Designing Exercises to Teach Programming Patterns

Nathaniel Weinman

Designing Exercises to Teach Programming Patterns

by

Nathaniel Weinman

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Armando Fox, Co-chair
Professor Marti Hearst, Co-chair
Professor Marcia Linn

Spring 2022

Designing Exercises to Teach Programming Patterns

Abstract

Designing Exercises to Teach Programming Patterns

by

Nathaniel Weinman

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Armando Fox, Co-chair

Professor Marti Hearst, Co-chair

As digital technologies continue to grow in importance and influence, the need for strong programmers continues to grow. “Programming,” though captured by a single word, actually requires the acquisition of many different skills. One such skill is identifying where and how to apply *programming patterns* — reusable abstractions and organizations of code. This skill has been shown to be a distinguishing characteristic between experts and novices.

However, our current Computer Science classrooms offer scarce opportunities for students to *deliberately practice* patterns through *exercises*. The most common code practice tool is the Code Writing exercise; the goal it presents to students is to construct a program that matches certain outputs from given inputs. Code Writing exercises offer minimal affordances to guide students towards particular solutions, making it difficult for instructors to ensure students will practice a specific pattern.

This work introduces three new exercises designed to support students in acquiring new patterns. In Faded Parsons Problems, students rearrange and complete lines of code to reconstruct a program. In Subgoal Decomposition exercises, students reconstruct the execution flow of subgoals across multiple files and languages, specifying where each subgoal fits in an architectural framework. In Data Flow exercises, students further concretize each subgoal by using code snippets to specify the data processing for which each subgoal is responsible. Each of these exercises were designed to be easy to adopt into existing curricula without significant changes, complementing existing Code Writing exercises. Studies with students revealed that each of these exercises addresses currently unmet needs in CS courses, that these exercises support students practicing patterns by allowing students to focus on the full solution, and that students want these exercises to be integrated into their courses. Additionally, a classroom-based study with 237 students found that Faded Parsons Problems are more effective than Code Tracing and Code Writing exercises at helping students acquire patterns while also improving their general code writing abilities. Finally, from the lessons learned in developing these exercises, five design goals are presented to motivate the creation of more exercises for new contexts.

To Dave, who decided every day to come along on this wild adventure.

Contents

List of Figures

List of Tables

Acknowledgments

This work would not have been possible without the support of many brilliant and wonderful people along the way. First and foremost, thank you to Armando Fox and Marti Hearst for the countless hours you have spent co-advising me on my work and developing me as a researcher. Thank you both for pushing me to develop my ideas further than I thought possible.

I would also like to thank my committee member, Marcia Linn, as well as my other qualifying exam committee member, Sarah Chasins. Your feedback has helped tie this work together. Thank you to Björn Hartmann for the mentorship you provided at the beginning of my Ph.D. as I was first learning about Human-Computer Interaction. Thank you to Paul Laskowski, for patiently teaching me to conduct more rigorous statistical analyses.

I would like to thank Michael Ball, for trusting me to run a study as part of his course. I'd like to think him along with Pamela Fox for the many discussions around this work, as well as working to bring this work into more classes and contue extending it.

Thanks to Rob Deline, Steven Drucker, and Titus Barik for the mentorship you provided during my intersnship at Microsoft Research. You taught me ways to think about research, allowing me to answer the questions that mattered to me.

Thanks to the members of research groups I've been a part of at my time at Berkeley. Thank you for sharing your wisdom, support, inspiration, and camaraderie. Thank you Andrew Head for mentorship, friendship, and an endless supply of interesting papers to read. Thank you Katie Stasaski for incredible brainstorming, tea breaks, and the frequent advice. Thank you Hezheng Yin, An Ju, Logan Caraco, and the members of Fox Group. Thank you Philippe Laban, Chase Stokes, Peitong Duan, and the members of Hearst Lab. Thank you Collette Roberto, Zephyr Barkan, Vron Vance, Dan Garcia, Nick Weaver, and the other members of ACELab. Thank you Molly Nicholas, Sarah Sterman, Elena Glassman, Ananya Nandy, Forrest Huang, Janaki Vivrekar, Jeremy Warner, J.D. Zamfirescu-Pereira, Yakira Mirabito, Bala Kumaravel, Cesar Torres, Christine Dierk, David Chan, Deniz Dogruer, Drew Sabelhaus, Eldon Schoop, Katherine Song, Mike Laielli, Shm Almeda, Yifan Wu, and the members of BiDLab.

Thanks to the various students I've been lucky enough to mentor for supporting this work. A particular thank you to Brian Hsu and Jack Boreckzky for being co-conspirators in creating new tools. Thank you also to: Alexander Ng, Alexia Camacho, Aman Sidhant, Aslı Akalin, Daniel Gultom, Danny Chu, Emily Zhong, Leanna Tran, Srujay Korlakunta, and Tlaloc Baraja.

Thank you to my friends, for helping me disconnect when it was helpful and making sure I continued living a full life these past years. Thanks to Mitha Nandagopalan, Jacob Little, Alex Rybak, Alina Shinkarsky, Kendra Albert, Regina Eckert, Sandra Hui, Emily Naviasky, Josh Sanz, and all of my other friends who have supported me on this journey.

Finally, thank you to my husband, David Gainsboro. You celebrated in the milestones of my program and the successes of my work. You were the first to lend a hand whenever I was struggling. Thank you for coming along, every step of the way.

Chapter 1

Introduction

Programmers build incredible tools that change the way we live our lives, be it the way we access information, communicate with others, and more. Both the demand for and number of individuals interested in becoming software engineers continues to grow incredibly quickly (Scaffidi et al., 2005; Barr et al., 2011; *U.S. Department of Labor, Occupational Outlook Handbook, Software Developers* 2019). As we continue to train more software engineers to do important work, we should be improving the methods we have to train them. In this work, I discuss how we can better prepare future software engineers, with a particular focus on scaffolding the instruction of *patterns*.

In college-level courses, students are expected to spend a substantial proportion of their learning time working on homework exercises (Seaton et al., 2014). Homework and lab exercises provide a space for students to actively learn the concepts covered in lecture. It more closely mimics a professional environment, since students are given a chance to practice reasoning through how to solve given problems by applying concepts in new ways.

In many CS1 courses, homework practice shifts from reading code (e.g., example solutions, Code Tracing exercises) to writing it. Though this can be described simply as a transition from reading to writing, it is actually much more complex than that. When we speak of "programming" or "code writing," we are actually discussing many different ideas and skills (du Boulay, 1986; Jenkins, 2002; Cutts et al., 2012). Programmers must think about syntax, data types, control flow, algorithms, data structures, problem comprehension, problem decomposition, interpreting error messages, debugging test case failures, and more.

To become a proficient programmer, students must eventually become familiar with tying these ideas and skills together. Students often practice with Code Writing exercises as part of their coursework, in which students construct a program that matches certain outputs from given inputs. In some ways, Code Writing exercises may provide the perfect practice, as they provide realistic, holistic programming practice. On the other hand, the transition from reading example programs to writing programs might be overwhelming for some students, missing an opportunity to scaffold some of these skills. In fact, there are many projects that propose different exercises to scaffold some part of the programming practice, suggesting others also believe in a significant opportunity for more scaffolding (Linn and Clancy, 1992;

Alphonce et al., 2005; Moritz et al., 2005; Sajaniemi et al., 2005; Parsons et al., 2006; Politz et al., 2014; Choudhury et al., 2016; Loksa et al., 2016; Zavala et al., 2017; Ericson, Foley, et al., 2018; Felleisen et al., 2018; Shi et al., 2019; Wrenn et al., 2019; Weinman, Fox, and M. Hearst, 2020; Milliken et al., 2021).

Below, I explore how exercises can help students acquire useful *patterns* (also known as plans, schemata, templates, or frameworks). Specifically, I focus on two different types of patterns. *Goal-oriented patterns* are partial implementations of reusable, higher-level concepts which can achieve a goal; for example, computing a count or the Observer pattern in software design (Gamma et al., 1995). *Architectural patterns* are abstractions which can be used to set clear boundaries between smaller problems that can be solved independently. For example, the Model-View-Controller framework in web architecture or the Builder pattern (Gamma et al., 1995).

I chose to focus on scaffolding the instruction of patterns for a range of reasons. Experts have been found to have access to more patterns than novices (Wiedenbeck et al., 1993; Robins et al., 2003; Ko et al., 2008), suggesting that patterns are important to learn. Patterns help in understanding code written by someone else (McCauley et al., 2008), help in debugging code (O'Dell, 2017), and are a key building block to problem decomposition (Soloway, 1986). That is, patterns are critical to enabling programmers to tackle problems of increasing complexity.

All of these use cases have compounding effects as programs get more complex. In a CS1 course, a student might be writing a function of 5 lines over the course of 15 minutes. If the student fails to recognize the relevant patterns, their solution may become more complex increasing to 10 lines. Or the student may pursue some false leads adding an additional 15 minutes. But, even in these cases, the problem is still be reasonable to complete. In upper-division courses, individual programming questions may be designed to take hours, and consists of hundreds of lines of code across multiple files. These numbers get even larger if there is a month-long project in the course. In these cases, a false start that does not pan out could add hours to the total time a student spends on the exercise. Poor problem decompositions can lead to incredibly complex interactions between components and classes, in some cases because students did not identify productive boundaries between different subgoals. Though the benefits of learning patterns may not be as strongly apparent in CS1 courses, learning to recognize and think of patterns sets students up well to continue learning larger patterns in later courses, where the impact is much more significant.

This work contributes the following:

- A set of five design goals to capture one approach for creating exercises in a range of contexts to productively teach patterns, motivated by existing literature and probed through exercises (Chapters 2, 6).

- Faded Parsons Problems, a new exercise designed for patterns that can be grounded in a reasonable amount of syntax ($<$ 20 lines of code) (Chapter 4). This exercise is evaluated and found effective in a large CS1 classroom study (Chapter 5).

- Subgoal Decomposition and Data Flow exercises, designed for patterns at a higher level of abstraction than code. These exercises are qualitatively evaluated and show promise for students in an upper-division Software Engineering course (Chapter 6).

1.1 Overview of this Dissertation

In this section, I provide some additional details for each of the remaining chapters.

Chapter 2 describes five design goals to capture an approach to create exercises that teach patterns. The chapter provides some additional context by exploring the extent to which six different exercises adhere to each of the design goals. This chapter appears anachronistically; these design goals are motivated by insights from the empirical work in Chapters 3 - 5 and probed in the work in Chapter 6.

The following chapters appear in chronological order. Chapter 3 describes my initial structured exploration into Parsons Problems. I ran a study with advanced undergraduate students comparing Parsons Problems to Code Writing exercises. The study was focused on teaching students how to use different algorithms, similar to questions that might be asked in a Software Engineer interview. While the study did not lead to any strong findings, the results suggested that Parsons Problems were not particularly effective practice for upper division students.

Chapter 4 introduces Faded Parsons Problems, a variant of Parsons Problems that provide students with *partially incomplete* lines of code instead of complete lines. I ran a small study with CS1 students (n=13) to compare Parsons Problems, Faded Parsons Problems, and Code Writing exercises. This study confirmed the opportunity for further scaffolding between Parsons Problems and Code Writing exercises, as students that solved a question as a Parsons Problem weren't able to demonstrate transfer by solving an identical Code Writing exercise later in the same study session. This study also suggested that both types of Parsons Problems were easier for students than Code Writing exercises and effective at showing them a particular approach, with the caveat that these findings were based on self-rated data.

Chapter 5 reports on a more rigorous study on the efficacy of Faded Parsons Problems. I ran a study (in 3 parts) as part of a CS1 course, offering extra credit throughout the semester. 237 students participated in some part of the study. This study compared Faded Parsons Problems to both Code Tracing and Code Writing exercises. This study provided compelling evidence that Faded Parsons Problems are well-suited to teaching patterns, as well as general programming skills. Additionally, students expressed a preference for working with Faded Parsons Problems. Overall, the results suggest that Faded Parsons Problems are an effective complement to other exercises in a CS1 context. However, it was unclear how powerful they would be for more advanced students.

Chapter 6 reports on the design of two exercises that were motivated by the success of Faded Parsons Problems. I hope that Faded Parsons Problems could be powerful at teaching syntax-level patterns to upper-division students, such as idioms in a new language. However, I was more curious how to design exercises that could teach patterns at higher levels of abstraction, where individual lines of code might not be a productive granularity to engage with the ideas. In this chapter, I repeat the set of design goals (covered in more detail in Chapter 2), which were motivated by trying to hypothesize *why* Faded Parson Problems are effective at teaching patterns. I also introduce two new exercises targeted at an upper-division Software Engineering course teaching web development. I evaluated these two new exercises with 12 upper-division students, and also used the exercises as a probe to evaluate the design goals. The results suggest promise for these new exercises, as well as for the design goals as a productive method to develop further exercises in other contexts.

Finally, Chapter 7 highlights some exciting opportunities for future work.

1.2 Statement of Prior Publication

Chapter 4 is an extended version of a poster that previously appeared in the proceedings of the SIGCSE Technical Symposium on Computer Science Education (Weinman, Fox, and M. Hearst, 2020). Chapter 5 previously appeared as a conference paper in the proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Weinman, Fox, and M. A. Hearst, 2021).

All of my co-authors on these articles have provided their consent for the articles to be reproduced in this dissertation. Much of this dissertation, therefore, represents the gracious contributions of my co-authors and advisors, Professor Armando Fox and Professor Marti A. Hearst.

Chapter 2

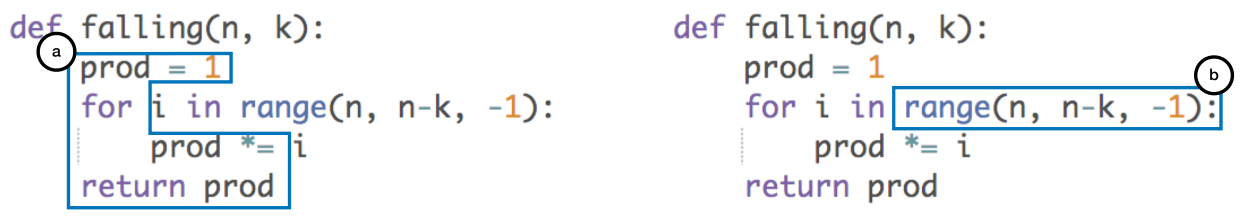
Design Goals for Exercises that Teach Patterns

Writing programs requires programmers to draw on a range of skills and different types of knowledge. While Code Writing exercises are a valuable opportunity to practice all of those skills in conjunction, students could benefit from exercises that provide opportunities to deliberately practice a subset of those skills. One such skill is identifying where and how to apply programming patterns when constructing a program. In this chapter, I present a set of five design goals to aid instructors and researchers in creating new exercises that help students practice this skill.

I explore patterns in two domains. First, I explore goal-oriented patterns relevant to a CS1 classroom. Goal-oriented patterns are partial implementations of reusable, higher-level concepts which can achieve a goal. In a CS1 context, these patterns consist of only a few lines of code. Consider computing the number of `k`-sized permutations of `n` objects, where the goal is to write a program that computes the product of `k` consecutive numbers, starting from `n` and working downwards. A solution can be seen in Figure 2.1. A student creating this solution may extract several different patterns.

In addition to CS1 patterns, I also explore those relevant in an upper-division setting; specifically, in web architecture. As in the simpler case, there are still goal-oriented patterns such as the use of asynchronous callbacks when leveraging a third party service (e.g., for authentication). However, there are also architectural patterns, such as the Model-View-Controller framework, which help keep programs reasonably organized and efficient. Architectural patterns are abstractions which can be used to set clear boundaries between sub-problems.

In this chapter, I present a set of five design goals to create new exercises that help students learn when and how to apply patterns. These proposed design goals emerged from the studies in Chapters 3 - 6. I then explore how several different exercises satisfy or fail to satisfy these design goals. I explore ubiquitously used exercises (Code Tracing and Code Writing), as well as less widely-used exercises (Parsons Problems, Faded Parson Problems, Subgoal Decomposition exercises, and Data Flow exercises).



```
def falling(n, k):
    prod = 1
    for i in range(n, n-k, -1):
        prod *= i
    return prod
```

```
def falling(n, k):
    prod = 1
    for i in range(n, n-k, -1):
        prod *= i
    return prod
```

Figure 2.1: **Examples of different goal-oriented patterns in a single solution.** A solution to compute the number of k-sized permutations of n objects, or a "falling factorial." We highlight 2 of the possible patterns students may recognize in this program. (a) The pattern of an accumulator variable, in this case a running product. (b) The pattern of a reverse range in Python, iterating down instead of up.

2.1 Design Goals

In this section, I describe five proposed design goals to create exercises that teach patterns. These design goals can be collectively referenced as creating exercises in which students *reconstruct many intentionally-designed solutions.*

The first three design goals focus on "reconstructing solutions." While it is important to make exercises interactive for students, the exercises should also limit parts of programming that might distract students from reflecting on the patterns. By providing pieces that can be combined into a solution (e.g., lines of code or subgoals), students can maintain their attention on the problem as a whole until it is solved. Without this aid, students might disrupt their own problem solving process to dive into implementation details and complex debugging. By focusing on the problem as a whole, students would also focus on the structure of the solution and the patterns contained within.

The final two design goals focus on students creating "many intentionally-designed solutions." Constructivism posits that people learn by comparing new experiences to what they already know, identifying similarities and differences, and synthesizing that into new knowledge (Fosnot, 2013). Based on this theory, one way to teach a pattern is for students to identify similarities and differences between two programs that share the pattern in common. An advantage of this is that students can internalize patterns without explicitly naming them. Exercises should allow students to rapidly see multiple examples of a pattern, to aid them in recognizing and internalizing the pattern by identifying similarities in solutions to multiple problems.

2.1.1 Provide Opportunity to Practice

Exercises should address learning goals by allowing students to construct relevant components of the solution. Formative assessments, such as homework assignemnts, are critical to the classroom. Exercises like Code Writing involve "active learning" – having students reflect on ideas and how to use them. Active learning is a powerful pedagogical tool (Michael, 2006). When assigning an exercise for a course, it is important to examine what is given to students and what they must actively reflect on. Exercises can focus on any combination of mental models (e.g, the notional machine (Boulay et al., 1981; Griffin, 2016) or UML diagrams), skills (e.g., program comprehension before writing (Wrenn et al., 2019; Basu et al., 2015)), particular knowledge (e.g., language-specific syntax), and more. With this design goal, I encourage reflection on *what* parts of programming students actively practice as they construct their solutions. Since many patterns are structural, spanning more than a single line of code, exercises must allow students to engage with how the different components that form a pattern interact.

2.1.2 Scaffold Vocabulary

Exercises should provide pieces of the solution to students, allowing students to focus their learning, time, and energy on how those pieces interact to form a solution. In a CS1 context, those pieces might be lines of code. In an upper-division context, those pieces might e more abstract, for example subgoals. Programming is a complex task consisting of many different skills and types of knowledge (du Boulay, 1986; Cutts et al., 2012; Jenkins, 2002). Cognitive Load theory poses that students have limited mental capacities while working on problems, and if overtaxed they will be unable to reflect on and store their learning from completing exercises (Renkl et al., 2002; Sweller, 1988). One way to reduce cognitive load is through *scaffolding*, or providing structure or material to students.

