programming [19]. Studies have been done on the efficacy of the Cal Poly Chemistry curriculum as a whole, but the programming aspect of the curriculum has not yet been evaluated [19]. This thesis seeks to examine the programming aspect of the PChem curriculum taught at Cal Poly through two research questions:

RQ1. To what extent are PChem students developing their computational thinking ability?

RQ2. Do Chemistry students perceive themselves as effective programmers, and do they believe that programming is a valuable skill for their Chemistry careers?

RQ1 seeks to understand the challenges faced by the students as they learn to program, while RQ2 is aimed at understanding student attitudes towards computing.

Chapter 2

BACKGROUND

In this section, we will cover what it means to be "computationally literate", why knowledge transfer from one domain to another is difficult, how metacognition plays into problem solving, how academic motivation impacts learning outcomes, and how people learn to program.

2.1 Computational Thinking and Literacy

Computing can be thought of as a new medium through which people can express themselves. People who are able to utilize computing are *computationally literate* [11]. As with traditional literacy, there is a gradient to computational literacy: people can be end-users of software, creators of software, or anywhere in between. It is difficult to develop computational literacy, but the benefits are numerous [11]. For example, people who leverage computing are more productive and are able to think about their domains in new, interesting ways. In contrast, people who are not computationally literate will often suffer productivity costs.

Computational thinking is a term coined by Jeannette Wing to describe the thought processes that are involved in solving programming problems [34]. Mark Guzdial describes computational thinking as the practice of "applying computing ideas to facilitate computing work in other disciplines" [11]. As computational thinking involves expressing oneself through computing, it is a form of computational literacy. The ability to program is not the end goal of computational thinking; it is another tool to be utilized by students to explore their domain. Students who are able to think compu-

tationally will see productivity gains and will be able to interact with the material differently than students who have not been taught computational thinking [11].

Defining computational thinking in a way that is useful in practice is difficult. Given that computational thinking has been adopted as a core practice under the Next Generation Science Standards, Weintrop et al. developed a taxonomy as a definition [31]. The taxonomy supports practical lessons that infuse math and science curricula with computational thinking.

While computational thinking can be applied to any domain, this thesis will focus on its application to science. Computational thinking and science have a reciprocal relationship: computing can deepen a student's understanding of science concepts, while science provides an authentic context for the application of computing [31].

2.2 Knowledge Transfer

Applying prior knowledge to novel situations is known as *knowledge transfer*. Sometimes, prior knowledge is applicable and applying it is appropriate. This is known as *positive transfer*. Other times, applying prior knowledge is detrimental to solving the problem. This is known as *negative transfer*. Transfer issues in computer science are well known, but explicitly teaching for transfer can help [11, 25]. Robust, abstract mental models are key for transfer.

Positive transfer is affected by many factors. Among them is learning with understanding rather than learning by memorization [4]. Wertheimer found that students who learn and apply a procedure via rote memorization are unable to transfer their knowledge to similar problems [33]. On the other hand, students who are explicitly taught the reasoning behind the procedure are able to solve the problems. This makes

sense, as understanding involves learning an abstract concept that can then be applied to similar problems. Students who memorize do not learn at the abstract level, and so are unable to identify that the problems are similar. When presented with the similar problems, the memorization students said: "We haven't learned this yet".

Another factor that impacts transfer is the context in which knowledge is acquired. When learning occurs in only one context students struggle to disentangle their knowledge from that context, negatively impacting transfer [10]. In contrast, teaching across multiple contexts with explicit demonstration of a wide range of applications improves knowledge transfer. This is because people learning in multiple contexts are more likely to identify the key components of what they are learning and form abstract mental models that are applicable to a wide array of problems. One way to encourage abstract mental models is to give students multiple, slightly different, specific cases of a concept. As they go through the specific cases they are likely to construct a more abstract mental model of the concept.

Teaching students to use multiple abstract levels can help students develop robust mental models that transfer well [4]. Students who are taught only at a specific level are unlikely to understand the limits of what they have learned. Bransford uses the example of students constructing business plans: students who construct business plans for a complex problem may not realize that their plan works well for "fixed-cost" businesses, but not for others. However, if they are taught to represent their solution at multiple different levels of abstraction, they are likely to realize the broader classes of problems that it can and cannot solve.

Negative transfer can range from outright preventing people from solving a problem to reducing the efficiency of their solution. Luchins and Luchins found that people's prior knowledge can blind them to simpler, more efficient solutions when faced with a new problem [18]. They instructed participants on one method of solving a complex

series of problems, and then gave them a much simpler problem. The control group, who did not receive the instruction on complex problem solving, were able to solve the simple problem with a simple, efficient solution. In contrast, the experimental group applied the complex solution to the simple problem. While both worked, the control group was much more efficient.

2.3 Metacognition and Problem Solving

Developing a program as a solution to a problem is a complicated process that goes far beyond simply writing code. *Metacognition* is an awareness of one's thought processes. Solving complex problems involves high levels of metacognitive awareness; expert programmers explicitly monitor their progress and evaluate the effectiveness of their problem solving strategies [17]. Frameworks designed to teach students how to solve problems can improve metacognition. One of the most popular frameworks is the *design recipe*, described in How to Design Programs [7].

The design recipe is a metacognitive aid that helps students problem-solve with programming. It is a 6-step process, briefly described below.

1. Determine data definitions from the problem statement. What information must be represented? How can it be represented?

2. Define a function stub. What does a function that solves this problem take as inputs and produce as outputs?

3. Work through examples to illustrate the purpose of the function.

4. Create a function template. Incorporate the data definitions into the function

5. Define the function. Leverage the examples to fill out the function template to solve the problem.

6. Test the function. Translate examples into test cases and ensure that the function passes.

As students work through a problem using the design recipe, they build up their solution in small steps. They then iteratively improve on this solution.

Metacognition can also improve knowledge transfer. Students who understand themselves as learners and can reason about their learning process are more likely to transfer knowledge without explicit prompting [4]. Emphasizing metacognition helps students evaluate their progress towards understanding and develop knowledge acquisition skills. Students who realize that they do not understand a concept will seek out knowledge and incorporate it into their mental models until they are able to solve a given problem. Students who do not have these metacognitive skills may negatively transfer inappropriate knowledge because they fail to realize that they do not understand the problem, fail to seek out supplementary knowledge, or fail to properly incorporate new knowledge into their mental models.

2.4 Academic Motivation

Academic motivation is an important factor in student success; high motivation begets higher academic achievement because motivated students are more likely to engage in activities that help them learn [14]. Motivation is affected by many factors, some of which are out of the instructors control (i.e. natural interest in the subject). However, instructors have a large amount of influence over student motivation and they can

design courses to maximize motivation, as detailed by Brett Jones with the MUSIC model of academic motivation [14].

The MUSIC model breaks motivation into 5 components: "eMpowerment", "Usefulness", "Success", "Interest", and "Caring". While it is not clear the minimum number of components that must be satisfied to sufficiently motivate students, it appears that more is better. Instructors have varying degrees of control over each of the 5 components, but they can still make a positive impact in every component.

Authenticity is important to students' perception of "usefulness". Students perceive skills taught in an authentic context to be more broadly useful than skills taught in a contrived, or "toy" context. As an example, students learning to program with block-based programming languages perceive the class as unauthentic, even though they are still learning important programming skills [32]. So, teaching as authentically as possible while monitoring and addressing student perceptions of authenticity can increase student motivation.

2.5 Learning to Program

It is widely known that students experience difficulty when learning to program. This is partly due to students needing to develop a *notional machine* in order to reliably solve even trivial programming questions [11]. A notional machine is an abstract idealization of a computer through which the programmer understands the capabilities of a computer and how to control it [6]. Developing an accurate mental model of a notional machine is considered one of the most challenging components of learning to program [26]. As a result, novice programmers' mental models are frequently incomplete or rife with misconceptions. These misconceptions recur systemically, and so can be classified into meta-misconceptions, which Roy Pea calls "super-bugs" [23].

Pea proposes three language-independent super-bugs: the "hidden mind", "goal-plan merge", and metacognitive bugs. Pea also proposes two language-dependent super-bugs: knowledge unavailability and knowledge inaccessibility.

The *hidden mind* is the idea that a computer is an intelligent agent that collaborates with a programmer. Novices conceive of programming as a conversation with a computer, which, although pedagogically helpful, leads them to overgeneralize and believe that the computer can "understand" implied intent as a human would. Modern programming languages further this misconception by hiding certain aspects of programming, such as memory management. It is difficult for novices to construct an accurate notional machine when they do not understand what parts of a program must be explicitly defined and which can be left implicit. This causes them to leave out key parts of their programs that should be "obvious" to a human interpreter, such as else statements or variable declarations [23].

The *goal-plan merge* class of bugs arise when students understand the components of a program, but can not connect them together to achieve a goal. To solve a problem, programmers must identify and address the various sub-goals contained in the problem. These sub-goals often appear similar but contain important, subtle differences. In merging their solutions to the sub-problems, novices are prone to drop out at least one sub-goal, thereby producing a buggy solution that does not reliably solve the original problem [23].

Metacognitive bugs occur when students do not accurately monitor their cognitive process as they are solving a problem [23]. These bugs occur during program reading, writing, and debugging. Students struggle to act as a computer would while tracing a program, often forgetting or redoing commands. This can be attributed to poor bookkeeping: few students use metacognitive aids such as writing out what their notional machine is doing at each step. As discussed in Section 2.3, careful reflection

on one's knowledge and programming skills and process could be key to developing programming expertise.