Scaffolding is an effective method to help students focus on particular learning goals. For example, a series of problems around navigating Trees might provide an already implemented Tree class, since the learning goals are around using Trees rather than implementing the data structure itself. Or, some courses use cloud-based virtual machines so that students do not need to learn about installing libraries and packaging in their local development environment, a task which can be quite time-consuming and distracting for students.

In the context of learning patterns, we highlight the opportunity to limit the levels of abstractions students must navigate. Patterns focus on how subgoals or different pieces of syntax *combine* together. For example, by scaffolding vocabulary by providing lines of code, these exercises prevent students from getting distracted by debugging issues at a low-level of abstraction, such as missing a colon at the end of a line. By providing vocabulary to students, students can focus their efforts on understanding *how* that vocabulary can be reconstructed into a solution.

2.1.3 Scaffold Process

When relevant, exercises should be designed such that students follow good processes as they reconstruct solutions. In programming courses, it is common to grade students primarily on the completeness of the final program. However, though it can be difficult to grade, instructors also may desire students to practice different processes (e.g., top-down problem decomposition, test-driven development, pair programming) (Ju et al., 2018).

One way to scaffold process is to require students to submit artifacts at different stages. For example, to encourage Test-Driven Development, CaptainTeach (Politz et al., 2014) requires students to submit and peer review test suites before the program implementation. Or, to encourage problem comprehension, OK[1] (Basu et al., 2015) requires students to validate example outputs before beginning their program implementation. Alternatively, exercises can require students to submit artifacts at a level of abstraction different than code, such as the work by Cunningham et al. (2021) in which novice programmers work on goals before code. Architectural patterns, which provide an organization framework for sub-problems, are particularly relevant when planning a solution. If teaching patterns like these, exercises should require students to create artifacts in an order such that students are following good processes.

2.1.4 Guide Students to One Solution

Exercises should intentionally expose students to particular solutions selected by the instructor. There are many ways to construct a program that achieves a specific goal. For example, for loops can be replaced by while loops, or an algorithm using iteration can also be solved with recursion. Pedagogically, there are several reasons instructors might want to guide students towards a specific implementation: one approach may be much simpler to reason through than the other, or one approach may require students grapple with a newer approach, or one approach might be better-styled and more compact. In the context of patterns, this design goal becomes even more relevant, as the instructor can ensure that students reconstruct a program that matches the intended pattern. Instructors can then create a series of exercises that all leverage the same pattern, given students multiple sequential examples that highlight the pattern as a similarity between each exercise. If this design goal is not achieved, there is no guarantee that a student will even see the pattern at all. Though not explored in this work, instructors could also use this design goal to expose students to multiple solutions to the same problem, giving students an opportunity to reflect on which solutions or patterns are a best fit for the problem. Exercises should ensure students have the opportunity to discover patterns by careful design of activities.

2.1.5 Present Examples Efficiently

Exercises should be efficient enough to expose students to multiple solutions using a pattern within a single assignment. Constructivism posits that people learn by comparing new experiences to what they already know, identifying similarities and differences, and synthesizing that into new knowledge (Fosnot, 2013). Indeed, programmers become more effective over

[1]https://okpy.org/

time by seeing more programs and developing patterns they can apply to future problems (Rist, 1991).

In the limited amount of time students have to work on homework or lab exercises, they should see enough examples to see commonalities in solutions that generalize into patterns. This design goal becomes particularly relevant in upper-division courses. In introductory courses, problem sets often consist of a sequence of problems that can each be solved in minutes, but in upper-division courses, many programming assignments take days if not weeks. While these longer exercises play a critical role, students would be supported they were also able to work on shorter exercises to better prepare for the longer ones. Exercises should be fast enough that students can see multiple solutions to internalize their own generalizations.

2.2 Exploring Specific Exercises

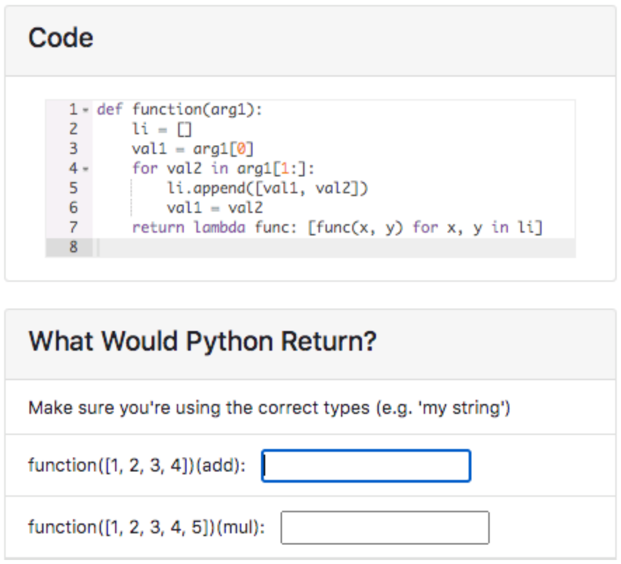
In this section, I describe how six exercises adhere to each of the design goals described above. The first two exercises are ubiquitous to CS classrooms: Code Tracing and Code Writing exercises. In Code Tracing exercises, students must read a program and predict the output from specified input. In Code Writing exercises, students must construct a program that solves a given prompt. The third exercise is Parsons Problem. In a Parsons Problem, students are given a prompt and must create a solution by selecting and reordering lines of code that are given to them. The final three exercises are novel to this thesis: Faded Parsons Problems, Subgoal Decomposition exercises, and Data Flow exercises. Faded Parsons Problems are the same as Parsons Problems, except the given lines of code are incomplete, so students must reorder and complete the lines of code. Both Subgoal Decomposition and Data Flow exercises were designed for use in a web architecture course. In Subgoal Decomposition exercises, students must order subgoals by execution flow and select where they fit into the Model-View-Controller framework. In Data Flow exercises, students must select which data is modified by each subgoal as well as the code that connects execution between each subgoal.

2.2.1 Code Tracing

In Code Tracing exercises (Figure 2.2), students are given a program and must predict what a program will output given certain arguments. In some cases, students must also record the value of each variable at certain points of execution within the program.

Below, I explore how well Code Tracing exercises satisfy each design goal.

- *Provide Opportunity to Practice:* Low adherence. Code Tracing exercises provide practice of the notional machine (Boulay et al., 1981; Griffin, 2016), since students must understand how state is updated as the program is executed. However, students do not construct any of the program themselves, so these exercises do not offer practice for syntax, debugging, control flow, and many other skills that are part of programming.



```
def function(arg1):
    li = []
    val1 = arg1[0]
    for val2 in arg1[1:]:
        li.append([val1, val2])
        val1 = val2
    return lambda func: [func(x, y) for x, y in li]
```

Make sure you're using the correct types (e.g. 'my string')

function([1, 2, 3, 4])(add):

function([1, 2, 3, 4, 5])(mul):

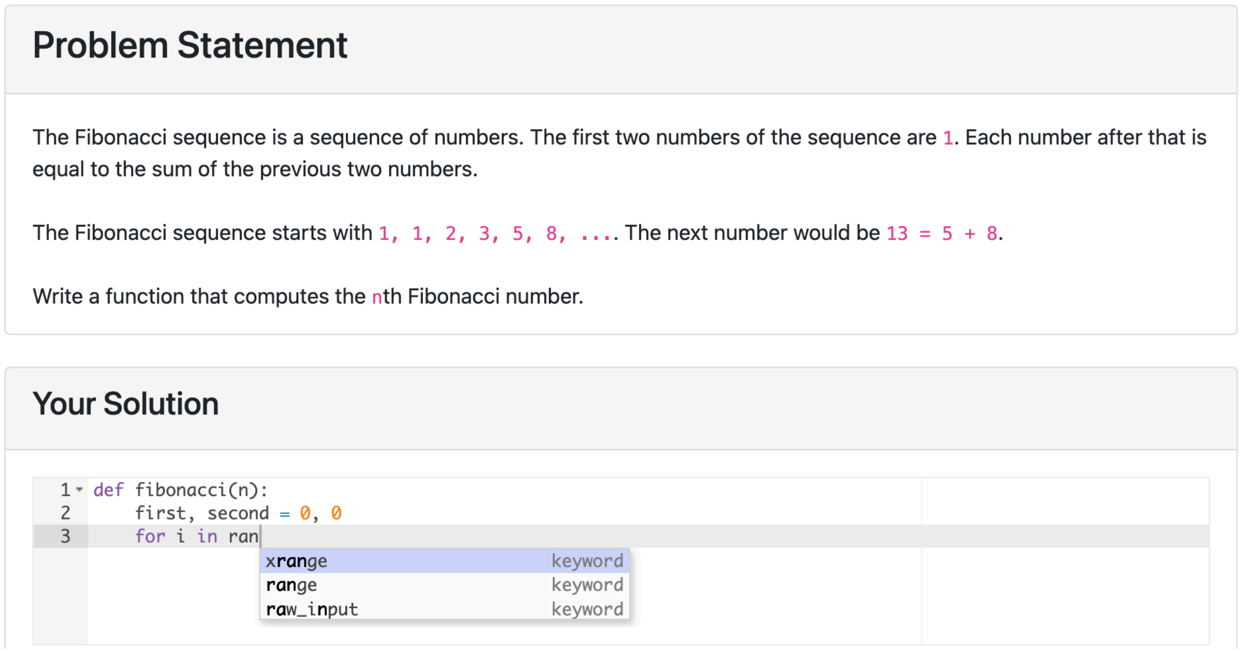
Figure 2.2: **A Code Tracing exercise.** Students must read the obfuscated code (top) and determine the output from specified argument prompts (bottom)

- *Scaffold Vocabulary:* High adherence. Code Tracing exercises provide a complete program.
- *Scaffold Process:* Low adherence. Code Tracing exercises fully scaffold the process of getting to a complete program without representing any intermediate steps.
- *Guide Students to One Solution:* Medium adherence. Code Tracing exercises provide all students with the same complete program. However, these exercises often use obfuscated variable names to ensure students walk through the program step-by-step, and sometimes use obscure syntax to introduce it to students. By design, Code Tracing exercises may not be providing a *well-designed* example of a program.
- *Present Examples Efficiently:* High adherence. Code Tracing exercises are quite fast compared to Code Writing exercises.

2.2.2 Code Writing

In Code Writing exercises (Figure 2.3), students are given a goal as a problem prompt and must construct a program which achieves that goal. In some cases, students are given parts of the program (e.g., function signatures) or test cases with or without expected outputs.

Below, I explore how well Code Writing exercises satisfy each design goal.



Problem Statement

The Fibonacci sequence is a sequence of numbers. The first two numbers of the sequence are `1`. Each number after that is equal to the sum of the previous two numbers.

The Fibonacci sequence starts with `1, 1, 2, 3, 5, 8, ....` The next number would be `13 = 5 + 8`.

Write a function that computes the `n`th Fibonacci number.

Your Solution

```
def fibonacci(n):
    first, second = 0, 0
    for i in ran
```

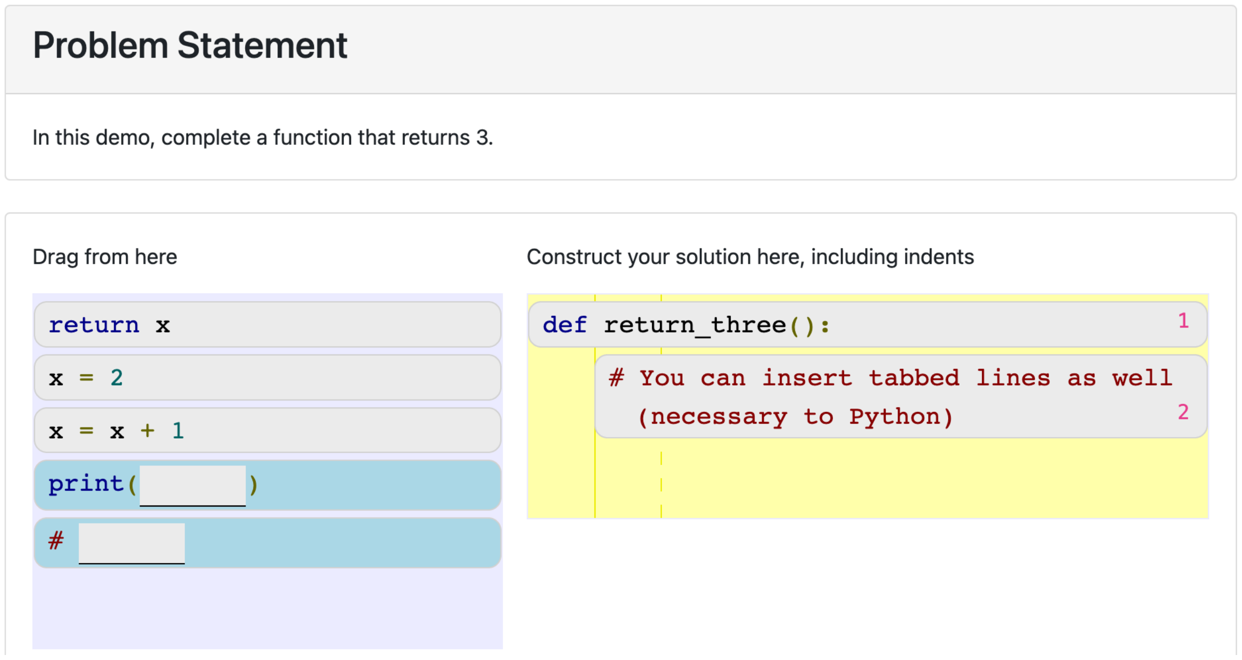
Figure 2.3: **A Code Writing exercise.** Students must create a functioning program to solve a given exercise prompt using an editor with basic IDE functionality like syntax highlighting and auto-completion.

- *Provide Opportunity to Practice:* High adherence. Code Writing exercises provide practice of every part of the programming process given a problem prompt. Instructors can use the problem prompt to an extent to adjust the complexity of each of these, but this includes: recollecting the correct syntax, problem decomposition and planning, and selecting appropriate algorithms to use. Outside of instructor control, this also includes debugging, as each student will make different syntactic or semantic mistakes as they work through a problem.

- *Scaffold Vocabulary:* Low adherence. Code Writing exercises need not provide any vocabulary, and often only include function signatures.

- *Scaffold Process:* Low adherence. Code Writing exercises can scaffold problem decomposition and planning through problem prompts or provided code, but otherwise do not scaffold the program construction process.

- *Guide Students to One Solution:* Low adherence. Code Writing exercises provide limited mechanisms for instructors to ensure students are guided to a single solution. Instructors can achieve this to an extent by explicitly instructing students to follow an


Problem Statement

In this demo, complete a function that returns 3.

```
return x
x = 2
x = x + 1
print(      )
#
```

```
def return_three():
    # You can insert tabbed lines as well
      (necessary to Python)
```

Figure 2.4: **An example of an unsolved Parsons Problem.** Students rearrange the given lines of code by dragging them from the left to the right.

approach, but in that students do not have an opportunity to recognize for themselves which approaches are applicable.

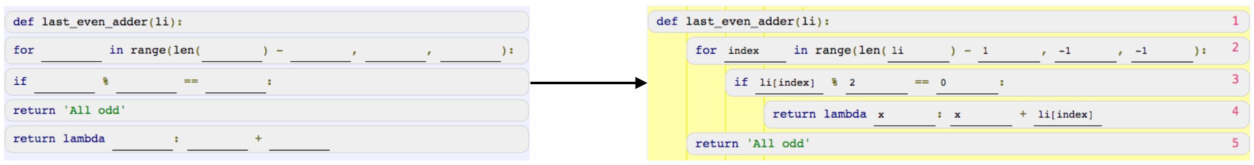
- *Present Examples Efficiently:* Medium adherence. In a CS1 context, Code Writing exercises are fast enough that students can work through several problems on a weekly homework. However, in upper-division courses, project-based assignments are much more common and can take weeks to complete a single assignment.

2.2.3 Parsons Problems

In Parsons Problems (Parsons et al., 2006) (Figure 2.4), students are given both a problem prompt as well as lines of code. Students must select and correctly rearrange the appropriate lines into a program.

Below, I explore how well Parsons Problems satisfy each design goal.

- *Provide Opportunity to Practice:* Medium adherence. Parsons Problems have been found to correlate more closely with Code Writing exercises than Code Tracing exercises (Denny et al., 2008; Ericson, Margulieux, et al., 2017). Though Parsons Problems provide code snippets, students must still reason *how* the lines connect to correctly construct the logical flow of the program. However, with sufficiently advanced students,



```
def last_even_adder(li):
for ________ in range(len(________) - ________, ________, ________):
if ________ % ________ == ________:
return 'All odd'
return lambda ________: ________ + ________
```

```
def last_even_adder(li):
    for index in range(len( li ) - 1 , -1 , -1 ):
        if li[index] % 2 == 0 :
            return lambda x : x + li[index]
    return 'All odd'
```

Figure 2.5: **An example of an unsolved Faded Parsons Problems (left) and the completed solution (right)** using a Premature Return pattern to solve a Higher-Order Function problem. Students solve the question by both rearranging and completing the given lines of code.

there are some concerns that they can use techniques like data flow analysis to nearly solve a Parsons Problem, reducing the effective practice (Weinman, Fox, and M. Hearst, 2020).

- *Scaffold Vocabulary:* High adherence. Parsons Problems provide students with the full lines of code necessary to complete the program. They scaffold syntactic knowledge and may be particularly effective when asking students to practice new syntax and concepts, as Bloom's taxonomy suggests students will be able to recognize how to use new syntax before being able to compose with it unaided (Bloom et al., 1956).

- *Scaffold Process:* Medium adherence. Parsons Problems offer a different process than Code Writing exercises. It is possible that they encourage students to take a top-down approach, focusing on either key control flow or data flow elements. Further researcher similar to that done by Helminen et al. (2012) could shed further light on this.

- *Guide Students to One Solution:* High adherence. Parsons Problems, by design, often have a single solution. In fact, some popular implementations (e.g. Runestone[2], PrairieLearn[3]) grade Parsons Problems by simply checking the order of the lines of code without any code execution. Because code fragments are provided to students, an instructor can guarantee that students create a particular solution.

- *Present Examples Efficiently:* High adherence. Ericson, Margulieux, et al. (2017) found that Parsons problems take only 70% of the time as Code Writing exercises in introductory Computer Science courses. However, this efficiency may be lost if the given lines of code differ greatly from how the student would approach their solution (Haynes et al., 2021).

2.2.4 Faded Parsons Problems

In Faded Parsons Problems (Weinman, Fox, and M. A. Hearst, 2021) (Figure 2.5, Chapters 4, 5), in addition to rearranging the lines of code as in Parsons Problems, students must also complete certain lines of code (e.g., referencing variables, writing conditions, etc.). The design of Faded Parsons Problems is motivated in Chapter 4 and evaluated in Chapter 5.