The *knowledge unavailability* bugs arise when students simply do not have adequate knowledge to solve a problem [23]. This can be as clear as not knowing the syntax for a conditional in a particular language, or as abstract as misunderstanding how control flows in a program. Many flaws in student's notional machines are due to knowledge unavailability. *Knowledge inaccessibility* bugs, on the other hand, occur when students possess the correct knowledge, but are unable to apply it [23]. Students struggle to transfer their programming knowledge from one context to another when the contexts have surface-level differences, despite underlying deep similarities [1, 29]. Students may grasp a programming construct in a given context, but do not see how it applies to a different context. This super-bug occurs frequently in science courses that emphasize computational thinking: students do not always learn as much general purpose programming as expected [11]. As discussed in Section 2.2, this is a problem of knowledge transfer — students know how to program, but that knowledge is tightly coupled to the context in which they learned to program, so they struggle to apply it elsewhere. These students may pass a test, but be unable to write a successful program.

2.6 Summary

Computational thinking can improve productivity and success in domains outside of computer science. However, transferring knowledge from one domain to another is difficult. There are teaching strategies that have been shown to facilitate knowledge transfer. Writing code is only part of the programming process. Metacognitive self-regulation of ones problem-solving process is necessary to consistently develop

effective solutions. Motivation is key to student success; students who are more academically motivated in a given course are likely to perform better than students who are less motivated.

Domain-specific computer science courses teach programming in the context of the target domain. This has the potential to reduce the necessity for knowledge transfer and improve student motivation to learn programming by providing an authentic context for its application. In the next chapter, we discuss how we measured Cal Poly PChem students' programming abilities.

Chapter 3

UNDERSTANDING CHEMISTRY STUDENTS' PROGRAMMING ABILITIES

In this chapter, we cover how we systematically classified the programming knowledge of the PChem students. We first cover our initial quantitative approach, then our interview process, and finally discuss our findings.

3.1 Measuring Knowledge of Programming

While there are many ways to assess student knowledge, multiple-choice tests are frequently used due to their ease of administration and high interpretability. *Concept inventories* are carefully developed tests that evaluate student understanding of a set of concepts [12]. If a concept inventory is *validated*, then it has been shown to accurately test the concepts that it claims to be testing. Validated concept inventories have been applied to many fields, including computer science education research. The Second CS1 Assessment (SCS1) is a validated, multiple-choice concept inventory that tests introductory computer science knowledge [21]. After examining the PChem curriculum, we determined that an introductory-level assessment would be appropriate for the PChem students. The MATLAB Computer Science 1 Assessment (MCS1) is an assessment created by researchers at the Ohio State University for first year engineering students [2]. The MCS1 is isomorphic to the SCS1, which means that it tests the same concepts and learning outcomes as the SCS1. As the MCS1 uses MATLAB, it is better suited for our purposes. We elected to use the MCS1 to evaluate PChem student's MATLAB knowledge.

While the MCS1 assesses students' *knowledge*, it is also important to understand student's *attitudes*, which we measure with the Attitudes Towards Computing (ATC) survey [30]. Attitudes are tightly tied to academic motivation, which is a key factor in student success. The MUSIC model of academic motivation lists "empowerment", "usefulness", "success", "interest" and "caring" as components of motivation [14]. The ATC directly measures "usefulness" and "interest", as well as motivation as a whole. Intuitively, students who do not perceive programming as useful or interesting are less likely to fully engage with course material. On the other hand, students with positive attitudes towards computing are more likely to persist in the field and demonstrate more advanced computing knowledge. Either way, student attitudes are an important factor in their performance.

Performance on the MCS1 could be impacted by confounding variables such as prior programming experience. To account for this, we created a set of demographic questions to measure extracurricular programming experience. We also collected data on academic progress, major, gender, ethnicity. The question about languages used in prior programming experience had a free-response "other" option that some students used to provide unsolicited feedback on the assessment.

We administered the MCS1, ATC, and demographic questions as a combined survey on Canvas, the learning management system that Cal Poly uses. We created the survey by first presenting the MCS1, then the ATC, and finally the demographic questions. With the cooperation of the Chem faculty at Cal Poly, we were able to require that all students in PChem 2 and PChem 3 complete our survey (n=68). While students were required to complete the survey, their performance on the MCS1 was not used in their course grade.

In general, students performed poorly on the MCS1. There are a variety of possible explanations for this. It could be that the test was long and did not count for a grade,

so students may have sacrificed accuracy for speed. Or, it could be that the test was context-agnostic, so the students may have struggled to transfer their Chemistry-specific programming knowledge. It could have been that students did not recognize the syntax for certain expressions, because it is different than the syntax that they learned in class. It is likely a combination of these factors and others. Regardless, student responses in the free-response section of the survey led us to believe that most students were guessing on many of the questions. One student said:

> *I feel like I am actually very good when it comes to MATLAB, but only if I have been taught the material or spend a lot of time on Mathworks teaching myself. I know I did not do well on this assessment, but that is just because I am not a super genius when it comes to code. If I had been taught all of the stuff on this exam (aka if I took another CS class down the line) I think I would excel at it*

This quote exemplifies the general student attitude after taking the MCS1; they felt that it was very difficult and not related to what they have learned in class, which caused them give up on problems and guess. This made the data noisy, which made it difficult to discern students' abilities from this test.

Upon further review of the MCS1 and consultation with the Chemistry faculty, we found that many of the MCS1 questions were not accessible to the PChem students because they used constructs that were not covered in class. This is congruent with prior research that shows that the SCS1 is a difficult exam, and so is not suitable as an introductory-level CS pre-test [22]. As the programming portion of the PChem curriculum amounts to an introductory CS course, giving the exam to any students in PChem 1 or 2 is a de facto pre-test. In light of this, we decided that a shorter, more accessible test would give us better results. We elected to give this modified MCS1

as a think-aloud interview instead of a multiple-choice exam because the interview format allows us to gain richer insights into the student's programming abilities. The interview process is described in the next section.

3.2 Interview Process

Our goal with interviews was to develop a theory about the extent of PChem students' computational thinking abilities, as well as their attitudes towards programming. We structured the interviews as a selection of seven MCS1 problems followed by three open-ended questions to gauge student's attitudes towards computer science. In order to collect rich data about students' problem solving process, we asked them to think aloud as they worked and explain their rationale for each answer. The interviews were given in-person or over Zoom, and the audio was recorded and transcribed.

The MCS1 questions were selected to test each PChem learning objective multiple times. The PChem learning objectives were directly taken from the paper describing the creation of the PChem curriculum [19]. We then coded each MCS1 question with the PChem learning objectives involved in solving it. Each researcher independently coded the questions, and we then discussed until we arrived at a consensus. The learning objectives, keys, and frequency with which the objective was tested in the MCS1 can be found in Table 3.1. There were some PChem learning objectives that are never tested in the MCS1, such as "Symbolically solve and integrate using MATLAB". This makes sense because the MCS1 was designed to test general introductory CS knowledge, while the PChem curriculum was designed to teach physical chemistry with the help of programming. While there are many concepts shared between the two, they do not overlap completely. We cannot test learning objectives that are not included in the MCS1, and we should not include MCS1 questions that don't

test learning objectives. So, to create an effective assessment we selected a subset of MCS1 problems that test each expressed learning objective multiple times.

Table 3.1: PChem learning objectives and the frequency with which they appear in the MCS1

Learning Objective	Frequency
Perform symbolic math (multiplication, addition, subtraction, division, exponentiation, and differentiation) using MATLAB.	4
Write scripts to execute a sequence of commands.	3
Symbolically solve and integrate using MATLAB.	0
Assign values to arrays, address elements in an array, perform element-by- element multiplication, addition, subtraction, division, and exponentiation on arrays; use arrays to plot functions.	1
Perform numerical integration using MATLAB.	0
Write and use custom functions.	4
Solve single-variable differential equations numerically and plot the resulting functions.	0
Solve multivariable differential equations numerically and plot the resulting functions.	0
Create and diagonalize a matrix in MATLAB.	0
Write and execute simple code, including FOR loops, nested loops, and conditionals.	2

We adapted the MCS1 questions to ensure that students would be able to solve each question using only what they were taught in the PChem series. For example, the MCS1 uses string operations in several problems, but PChem students do not learn to work with strings. So, we replaced the string operations with arithmetic operations, while preserving the other concepts that the question was designed to test. We worked with the PChem faculty to ensure that all of the interview questions were solvable by the PChem students.

Next, we adapted the ATC questions to better fit the interview format. Multiple-choice questions are ideal for processing on a large scale, but we wanted richer data from our interviews. So, we created three open-ended questions, adapted from the ATC. The questions were:

1. How do you feel about MATLAB?

2. Do you use MATLAB if it's not specifically asked for in a question or problem?

3. How do you feel about your computing abilities?

As we began interviewing students, it became clear that there were subtleties in their programming knowledge that were not being captured by the initial set of interview questions. To maximize the power of the interviews, we decided to follow a *grounded theory* approach. In grounded theory, a theory is iteratively built up over multiple rounds of data collection [27]. There are multiple interpretations of grounded theory. We are using the Straussian version of grounded theory which allows for engaging with the literature as the theory is constructed. We modified our theory after each round of interviews, and used the subsequent round to validate our modifications. The classes of changes that we made to the interview questions are described below.

Brute Forcing Answers We observed students brute-forcing problems by checking and discarding answers. For some problems this is acceptable, as verifying an answer requires the skill that the question is designed to test. However, other problems were designed to test deeper computational thinking abilities; providing answers as scaffolding made it difficult to determine if students were demonstrating true computational thinking ability or simply good test taking strategies. To solve this, we changed some of the multiple choice problems to short answer problems. In particular, problem 5 requires students to construct an ordering of statements from a list to manipulate a variable into a particular value. As a multiple choice problem, this tests a student's ability to trace a series of variable mutation statements. However, as a short answer problem the solution space is too large for students to brute force. They must understand the relationships between different statements, weed out the unnecessary

statements, and construct a plan to achieve their goal. All of these are complex computational thinking skills. Removing the answer choices allowed us to more clearly understand student's computational thinking abilities.

Stumbling Blocks As trained computer scientists, it was challenging for us to anticipate what aspects of a problem students would have difficulty with. Ideally, students would only struggle with the computing constructs that a problem is designed to test. When students stumble over unrelated aspects we cannot discern with certainty the extent of their knowledge in the target construct. For example, problem 3 tests a student's knowledge of functions and control flow through a function that squares all of the odd numbers in an array. In the first version, we used the modulo operator (denoted by %) to determine if a number is even or odd, called its *parity*. The modulo operator produces the remainder after integer division. So, **x % 2 == 0** is **True** when x is even and **False** when x is odd. This is a frequent pattern for determining parity in computer science. However, since PChem students do not learn the modulo operator, they would be unlikely to solve this problem, even if they understood functions and control flow completely. Since modulo is not a core concept that we are testing, we modified the problem to test for parity by comparing integer division by two and division by two. The students were generally able to trace this line, which allowed us to gain insight into the target constructs of functions and control flow.

Tracing vs. Explaining In some problems, we are trying to test students' ability to correctly *trace* a program. Tracing a program involves stepping through it as a computer would, keeping track of state in order to determine what will be output. In this case, we are focused on evaluating how closely their notional machines approximate a computer at a low level, because tracing a program

requires an accurate notional machine. Other times, we are trying to evaluate how they are *understanding* the program at an abstract level: what patterns do they recognize, are they identifying the purpose of a block of code, etc. The skills required to understand a program are a superset of the skills required to trace the same program. This means that students who can trace a program and students who understand the program will both be able to solve many of the same problems. In order to differentiate between these two ability levels, we changed problem 3 from a multiple choice problem to an "explain" problem. Students were asked to "explain in plain English" what would be output, rather than explicitly asked to solve the problem. Some students mechanically traced the entire problem while others analyzed it and explained the purpose of each section. Most students fell somewhere in the middle, which gave us rich data about their understanding of several different constructs.

In order to systematically process the data from the interviews, we needed a *learning taxonomy*. A learning taxonomy is a tool used to classify different levels of cognition. We used the *Matrix Taxonomy*, a two dimensional adaptation of Bloom's learning taxonomy [8]. The key insight that led to the creation of the Matrix taxonomy is that the ability to comprehend a program and the ability to produce a program are semi-independent skills. In the Matrix taxonomy, levels of program production are on the Y-axis, while levels of program interpretation are on the X-axis. This allows us to visualize where students fall on the interpretation production plane. Figure 3.2 depicts the Matrix Taxonomy with various key programming activities placed on it.

To capture student's abilities across different constructs, we created a multi-strand taxonomy, where each strand captures ability in an independent area. Learning to program effectively involves understanding multiple different computing constructs. Often, these constructs are not directly related, and mastery of one does not imply

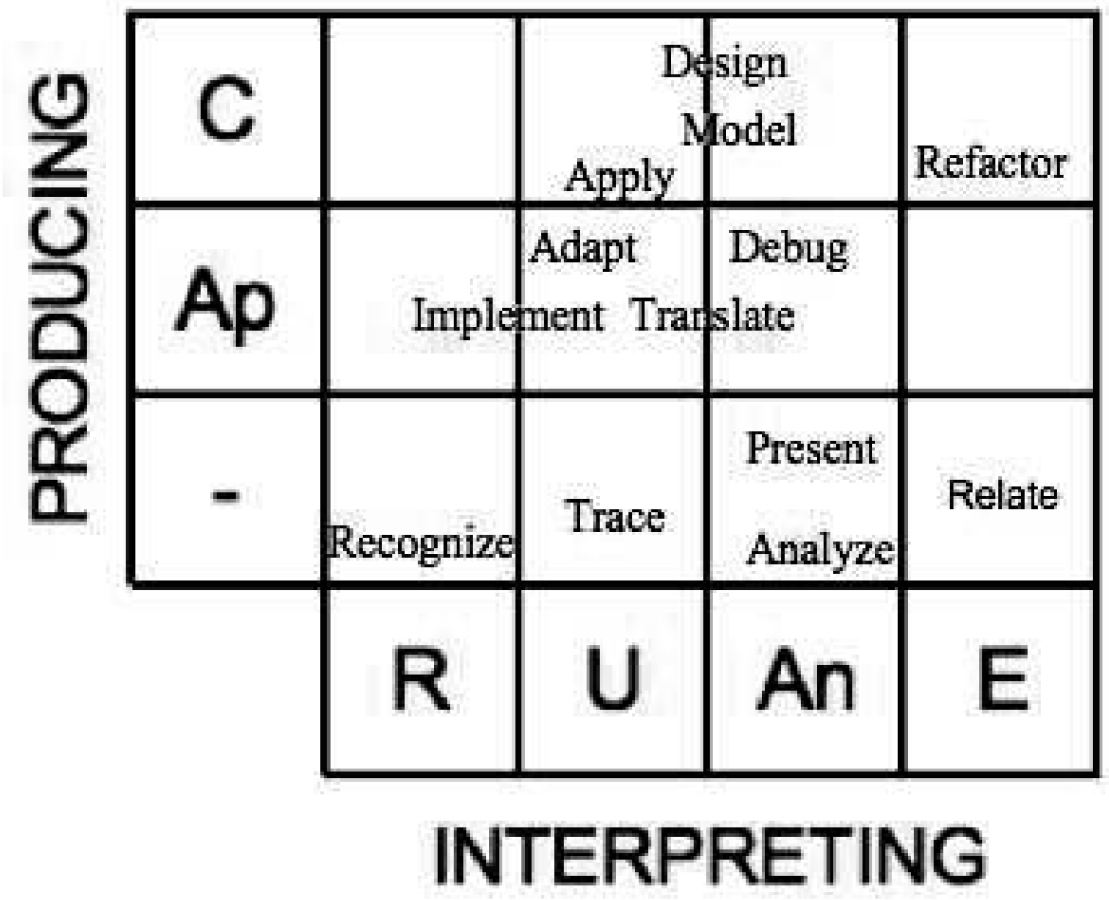


Figure 3.1: The Matrix Taxonomy, reproduced from Fuller et. al [8]
C - Creating, Ap - Applying, R - Recognizing, U - Understanding, An - Analyzing, E - Explaining

mastery of any others. Student's mastery of each construct can be placed on its own taxonomy. Castro and Fisler applied this to programming students using the SOLO taxonomy, a different learning taxonomy [5]. We applied the same idea, but with the Matrix taxonomy instead. We settled on three strands: Control Flow, Variables & Arrays, and Functions. *Control Flow* covers statement ordering, conditionals, and loops. *Variables & Arrays* cover variable assignment and mutation, as well as all array operations. We chose to combine variables and arrays since all MATLAB variables

are actually arrays. *Functions* covers creating custom functions, as well as calling functions and scope of the variables and parameters local to the function.

Most questions only test one or two strands. Some questions require students to produce a program, while others only required them to interpret a program. So, the highest that a student could score on each axis for a given question was capped by that question. For example, it is impossible to demonstrate the ability to produce a program on question 3, since it asks only for an explanation of the provided program.

After each interview, we placed the student on each strand of our taxonomy. In grounded theory, these documents are referred to as *memos*, and are used to help construct a theory. We constructed memos by observing the level of ability expressed in each strand for each question. We found quotes from the student that illustrated their ability level for every construct tested in that problem and placed these quotes on the taxonomy appropriately. Misconceptions were placed in the bottom left box, which is outside of the producing-interpreting plane. The first three interviews were coded by one researcher. We then discussed the coding until we arrived at a consensus. After the first three interviews, both researchers had a shared understanding of the meaning of each level in the Matrix taxonomy, so only one person coded the remaining interviews.

Once we finished coding an interview, we observed where the student's abilities were clustered. Usually, students consistently demonstrated the same ability level, although some students demonstrated significantly different ability for the same construct in different problems. This phenomenon is further discussed in the following section.

Listing 3.1: Named statements similar to those used in Problem 5

```
1 alpha: a = b / z;
2 bravo: b = c + x;
3 charlie: y = y * b;
4 delta: x = c * a;
5 foxtrot: y = z - y;
```

3.3 Interview Results

A tabular summary of the students' programming abilities, determined qualitatively from the interviews, can be seen in Table 3.2 for the Control Flow strand, Table 3.3 for the Variables & Arrays strand, and Table 3.4 for the Functions strand.