Below, I explore how well Faded Parsons Problems satisfy each design goal.

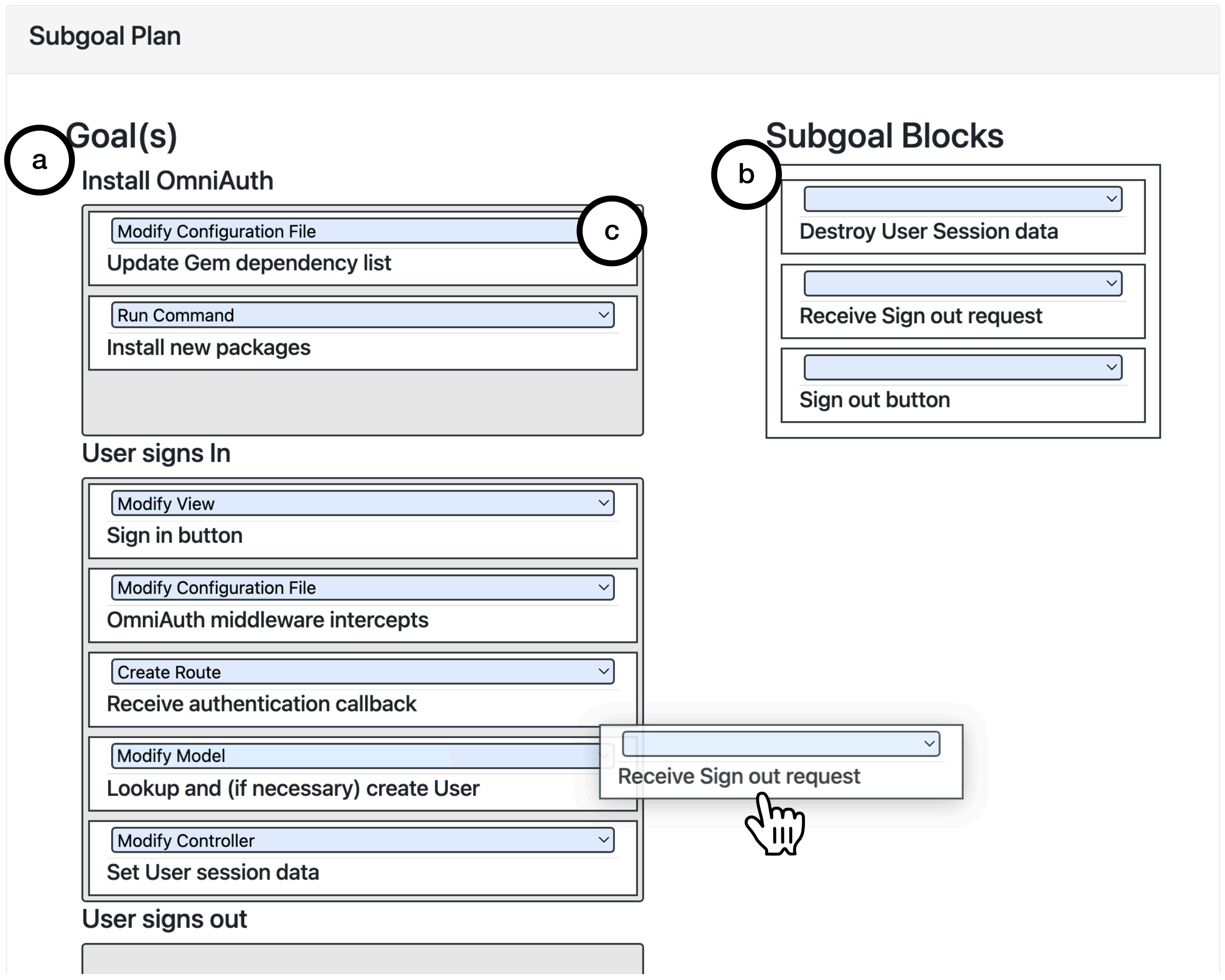
- *Provide Opportunity to Practice:* High adherence. Similar to Parsons Problems, Faded Parsons Problems correlate more closely with Code Writing exercises than Code Tracing exercises (Weinman, Fox, and M. A. Hearst, 2021). In Faded Parsons Problems, students must also complete the blanks, which might require reasoning about the relevant variable or constructing the correct syntax to achieve a goal.
- *Scaffold Vocabulary:* High adherence. Faded Parsons Problems provide students with partially complete lines of code. As long as not too much of the line is left blank, it still scaffolds most of the code used by the final solution. Similar to Parsons Problems, they scaffold syntactic knowledge.
- *Scaffold Process:* Medium adherence. Similar to Parsons Problems, it is possible that these exercises encourage students to take a top-down approach.
- *Guide Students to One Solution:* High adherence. Similar to Parsons Problems, Faded Parsons also guarantee a particular solution by providing code fragments to students, though it depends on which parts of the lines are admitted. For example, a Faded Parsons Problem might guarantee an algorithmic approach, but allow students to create their own individual variable names. Faded Parsons Problems can guarantee that students are being exposed to well-structured solutions and can target specific learning goals (e.g., the use of a particular syntax feature or algorithm) (Weinman, Fox, and M. A. Hearst, 2021).
- *Present Examples Efficiently:* Unknown adherence. Faded Parsons Problems have not yet been tested for efficiency, but the efficiency would likely be influenced by the lines of code given as well as the nature of the omitted code that students must complete.

2.2.5 Subgoal Decomposition

Subgoal Decomposition exercises (Figure 2.6, Section 6.4.1) were initially designed to be used in a software engineering course that teaches web programming and architecture. In these exercises, students are given a problem prompt, a list of goals, and a list of subgoals. Students must arrange the subgoals into the correct goal as well as order each subgoal based

[2] https://runestone.academy/ns/books/published/overview/Assessments/parsons.html

[3] https://prairielearn.readthedocs.io/en/latest/elements/#pl-order-blocks-element


Subgoal Plan

Goal(s)

Install OmniAuth

Modify Configuration File
Update Gem dependency list

Run Command
Install new packages

User signs In

Modify View
Sign in button

Modify Configuration File
OmniAuth middleware intercepts

Create Route
Receive authentication callback

Modify Model
Lookup and (if necessary) create User

Modify Controller
Set User session data

User signs out

Subgoal Blocks

Destroy User Session data

Receive Sign out request

Sign out button

Receive Sign out request

Figure 2.6: **A mostly solved Subgoal Decomposition exercise.** (a) Students are given a list of goals, each of which begin empty. (b) Students drag Subgoal Blocks from the right into the appropriate goal on the left, ordered within each goal by execution flow. (c) Students also specify where in the Rails Model-View-Controller framework they would modify code.

on the execution flow within the goal. Additionally, students must specify where in the Rails Model-View-Controller framework they would modify code to implement each subgoal.

Below, I explore how well Subgoal Decomposition exercises satisfy each design goal.

- *Provide Opportunity to Practice:* Unknown adherence. Subgoal Decomposition exercises provide a list of concrete subgoals, allowing students to practice how those subgoals work together to achieve a goal as well as their understanding of the boundaries in the Rails Model-View-Controller framework. This exercise also forces students to practice explicitly using subgoals as part of their problem decomposition process. Participant feedback suggests that this exercise helps students understand how to use the Model-View-Controller framework and the "big picture" of how to satisfy the problem prompt (Section 6.6.1). While this suggests adherence, actual learning gains have not yet been assessed.

- *Scaffold Vocabulary:* High Adherence. Subgoal Decomposition exercises provide subgoals to students as well as an extended list of where they will modify code within the Model-View-Controller framework.

- *Scaffold Process:* High adherence. Subgoal Decomposition exercises are complete when students have correctly placed each subgoal and specified where code would be modified. This has two implications for scaffolding process. First, the scope of this exercise does not include any part of the process beyond subgoals; that is, this exercise does not allow practice for parts of the process like code writing or debugging. As this exercise targets a particular part of the problem decomposition process, instructors should be sure to complement it with other exercises (e.g., Faded Parsons Problems, Code Writing exercises). Second, this exercise requires students to reconstruct a solution at the level of abstraction of subgoals. This exercise adds an explicit check of student understanding at an intermediate stage, which otherwise students may skip over entirely.

- *Guide Students to One Solution:* High adherence. Subgoal Decomposition exercises solve problems at the level of subgoals, not code. Because these subgoals are created by the instructor and are ordered by execution flow, there is a single correct solution. Since the subgoals are at a higher level of abstraction than code, however, students could construct countless different implementations from the same subgoal solution.

- *Present Examples Efficiently:* High adherence. Subgoal Decomposition exercises have not been compared directly to Code Writing exercises. However, students reported spending hours on each Code Writing exercise in the course, but were able to complete a Subgoal Decomposition exercise in under 20 minutes (Section 6.6.2). Additionally, students self-reported that they expected to take less time on a Code Writing prompt after working on this exercise, even if they included the time it took to complete this exercise.

2.2.6 Data Flow

Data Flow exercises (Figure 2.7, Section 6.4.2) were also initially designed to be used in a software engineering course that teaches web programming and architecture, as a follow-up to the Subgoal Decomposition exercise. Data Flow exercises were designed to ground the abstract subgoals from the Subgoal Decomposition exercise, focusing on how data flows through the subgoals to eventually achieve the desired effect. In these exercises, students are given a problem prompt and an already ordered list of subgoals. Students are also given a set of Data Block code snippets representing data access and modification (e.g., function arguments, instance variables) along with another set of Connector code snippets that represent how a programmer instructs the framework to move from one component to the next (e.g., function calls, route specifications). In this exercise, students must specify any Data Blocks that are *necessary* or *produced by* each subgoal from. Additionally, they must correctly place Connectors between the appropriate subgoals.

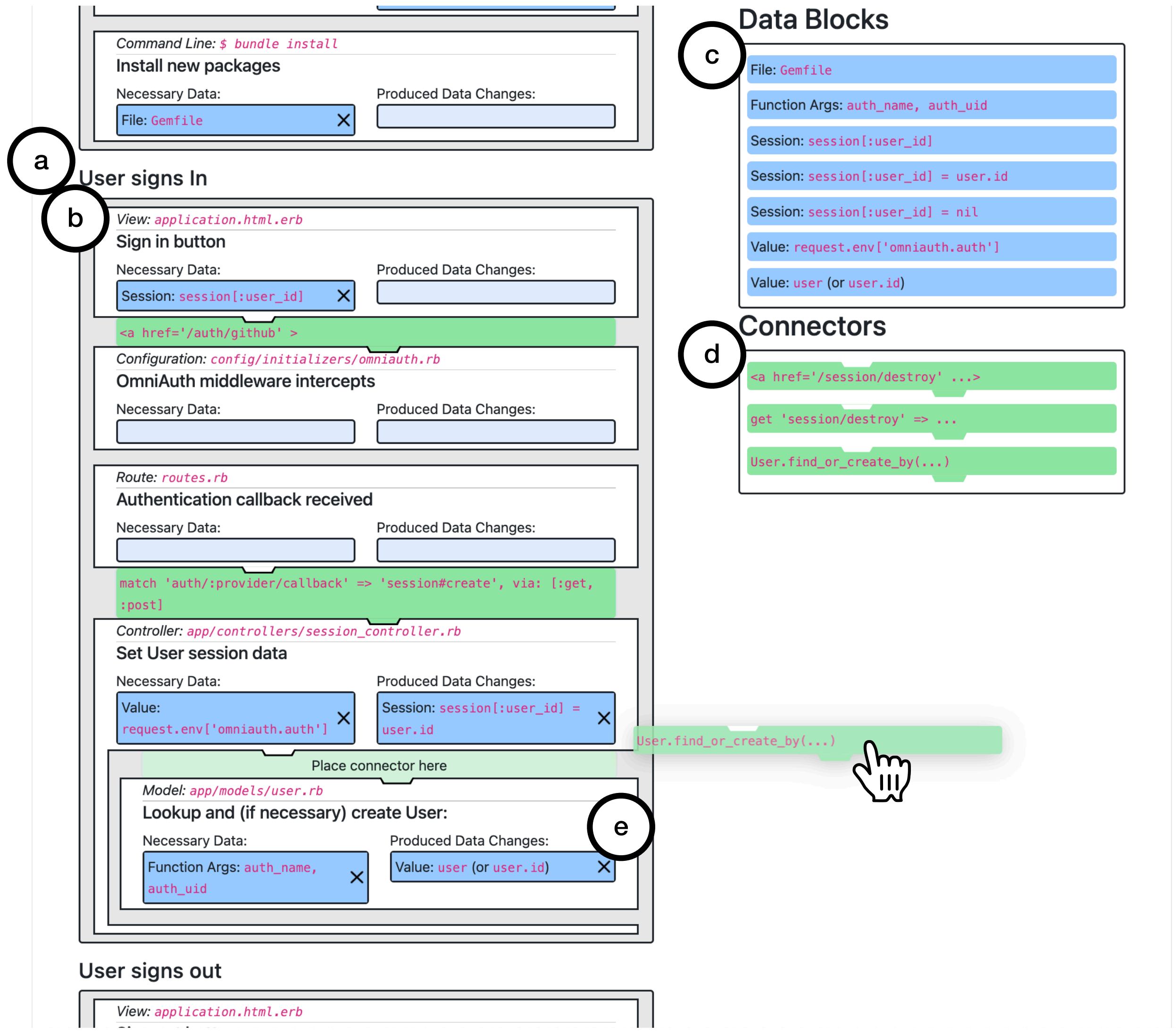
Below, I explore how well Subgoal Decomposition exercises satisfy each design goal.

- *Provide Opportunity to Practice:* Unknown adherence. Data Flow exercises provide an ordered list of concrete subgoals, allowing students to practice how data flows through these subgoals as well as their understanding of how syntactic conventions work with the framework to ensure code execution follows the intended path. Participant feedback suggests that this exercise helps students understand particular conventions of the Rails Model-View-Controller framework and further concertizes how the "big picture" satisfies the problem prompt (Section 6.6.1). However, while this suggests adherence, actual learning gains have not yet been assessed.

- *Scaffold Vocabulary:* High adherence. Data Flow exercises provide Data Block and Connector code snippets to students. These exercises provide examples of well-styled syntax, though students don't construct this syntax themselves, similar to Code Tracing exercises.

- *Scaffold Process:* High adherence. Similar to Subgoal Decomposition exercises, Data Flow exercises target a specific part of the problem decomposition process that students might otherwise skip. This exercise makes explicit where each key piece of data is read, created, or modified, as well as ensuring that each code segment across different parts of the framework will be connected as intended.

- *Guide Students to One Solution:* High adherence. Data Flow exercises provide students with an already complete list of well-separated subgoals. Given that list, there is a single correct solution to how each subgoal interacts with each piece of data for a program that follows the *conventions* of the Rails Model-View-Controller framework. While this exercise provides concrete code snippets, many implementation details are not covered by this exercise, so students could still construct countless different implementations from the same Data Flow solution.



Command Line: `$ bundle install`
Install new packages
Necessary Data: File: `Gemfile`
Produced Data Changes:

User signs In

View: `application.html.erb`
Sign in button
Necessary Data: Session: `session[:user_id]`
Produced Data Changes:

`<a href='/auth/github' >`

Configuration: `config/initializers/omniauth.rb`
OmniAuth middleware intercepts
Necessary Data:
Produced Data Changes:

Route: `routes.rb`
Authentication callback received
Necessary Data:
Produced Data Changes:

```
match 'auth/:provider/callback' => 'session#create', via: [:get,
:post]
```

Controller: `app/controllers/session_controller.rb`
Set User session data
Necessary Data: Value: `request.env['omniauth.auth']`
Produced Data Changes: Session: `session[:user_id] = user.id`

Place connector here

`User.find_or_create_by(...)`

Model: `app/models/user.rb`
Lookup and (if necessary) create User:
Necessary Data: Function Args: `auth_name, auth_uid`
Produced Data Changes: Value: `user` (or `user.id`)

User signs out

View: `application.html.erb`

Data Blocks

File: `Gemfile`
Function Args: `auth_name, auth_uid`
Session: `session[:user_id]`
Session: `session[:user_id] = user.id`
Session: `session[:user_id] = nil`
Value: `request.env['omniauth.auth']`
Value: `user` (or `user.id`)

Connectors

`<a href='/session/destroy' ...>`
`get 'session/destroy' => ...`
`User.find_or_create_by(...)`

Figure 2.7: **A mostly solved Data Flow exercise.** (a) Students are given the a list of goals and subgoals similar to Subgoal Decomposition exercises. (b) Each subgoal specifies the file and location in the framework where its code belongs. There are containers in which students can drop Data Blocks to specify necessary data and produced data changes by each subgoal. (c) Students must correctly drag Data Blocks into the correct subgoals. Data Blocks can be used multiple times. (d) Students must also drag Connectors between subgoals where appropriate. (e) Students can remove any incorrectly placed Data Blocks by pressing an X button.

	Code Tracing	Code Writing	Parsons Problem	Faded Parsons	Subgoal Decomposition	Data Flow
Provide Oppt. to Practice	Low	High	Medium	High	Unknown	Unknown
Scaffold Vocabulary	High	Low	High	High	High	High
Scaffold Process	Low	Low	Medium	Medium	High	High
Guide Students to One Solution	Medium	Low	High	High	High	High
Present Examples Efficiently	High	Medium	High	Unknown	High	High

Table 2.1: Design Goal Adherence by Exercise

- *Present Examples Efficiently:* High adherence. Similar to the Subgoal Decomposition exercises, Data Flow exercises have not been compared directly to Code Writing exercises. More than half of students were able to complete a Subgoal Decomposition exercise in tens of minutes (Section 6.6.2). Additionally, students self-reported that they expected to take less time on a Code Writing prompt after working on this exercise, even if they included the time it took to complete this exercise.

2.3 Discussion

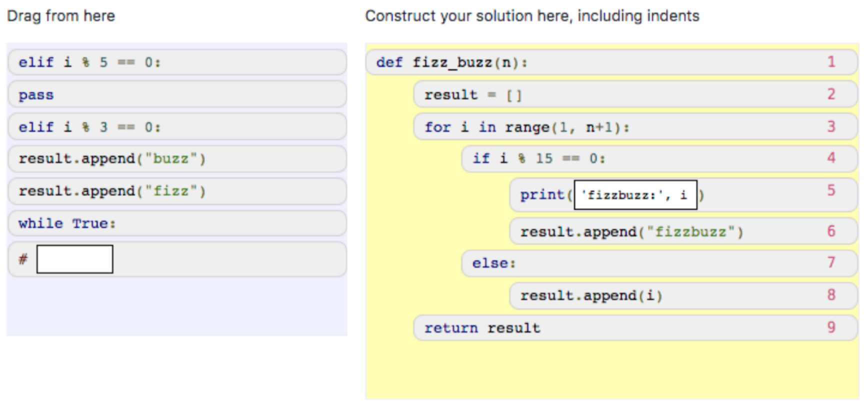
This chapter proposes five design goals to motivate the creation of exercises that teach programming patterns. It also explores how six different exercises adhere to each of the design goals. A summary of the exercise adherence can be seen in Table 2.1. These design goals are meant as motivation to evaluate and create new exercises, representing one perspective on how exercises can be used to teach patterns. The following chapters describe the work from which these design goals emerged.

Chapter 3

Taking Parsons Problems Beyond CS2

In this section, I discuss my initial structured exploration into Parsons Problems. Given the success of Parsons Problems in CS1 contexts, I set out to explore their effectiveness for upper-division students. In this between-subject study, we used Parsons Problems (Figure 3.1) for more complex concepts (e.g., topological sort, Tries) as well as complex programs (up to 28 lines of code to rearrange across up to four functions or methods).

This chapter is structured slightly differently from the following as it was preliminary work that laid the groundwork for the approach and results of Chapter 4. This study provided early motivation that Parsons Problems had some limitations, particularly when used as an exercise for upper-division students.



```
elif i % 5 == 0:
pass
elif i % 3 == 0:
result.append("buzz")
result.append("fizz")
while True:
# 
```

```
def fizz_buzz(n):
    result = []
    for i in range(1, n+1):
        if i % 15 == 0:
            print('fizzbuzz:', i)
            result.append("fizzbuzz")
        else:
            result.append(i)
    return result
```

Figure 3.1: **A partially complete Parsons Problem** of fizzbuzz, an easy problem used to demonstrate the system. Unlike other Parsons Problem interfaces, this one introduced blanks so that students could use comments to organize their code and print statements to debug it. Though certain lines of code do allow text input, these are not Faded Parsons Problems, which are introduced in the next Chapter.

3.1 Methodology

I recruited 84 student volunteers from UC Berkeley through a forum post on Facebook. Participants were required to have taken CS2 or an equivalent course. The study was run as part of a Software Engineering (SWE) Interview Preparation workshop. Participants were not monetarily compensated for their participation.

The study ran over 4 weeks, each week consisting of a 90 minute session. There were two tracks of the study running in parallel on different days of the week (treatment and control), and participants were required to say within a single track. During weeks 2 and 3, participants in the treatment group practiced with Parsons Problems, while participants in the control group practiced with Code Writing exercises. The first 15 minutes and final 15 minutes of each session were mini-presentations around preparing for SWE interviews. The remaining 60 minutes were dedicated to solving problems as part of the study. To ensure both groups had as similar experiences as possible, all presentation was done with a verbatim script (excluding participant questions) and identical materials. Any mistakes (e.g., typos in the slides) made in the earlier session were repeated in the later session.

In this study, I focused on Depth-First Search, Topological Sort, Tries, and Huffman Coding, all topics typically taught after CS2 or late in a CS2 curriculum. Problems involved extending these algorithms in interesting ways. Problems were designed to be numerous and difficult enough that participants would not finish them within the time limit; that is, all participants, regardless of speed, were intended to have a full 60 minutes of practice.

3.1.1 Weeks 1 and 4: Pre-/Post-Test

In the first and last week, participants worked through identical problems as a pre- and post-test. After being introduced to the system, participants had 10 minutes to answer multiple choice and short answer comprehension questions about the four algorithms covered in the study. As an example, one of these questions asked participants to report on the DFS traversal order of a given graph. After this, participants had 45 minutes to work through up to five coding questions. For each of these questions, participants were asked to write their approach in English along with code to solve the problem. They were allowed to use outside resources. However, they were not allowed to run their code and were not given test cases.

Participants in the treatment and control groups had the exact same experience for weeks 1 and 4.

3.1.2 Weeks 2 and 3: Practice

In the two middle weeks, participants worked on a series of exercises designed to help them practice the four algorithms used in this study. In these sessions, participants had access to an autograder to validate if their solution was correct. The autograder provided the sample input, expected output, actual output, and print statements for every test case. In addition, questions for each algorithm were designed in a sequence to get progressively more challenging.

Each practice session lasted 60 minutes. In each week, participants worked on a maximum of five problems; three for the first algorithm of the week, two for the second algorithm. Participants were able to move on the next question after working on it for 15 minutes or passing all test cases. After 40 minutes, all participants were forced to move onto the fourth problem of the week (the first problem of the second algorithm).

Participants in the control group worked with a standard coding interface both weeks. Participants in the treatment group worked with a Parsons Problems interface both weeks, including print statements and comments. Because the Parsons Problems were 11-28 lines long and consisting of up to four functions or methods, I decided to not use any distractors (i.e., lines of code that would not be used in the solution). At the time of running this study, studies that used distractors with Parsons Problems primarily used *syntactic* distractors (e.g., missing a colon at the end of a control flow), which I believed would not be interesting for upper-division students. Additionally, I was concerned the problems might already have enough lines of code to overload participants, so I did not want to add additional unnecessary lines of code.

3.2 Results

3.2.1 Limitations

There were two major limitations that prevented me from presenting any strong conclusions from this study. First, there was significant attrition in the study (Figure 3.2). Though the study began with 84 participants, that number dwindled to 19 participants by the end of Week 4. This prevented me from reporting on any statistically significant results between the pre-test and post-test. However, I did have 64 participants during Week 2, so can still conduct some analysis comparing the practice of Parsons Problems and Code Writing exercises.

Second, even if I had not lost many participants, the pre-test and post-test were not well designed to find learning differences. In general, when one sees neutral learning gain results, it's possible that participants did not learn *or* that the tests did not assess what participants were actually learning. The pre-test and post-test consisted only of straightforward questions and challenging Code Writing questions. This study would have benefited from problems of varying difficulty, which could have better distinguished participants. Despite these limitations, this study still provided some valuable insights that motivated work described later in this dissertation.

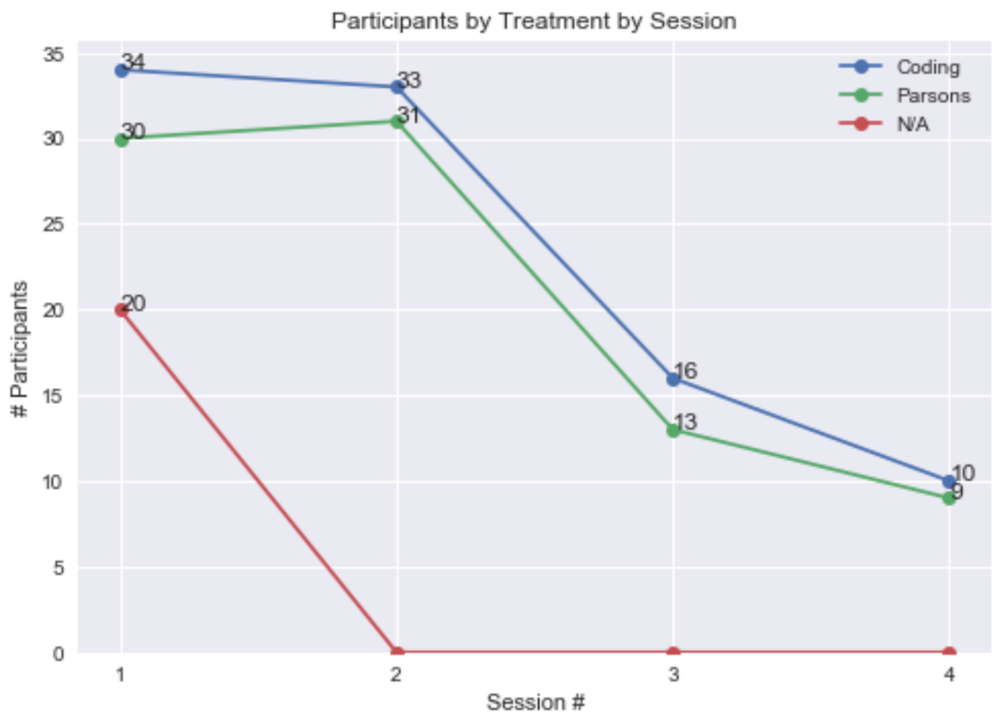


Figure 3.2: **The weekly number of participants by treatment condition.** The N/A condition represents participants that dropped out before week 2, and therefore were never exposed to treatment or control conditions

Table 3.1: **Efficiency by Interface and Week.** Efficiency is reported as a percentage change of Parsons Problems compared to Code Writing. * indicates $p < .01$

	Code Writing	Parsons Problems	Efficiency
Week 2			
Attempted	3.36*	4.23*	20.6%
Solved	.94*	2.32*	59.5%
Progress	7.36*	10.548*	30.2%
Week 3			
Attempted	2.63*	4.54*	42.1%
Solved	0.94*	3.62*	74.0%
Progress	5.75*	12.69*	54.7%

3.2.2 Efficiency of Parsons Problems

Ericson, Margulieux, et al. (2017) found that CS1 students worked through Parsons Problems in approximately 70% of the time as Code Writing exercises. That is, Parsons Problems were 30% more efficient to work with than Code Writing exercises. In that study, all participants worked through the same set of questions, but took varying amounts of time to complete it. Additionally, participants had a fixed amount of time (1 hour), were able to work through as many problems as they can. Finally, participants were allowed to move on from a question after 15 minutes even if they had not solved it.

I report on the efficiency with which participants solved Parsons Problems compared to Code Writing exercises through three metrics collected from Weeks 2 and 3. Due to the significant drop-off in participation, we analyze Week 2 and Week 3 separately. First, I report on the number of **attempted** problems, which ranges from a minimum of two to a maximum of five by design. All participants worked on the first problem of each algorithm (first and fourth problem in the weekly sequence) and had a maximum of five problems to work through each week. Second, I report on the number of correctly **solved** problems, ranging from a minimum of zero to a maximum of five. Since participants could move on from a problem after working on it for 15 minutes, some participants may have attempted all five problems but solved none of them. Finally, II report on an aggregate **progress** metric by combining these metrics into a single score; for each question, a participant receives a score of 3 if they solve the question correctly, a score of 2 if they moved on from the question (i.e., spent 15 or more minutes on it), a score of 1 if they attempted the question at all (i.e., spent less than 15 minutes on it), and a score of 0 otherwise. These point values are summed for each participant. The results of this analysis can be seen in Table 3.1. Statistical significance was assessed using a Mann-Whitney U Test.

3.3 Discussion

The reported values suggested that, for upper-division students, Parsons Problems might be meaningfully more efficient than the 30% efficiency previously reported (Ericson, Margulieux, et al., 2017). In short answer responses, participants noted that Parsons Problems “helped [them] think of ideas for some of the problems where [they] initially felt a little stuck” and “gave hints about the correct way of coding up the solution...allowing [them] to focus more on the conceptual part”. However, despite these advantages and that the treatment group was working through many more problems, I did not observe any changes in learning gains between Weeks 1 and 4. While this could have been the fault of my own measurements (as mentioned above), it also might have been because participants were solving Parsons Problems in ways that were less beneficial to learning. At least one participant “was able to guess [their] way through using the code blocks - looking at the variable names, [they] could deduce which statements belonged to which functions”. That is, Parsons Problems give strong contextual hints, like edge pieces in a jigsaw puzzle. In fact, some of the more challenging exercises in the study could be nearly entirely solved simply by focusing on data flow and recognizing initialization statements. It also made sense that this concern might not be as apparent in a CS1 context, as students needed a level of expertise and patterns to be able to confidently and efficiently take advantage of contextual hints. While I could have run a similar study to get more concrete results, this highlighted an opportunity to adapt Parsons Problems to remove these edge pieces, motivating the creation of Faded Parsons Problems.

Chapter 4

Exploring Challenging Variations of Parsons Problems

Though the previous study did not lead to as concrete results as I had hoped, it suggested that the efficacy of Parsons Problems might be limited for students that could take advantage of syntactic heuristics. This study was intended to better understand whether that limitation was present, as well as introduce Faded Parsons Problems as an attempt to navigate around the limitation. In addition to being able to obfuscate some syntactic hints by removing them from the given lines of code, Faded Parsons Problems also offered a more direct opportunity for automatically converting existing Code Writing exercises. For example, an Abstract Syntax Tree parser could find all variable references and automatically remove them from the instructor solution to create a Faded Parsons Problem. An in-lab study with 13 students found that both types of Parsons Problems are effective at teaching solution structures compared to Code Writing exercises, and that solving standard Parsons Problems don't necessarily lead to an understanding of the solution.[1]

4.1 Introduction

Learning to successfully write code to solve a task is complex. Both du Boulay (1986) and Cutts et al. (2012) argue that programmers move among multiple domains and levels of abstraction while solving programming problems. CS1 classes typically use code-reading exercises, such as worked examples and code tracing questions, to help scaffold students into code writing. However, even with these scaffolds, a large gap remains between code-reading mastery and code-writing mastery.

[1]The condensed version of this work was published as a poster (Weinman, Fox, and M. Hearst, 2020) and demo (Weinman, Hsu, et al., 2020) at SIGCSE 2020. This work was criticized for the duration of the treatment. The treatment lasted one hour. The critics argued that this was too short to impact the intended study goals. Despite this limitation, I still believe there are some interesting findings to this study, so the original paper submission is included here.

The learning sciences suggest that students benefit from tackling problems in their *Zone of Proximal Development* (ZPD), or problems that are solvable but only with guidance or scaffolding (Berk et al., 1995). Parsons Problems (Parsons et al., 2006), in which students unscramble provided lines of code, are an example of such scaffolding: unlike code tracing and worked examples, they provide an opportunity for students to construct solutions, but unlike code writing, the solution space is much more limited and a lighter mastery of language syntax is required. Ericson, Margulieux, et al. (2017) found Parsons Problems to be faster than code writing exercises while producing similar learning gains in CS1 classrooms.

However, Denny et al. (2008) raise concerns that students can "game" Parsons Problems, or make progress while avoiding learning. "Gaming" behavior is correlated with missed learning gains in Intelligent Tutoring Systems (Baker et al., 2004). This chapter introduces a more challenging and less "gameable" variant of Parsons Problems, **Faded Parsons Problems**, in which the provided lines of code can be partially blank or incomplete. This provides a tunable opportunity to scale the difficulty of Parsons Problem exercises by adjusting the proportion of incompleteness in the given code.

We explore the efficacy of Faded Parsons Problems in a CS1 context. We run a study on current CS1 students focused on *Multiple Recursion*: recursive functions that contain multiple self-references. This topic was selected because instructors and previous students reported it as one of the more challenging topics. We introduce and study **Blank-Variable Parsons Problems**, in which all variable names are removed from the provided lines. This instantiation of Faded Parsons Problems requires no extra work for instructors to create compared to Parsons Problems and removes a class of syntactic cues present in the provided lines. This work compares Blank-Variable Parsons to code writing and to Parsons Problems (without distractors), making the following three contributions:

1. We introduce Faded Parsons Problems and explore Blank-Variable Parsons as a particular instantiation.
2. We demonstrate that both types of Parsons Problems provide opportunities to achieve different learning objectives than code writing.
3. We demonstrate that Blank-Variable Parsons fall in difficulty between standard Parsons Problems and code writing.

After reviewing existing related research comparing Parsons Problems to code tracing and code writing, we describe our Faded Parsons Problems and present the design, results, and potential implications of a study of their effects on CS1 students.

4.2 Related Work

The learning sciences suggest several helpful teaching techniques, including *faded scaffolding*. Faded scaffolding incrementally removes scaffolding structures to gradually make problems more challenging. This allows instructors to design a series of increasingly difficult problems that remain inside a students' Zone of Proximal Development as they acquire new knowledge. Cognitive Load Theory (CLT) proposes that students have limited mental capacities when working on problems (Sweller, 1988). Faded scaffolding is also supported by CLT, as it limits the new knowledge that must be integrated by the student compared to fully removing the scaffolding. Faded scaffolding has been found as an effective technique in a range of domains (Renkl et al., 2002).

Parsons Problems (or Parsons Puzzles, code scrambles, Code Mangler problems) were originally designed to be an engaging task for students to practice syntax drills (Parsons et al., 2006). Several studies have found that the skills required to solve Parsons Problems lie between those of code tracing and code writing (Lopez et al., 2008; Zavala et al., 2017). Lopez et al. (2008) found that code tracing questions and Parsons Problems on an exam required less mastery to solve than code writing questions. Zavala et al. (2017) further separated these and found support that students should master code comprehension followed by code manipulation as precursor skills to writing code.

Parsons Problems have also successfully been integrated into teaching. Ericson, Margulieux, et al. (2017) found that Parsons Problems, when combined with worked examples, provide similar learning gains to code writing or bug detection in a CS1 classroom while taking only 70% of the time. Harms, Rowlett, et al. (2015) compared Parsons Problems to a guided tutorial with students replicating code in Looking Glass[2], a block-based programming environment to author 3D animations. They found that students using Parsons Problems did 26% better on transfer tasks despite taking less time.

A common approach to adjust the difficulty of Parsons Problems is with distractors (additional lines of code that do not fit in the solution), though it can be challenging and time-consuming to generate them. Ericson, Foley, et al. (2018) explored Adaptive Parsons Problems, which improve learning speed by modulating difficulty with arrangement and number of hand-crafted distractors. In contrast, Harms, Chen, et al. (2016) found evidence that certain logical distractors are harmful to learning with Parsons Problems.

A downside suggested by Denny et al. (2008) is that students can use certain syntactic heuristics, such as data flow dependencies between variables or placing return statements at the end of functions, as hints that Parsons Problems unavoidably provide. This "gaming", if present, could become more problematic as students gain more mastery over common programming patterns to leverage as hints. Faded Parsons problems offer a new way to adjust the difficulty of Parsons Problems. In addition to offering different ways of fading the difficulty, they also address Denny et al.'s concern by allowing instructors to remove these syntactic heuristics. For example, Blank-Variable Parsons was designed to obscure data flow dependencies between variables.

There is previous work combining blanks with Parsons Problems. Garner (2007) explored removing full lines of code from Parsons Problems. Zhi et al. (2019) explored Parsons Problems with Snap! and allowed user input for constants, which are individual fragments

[2]https://lookingglass.wustl.edu/

in this programming language. Ihantola, Helminen, et al. (2013) designed an interface where parts of code lines, such as comparator operators, had to be selected from pre-configured choices, but did not measure its effectiveness. Unlike these works, Faded Parsons Problems encourage arbitrary user input as *part* of a fragment (e.g. a code line in Python). This chapter also differs by directly comparing standard Parsons Problems to Blank-Variable Parsons on the same questions.