In Control Flow, Table 3.2, most students were able to reach the Understand/Apply level. Students at this level are able to *understand* MATLAB control flow and *apply* it to achieve their goals. For example, Problem 5 asks students to manipulate a variable using only predefined statements like those in Listing 3.1. Students at the Understand/Apply level *understand* that lines of code are executed sequentially and *apply* that to construct and test different orderings. However, they often exhibit a "brute force" style approach by testing seemingly random orderings until they find one that works. In contrast, students at the Analyze/Apply level *analyze* the provided statements to determine general rules that they can then use to construct an ordering. While solving problem 5 and thinking aloud, one student said:

> *So the thing that I'm looking for first is which one will actually affect A, because that's the one that I'm trying to change, and then I just need to set the variables that are in the one that affects A to be the the correct form.* -P7

Which demonstrates the Analyze/Apply level.

Similarly, most students in Variables & Arrays, Table 3.3, were able to reach the Understand/Apply level. Students at this level *understand* how variables work, which they can then *apply* to their programming. Importantly, students at the Understand level do not express misconceptions about variables. In contrast, students at the Remember/Apply level can *remember* facts about variables and *apply* them to code, but they often express misconceptions. Students at the Remember/None level can recall facts about variables, but they are unable to apply them to programming problems.

Functions, Table 3.4, proved to be difficult for students. Students also expressed the most varied levels in this strand. A student at the Remember/Apply level could be expected to *remember* some facts about functions and *apply* them to programming. However, they often express important misconceptions that can sideline their attempts to apply their knowledge. Students at this level use a trial-and-error approach, which can lead to frustration as they encounter errors [8]. Problem 3 asked students to explain a function. A student at the Understand/Apply level would correctly explain the function line by line. While technically correct, this explanation shows that they are considering the function as a collection of lines of code, not as a single entity that accomplishes a task. On the other hand, students at the Analyze/Apply level were able to read the function and describe it in terms of its inputs, outputs, and purpose. Going further, a student at the Evaluate/Create level could *evaluate* the quality of a function and *create* alternative solutions. This could look like recognizing that a function should be decomposed into smaller functions, and performing the decomposition.

In the following subsections, we discuss the interview results for the programming questions, as well as the attitude questions.

Table 3.2: Number of students in each cell on the Control Flow strand of the Matrix Taxonomy

Control Flow				
Create				
Apply	1	5	2	
None				
	Remember	Understand	Analyze	Evaluate

Table 3.3: Number of students in each cell on the Variables & Arrays strand of the Matrix Taxonomy

Variables & Arrays				
Create				
Apply	1	6		
None	1			
	Remember	Understand	Analyze	Evaluate

Table 3.4: Number of students in each cell on the Functions strand of the Matrix Taxonomy

Functions				
Create				
Apply	2	1	1	
None	2			
2	Remember	Understand	Analyze	Evaluate

Note that two students only expressed misconceptions, and so are in the bottom left corner of the taxonomy.

3.3.1 Recurring Patterns

While processing the interviews, we noticed recurring patterns of thought across the groups of students. In this section, we detail the different patterns that we have found.

Belief that "if-and" is a programming construct Many students (P2, P3, P4, P5, P6, P8) could not correctly evaluate an expression like **and(true, and(true, false))**, but had no problem evaluating a similar expression inside of an **if** statement. When asked to solve problem 1, a Boolean logic problem, one student said:

> *We've never gone over a problem like this, or the logic behind this... I would really just be guessing on this.* -P5

Then, while solving problem 4, they correctly traced an **and** inside of an **if** statement:

> *A is greater than B and B is greater than C, so that's not true because 3 is not greater than 7.* -P5

Furthermore, only one student (P1) recognized the **and()** syntax in problem 1 after seeing Boolean logic used within an **if** statement. This furthers the idea that students are viewing "if-and" as its own MATLAB construct. Students who hold this misconception would likely place in the Understand/Apply cell of the Control Flow strand, Table 3.2. They can write an "if-and" statement like option A in Listing 3.2, but they could not analyze alternatives, such as option B.

Unable to recognize "matrix append" When presented with code like Listing 3.3, only one student (P7) was able to recognize that it would create a matrix with

Listing 3.2: Two ways to write an "if-and"

```
1  % Option A
2  if (and(x, y))
3      disp("x and y")
4  end
5
6  % Option B
7  c = and(x, y)
8  if (c)
9      disp("x and y")
10 end
```

Listing 3.3: One way to append values to a matrix in MATLAB

```
1  x = [''];
2  x = [x 'B']
3  x = [x 'A']
```

the values **['B' 'A']** by appending them to the empty matrix. This works by assigning x the value of *a new matrix*, created from the old values of x and an additional value.

The other students expressed some familiarity with the matrix creation syntax, but they could not figure out that values were being appended to the matrix. One student said:

> *I don't think that is true because if you want to write in matrix format yeah, it's been in you have to do like X, say the first one, then the second one.* -P1

This student clearly recognizes the matrix syntax, but they think that the matrix could only be created with a matrix literal like **x = ['B' 'A']**. Although appending to a matrix uses the same matrix creation operation as initializing the matrix with static values, students were unable to identify the matrix creation operation in the append code.

Listing 3.4: A short function that does not return anything

```
1 function [ res ] = compute(x, y)
2     x = x - y;
3     y = x / y;
4 end
5 a = 9;
6 b = 6;
7 compute(a, b)
```

Students who hold this misconception would land in the Understand/Apply cell in the Variables & Arrays strand of the taxonomy. They understand how to work with arrays in one way, but they do not have a robust or abstract enough mental model to analyze alternative solutions, such as appending with the method used in Listing 3.3.

Negative transfer of math knowledge to MATLAB Some students appeared to inappropriately apply their prior math knowledge to programming. For example, consider the following snippet:

When given code like Listing 3.4, one student said that they could not solve it because:

> *calculating X & Y based off of each other I would think that we would need a value to start with* - P6

They came to this conclusion after trying to solve for **x** (line 2) and **y** (line 3) by substituting, as they would solve a system of equations. However, **x** and **y** were previously defined as parameters to the function, so this problem is solvable by substituting in the arguments **a** and **b** (line 7).

Other students appeared to think of functions in mathematical terms, such as this student when presented with a function similar to Listing 3.5 to trace:

Listing 3.5: A complex function involving conditionals, prints, and looping

```
1  function [ ret ] = square_evens(numbers)
2      ret = zeros(1, length(numbers));
3      for i=1:length(numbers)
4          val = numbers(i);
5          if(floor(val / 2) == (val / 2))
6              val = power(val , 2);
7              disp(val);
8          end
9          ret(i) = val;
10     end
11 end
12
13 in_numbers = [1, 2, 4, 3, 5];
14 output = square_evens(in_numbers);
```

> [Referring to line 14 in Listing 3.5] *That would generate a function of those numbers that would do some math on those numbers to create some value*
>
> -P8

The **square_evens** function above returns a list with all of the even numbers squared. However, the student felt that it should return only a single value, as a mathematical function would. This negative transfer prevented them from solving the problem.

While mathematical principles overlap significantly with programming principles, some students' appear to believe that MATLAB follows mathematical principles at inappropriate times.

Difficulty with Functions As can be seen in Table 3.4, Functions were more difficult for students than either of the other two strands. Five students (P3, P4, P5, P6, P8) expressed misconceptions about functions. Of them, two students (P4, P8) exclusively expressed misconceptions about functions. While the misconceptions were varied, they all fall under the umbrella of "Functions". The different misconceptions are described below.

Some students (P4, P8) appeared to have a general lack of knowledge about functions. They exclusively expressed misconceptions. Referring to line 2 of Listing 3.5, one student said:

> *And then numbers are also a set of the zeros.* -P8

It is likely that there are several misconceptions at play in this response. This student believes that the **zeros** function is going to modify the **numbers** variable. This implies that they do not recognize that **length(numbers)** is itself a function call that returns a value. They also do not understand that MATLAB is "pass-by-value", so a function cannot modify its parameters. So, **zeros** could never modify **numbers**, even if it were passed **numbers** directly. The rest of this student's explanation of the **square_evens** function (Listing 3.5) continued similarly. Overall, it appeared that they were trying to construct a mental model of how functions work on the fly.

Another student, when presented with a function like Listing 3.4, said:

> *I feel like I haven't seen this before* -P4

They were referring to the function declaration syntax. While it is possible that they have not been exposed to this particular way of defining a function, they have been taught to "Write and Use Custom Functions" during PChem [19]. Neither of these students were able to explain or solve any of the function problems.

Some students held more nuanced misconceptions about arguments, parameters, and scope. Referencing code similar to lines 2 and 3 of Listing 3.4, one student said

> *calculating X & Y based off of each other I would think that we would need a value to start with* - P6

Listing 3.6: An example problem that demonstrates scope

```
1 function [solution] = calculate(a, b, c)
2     solution = a * c + 0.5 * b * c * c;
3 end
4
5 x = 4;
6 y = 16;
7 z = 12;
8 solution = calculate(x, y, z);
9 disp(b);
```

While we previously used this example to demonstrate negative transfer of math knowledge, it also shows a lack of understanding about how parameters work inside functions. Since **x** and **y** are defined as parameters, they must have a value in the function body. As the student does not recognize this, they may not understand the concept of parameters.

Other students correctly mapped arguments to parameters, but did not understand the scope of the parameters. When solving a problem similar to Listing 3.6, one student said

> *A would be like X and then B would be Y and C would be Z so B would probably be like 16* -P5

This demonstrates that they correctly mapped the arguments to parameters, but they incorrectly believed that the parameters to **calculate** are still defined on line 9, outside of the function body.

Only P7, who had several prior programming classes, could explain that line 9 would produce an error because **b** is undefined outside the scope of the function body.

3.3.2 Attitudes Towards Programming

Patterns emerged in the students' attitudes towards programming as well. Overall, most students viewed their programming skills as specific and limited. One student (P7) was pursuing a CS minor and displayed higher self-efficacy. However, unless otherwise mentioned, the following observations hold true for the rest of the sample.

Students are confident and capable in the subset of MATLAB skills that they view as helpful PChem students often treat programming as an advanced graphing calculator, using it to solve difficult equations and create high-quality plots. Prior to PChem, they would do this with a calculator and a spreadsheet program such as Excel. For many students, MATLAB is an easier option, so they use it frequently and are confident in their abilities. All students except P7 and P8 expressed this sentiment. A quote from one student sums up their collective attitude nicely:

> *So I literally threw my calculator away and start doing everything on MATLAB.* -P1

Students have low self-efficacy for general purpose programming Although students are confident in the subset of skills that they regularly use in PChem, they do not believe that they are generally strong programmers. When asked "How do you feel about your computing abilities?", one student responded simply:

> *I feel like they're pretty weak* -P3

Another said:

> *Within the limits of chemistry, I think I feel like very good about my MATLAB skills... But if you gave me something like outside of that I wouldn't really know what I'm doing* -P5

Students have learned a large amount of programming through the PChem course, but seem to believe that it only applies to PChem. In reality, they have learned general-purpose MATLAB, in the *context* of PChem. That being said, transfer problems are very common in CS education, and current research holds that transfer must be taught explicitly [11].

Students do not feel that they understand the fundamentals of MATLAB This makes them feel lost as they learn new constructs, because they do not know where to fit them into their mental model of programming. Students repeatedly expressed the sentiment that they feel they are missing key components of MATLAB, despite their ability to solve PChem problems programmatically. One student said:

> *I think with chemistry they do a really good job of teaching us. But there's no like okay we're only going to be doing like MATLAB and like learning the very basics of the basics.* - P5

Students do not always perceive programming as a useful skill Although they acknowledge that programming is a large, and growing, part of research in many domains, they do not know if programming will be directly useful to them in the future. One student said:

> *I should probably understand everything computationally little bit more because that's the way that the research is moving and I think that's you know, an important part of the research future. So I want to develop these*

skills more but I'm like I don't know what specific applications it really looks like. -P3

While another student appeared to view the PChem programming curriculum as non-authentic because they are using MATLAB instead of Python:

we're not learning Python or anything which I know is what's generally used on industry and a lot more out in the field -P8

It is possible, perhaps even likely, that students are mistaken in their perception of the authenticity of the course. Authenticity falls under "usefulness" in the MUSIC model of academic motivation [14]. Students have a vague sense that programming is important, but they do not believe that the programming that *they* are doing is useful or authentic. This results in lower student motivation, which can lead to worse learning outcomes.

3.4 Discussion

From the interviews, we found that students are comfortable using MATLAB as essentially a graphing calculator: they can create plots, solve differential equations, etc. However, many students lack an understanding of programming at an abstract level. This means that they can modify examples to achieve their goals, but when the examples do not directly apply to the problem at hand they are at a loss. This shows up in all strands of the taxonomy as the Understand/Apply or Remember/Apply cells, because students are able to apply what they know, but they do not possess deep enough knowledge to analyze or evaluate alternatives. In this section, we discuss our theory about why the students are at this level.

While students often demonstrated the same level of ability consistently throughout the interview, some students were more sporadic. Occasionally, students demonstrated an understanding of a construct during one question, but demonstrated a misconception about the same construct in a different question. One example of this is the "if-and" misconception discussed in Section 3.3.1. As several students demonstrated the same phenomenon, we theorize that this is because students learned Boolean logic in the context of conditionals, and do not understand the abstract concept of *expressions*. This is supported by the students' struggles with the "matrix append" operation, as in Listing 3.3. Even if students had never seen this specific pattern before, they should be able to reason about it if they understand abstract constructs such as expressions. After all, "matrix append" is a specific combination of expressions and operators, but those expressions and operators follow the same rules as anywhere else in MATLAB. It appears that PChem students do not have a good mental model of the abstract rules of programming, which manifests as misconceptions.

It is possible that many of the students' misconceptions are a result of them trying to construct a mental model of programming without understanding the fundamentals. In the PChem curriculum, students learn primarily through practical, context-specific examples [19]. So, they construct their mental models from a small number of examples, rather than from an abstract understanding of the components of a program. While this empowers students to begin programming quickly, it likely contributes to student's knowledge transfer issues, as their programming knowledge is locked to the context of the examples through which they learned. This can lead to shallow, internally inconsistent mental models, which are exposed by misconceptions like the "if-and" and "matrix append" patterns discussed previously. Additionally, as stated in Section 3.3.2, students feel that they have gaps in their MATLAB knowledge. It is possible that additional instruction on the fundamentals of programming languages

would help students construct a robust mental model of programming and increase their confidence. This could improve their ability to transfer their knowledge from one context to another.

Supporting the idea that students have not learned to program at an abstract level, we found that students are able to experience some benefits of MATLAB with only a shallow understanding of programming. For instance, once students become comfortable using MATLAB to solve differential equations, they prefer to use MATLAB over solving the problem by hand. As mentioned in Section 3.3.2, one student went so far as to stop using their calculator completely.

Although students are successful at using MATLAB as a calculator, they seem to rely on copying and modifying examples. When asked how they feel about MATLAB, one student said:

> *I want to say it's from my perspective at least, as a calculator, so to why calculating huge derivative or taking huge you know differentials, right? So it's a lot of copy and paste code.* -P1

This implies that students are frequently referencing examples to solve simple problems. In this case, a shallow understanding allows the students to be successful, so they have not developed a deeper understanding. This shallow understanding constitutes fragile knowledge, which contributes to the difficulty students have transferring their programming skills to different domains. It is possible that explicitly drawing connections between seemingly disparate examples would encourage students to create a more abstract mental model of programming. This could deepen their knowledge and improve their programming abilities.

While students possess practical knowledge of some programming constructs, they struggle with other constructs. Examples of this are students' collective difficulty with functions and loops. The "Functions" strand, Table 3.4, showcases this well. There are a variety of potential reasons why students could be failing to learn these constructs; one explanation is a perceived lack of utility. Functions only show their full utility when programs become complex and code re-use becomes a necessity. Likewise, looping is only necessary when a block of code must execute many times. If a block must only repeat a few times, students may simply repeat the necessary code, rather than learning a new looping construct. If the programming required in PChem does not showcase the utility of these constructs, students are liable to consider them superfluous. Since "usefulness" is a key component of academic motivation [14], a perceived lack of utility could cause students to lose motivation, which leads to poorer learning outcomes. It is possible that additional demonstrations of the utility of each construct could better motivate students and improve learning outcomes.

While our interviews rarely allowed students the opportunity to demonstrate true "Create" abilities, we believe that students struggle to create programming solutions for novel problems. For example, Problem 5 requires students to construct an ordering of statements to manipulate a variable to hold a specific value. This problem can be solved by brute forcing all of the available orderings, or by creating an ordering. Only three students decided to create their own ordering. We believe that this is because students struggle to translate their understanding of a problem to a MATLAB program. This is a daunting, multi-step process, so it may be easier for students to brute force orderings instead of developing their own ordering. We believe that explicit instruction in problem decomposition and problem solving would improve student's ability to solve novel problems.

Compounding this issue, we found that most PChem students do not perceive themselves as general-purpose programmers. When asked "How do you feel about your computing abilities?", almost all students expressed a lack of confidence in their abilities outside of the PChem context. One student said:

> *I don't think I'd be able to take these skills outside of class and use them elsewhere*

Another student responded:

> *I know how to do explicitly what I've learned, and not much else.*

Students believe that they have only learned "PChem programming", and so are unlikely to use programming to solve problems outside of the context of the PChem series. This perception is limiting.

3.4.1 Our Theory

Our theory is two pronged: first, the PChem students do not have a robust mental model of programming, and second, they do not have the tools to solve complex problems with programming. The first prong causes them to struggle and become frustrated as they write programs, which erodes their confidence and makes programming a tedious task. The second prong causes them to not even know where to begin when faced with a complicated problem that they can not easily modify an example to solve. All of this combines to discourage PChem students from applying programming to novel problems. We believe that a workshop designed specifically for the PChem students could improve their programming experience and, ideally, help

them see programming as a powerful tool that they can apply to a diverse array of problems.

As an example, consider two students, A and B. Student A has an abstract mental model of programming, much like the one in Figure 3.2. This model shows how lines of code could be mapped to abstract constructs like "Expressions", "Operators", and "Statements". Since this model is so abstract, it encompasses much of the MATLAB syntax with only three constructs. Whenever student A encounters a new line of code, they can understand it in terms of their mental model, *without updating their mental model*. This is shown in Figure 3.2 with the line **d = and(b,c)**. Student B, on the other hand, has a much more specific mental model, like the one visualized in Figure 3.3. Since each of their "abstract" constructs are more specific, they are not re-used as easily. Even though they recognize variable assignment and understand **and(b, c)** in the context of an if statement, they can't fully understand **d = and(b, c)** with their mental model because they don't have an appropriate abstract construct for **and(b, c)**. Compared to student A, student B will frequently encounter lines of code that do not fit into their mental model. This forces them to update their mental model to understand the new line, which can be frustrating and difficult.