4.3 Implementing Blank-Variable Parsons

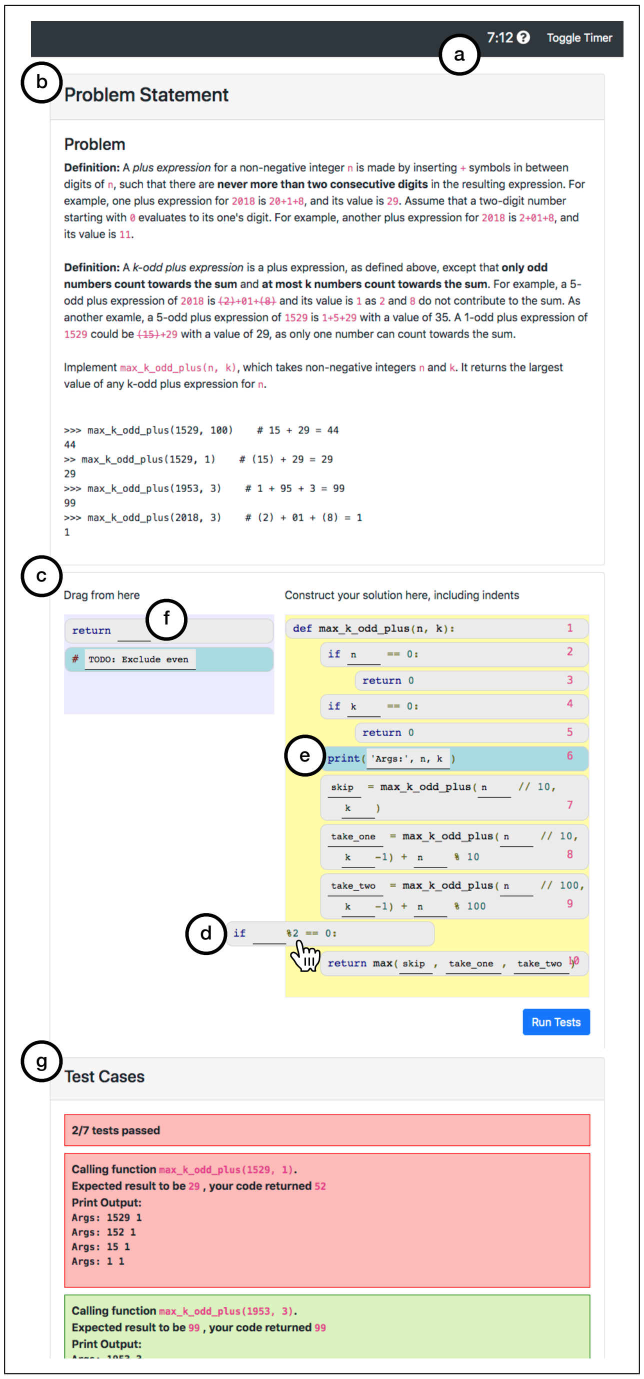
We built a Flask app that extends Karavirta et al.'s js-parsons library (Ihantola and Karavirta, 2010) to support Blank-Variable Parsons Problems, as well as Faded Parsons Problems more generally. The system supports Python programming exercises—traditional code writing, standard Parson Problems, and Blank-Variable Parsons—as well as survey questions (multiple choice and short answer questions). A nearly complete Blank-Variable Parsons exercise can be seen in Figure 4.1(c). In Parson Problem exercises, the user is initially given a set of blocks containing code on the left including optional print and comment statements (e) for debugging, with the initial function signature populated on the right. The lines on the left are initially alphabetized, as suggested by Cheng et al. (2017). To solve a Parsons Problem, students drag fragments in a correct order and indentation on the right. In Blank-Variable Parsons exercises, some blocks will have lines indicating omitted code and support for text input (f), similar to a fill-in-the-blank question an an exam. The code writing interface can be seen in Figure 4.2.

Exercises in any of these interfaces display the current time spent on a problem to participants (a), followed by the problem statement (b). Participants can run pre-configured tests as often as they want, which displays detailed output from the test cases (g) including: function arguments, expected output, actual output, any standard output from print statements, and any raised exceptions. The type of feedback is consistent across interfaces. Unlike some other work with Parsons Problems, no line-level placement feedback is given (Helminen et al., 2012; Ericson, Guzdial, et al., 2015).

The Flask app logs anonymized data from participants, enforces time limits, and randomizes treatment selection. The autograder is a separate worker that uses RQ[3] to safely execute arbitrary Python code. We manually selected the blanks in Blank-Variable Parsons, but with minor modifications, the system could do this automatically from a reference solution using Python's AST library[4].

[3]https://python-rq.org/

[4]https://docs.python.org/3.7/library/ast.html



Problem Statement

Problem

Definition: A *plus expression* for a non-negative integer `n` is made by inserting `+` symbols in between digits of `n`, such that there are **never more than two consecutive digits** in the resulting expression. For example, one plus expression for `2018` is `20+1+8`, and its value is `29`. Assume that a two-digit number starting with `0` evaluates to its one's digit. For example, another plus expression for `2018` is `2+01+8`, and its value is `11`.

Definition: A *k-odd plus expression* is a plus expression, as defined above, except that **only odd numbers count towards the sum** and **at most k numbers count towards the sum**. For example, a 5-odd plus expression of `2018` is `~~(2)~~+01+~~(8)~~` and its value is `1` as `2` and `8` do not contribute to the sum. As another examle, a 5-odd plus expression of `1529` is `1+5+29` with a value of 35. A 1-odd plus expression of `1529` could be `~~(15)~~+29` with a value of 29, as only one number can count towards the sum.

Implement `max_k_odd_plus(n, k)`, which takes non-negative integers `n` and `k`. It returns the largest value of any k-odd plus expression for `n`.

```
>>> max_k_odd_plus(1529, 100)    # 15 + 29 = 44
44
>> max_k_odd_plus(1529, 1)    # (15) + 29 = 29
29
>>> max_k_odd_plus(1953, 3)    # 1 + 95 + 3 = 99
99
>>> max_k_odd_plus(2018, 3)    # (2) + 01 + (8) = 1
1
```

Drag from here

```
return ____
# TODO: Exclude even
```

Construct your solution here, including indents

```
def max_k_odd_plus(n, k):
    if n     == 0:
        return 0
    if k     == 0:
        return 0
    print( 'Args:', n, k )
    skip   = max_k_odd_plus( n     // 10,
      k     )
    take_one   = max_k_odd_plus( n     // 10,
      k    -1) + n     % 10
    take_two   = max_k_odd_plus( n     // 100,
      k    -1) + n     % 100
if ____ %2 == 0:
    return max( skip , take_one , take_two )
```

Test Cases

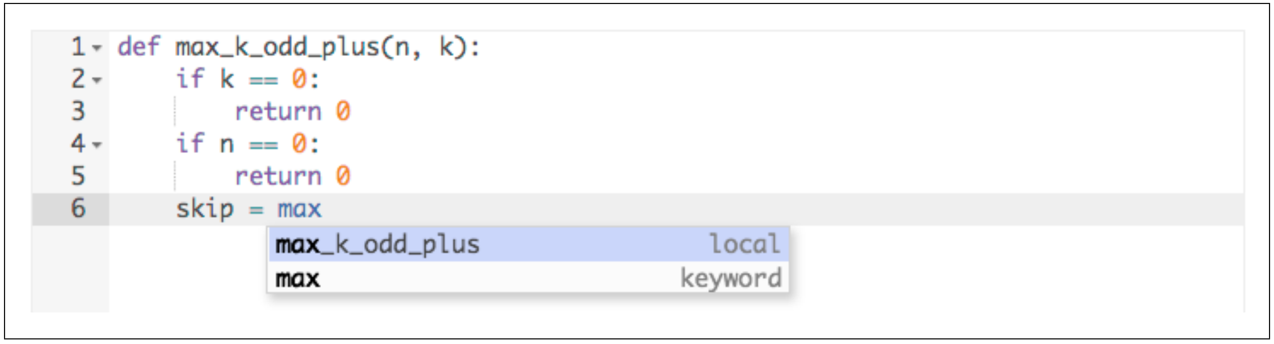
2/7 tests passed

Calling function `max_k_odd_plus(1529, 1)`.
Expected result to be `29` **, your code returned** `52`
Print Output:

```
Args: 1529 1
Args: 152 1
Args: 15 1
Args: 1 1
```

Calling function `max_k_odd_plus(1953, 3)`.
Expected result to be `99` **, your code returned** `99`
Print Output:

Figure 4.1: **A nearly correct Blank-Variable Parsons exercise of k-Odd Plusses.** (a) Timer (b) Problem Description (c) Parsons Problem interface, participants can drag blocks between the bin (left) and solution (right) (d) A block being dragged to the right (e) Optional print block (f) Blanks for Faded Parsons Problems (g) Test case results.



```
def max_k_odd_plus(n, k):
    if k == 0:
        return 0
    if n == 0:
        return 0
    skip = max
```

Figure 4.2: **A Code Writing exercise.** It supports basic IDE functionality, such as syntax highlighting and autocomplete.

4.4 Study Design

We ran a study in the summer of 2019 at a large US university. All participants were currently enrolled in a CS1 class, in week 5 of an 8 week course, and were recruited through a post on a class forum and in-person announcement. 13 participants (11 Male) were compensated monetarily for their time and offered a chance to review a topic covered in their CS1 class. The researchers were not course instructors.

To calibrate the difficulty of the questions and smoke-test the system design, two CS1 students were recruited to test a pilot version of the system. Both participants worked through two Multiple Recursion problems in a range of interfaces. They were able to solve both questions in well under the expected time, so the questions were modified to be slightly more challenging in the larger study. Based on their performance, we decided to include variable names in the function signature to match the problem description and reduce unnecessary confusion for Blank-Variable Parsons. Due to practical constraints the new questions were not piloted, and results indicate they were quite challenging in difficulty.

4.4.1 Study Materials

Programming Questions

All three programming questions were designed to be representative of those in a typical CS1 class. The first and last problem of the study was *k-Odd Plusses*, which can be seen with a nearly complete solution in Figure 4.1. The second problem was *Fizz Buzz*[5]. We include Fizz Buzz as an easier question. The third problem was *Coin Game*, determining the winner of a game of Nim[6] with only 2 rows.

[5]https://www.hackerrank.com/challenges/fizzbuzz/problem
[6]https://www.hackerrank.com/challenges/nim-game-1/problem

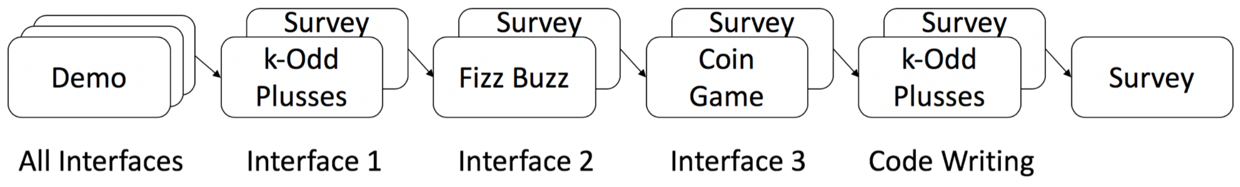


Figure 4.3: **Study Design.** After being introduced to each exercise type, participants worked through three problems in a fixed order but with their own randomized interface ordering. Finally, participants worked on the first problem again as a Code Writing exercise before completing survey questions.

Post-Exercise Survey Questions

After each exercise, participants were asked to rate the Mental Demand and Effort expended on a 1-7 scale. They were also asked if the exercise helped them to (a) learn ways others use Python, (b) learn how to solve the question they just worked on, (c) learn the topic they just worked on (Multiple Recursion, or loops for *Fizz Buzz*), and (d) learn Python Syntax. Additionally, on the final exercise, they were given a text prompt to describe any benefit from working on the same question again.

Post-Study Survey Questions

Participants were asked to rate the difficulty of solving a problem with both types of Parsons Problems compared to code writing in terms of required time and assistance. They were also given text prompts to describe (a) which interfaces they would prefer to use for homework problems to master the topics, (b) how each of the Parsons Problem interfaces could be best integrated into their class, and (c) any other comments.

4.4.2 Procedure

Figure 4.3 illustrates the study design. Each *exercise* consisted of working on a *problem* in a particular *interface*. Participants were first introduced to the system with a simple task in all three interfaces, writing a function that returns 3. They then worked through three problems in a fixed order, each exercise in one of the three interfaces described above, having 12 minutes for each exercise. The interface order was assigned randomly for each participant; each participant used each of the three interfaces once in these three exercises. The first and third questions were different Multiple Recursion questions, which were designed to be challenging, while the second question was Fizz Buzz, designed to be an easy for-loop question.

After completing these exercises by constructing a correct solution or running out of time on each, there was a final exercise that repeated the first question. The final exercise was always in a code writing interface, as code writing is a commonly used to measure student understanding of a problem in CS1. Participants were told at the start of the study that one of the questions would be repeated, but not which one.

As described in *Study Materials*, participants were asked to fill out a brief survey after each exercise about their experience with the exercise. At the end of the study session, participants were asked to complete a longer survey.

The study is designed to answer the following four research questions:

- **RQ1:** How effective is each interface at achieving specific learning objectives?
- **RQ2:** How do Blank-Variable Parsons affect the difficulty of a problem?
- **RQ3:** How does practice in each of these interfaces support short-term coding ability?
- **RQ4:** How do students feel about using Blank-Variable Parsons?

4.5 Results

4.5.1 RQ1: Varying Learning Objectives

Compared to code writing, participants found both types of Parsons Problems better for learning how others write Multiple Recursion programs but not Fizz Buzz. Recall that after each exercise, participants were asked whether that exercise helped them learn four topics: ways others use Python, how to solve the question they just worked on, Multiple Recursion (or loops), and Python Syntax. Table 4.1 shows the responses. We analyze these self-reported responses on the first three exercises, excluding the fourth as it was a repeated problem and always code writing. Furthermore, we combine the responses from the two Multiple Recursion problems. A Kruskal-Wallis test on Fizz Buzz, the easy problem, finds no significant differences between the interfaces on any of four learning topics. A Kruskal-Wallis test on the Multiple Recursion problems finds a difference between the interfaces only for learning how others use Python. Mann-Whitney U tests between each pair of interfaces found no significant difference between either type of Parsons Problems, but did find significant differences between each type of Parsons Problem and code writing. P12 noted that for both types of Parsons Problems "sometimes your idea for a solution looks very different from what is given." P6 similarly noted that, in standard code writing interfaces, they would "come up with an approach that is more complicated than necessary and miss out on simpler solutions."

While all participants wrote runnable (but often incorrect) code for the final exercise, participants who worked with either Parsons Problem interface initially were more likely to write Multiple Recursion code. After the final exercise, in which participants repeated *k-Odd Plusses* in a code writing interface, participants were asked if there was a benefit to working on the question again. P12, who originally worked on it in Blank-Variable Parsons, noted that they "had already thought at least about the base cases"

Table 4.1: **Self-reported learning measures.** Proportion of participants indicating that they learned 4 topics, by question set and interface. Multi Rec represents aggregate data from Multiple Recursion exercises. * represent p < 0.05 in-group or pairwise statistical test.

	Standard Parsons Problem	Blank-Variable Parsons	Code Writing
Learn Others Code			
Multi Rec*	.89*	.75*	.22*
Fizz Buzz	.75	.6	.75
Repeated Q	-	-	.23
Learn This Problem			
Multi Rec	.78	.5	.67
Fizz Buzz	.75	.6	1.0
Repeated Q	-	-	.46
Learn This Topic			
Multi Rec	.56	.5	.33
Fizz Buzz	1.0	.8	.75
Repeated Q	-	-	.54
Learn Python Syntax			
Multi Rec	.56	.75	.67
Fizz Buzz	1.0	.6	.75
Repeated Q	-	-	.42

and "mainly focused on the structure, not so much the algorithm itself". We mark code submissions of the final exercise, which was code writing for all participants, on whether it implements a Multiple Recursion structure. 4/7 participants who first worked on the problem as code writing (compared to 1/6 in the other groups) failed to write Multiple Recursion code, either by not writing recursive code or writing recursive code with only one self-referential call.

4.5.2 RQ2: Difficulty of Interfaces

The relative difficulty of solving problems in the three interfaces is measured in three ways.

Completion Rate of Problems

Standard Parsons Problems make problems more solvable compared to the other two interfaces. Participants ended each exercise either by running out of time or constructing a correct solution. First, we observe whether participants were able to successfully complete each question. As shown in Table 4.2, the only interface in which participants could

Table 4.2: **Problem completion rate by interface.** Completion counts over participant count by question and interface

	Standard Parsons Problem	Blank-Variable Parsons	Code Writing
k-Odd Plusses	3/3	0/3	0/7
Fizz Buzz	4/4	5/5	3/4
Coin Game	4/6	0/5	0/2

solve the Multiple Recursion questions—*k-Odd Plusses* and *Coin Game*—was the standard Parsons Problem interface, and in this case 7/9 participants solved them.

A Kruskal-Wallis test on the Multiple Recursion questions finds a difference between the interfaces (p¡.01). Two-Proportion Z-tests between each pair of interfaces indicate that standard Parsons Problems have a higher success rate than Blank-Variable Parsons (p¡.01) and code writing (p¡.01). This suggests that standard Parsons Problems, compared to Blank-Variable Parsons and code writing, move problems into a student's Zone of Proximal Development that would otherwise be too challenging in a timed environment with less scaffolding. This data shows no difference between Blank-Variable Parsons and code writing, though this may be because the questions were too challenging to separate the two.

Self-Rated Mental Demand and Effort TLX scores

No significant difference was found in difficulty-related TLX scores. To assess the participants' perceived workload of completing the exercises, they were asked to answer the NASA TLX questions after each exercise (Hart et al., 1988). We find no interesting differences along the Mental Demand and Effort dimensions. This may be because the scale on these questions were not concretely anchored, so participant responses depended on varying scales and the previous interfaces they had encountered in the study.

Self-Rated Difficulty of Interfaces

Participants perceive Parsons problems as easier than Blank-Variable Parsons, and Blank-Variable Parsons as easier than code writing. At the end of the study, participants were asked to rate the amount of time and assistance they would need to solve problems comparing each of the two Parsons type interfaces to code writing. These questions were on a Likert scale of 1 (much easier than code writing) to 5 (much harder than code writing). The average rating for Parsons Problems was 1.08, and for Blank-Variable Parsons was 1.85. Wilcoxon Signed-Rank tests compared to an expectation of 3 show that both types standard Parson Problems (p¡.01) and Blank-Variable Parsons (p¡.02) were self-rated as easier than code writing. A Wilcoxon Signed-Rank test between answers to the two questions show that Parsons Problems are self-rated as easier compared to Blank-Variable Parsons (p¡.02).

P8, the only participant who ranked Blank-Variable Parsons as more difficult, explained that “the number of blanks given means that [they] have to write [their] code as efficient as the general solution,” losing the “freedom to write inefficient code which still works.” Though this participant found Blank-Variable Parsons as more difficult than writing code, they also indicate pedagogical value attached to that difficulty.

Summary

Completion rates clearly show that problems at a certain level of difficulty are solvable as standard Parsons Problems but not as Blank-Variable Parsons or code writing exercises. Though we find no significant differences in difficulty-related TLX scores, self-rated difficulties indicate that students perceive Blank-Variable Parsons as harder than standard Parsons Problems but easier than code writing. As P10 explains, “[Blank-Variable Parsons] are similar to [standard Parsons Problems], but requires more thought on what each step is supposed to do. But at least you’re given a guiding hand on how the problem could be solved.”

4.5.3 RQ3: Short-Term Coding Mastery

Standard Parsons Problems do not immediately lead to short-term coding mastery. Participants worked on *k-Odd Plusses* first in a random interface, and then again as a code writing exercise. For each interface, we wanted to observe the short-term coding master of an identical question roughly 30 minutes later. We identify students that successfully solved the first exercise, *k-Odd Plusses*, in any interface and analyze their success on the fourth exercise, the same question in code writing. However, likely due to the difficulty of the question, all of these participants are ones that worked on the first exercise as a standard Parsons Problem as seen in Table 4.2. None of these participants were able to successfully write code for the same problem, indicating that solving standard Parsons Problems don’t immediately lead to an ability to write code for the same problem. Our data does not elucidate if this applies to Blank-Variable Parsons or code writing. Of note, the only participant to complete this exercise first attempted it as a Blank-Variable Parsons, but this single data point is not enough to draw any conclusions.

One possible explanation comes from P5, stating that, with Parsons Problems they “can just guess and check the right order” and “[they] didn’t really understand [the solution], [they] just kind of logic-ed out how the cases have to be.” Despite the “gaming” behavior, this suggests that Parsons Problems might be effective at exposing students to common logical templates.

4.5.4 RQ4: Student Affinity

Participants responded positively to Blank-Variable Parsons. At the end of the study, participants were given a text prompt for which interface(s) they would like to use for formative assessments to best prepare them for exams. Participants were free to mention as many interfaces as they wanted in this prompt. One participant did not answer the question, and one gave too vague an answer to interpret. For the remaining 11 participants, responses were coded to tally which interfaces they mentioned in a positive way. Of these 11 participants, 9 mentioned a preference for Blank-Variable Parsons, 6 for code writing, and 4 for standard Parsons Problems.

The results of RQ1 and RQ2 suggest one possible explanation as to why students find Blank-Variable Parsons so helpful; student feedback suggests that they teach new ways to solve a problem and are challenging enough to be educational.

4.6 Discussion

4.6.1 Observations/Opportunities

The results of this study suggest several immediate opportunities for integrating Blank-Variable Parsons into classrooms.

Programming requires many skills. This study explored not only whether Parsons Problems and Blank-Variable Parsons represent a teaching opportunity, but also what they teach effectively compared to code writing exercises. As Shunryu Suzuki said, “In the beginner’s mind there are many possibilities, but in the expert’s there are few” (Suzuki, 1970), and both types of Parsons Problems offer valuable opportunities to constrain the possibilities attempted by beginners to match teaching goals. P6, for example, usually would “come up with an approach that is more complicated than necessary”, but found Blank-Variable Parsons to help them “focus on the process of the program instead of several different approaches.” This study found participants better following the intended structure of Multiple Recursion after working with Parsons Problems. Instructors could also leverage Parsons Problems to, for example, ensure students are exposed to certain helpful code idioms, or make it easier for students to construct solutions with good coding habits.

Code Skeleton questions are a common CS1 exercise: students must fill in the missing parts of a partially complete program, which both provides and constrains the structure of the solution. Several students expressed sentiments similar to P7, that Blank-Variable Parsons are “similar to the structure provided in skeleton code as seen on exams.” Blank-Variable Parsons may therefore offer students an opportunity to practice questions in formative assessments that are closer to the questions that appear on exams. Both types of Parsons Problems can further be used for exam preparation as P12 expresses the speed with which they can be solved is desirable once a student “just needs to do a lot of practice problems” to prepare for an exam.

Though not directly explored in this study, Blank-Variable Parsons Problems were designed, in part, to provide a scaffold between Parsons Problems and code writing exercises. This study supports that solving a programming exercise as a Blank-Variable Parsons is, indeed, more difficult than as a Parsons Problem and less difficulty than as a code writing exercise. Feedback from two of the three participants that solved *k-Odd Plusses* as a standard Parsons Problem and then failed to solve it as a code writing exercise further highlights the need to create a scaffold between standard Parsons Problems and code writing exercises. Both of these participants mentioned that failing the code writing exercise hurt their self-efficacy, which is troubling as self-efficacy has been found to correlate with positive learning behaviors (Schunk, 1989). P5 expresses that the repeated exercise "made [them] feel more discouraged", while P2 shared that it challenged their initial impression and "that [they] did not have a solid understanding of how the problem should work."

Lastly, P13 offers a compelling suggestion to let students "try out the problems with [either type of Parsons Problem] as hints before going to the solution" when homework and exam solutions are released, possibly offering them partial credit. This could provide an opportunity to insert one more active learning opportunity before students view solutions to problems on which they failed.

4.6.2 Limitations

This study had some limitations. First, the small sample size of n=13 limits the analysis that could be done. The study was run on volunteers in a Summer session of CS1, which may make them meaningfully different than all CS1 students during the regular Fall and Spring sessions. As described above, analysis was mostly done on *challenging* Multiple Recursion questions and in a timed environment, preventing an objective completion-based difficulty analysis to separate Blank-Variable Parsons and code writing exercises. Lastly, students were unfamiliar with both Parsons Problems and Blank-Variable Parsons before starting this study, and their behavior may differ as they gain expertise.

4.6.3 Future Work

There is significant work to be done exploring other instantiations of Faded Parsons Problems. We designed Blank-Variable Parsons for this study to focus on the "gaming" concerns present in Parsons Problems while being trivial for instructors to implement from existing content. However, there are many exciting opportunities for Faded Parsons Problems beyond Blank-Variable Parsons. First, instructors could manually add blanks to target particular concepts in the solution code that need more practice. For example, if students were learning list syntax in Python, instructors could ensure all list indexing code with square brackets were left blank. Or, when introducing recursion, instructors could gradually remove how much base case code is given to help students gradually learn how to reason about base cases. Another Faded Parson Problem opportunity is leveraging old student solutions to focus on common misconceptions. One could create an algorithm to, for example, remove the top N partial code fragments covering the most common incorrect student submissions. In *k-Odd Plusses*, for example, several students returned the `max` of two values instead of 3; making the entire inside of the `max()` call blank would force students to address this mistake to get a correct solution while still making it clear that `max` is the desired aggregation. Misconception-based Faded Parsons Problems could still provide some guidance and assurance compared to writing code but also force students to confront and reconcile these misconceptions. It would also be interesting to understand the opportunities offered by different types of Faded Parsons Problems compared to code writing exercises and to Parsons Problems with distractors.

4.7 Conclusion

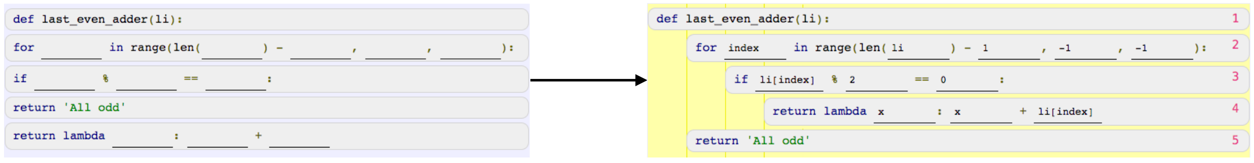
Parsons Problems are an exciting tool for CS1 classrooms. We introduce Faded Parsons Problems to explore more ways these types of problems can be integrated into classrooms. We confirm that Blank-Variable Parsons provide an opportunity for fading the scaffolding afforded by standard Parsons as students work to master unscaffolded coding. Additionally, we find that Parsons Problems and Blank-Variable Parsons can be more effective than code writing at teaching certain skills. However, we do not yet find a difference in how they support programming holistically. This study motivates exciting new opportunities for leveraging all of these types of exercises in classrooms.

Chapter 5

Improving Instruction of Programming Patterns with Faded Parsons Problems

The previous study showed promise for Faded Parsons Problems as a pedagogical tool. However, many of the interesting findings were based on self-rated metrics by students in an in-lab setting. To better understand Faded Parsons Problems, I wanted to run a large-scale in-situ classroom study. This 237-student study compared Faded Parsons Problems to both Code Tracing and Code Writing exercises. Additionally, since the study was run over the course of a full semester, I was actually able to run the study in three distinct parts, each focused on teasing out different insights. This study also crystallized the opportunity to use Faded Parsons Problems to teach patterns in particular. I was excited about the opportunity to help instructors teach patterns without a significant amount of work creating new materials or reworking their curriculum. This study revealed that Faded Parsons Problems are particularly effective at teaching patterns while still improving overall code writing ability similar to Code Writing exercises.[1]

[1]This work was published at CHI 2021 (Weinman, Fox, and M. A. Hearst, 2021).



```
def last_even_adder(li):
for ________ in range(len(________) - ________, ________, ________):
if ________ % ________ == ________:
return 'All odd'
return lambda ________: ________ + ________
```

```
def last_even_adder(li):
    for index in range(len(li) - 1, -1, -1):
        if li[index] % 2 == 0:
            return lambda x: x + li[index]
    return 'All odd'
```

Figure 5.1: **An example of an unsolved Faded Parsons Problems (left) and the completed solution (right)** using a Premature Return pattern to solve a Higher-Order Function problem. Students solve the question by both rearranging and completing the given lines of code.

5.1 Introduction

Despite the increasing relevance of programming (Scaffidi et al., 2005; Barr et al., 2011; *U.S. Department of Labor, Occupational Outlook Handbook, Software Developers* 2019), developing this skill continues to be difficult for novices to learn in part because programming is a complex task consisting of many skills and types of knowledge (du Boulay, 1986; Jenkins, 2002; Cutts et al., 2012).

One such skill, the ability to recognize and use programming patterns, is a distinguishing characteristic between experts and novices (Robins et al., 2003). *Programming patterns* are partial implementations of reusable, higher-level concepts which can achieve a goal. For example, the solution in Figure 5.1 represents a “Loop: Premature Return” pattern, searching the list backwards and returning a higher-order function as soon as an even number is found. This pattern consists of a conditional return within a loop, with a catch-all return outside of that loop. This same pattern, with different code, could be used for many similar purposes, such as to find the first occurrence of a character in a string or the largest even number in an increasingly sorted list. These patterns provide the building blocks that allow programmers to efficiently tackle more complex tasks. However, despite their importance, they are often not explicitly taught in CS1 classrooms (Muller et al., 2007).

Programming patterns can be challenging to teach because they rely on a foundation of conceptual knowledge (Xie et al., 2019). For example, the pattern in Figure 5.1 requires understanding of `for` loops and `range()`. Researchers have developed methods to teach programming patterns, but these involve significant modifications to the structure of the course curriculum (Linn and Clancy, 1992; McGill et al., 1997; Sajaniemi et al., 2005; Loksa et al., 2016; Xie et al., 2019). For example, Pattern-Oriented Instruction (Muller et al., 2007), as part of their guidelines, proposes discussions after all problem-solving activities and prepared material to compare patterns after complex exercises. The practical difficulties of introducing large-scale innovations to classroom teaching suggest that an approach that minimizes changes to the status quo will lead to better adoption over major structural changes.

Most computer programming classes already make use of assigned programming exercises, highlighting opportunity for easily integrated improvements. Many programming courses use a combination of Code Tracing exercises, in which students predict the behavior of instructor-provided code, and Code Writing exercises, in which students write code in response to an exercise prompt. In this work, we compare these commonly used exercise interfaces, as well as a recently-introduced exercise interface, Faded Parsons Problems (Figure 5.1) (Weinman, Fox, and M. Hearst, 2020), as a low-friction way to teach programming patterns. In this chapter, we emphasize exercise interfaces as one way to distinguish between different types of exercise. We find that the content of existing Code Tracing and Code Writing exercises can be easily re-purposed as Faded Parsons Problems.

Faded Parsons Problems are a particular variation of Parson Problems (Parsons et al., 2006). In Faded Parsons Problems, creators give students a set of partially complete lines of code. Students must fill in blanks in the lines of code with valid expressions as well as rearrange the lines of code to construct a valid program. Unlike Code Tracing exercises, with Faded Parsons Problems, students must actively construct the structure and logic of a program. However, students are heavily constrained by the given lines of code, thus deliberately excluding many valid alternative solutions to the exercise.

We ran an in-situ study as part of an introductory Computer Science class and found strong evidence to confirm that Faded Parsons Problems are effective at teaching programming without any modifications to instruction. Our study also explores how Faded Parsons Problems affect transfer to student code writing ability in general, as well as qualitative data to understand how students engage with Parsons Problems and limitations of Code Writing exercises.

Our contributions are as follows:

- We deploy a system which supports a range of programming exercise interfaces which scales to a class of hundreds of students and is well-suited for remote learning.
- We confirm the efficacy of Faded Parsons Problems in teaching students to recognize and apply programming patterns to relevant exercise prompts.
- We confirm that practicing with Faded Parsons Problems produces similar transfer to code writing ability as directly practicing with Code Writing exercises.
- We probe student attitudes towards Faded Parsons Problems, finding students prefer working with Faded Parsons Problems when given a choice and providing insights as to why.

5.2 Related Work

5.2.1 Programming Patterns

Patterns (or plans, schemas, templates) have several subtly different definitions in the literature (Spohrer et al., 1986; Wiedenbeck et al., 1993; Marshall, 1995; Astrachan et al., 1998; Clancy et al., 1999; Muller et al., 2007; Xie et al., 2019). In general, they are higher-level, reusable abstractions of code that achieve a specific goal. Patterns are also hierarchical and multi-layered, with some smaller patterns contained in other larger ones. The importance of patterns is supported through chunking from cognitive theory, in which people, as they view examples with identifiable similarities, construct and store more complex patterns as single cognitive "chunks" (Chase et al., 1973; Marshall, 1995).

Studies have found evidence that one way the behavior of expert programmers is different from novices is in their ability to leverage patterns in understanding and writing code (Wiedenbeck et al., 1993; Robins et al., 2003). However, patterns remain out of the focus in many Computer Science classes, leading to proposals for significant instructional shifts to address this concern.

Muller et al. (2007) propose Pattern-Oriented Instruction in introductory Computer Science classes, where patterns are explicitly incorporated into a course's instruction. They found that this explicit change to instruction led to novices improving their problem decomposition and solution construction skills. Xie et al. (2019) also designed a curriculum focused on explicit instruction of templates. They find some evidence that this new template-oriented curriculum improved student coding ability.

Xie et al. (2019) also note the importance of learning syntactic or conceptual knowledge before learning patterns. That is, students must understand all the building blocks of a pattern before being able to understand the pattern itself, and similarly for writing building blocks and patterns. That is, though patterns are a foundational skill for programming, they are not entirely introductory. This may explain why many CS1 classes today still do not focus on explicitly teaching patterns.

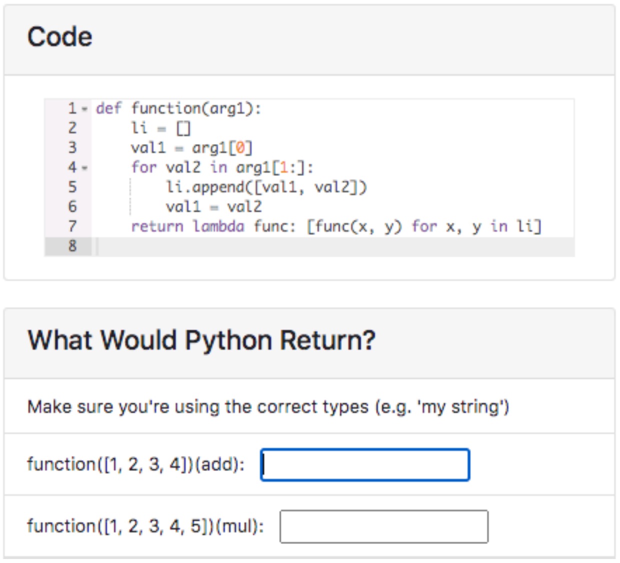
We build on the existing work showing that patterns are both important and could be taught better in most CS1 classrooms. Our work differs, however, in its focus on a much simpler instructional change. Our work focuses on assigned programming exercises – a part of the course where students already spend considerable time. Instead of updating curricula to explicitly teach patterns to students, we focus on whether students can recognize and incorporate patterns more effectively by working on programming exercises with different user interface.

5.2.2 User Interfaces for Program Exercises

Several frameworks have been proposed for conceptualizing user interfaces for programming exercises. Cutts et al. (2012) propose that programmers must master moving between three levels of abstraction: English, pseudocode, and code, proposing different exercise interfaces to target different levels of abstraction. Bryant et al. (2008) extend this further, proposing five levels of abstraction. They posit that expressing algorithms in human language, when compared to code, changes how programmers engage with their programming task, suggesting that pair programming causes programmers to focus more on an intermediate level of abstraction compared to when programming alone.

Shi et al. (2019) motivate their work from a different perspective, focused on the different stages of problem solving. They design Pyrus, an interface where students work together with limited knowledge to focus student attention and growth on the planning stage of the problem-solving process. By constraining students' actions in solving the exercise, they find students spend more time planning. Python Tutor (Guo, 2013) has become a popular educational tool for introductory students because its granular visualizations and debugging functionality are well-suited to novice programmers.

Our work is motivated by these frameworks and interfaces, exploring how the design behind programming exercise interfaces affects how students engage with the content. Thoughtfully considering these designs can support programmers in more effectively practicing specific programming knowledge or skills. Our work reveals insight into our recently introduced interface, Faded Parsons Problems (Weinman, Fox, and M. Hearst, 2020).



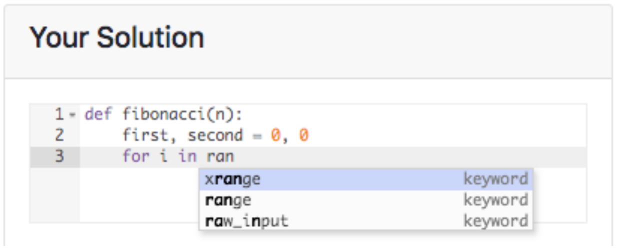
```
def function(arg1):
    li = []
    val1 = arg1[0]
    for val2 in arg1[1:]:
        li.append([val1, val2])
        val1 = val2
    return lambda func: [func(x, y) for x, y in li]
```

Make sure you're using the correct types (e.g. 'my string')

function([1, 2, 3, 4])(add):

function([1, 2, 3, 4, 5])(mul):

Figure 5.2: **A Code Tracing exercise.** Students must read the obfuscated code (top) and determine the output from specified argument prompts (bottom)



```
def fibonacci(n):
    first, second = 0, 0
    for i in ran
```

```
xrange      keyword
range       keyword
raw_input   keyword
```

Figure 5.3: **A Code Writing exercise.** Students must create a functioning program to solve a given exercise prompt using an editor with basic IDE functionality like syntax highlighting and auto-completion.

5.2.3 Parsons Problems

Parsons Problems (or Parsons Puzzles, code scrambles, Code Mangler Problems) require students to unscramble lines of code to construct a syntactically and logically correct program. They were originally designed to be an engaging task for students to practice syntax drills (Parsons et al., 2006). They share many similarities with block-based programming languages such as Scratch, Snap!, Alice, and Blockly, which provide an engaging introduction to certain introductory topics (Rizvi et al., 2011; Meerbaum-Salant et al., 2013; Weintrop et al., 2017). However, unlike block-based languages, Parsons Problems support the full expressivity of text-based programming languages such as Python.