When student A sees a line of code like **if and(a, b)** they understand it as an **if** *statement* that contains an **and** *expression* as the conditional. Since they know that an **if** statement only executes if the conditional evaluates to true, they know that it will not run unless both **a** and **b** are true. When student B sees the same line of code, they view it as an "if-and" statement. While they arrive at the same conclusion that it will only run if both **a** and **b** are true, it is not a robust mental model. This model will break down when student B needs to do something more complex, like construct a more complex conditional such as **if and(b, or(c, d))**.

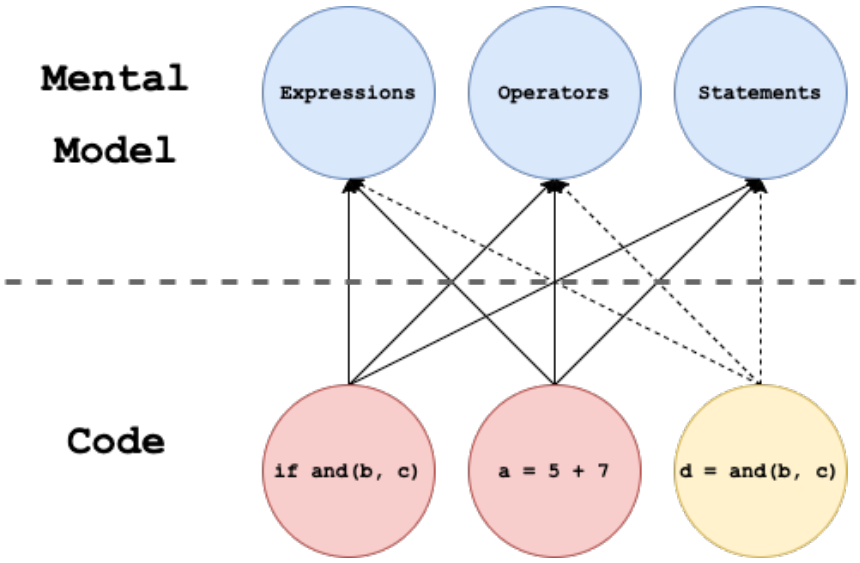


Figure 3.2: Example of an abstract mental model
This model maps lines of code to the abstract constructs used in the line. Since the model is abstract, new lines of code are easily assimilated without modifying the mental model. This mental model aligns well with MATLAB.

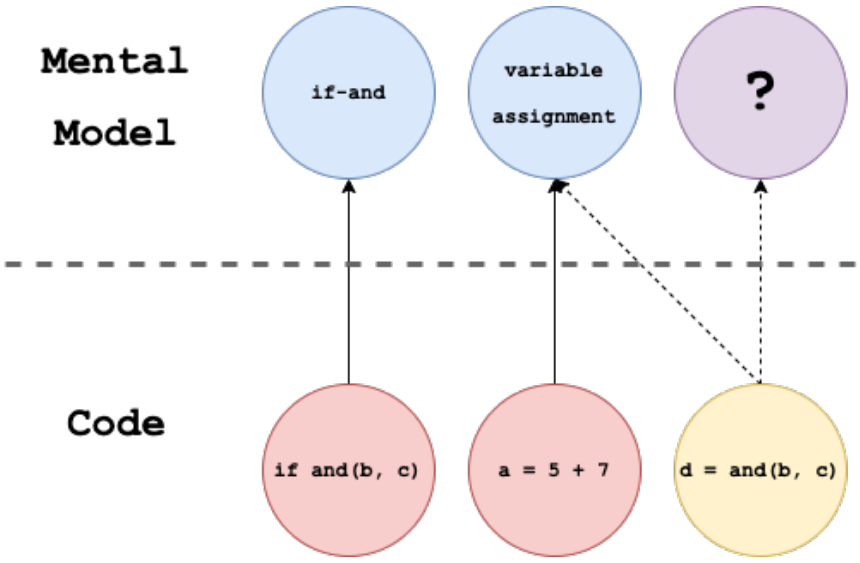


Figure 3.3: Example of a less abstract mental model
This model maps lines of code to semi-abstract constructs. Since the model is specific, assimilating new lines of code often involves updating the mental model. The line **d = and(b, c)** does not fit entirely into either construct in the mental model, so a new construct must be created. This mental model does not align well with MATLAB.

We believe that most of the PChem students have mental models of programming that are closer to student B than they are to student A. This makes it more difficult for the students to comprehend and write programs. In the next chapter, we discuss the development of a workshop based on these findings to help students choose to "reach for computing" to solve future problems.

Chapter 4

CREATING AN INTERVENTION

We were given the opportunity to put our theory into practice by running an 80-minute workshop with the PChem 3 students. In this chapter, we will cover the design of the workshop, lessons learned from the implementation, and recommendations going forward.

4.1 Designing the Workshop

Our goal is to help the PChem students "reach for programming" as a tool to solve problems that they will encounter in the future. As the students have already learned a large amount of programming through the PChem curriculum, we do not need to teach them any new MATLAB constructs. Instead, we can focus the workshop on teaching them to transfer their existing knowledge. We planned to do this by showcasing the abstract relationships between constructs that they have already learned and teaching a "design recipe" to help them approach new problems.

4.1.1 Abstract Programming Fundamentals

The "if-and" and "matrix append" misconceptions discussed in Section 3.3 imply that the PChem students faced challenges recognizing known constructs (e.g. the **and** operator) when they were presented in a different context. As discussed in Section 3.4.1, we believe that they are struggling because they have not learned how all of the constructs are connected. We theorize that making these connections will

Listing 4.1: Expressions used in the workshop

```
1 12 - 8
2 and(True, False)
3 false
4 sym('a')
5 or(True, False)
6 17
7 12 > 7
```

Listing 4.2: Statements used in the workshop

```
1 % What do y, f, and k evaluate to?
2 x = 12;
3 y = 8 / 2;
4 c = @(a, b) (a^2 + b^2)^0.5;
5 h = true;
6 f = c(x/y, y);
7 k = and(h, f > 4.5);
```

improve their mental models, which will allow them to solve problems more robustly. So, a primary goal of the workshop was to help students transition to a more abstract mental model.

To facilitate this, we spent the first half of the workshop on the "building blocks" of programming: statements and expressions. We introduced statements as "anything that causes something to happen in a program", and expressions as "anything that has a value". Intermittently throughout the presentation, we had the students work on short problem sets in small groups. These problem sets can be found in Listings 4.1 and 4.2. The problem sets were designed to showcase everything that had been covered in the previous few slides in order to solidify students' knowledge and help them identify areas of confusion.

4.1.2 Design Recipe

Although being able to write code is a necessary condition for programming a solution to a problem, it is not sufficient. In order for students to be able to utilize their programming knowledge as the tool that it is, they must know how to break a problem down into programmable chunks. In software engineering, this is referred to as *problem decomposition*, and students are often taught the *design recipe* [7]. We felt that the full design recipe, while helpful for software engineering, was not a good fit for the PChem students. To this end, we developed a simplified design recipe (Figure 4.1) that is more appropriate for the style of problem that the PChem students face.

We introduced our design recipe in the context of an example problem. The example problem was:

> *Given a list of grades and an average class grade to meet, create a program that calculates the average excluding the highest N grades and lowest N grades from the calculation and returns whether curving is required or not. Curving is required if the calculated average is below a given minimum required average.*

We designed the example problem to be simple enough to compute examples by hand, while being complicated enough to be an authentic vessel for the design recipe. One way to apply our design recipe to this problem can be seen in Figure 4.2, which shows Step 1, and Figure 4.3, which illustrates Step 3. After Step 3, we would convert each "sub-problem black box" to pseudo-code, and then convert the pseudo-code to MATLAB.

A Simplified Design Recipe

1. **Understand the goal(s) & data** What are you trying to accomplish? What is the data that you are given to accomplish this? Think of your program as an opaque "black box" that takes inputs and produces outputs. If you had a magic box that could solve your problem, what would you input? What would be output?

2. **Do an Example** Solve a simple example by hand. As you go through this process, think about what steps you are taking. If an example is not appropriate for your problem, you can substitute another activity that requires you to engage thoughtfully with the problem.

3. **Identify sub-goals & create plans** What needs to happen to move from the input data to the output solution? What are the steps that you identified as you worked through an example? What do they each take as input and produce as output? Break your original "black box" into a pipeline of smaller "black boxes" that feed into each other to solve the original problem.

4. **Write & check pseudo-code** Fill out each of your small "black boxes" with pseudo-code that converts the input into the output. Then, check the psuedo-code to make sure that it is actually solving every sub-problem, and the original problem. It is tempting to skip this step, but it is much easier to catch errors at this stage than it is to debug MATLAB code.

5. **Translate pseudo-code into MATLAB** Convert your pseudo-code into syntactically correct MATLAB code.

6. **Test your solution** Run your solution on the input data. Verify that it works for all conceivable types of input data, not only the test cases that you have. To accomplish this, you may have to fabricate "tricky" input data.