Zavala et al. (2017) found support that code comprehension (e.g., Code Tracing exercises), code manipulation (e.g., Parsons Problems), and code writing are separate skills that should be mastered in that order. Ericson, Margulieux, et al. (2017) successfully integrated Parsons Problems into teaching, finding that Parsons Problems, when combined with worked examples, provide similar learning gains to code writing or bug detection in a CS1 classroom while taking only 70% of the time. Ericson, Foley, et al. (2018) extended this study, finding similar results for Adaptive Parsons Problems, which dynamically adjust difficulty by adding or removing unnecessary or incorrect lines of code.

However, Denny et al. (2008) raise concerns that Parsons Problems may be easily "gamed" by sufficiently mature students using syntactic heuristics. In our formative study for the present work, a student described their strategy for solving Parsons Problems as "just kind of moving things around based on test cases, not really thinking about the logic." In previous work (Weinman, Fox, and M. Hearst, 2020), we found that solving a Parsons Problem did not transfer to being able to write code for the same question, and attempted to address this limitation by introducing Faded Parsons Problems. In Faded Parsons Problems, provided lines of code can be partially or fully incomplete. Faded Parsons Problems integrate code writing with Parsons Problems, providing less syntactic and logic scaffolding than standard Parsons Problems.

The present work differs from previous work on Parsons Problems two ways. First, this work primarily focuses on Faded Parsons Problems. Second, this work focuses on teaching programming patterns, a targeted subset of general programming ability.

We focus on three interfaces. Code Tracing exercises (Figure 5.2), in which students predict the behavior of instructor-provided code, require students to *understand* a *well-designed solution*. Code Writing exercises (Figure 5.3), in which students write their own code in response to an exercise prompt, require students to *construct* a solution, though possibly a poor one. Faded Parsons Problems require students to *reconstruct* a *well-designed solution* by using constraints to both lock out valid solutions and scaffold the solving process.

5.3 Programming Patterns

Following existing definitions in the literature, we define **programming patterns** as partial implementations of reusable, higher-level concepts which can achieve a goal. The programming patterns used in this study are specific to Python and are based in coding structures.

We ground our definition of programming patterns in actual code to ensure they contain certain language idioms, some of which are specific to Python. For example, L2 (defined in Table 5.1) includes the use of `enumerate(li)` in a `for` loop. A more abstract, language-agnostic definition would include more verbose implementations such as iterating through `range(len(li))` and then also defining a local variable from indexing into the list. This example also highlights how patterns must be built on both conceptual knowledge, e.g., un-

Table 5.1: **Examples of programming patterns and their descriptions.**

Goal/Name	**Description**
Loop: Premature Return (L1)	A return within a loop based on a conditional, with a final return outside of that loop
Loop: Index Value (L2)	Use of `enumerate` to access and update both the index and value of elements, using both inside the loop.
Loop: Last/Current (L3)	Inside a loop, use last and then update last to current.
Stateful Tree Traversal (MR1)	Traverse a Tree in depth-first-search using a default argument to pass state.
Multiple Recursion Game Simulation (MR2)	Check for a base case, simulate moves, then return based on the results.
Stateful Generators (OOP1)	Define generators as part of a class for stateful instance variables, yielding at the end of loops.
Mixins (OOP2)	Use `super` to access parent methods.

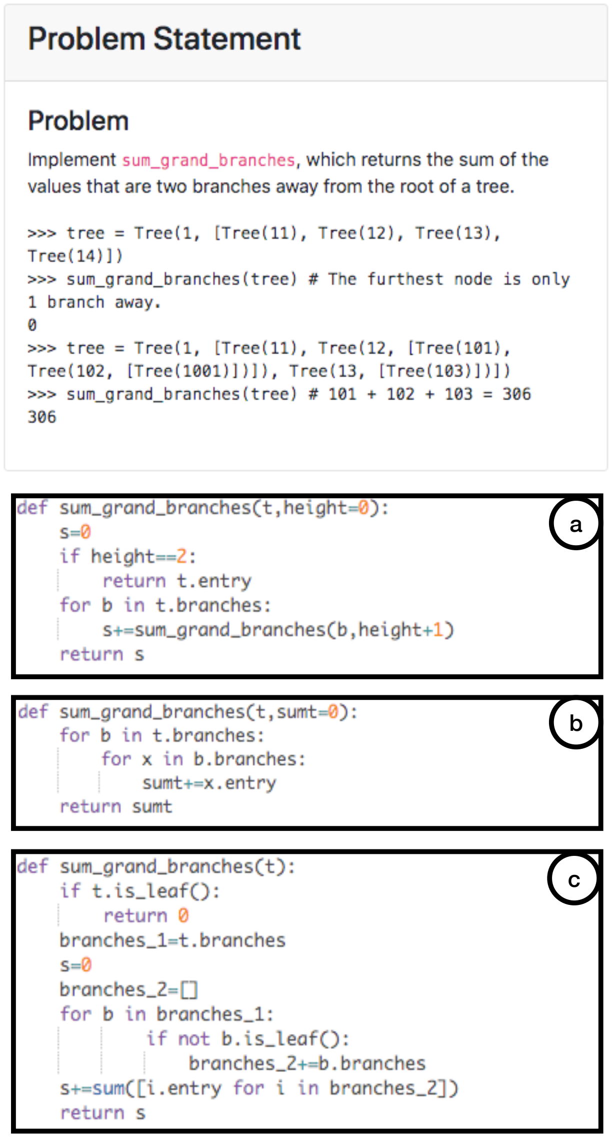
derstanding loops, and syntactic knowledge, e.g., `enumerate`. Some of the syntactic features on which patterns are built, such as `for`, may be more fundamental than others.

The pattern adherence of a given program can be evaluated both by the use of control flow elements as well as specific syntactic elements. An advantage of this is that patterns are specific enough to be effectively captured with automated parsing logic, allowing efficient analysis across tens of thousands of submissions. Additionally, we can check the pattern adherence of a program even if the program contains unrelated syntax errors, which happens frequently for introductory student submissions.

Table 5.1 lists the programming patterns used in this study, along with definitions. These programming patterns were selected by analyzing programming assignments from previous versions of two introductory Computer Science courses. That is, existing course materials, developed by the instructor, determined which programming patterns were used in this study.

5.3.1 Examples of Programming Patterns

The patterns were selected so the three Loop patterns, L1-L3, would appear appropriately early in the course, focusing on foundational material. Though these patterns were used in exercises over multiple weeks, tangential concepts covered within the exercises changed over the course of the semester. For example, lambdas and higher order functions were introduced into the patterns later in the semester. The example in Figure 5.1, for instance, returns a function which takes an argument `x` and adds it to the last even element in the given list. Another exercise using this pattern earlier in the course returned True if any numbers in a list are divisible by five. The next two patterns (MR1, MR2) were selected to match course content on multiple recursion, focusing on algorithmically challenging exercises. The final two patterns (OOP1, OOP2) included class definitions in order to match course content



Problem Statement

Problem

Implement `sum_grand_branches`, which returns the sum of the values that are two branches away from the root of a tree.

```
>>> tree = Tree(1, [Tree(11), Tree(12), Tree(13),
Tree(14)])
>>> sum_grand_branches(tree) # The furthest node is only
1 branch away.
0
>>> tree = Tree(1, [Tree(11), Tree(12, [Tree(101),
Tree(102, [Tree(1001)])]), Tree(13, [Tree(103)])])
>>> sum_grand_branches(tree) # 101 + 102 + 103 = 306
306
```

```
def sum_grand_branches(t,height=0):
    s=0
    if height==2:
        return t.entry
    for b in t.branches:
        s+=sum_grand_branches(b,height+1)
    return s
```

```
def sum_grand_branches(t,sumt=0):
    for b in t.branches:
        for x in b.branches:
            sumt+=x.entry
    return sumt
```

```
def sum_grand_branches(t):
    if t.is_leaf():
        return 0
    branches_1=t.branches
    s=0
    branches_2=[]
    for b in branches_1:
            if not b.is_leaf():
                branches_2+=b.branches
    s+=sum([i.entry for i in branches_2])
    return s
```

Figure 5.4: **Three different correct student submissions to an exercise for which students could apply the MR1 pattern.** (a) A reasonable solution to the exercise which follows the pattern. (b) A reasonable solution to the exercise which does not follow the pattern. (c) An overly complex solution to the exercise which does not follow the pattern.

on Object-Oriented Programming, focusing on syntactically and conceptually challenging exercises. These patterns were chosen to cover a range of different aspects of programming.

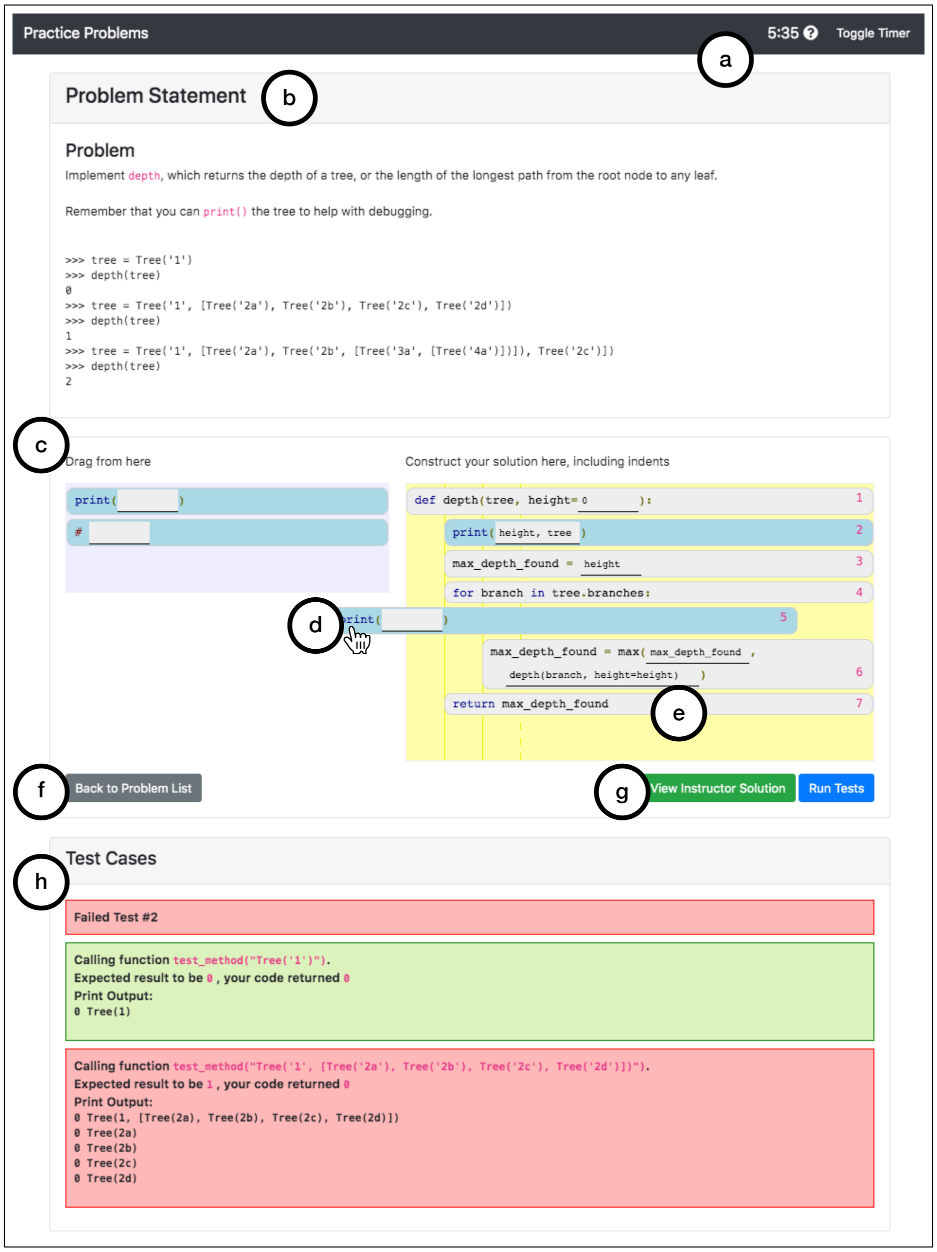
5.3.2 Programming Patterns in Code

To better illustrate the role of programming patterns in student work, Figure 5.4 contrasts three correct student submissions to a problem — summing nodes of a Tree at depth 2 — in an open-ended Code Writing exercise. These submissions were selected from the study described in Section 5.5. All students had immediately previously worked on a question computing the depth of a Tree of arbitrary height. Submission (a) follows the MR1 pattern. It is very similar to the instructor solution for the previous exercise and a reasonable solution to this one. Submission (b) is a reasonable submission that does not follow the pattern. Instead, it takes advantage of the fact that this exercise is asking about a fixed depth within the Tree, using nested for loops instead of recursion. While this is a valid solution, it defeats the instructor's motivation to have students recognize this exercise as an opportunity to practice recursion to navigate Trees. This highlights a challenge of teaching programming patterns, as there can sometimes be other reasonable solutions to specific exercises. Submission (c) is a correct submission which does not follow the pattern, and is overly complex and verbose. It is not uncommon for students to create poorly structured solutions such as these in auto-graded Code Writing exercises, due to the lack of granular feedback and the iterative manner with which students can create solutions.

Though Code Writing exercises are an effective form of practice for programming, the example above illustrates why they may be ill-suited for helping students learn how to use patterns as students only sometimes construct solutions following a relevant pattern. If students are given the freedom to solve two related exercises in entirely different ways, they miss an opportunity to compare and contrast two applications of the same pattern, an important opportunity to generalizing the abstraction. Code Tracing exercises allow instructors to control the structure of code students are interacting with, providing a good opportunity for teaching patterns. However, they do not give students any practice in constructing algorithms or syntax to create a program.

5.4 User Interface For Programming Exercise Comparison

We built a Flask app that extends Karavirta et al.'s js-parsons library (Ihantola and Karavirta, 2010) to support all exercise interfaces used in this study, including Faded Parsons Problems. The system supports Python programming exercises—traditional Code Writing exercises, Faded Parsons Problems, Code Tracing exercise, Code Skeleton exercises, and multiple-choice comprehension—as well as short answer survey questions. A nearly complete Faded Parsons Problem exercise can be seen in Figure 5.5(c). In Faded Parson Problem ex-



Problem Statement

Problem

Implement `depth`, which returns the depth of a tree, or the length of the longest path from the root node to any leaf.

Remember that you can `print()` the tree to help with debugging.

```
>>> tree = Tree('1')
>>> depth(tree)
0
>>> tree = Tree('1', [Tree('2a'), Tree('2b'), Tree('2c'), Tree('2d')])
>>> depth(tree)
1
>>> tree = Tree('1', [Tree('2a'), Tree('2b', [Tree('3a', [Tree('4a')])]), Tree('2c')])
>>> depth(tree)
2
```

```
print(          )
#
```

```
def depth(tree, height= 0          ):
    print( height, tree )
    max_depth_found = height
    for branch in tree.branches:
print(          )
        max_depth_found = max( max_depth_found ,
          depth(branch, height=height)    )
    return max_depth_found
```

Test Cases

Failed Test #2

```
Calling function test_method("Tree('1')").
Expected result to be 0 , your code returned 0
Print Output:
0 Tree(1)
```

```
Calling function test_method("Tree('1', [Tree('2a'), Tree('2b'), Tree('2c'), Tree('2d')])").
Expected result to be 1 , your code returned 0
Print Output:
0 Tree(1, [Tree(2a), Tree(2b), Tree(2c), Tree(2d)])
0 Tree(2a)
0 Tree(2b)
0 Tree(2c)
0 Tree(2d)
```

Figure 5.5: **A nearly correct Faded Parsons Problem finding the depth of a Tree.** (a) Optional Timer. (b) Problem Description. (c) Faded Parsons Problem interface; participants can drag blocks between the bin (left) and the solution (right). (d) An optional print block being dragged to the right. (e) A blank that has been filled in with code by the student. (f) Students can always navigate back to the exercise list or (g) run tests on their current solution. After effort-completing a exercise, they can view the instructor solution (g). (h) Descriptive test case results up to the first failed test.

ercises, the user is initially given a set of blocks containing partially incomplete code on the left including optional print and comment statements (d) for debugging, with the initial function signature populated on the right. The lines on the left are initially alphabetized, as suggested by Cheng and Harrington (Cheng et al., 2017). To solve a Faded Parsons Problem, users drag fragments in a correct order and indentation on the right and complete the blanks (e).

All exercises display the problem statement (b) above the interface. Users can run preconfigured tests as often as they want, which displays detailed output from the test cases (h) including: function arguments, expected output, actual output, any standard output from print statements, and any raised exceptions. The type of feedback is consistent across all programming interfaces. Not pictured: At any point, users can return to a list of exercises for that week. Additionally, users can view a correct solution to the exercise after solving the question or expending enough effort on the exercise based on custom time- and submission-based logic.

The Flask app logs anonymized data from participants, and randomizes treatment selection. The autograder is a separate worker that uses RQ[2] to safely execute arbitrary Python code, allowing execution-based feedback. Instructors manually configure the blanks in Faded Parsons Problems.

5.5 Evaluation

We run a study to explore the following research questions:

RQ1: How does practice with different exercise interfaces affect student acquisition of programming patterns? To be able to learn a pattern, students must first be exposed to examples of it. We measured **Pattern Exposure** to see if students encounter pattern-adherent code as they craft their own solution or by viewing the instructor solution. We also measured **Active Pattern Exposure**, a subset of this, to see if students craft a pattern-adherent solution. In later exercises, we measured **Pattern Acquisition** to see if students, after being exposed to examples of a pattern, are able to recognize the opportunity and apply a pattern in a Code Writing exercise. We note that students can apply a pattern without correctly solving an exercise, such as the code in Figure 5.5.

RQ2: What is the general efficacy of practice with different exercise interfaces? This work is motivated by the ease in which instructors can replace existing exercises with different exercise interfaces. To that end, we must understand the educational impact of different exercise interfaces beyond patterns. Since Code Writing exercises are often used as a de facto measurement of programming expertise, we measured **Code Writing Transfer**

[2]https://python-rq.org/

to see if students can successfully transfer (Perkins et al., 1992) what they learned in each interface to similar Code Writing exercises regardless of any pattern use.

RQ3: What is student perception of working with Faded Parsons Problems? In several exercises, students were able to select which exercise interface they worked with, which implicitly suggests preference. Students were also asked Likert-scale and open-ended survey questions to understand their **Preference** surrounding Faded Parsons Problems. These survey questions also provide insight into the **Perceived Difficulty** of these exercise interfaces, which we compare to the **Actual Difficulty**.