Figure 4.1: A modified design recipe

After the example problem, we gave the class a more complicated problem to solve in small groups. We wanted our problem to be simple enough to solve in 20 minutes, but complex enough to show the benefit of the design recipe. An ideal problem would:

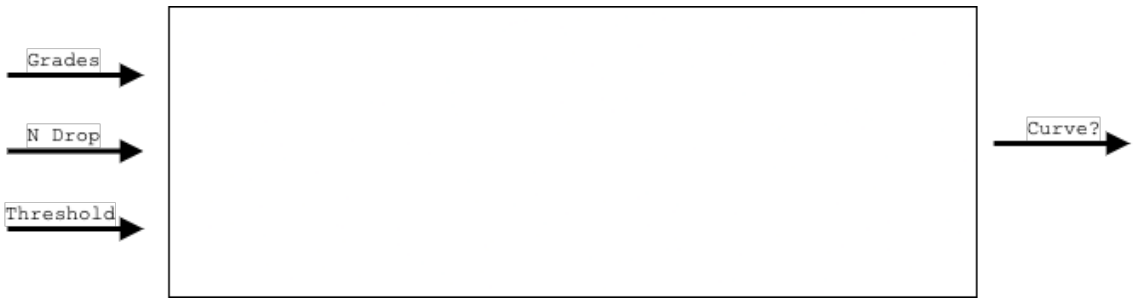


Figure 4.2: An example "black box" diagram
This diagram corresponds to Step 1 of our design recipe, Figure 4.1

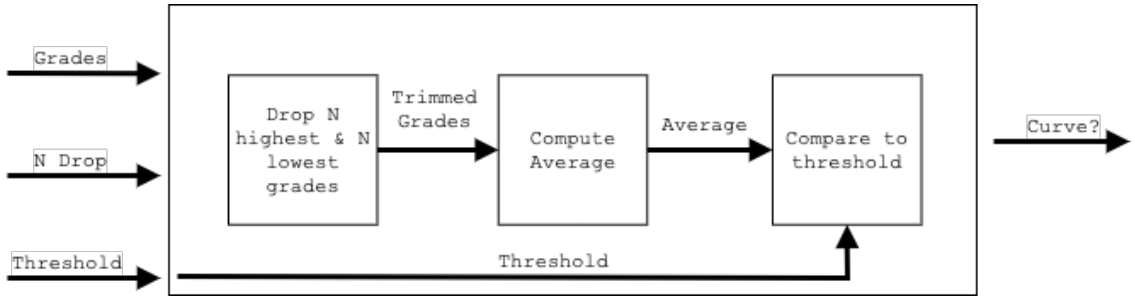


Figure 4.3: An example of sub-problems in a "black box" diagram
This diagram corresponds to Step 3 of our design recipe, Figure 4.1. In more complicated problems, it may be necessary to further decompose each "sub-problem black box" into a series of sub-sub-problem black boxes. This process can continue until the black boxes are small enough to easily be translated into pseudo-code.

1. Have nuances that could be missed without going through an example

2. Be difficult or impossible to solve without decomposing into smaller problems

3. Be easy to verify that each sub-problem is working correctly

We decided that loading and filtering a data set before fitting a curve and interpolating a value would satisfy these criteria. To make the problem more authentic for the PChem students, we worked with Dr. McDonald, a Chemistry professor, to find a suitable data set. Dr. McDonald gave us Morse potential data and an equation that it should fit. We modelled bad sensor readings by inserting '-1' values randomly in the data.

We felt that students would be more likely to follow the design recipe steps if they were scaffolded, so we created a design recipe worksheet. The worksheet includes an empty "black box" model for students to fill out, as in Figures 4.2 and 4.3. It also includes a section for students to translate their "sub-problem black boxes" into pseudo-code. To drive home the point that students should not begin coding until they have theoretically solved the problem, we asked that they run through the entire worksheet before writing any MATLAB code.

4.2 Running the Workshop

Student engagement during the workshop was excellent. Students were keen to volunteer answers, energetic in their group work, and asked insightful questions. We ran the workshop during Week 9, the second to last week of instruction in the quarter. Our workshop was prefaced with a reminder from Dr. McDonald that the MATLAB portion of the final exam is rapidly approaching. It is possible that this contributed to the high level of student engagement.

The abstract programming fundamentals section went smoothly. Students quickly grasped the concepts of expressions, statements, and operators. They had no trouble evaluating the first set of expressions, seen in Listing 4.1. We then moved on to variables, which they were already familiar with. Despite this, they remained engaged throughout the variables section. The next exercise asked them to trace a more complicated program and report the values of the variables at the end, seen in Listing 4.2. This problem includes an **and** outside of an **if** statement in line 7. While many students struggled to recognize the **and** outside of an **if** during the interviews, plenty of students were able to correctly evaluate the program during the workshop, even though we did not explicitly call out the "if-and" misconception. I believe that the

fundamentals portion of the workshop pushed students to expand and abstract their mental models to the point where something previously intractable became trivial.

In the design recipe portion of the workshop, we guided students through the example problem as a whole class on the whiteboard. To maintain engagement, we solved the problem interactively with the class. For example, when we reached step 3 of our design recipe (Figure 4.3), we asked the class to come up with sub-problems. The class quickly arrived at a set of correct sub-problems, which we used for the remainder of the example. In this way, we were able to work with the class to solve a problem, rather than simply lecturing to them. We feel that this helped the students better internalize the steps of the design recipe.

The final portion of the workshop was dedicated to applying the skills previously covered to a more complex problem. We ended up with around 20 minutes to spend on the problem. The workshop instructors floated around the room answering questions and checking in with the students as they worked. While most students embraced the design recipe and filled out the worksheet before coding, some students still attempted to problem solve directly in MATLAB. These students tended to ask questions that they would have answered themselves, had they stuck to the design recipe. Rather than giving the answers away, we asked them to go back to the design recipe. They usually quickly realized that they had misinterpreted or not fully understood part of the problem. They were then able to course correct and continue solving the problem.

It took around 15 minutes for most students to finish the design recipe worksheet, leaving only 5 minutes for them to implement their solutions. This time pressure was not ideal, as many students got stuck on filtering the data. This was an unanticipated stumbling block. We quickly ran through one pattern to filter the data in front of the class. This was a small missed opportunity; had we anticipated this difficulty, we could have made a point of showing students how to search through documentation

or online resources to find answers for themselves. While this would have been nice, it was not a goal of the workshop.

Several students came up to us after the workshop and expressed gratitude. Anecdotally, the general consensus was that the workshop was helpful, but could have been even more impactful if it came earlier in the PChem curriculum. Still, I believe that the workshop may help some students "reach for computing" in the future.

4.3 Lessons Learned

While the workshop was successful, several things could have been improved. In this section, we discuss lessons learned from the first running of the workshop.

An 80-minute workshop was an excellent first step, but more time would have been beneficial. The workshop could have been more effective as two 80-minute workshops, or one 3-hour workshop. This extra time would have allowed us to make sure that every student understands the material. For example, we could have given students more time on the final problem, or answered any lingering questions. We could also have spent more time on the fundamentals, challenging students to explain the "matrix append" operation discussed in Section 3.3. We did not want to bring this up without sufficient time to fully explain it, lest we risk introducing more misconceptions than we address. While there will never be enough time to cover everything in a workshop, a longer workshop would have allowed us to cover material in more detail.

The workshop could have been more effective in a different class. Students in PChem 1 and PChem 2 are in the midst of learning MATLAB. Holding the workshop during one of those courses could have made learning new constructs easier for the students. This could have a compounding effect on student motivation and enjoyment of pro-

gramming, which could lead to improved student outcomes. Additionally, several students told us that they would have loved to have a workshop like this during PChem 1 or 2.

Chapter 5

FUTURE WORK & CONCLUSION

5.1 Future Work

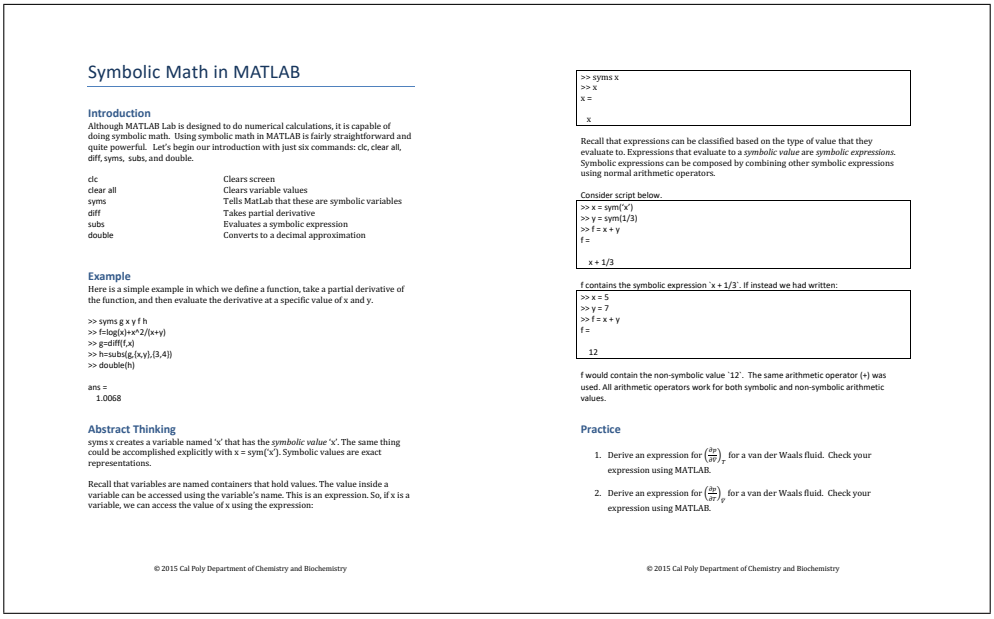
Future work in this area could focus on confirming and extending the results that we have found. In particular, the effectiveness of the workshop should be rigorously evaluated and the interview results should be confirmed with the general PChem population.

In order to properly carry out future work, one would need a concept inventory that is valid for the PChem students. A modified version of the MCS1 could work, but it would need to be shown to be valid for the concepts being tested. Once this is done, the assessment could be deployed to confirm the interview results and evaluate the workshop.

Our theory could be more thoroughly tested by deploying an appropriate assessment to all PChem students. The interview results were key to constructing our theory, but the sample size was small. One could then use students' performance on the assessment to confirm or reject the theory that we developed from interviews.

Evaluating the workshop is important to show that it is actually helping students. This can be done with a pre- and post- test of the new concept inventory. Additionally, it would be worthwhile to hold the workshop in an earlier PChem class to determine where it will be most effective. This can be done by looking at learning outcomes among the PChem students longitudinally.

Additionally, it is possible that a workshop is not the best format for instruction. A workshop is a one-time intervention, but the abstract programming ideas should be covered every time a new programming construct is introduced. With explicit instruction on how new constructs fit into the existing "map" of programming, instructors can encourage students to develop a particular mental model. This additional guidance could be helpful to students, particularly those who are struggling. The "Abstract Thinking" section of Figure 3.2 shows how one of the existing PChem MATLAB worksheets could be modified to refer back to abstract concepts while introducing new constructs.



Symbolic Math in MATLAB

Introduction

Although MATLAB Lab is designed to do numerical calculations, it is capable of doing symbolic math. Using symbolic math in MATLAB is fairly straightforward and quite powerful. Let's begin our introduction with just six commands: clc, clear all, diff, syms, subs, and double.

clc	Clears screen
clear all	Clears variable values
syms	Tells MatLab that these are symbolic variables
diff	Takes partial derivative
subs	Evaluates a symbolic expression
double	Converts to a decimal approximation

Example

Here is a simple example in which we define a function, take a partial derivative of the function, and then evaluate the derivative at a specific value of x and y.

```
>> syms g x y f h
>> f=log(x)+x^2/(x+y)
>> g=diff(f,x)
>> h=subs(g,{x,y},{3,4})
>> double(h)

ans =
   1.0068
```

Abstract Thinking

syms x creates a variable named 'x' that has the *symbolic value* 'x'. The same thing could be accomplished explicitly with x = sym('x'). Symbolic values are exact representations.

Recall that variables are named containers that hold values. The value inside a variable can be accessed using the variable's name. This is an expression. So, if x is a variable, we can access the value of x using the expression:


```
>> syms x
>> x
x =

   x
```

Recall that expressions can be classified based on the type of value that they evaluate to. Expressions that evaluate to a *symbolic value* are *symbolic expressions*. Symbolic expressions can be composed by combining other symbolic expressions using normal arithmetic operators.

Consider script below.

```
>> x = sym('x')
>> y = sym(1/3)
>> f = x + y
f =

   x + 1/3
```

f contains the symbolic expression `x + 1/3`. If instead we had written:

```
>> x = 5
>> y = 7
>> f = x + y
f =

   12
```

f would contain the non-symbolic value `12`. The same arithmetic operator (+) was used. All arithmetic operators work for both symbolic and non-symbolic arithmetic values.

Practice

1. Derive an expression for $\left(\frac{\partial p}{\partial V}\right)_T$ for a van der Waals fluid. Check your expression using MATLAB.
2. Derive an expression for $\left(\frac{\partial p}{\partial T}\right)_V$ for a van der Waals fluid. Check your expression using MATLAB.

Figure 5.1: A modified PChem MATLAB worksheet
We added the "Abstract Thinking" section to this worksheet as an example of how the existing PChem worksheets could be modified to encourage students to form an abstract mental model of programming.

Finally, the methodology covered in this thesis could be applied to other domain-specific programming courses. We have shown one way to create a data-driven intervention to improve student learning outcomes. Domain-specific programming courses already have numerous benefits. Enhancing them with our methodology could provide more opportunity for students.

5.2 Conclusion

In this thesis, we have presented a methodology for evaluating and improving domain-specific programming courses. We started by assessing students with a concept inventory and an attitudes survey. We found that the concept inventory was not appropriate for our student group, so we transformed it into an interview format to give richer data. From this data, we developed a theory: students do not have an abstract mental model of programming and do not feel confident applying their programming knowledge to novel problems. To address this, we created a workshop with the goal of improving students' mental models and providing them with the skills that they need to approach complicated problems. While we did not have time to rigorously evaluate the workshop, we anecdotally found it to be effective. The results from this case study are encouraging. Our methodology could be applied to other domain-specific programming courses to address student challenges and provide students with additional opportunities in computing.

BIBLIOGRAPHY

[1] M. Armoni, O. Meerbaum-Salant, and M. Ben-Ari. From scratch to "real" programming. *ACM Transactions on Computing Education (TOCE)*, 14(4):1–15, 2015.

[2] A. Barach. *A MATLAB Programming Concept Inventory for Assessing First-Year Engineering Courses: a Replication of the Second Computer Science 1 Assessment*. PhD thesis, The Ohio State University, 2020.

[3] B. S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. 13(6):4–16.

[4] J. D. Bransford, A. L. Brown, R. R. Cocking, et al. *How people learn*, volume 11. Washington, DC: National academy press, 2000.

[5] F. E. V. Castro and K. Fisler. Designing a multi-faceted solo taxonomy to track program design skills through an entire course. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, pages 10–19, 2017.

[6] B. du Boulay, T. O'Shea, and J. Monk. The black box inside the glass box: presenting computing concepts to novices. *International Journal of man-machine studies*, 14(3):237–249, 1981.

[7] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2018.

[8] U. Fuller, C. G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernán-Losada, J. Jackova, E. Lahtinen, T. L. Lewis, D. M. Thompson, C. Riedesel, et al.

Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, 39(4):152–170, 2007.

[9] A. Gautam, W. Bortz, and D. Tatar. Abstraction through multiple representations in an integrated computational thinking environment. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, pages 393–399. ACM.

[10] M. L. Gick and K. J. Holyoak. Schema induction and analogical transfer. *Cognitive psychology*, 15(1):1–38, 1983.

[11] M. Guzdial. Learner-centered design of computing education: Research on computing for everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, 2015.

[12] D. Hestenes, M. Wells, and G. Swackhamer. Force concept inventory. *The physics teacher*, 30(3):141–158, 1992.

[13] A. Hoegh and B. M. Moskal. Examining science and engineering students' attitudes toward computer science. In *2009 39th IEEE Frontiers in Education Conference*, pages 1–6. ISSN: 2377-634X.

[14] B. D. Jones. Motivating students to engage in learning: the music model of academic motivation. *International Journal of Teaching and Learning in Higher Education*, 21(2):272–285, 2009.

[15] Y. B. Kafai and C. C. Ching. Affordances of collaborative software design planning for elementary students' science talk. *The Journal of the Learning Sciences*, 10(3):323–363, 2001.

[16] A. J. Ko, R. Abraham, L. Beckwith, A. Blackwell, M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. Myers, M. B. Rosson, G. Rothermel,

M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. 43(3):21:1–21:44.

[17] D. Loksa, A. J. Ko, W. Jernigan, A. Oleson, C. J. Mendez, and M. M. Burnett. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pages 1449–1461, 2016.

[18] A. S. Luchins. Mechanization in problem solving: The effect of einstellung. *Psychological monographs*, 54(6):i, 1942.

[19] A. R. McDonald and J. P. Hagen. Beyond the analytical solution: Using mathematical software to enhance understanding of physical chemistry. In *Using Computational Methods To Teach Chemical Principles*, volume 1312 of *ACS Symposium Series*, pages 195–210. American Chemical Society. Section: 14.

[20] M. Parker, M. Guzdial, and S. Engelman. Replication, validation, and use of a language independent CS1 knowledge assessment. pages 93–101.

[21] M. C. Parker, M. Guzdial, and S. Engleman. Replication, validation, and use of a language independent cs1 knowledge assessment. In *Proceedings of the 2016 ACM conference on international computing education research*, pages 93–101, 2016.

[22] M. C. Parker, M. Guzdial, and A. E. Tew. Uses, revisions, and the future of validated assessments in computing education: A case study of the fcs1 and scs1. In *Proceedings of the 17th ACM Conference on International Computing Education Research*, pages 60–68, 2021.

[23] R. D. Pea, E. Soloway, and J. C. Spohrer. The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, 9:5–30, 1987.

[24] C. Scaffidi, M. Shaw, and B. Myers. An approach for categorizing end user programmers to guide software engineering research. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[25] B. Shneiderman. Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, 5(2):123–143, 1976.

[26] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education (TOCE)*, 13(4):1–64, 2013.

[27] A. Strauss and J. M. Corbin. *Grounded theory in practice*. Sage, 1997.

[28] A. E. Tew and M. Guzdial. The FCS1: a language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 111–116. Association for Computing Machinery.

[29] E. Tshukudu and Q. Cutts. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 227–237, 2020.

[30] D. Wanzer, T. McKlin, D. Edwards, J. Freeman, and B. Magerko. Assessing the attitudes towards computing scale: A survey validation study. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 859–865, 2019.

[31] D. Weintrop, E. Beheshti, M. Horn, K. Orton, K. Jona, L. Trouille, and U. Wilensky. Defining computational thinking for mathematics and science classrooms. 25(1):127–147.

[32] D. Weintrop and U. Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*, pages 199–208, 2015.

[33] M. Wertheimer. Experiments in productive thinking. 1949.

[34] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.