5.5.1 Study Environment and Participants

We partnered with an instructor of a CS1 class at a large US research university with IRB approval to run our evaluation study in three parts. Throughout the course of the semester, 237 students (P1-P237, 111 male, 120 female, 6 unreported) agreed to the Terms of Consent and interacted with our system. The study was deployed as extra credit assignments appended to 10 of the 12 weekly lab assignments. Students could begin the questions in an in-person lab, but had a week to continue working on the exercises on their own. The extra credit exercises were effort-based (as opposed to completion-based) to better match how labs were assessed. The instructor approved all exercises used in the study, but was not aware of the specific research goals to ensure other course content and instruction were not modified.

5.5.2 Method for Constructing Faded Parsons Problems

Researchers selected which code segments to blank out for Faded Parsons Problems. Required function arguments were never blanked out, unless determining the function arguments was critical to the exercise (e.g., recognizing and using an optional argument). If a variable name was blanked out in its assignment statement, all other references to that variable were also blanked out to avoid confusion in case students selected a different name. Top-level control flow tokens were never blanked out, though conditionals, constants, variable references, and other expressions were sometimes blanked out. The amount of code blanked out from exercises ranged from 0% (only rearranging was required) to 75% of the target code, on average blanking out 32% of the code.

5.5.3 Study Description

We report on three studies to answer the research questions. A high-level summary is provided in Table 5.2. Two studies, Study 1 and Study 2, compared the efficacy of three exercise interfaces — `tracing`, `parsons`, and `writing` — in teaching students to acquire programming patterns. A qualitative study was run in-between these two studies to better understand student perception of Faded Parsons Problems.

Table 5.2: **High-level description of each reported study.**

Study	**Research Questions**	**Participants**	**Exercise Interfaces**	**Duration**
Study 1	RQ1, RQ2	50	tracing, parsons, writing	4 weeks
Qualitative	RQ3	50	parsons, writing	2 weeks
Study 2	RQ1, RQ2	43	parsons, writing	2 * 2 weeks

Figure 5.6 summarizes the structure of the two studies. In both pattern-focused studies (Study 1 and 2), students knew the concepts (e.g., for loops, Tree structures) necessary to compose the patterns, but the patterns themselves were not explicitly taught as part of the class. These studies began with an `exposure` phase, in which students solved exercises involving different patterns and exercise interfaces. Knowledge Integration (Linn, 2005), an educational framework, suggests that students incrementally acquire generalized knowledge like patterns through new examples, motivating the use of multiple `exposure` exercises for each pattern. The `exposure` phase explores the efficacy of these exercise interfaces in successfully exposing students to relevant patterns. The `exposure` phase is followed by an `acquisition` phase, in which students wrote code in response to a prompt where one of the patterns was applicable. This phase explores if students can express their mastery of patterns through Code Writing exercises. The first study consisted of additional exercises between the two phases to explore other research questions. There was no explicit indication to students of these different phases, nor did they receive different types of instruction or feedback based on the phase.

In all studies, students worked through exercises in our system in a pre-determined order. Students **effort-completed** a question by correctly solving it and passing all test cases or by both spending at least 10 minutes on the exercise and making 10 consecutively distinct submissions (the course instructor determined this definition for effort-completion). After effort-completing a question, students were able to view an instructor solution or return to the exercise list, after which they could continue on to the next question. Students primarily worked in one of three exercise interfaces, as seen in Figures 5.2, 5.3, 5.5: Code Tracing exercises (`tracing`), Faded Parsons Problems (`parsons`), and Code Writing exercises (`writing`). Each exercise consisted of an exercise prompt (i.e., problem statement), an interface to work on it, and test cases with their results up to the first failed test case. Problems were designed such that problem statements within a pattern had meaningful differences and were not isomorphic to each other.

Study 1

This study was designed to compare the efficacy of three exercise interfaces — `tracing`, `parsons`, and `writing` — in teaching students programming patterns as well as more general programming ability.

The study consisted of patterns L1, L2, L3 (Table 5.1). In the `exposure` phase, each pattern was paired with *one* exercise interface such that each exercise interface was used once.

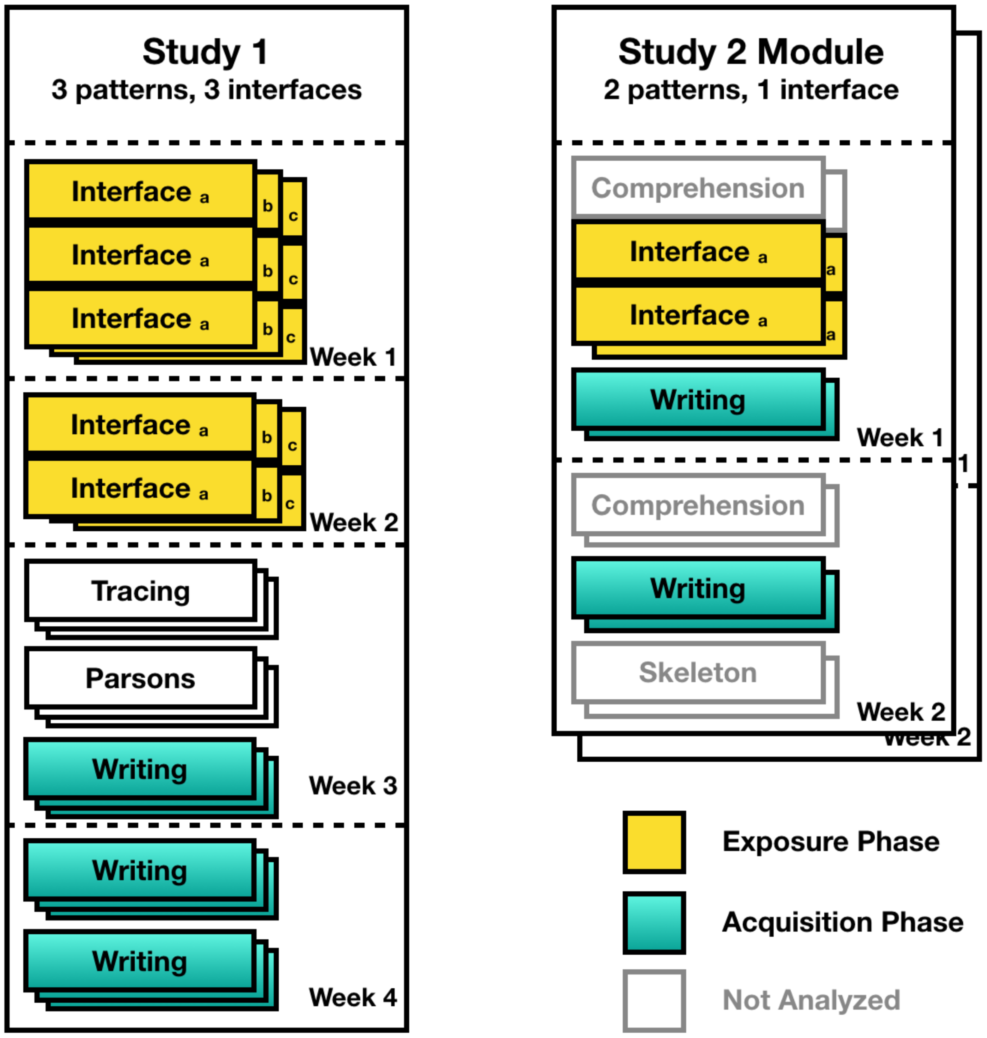


Figure 5.6: **Study description for Studies 1 and 2.** Example sequence of exercises for a single student for Study 1 and a Study 2 module, highlighting a *single* pattern. Each block represents a single exercise. If 2 or more blocks are vertically touching, it means those exercises were consecutive, otherwise some equivalent exercises for other patterns (possibly in different exercise interfaces) occurred between them. The "stacking" of blocks highlights the similar flow for other patterns. That is, in Study 1 Week 2, students worked on 2 exercises for each of 3 (pattern, exercise interface) pairs. Problems in yellow were in the exposure phase, where each pattern was associated with a single exercise interface. Problems in gradient blue were in the acquisition phase, always Code Writing. Grey exercises were not analyzed in this study. Study 2 consisted of 2 modules, one immediately after the other, represented by the "stacking" of the full study.

For each such pair, five exercises were presented over the course of two weeks. Each student effort-completed a total of 15 exercises in this phase, and all students saw all exercises in the same order with the same exercise interfaces. These exercises measured **Pattern Exposure** and **Active Pattern Exposure**.

In the following week, we instantiated all 9 *(pattern, exercise interface)* combinations: Each student worked on 9 exercises, grouped by exercise interface. The order in which patterns were presented to students was counterbalanced between exercise interfaces. These exercises measured **Actual Difficulty**, as the exercise interfaces were independent of the patterns they were paired with in the exposure phase.

Finally, in addition to the three `writing` exercises from the previous week, students completed 6 more `writing` exercises, 2 for each pattern. These 9 exercises in total represent the acquisition phase, and measured **Pattern Acquisition** and **Code Writing Transfer**. Both metrics are analyzed based on the exercise interface each pattern was paired with in the exposure phase.

All patterns were selected to be of comparable complexity, and the pairing of patterns and exercise interfaces in the learning phase was done randomly after all exercises were created. However, because the study was designed to give a consistent experience to every student and a given pattern was paired with only a single exercise interface in the exposure phase, better performance on that pattern could be due either to that exercise interface being more effective for learning, or to the pattern itself simply being easier to learn.

Study 2

This study was designed to more robustly compare the efficacy of two exercise interfaces — `parsons` and `writing` — in teaching students programming patterns as well as more general programming ability. This study explores more complex patterns (MR1, MR2, OOP1, OOP2) and randomizes interfaces within patterns (unlike Study 1). We do not explore `tracing` exercises in this study due to its poor **Code Writing Transfer** from Study 1.

This study consisted of two modules, one following directly after the other, with students randomly designated into treatment or control. Each module targeted two patterns. The first module covered MR1 and MR2 for all students regardless of treatment, with the second module covering OOP1 and OOP2. Assignment was counterbalanced, so each student was in the control group for one module and the treatment group for the other.

In the exposure phase of each module, patterns were paired with `parsons` for students in treatment or `writing` for students in control. For each pattern, two exercises were presented over the course of one week. Each student effort-completed a total of 4 exercises in this phase.

After the exposure phase, students were given two `writing` exercises for each of the same two patterns, for a total of 4 exercises in this phase.

The modules also consisted of exercises that were not analyzed for this study. First, at the start of every week, students were given a multiple choice comprehension question on the topics covered in lab. These questions were added because the exercises in Study

2 focused on more complex topics. Additionally, each module ended with a code skeleton exercise — effectively a Faded Parsons Problem with the lines already arranged. We hoped the code skeleton exercises would inform us if students were able to demonstrate mastery of the patterns in a more constrained exercise interface. However, upon further analysis, we did not present enough skeleton exercises to analyze.

Study Differences: Below, we emphasize some major differences between the studies that may be helpful in interpreting the results.

- Randomization: In Study 1, all students saw the same materials. In Study 2, students were randomly assigned to treatment or control.
- Dosage: In Study 1, students worked on 5 exercises for each pattern in the exposure phase. In Study 2, they only worked on 2.
- Pattern-Exercise Interface Pairings: In Study 1, each exercise interface was used for a single pattern in the exposure phase, which may have encouraged students to look for similarities. In Study 2, each exercise interface was used for two independent patterns.
- Topics: In Study 1, problem topics changed meaningfully from week to week to match course curriculum, for example focusing on higher-order functions or lambdas. In Study 2, the exercises focused on topics which extended but were supplementary to those in the course curriculum.

Subjective Study

Between the Study 1 and 2, we ran a study focused on gaining preference and self-reported insights from students. This study did not focus on any patterns. In this study, after seeing each problem statement, students were able to choose whether to work on the exercise as `parsons` or `writing`. Additionally, students filled out a survey with questions about their perception of Faded Parsons Problems.

5.6 Results

The following subsections first describe the statistical measures used and next the data cleaning process for the study. This is followed by analyses of the results corresponding to the major research questions. We report on the degree to which students acquire programming patterns (Section 5.6.3), what students learn using Faded Parsons Problems beyond learning programming patterns (Section 5.6.4), and students' subjective responses (Section 5.6.5). We then synthesize these results (Section 5.6.6).

5.6.1 Statistical Measures

Unless otherwise stated, statistical significance is computed as the proportion of relevant submissions matching the measurement in question grouped by student. For Study 1, we analyze the 50 students (17 male, 31 female, 2 unreported) that effort-completed all exercises in the study. We pair by student and use the Friedman test (Friedman, 1937) to determine if there is a difference between the three interfaces, then use pairwise Wilcoxon Signed-Rank tests to test for significance. For Study 2, we analyze the 43 students (17 male, 23 female, 3 unreported) that effort-completed all exercises in the study. We use Mann-Whitney U tests to test for significance.

5.6.2 Data Cleaning

Labeling Pattern Adherence

For all 7 patterns used in this study, researchers created code to automatically detect if a given code submission adhered to the pattern. The code relies on custom string parsing, as submissions might adhere to a pattern even if they cannot be parsed as a valid Abstract Syntax Tree, e.g. if a : was missing from a conditional statement. To test the validity of this approach, for each pattern, we randomly selected 25 submissions from the relevant exercises that adhered to the template and 25 submissions that did not. We then removed the algorithm-generated labels, shuffled the results within each pattern, then had two researchers manually annotate the pattern adherence of each submission. We computed Cohen's Kappa score (Fleiss et al., 1973) to measure agreement between the annotators and between each of them and the algorithm-generated labels. We found a kappa of 0.89 between the annotators, and an agreement of .94 and .88 respectively between each annotator and the algorithm-generated labels, indicating very good agreement. This gives us confidence that the algorithm can generate labels with comparable accuracy to a human.

Handling Corrupt Data

Researchers examined a sample of the logs and some aggregate data to detect and remove entries where students were abusing the system in some way.

First, because test cases were transparent, some students wrote code to return hard-coded values based on combinations of input arguments, clearly not trying to actually solve the exercise. To address this, researchers added three or more additional test cases to each exercise after the study completed. All submissions were re-tested for correctness based on success on the full set of test cases.

Second, for `parsons` and `writing` exercises, we removed submissions from students if they read and correctly solved the exercise in under 45 seconds. We also excluded submissions from students where their solution exactly matched the instructor solution including the comments explaining the solution.

Finally, from the remaining data to be analyzed, we manually inspected students that did not answer a single `writing` question correctly throughout the study. From this, we remove submissions that appeared designed entirely to get past the effort criteria, such as adding or removing random characters. Across all three criteria, we removed 5 students from analysis.

5.6.3 Pattern Acquisition

The primary motivation of these studies was to understand how practice with different exercise interfaces affects student acquisition of the programming patterns. For students to learn to recognize and apply patterns, they must first be exposed to the pattern. However, students can correctly solve exercises without ever writing pattern-adherent code or reading the pattern-adherent instructor solution. We first analyze if students are exposed to pattern-adherent code in the exposure phase, either as they construct their own solution or by viewing the instructor solution (Pattern Exposure). We then analyze if students recognize and apply the relevant pattern (Pattern Acquisition) to `writing` exercises based on the exercise interface paired with each pattern in the exposure phase. Results can be seen in Table 5.3.

Pattern Exposure: Are students exposed to the pattern-adherent code working on an exercise?

We posited that one advantage of `parsons` and `tracing` is that they both significantly constrain the solution space, introducing students to particular solutions. Therefore, we would expect that students are more likely to adhere to programming patterns when generating a solution with either interface compared to `writing`. We separately analyze **Active Pattern Exposure**, if students generate pattern-adherent code themselves while completing the exercise, and **Pattern Exposure**, if students ever generate or view pattern-adherent code (e.g., by viewing the instructor solution). We analyze both, because research on Active Learning (Michael, 2006), in which students practice applying skills instead of simply responding to questions, suggests that students will learn patterns better if they construct the patterns themselves.

For `tracing`, by design, Pattern Exposure is 100% and Active Pattern Exposure is 0% as students always view but never construct pattern-adherent code. We find find that `parsons` are more likely than `writing` to expose patterns. The rate of Pattern Exposure is higher for `parsons` than `writing`, 97.2% vs. 39.2% (p¡.001) in Study 1, 87.8% vs. 64.9% (p¡.001) in Study 2. The rate of Active Pattern Exposure is also higher for `parsons` than `writing` by a more significant margin, 92.4% vs. 4.4% (p¡.001) in Study 1, 70.9% vs. 36.9% (p¡.001) in Study 2.

Table 5.3: **Summary statistics related to Pattern Exposure and Acquisition, Code Writing Transfer, and Actual Difficulty.** (*) indicates a pairwise-significant difference with one other interface while (**) indicates a pairwise-significant difference with both.

	Study 1			**Study 2**	
	Tracing	Parsons	Writing	Parsons	Writing
Pattern Exposure	100%*	97.2%*	39.2%**	87.8%*	64.9%*
Active Pattern Exposure	0%*	92.4%**	4.4%*	70.9%*	36.9%*
Pattern Acquisition	44.0%*	55.3%*	20.7%**	43.3%*	33.7%*
Code Writing Transfer	55.3%**	85.3%**	74.7%**	27.7%	28.7%
Actual Difficulty	6.7%**	35.3%*	29.3%*	51.2%	53.5%

Pattern Acquisition: Do students recognize and apply the relevant pattern in `writing` exercises?

Though students are exposed to pattern-adherent code, they may not internalize the patterns as a general solution or may be unable to recall them when given a relevant exercise prompt. Unlike techniques like Pattern-Oriented Instruction (Muller et al., 2007), students were never given explicit instruction on the patterns or when to use them. We analyze whether students obtain sufficient mastery from the exposure exercises in each exercise interface to both recognize the opportunity to apply a pattern and generate code for that pattern in `writing` exercises in the acquisition phase.

We find that students are more likely to acquire patterns if they are first exposed to them as `parsons` (or `tracing`) compared to `writing`. In Study 1, there is no statistically significant difference in the rate of Pattern Acquisition of `parsons` (55.3%) and `tracing` (44.0%), though both are higher than `writing` (20.7%). In Study 2, the rate of Pattern Acquisition is higher in `parsons` than `writing` (43.3% vs. 33.7%, p¡.01).

A Special Case

Interestingly, we found one pattern, OOP1, where Code Writing exercises were more effective at teaching the programming pattern (though not statistically significant). Further investigation found that for this pattern's `exposure` exercises, Pattern Exposure was nearly identical between the `parsons` condition and the `writing` condition (73.9% vs. 73.7%). However, participants were more likely (not statistically tested) to view the instructor solution in the `writing` condition (57.9% vs. 28.3%).

This pattern involved creating classes with methods that returned generators, and part of the pattern was ensuring that `yield` was at the end of the end of the defining function. This requirement was included as a possibly misguided way to improve readability. The instructor solutions made an explicit reference to this, "# yield is the last line of the loop." This highlights a weakness of `parsons` for certain patterns. Though Active Pattern Exposure might be better in most cases, in this case students were less likely to view the well-documented instructor solutions after solving it correctly in `parsons`, so they did not notice this subtle attribute of the pattern. However, this issue only arose in one of the 7 patterns used in this study. For all exercises, instructors trying to teach patterns could benefit from tools that aggregate student solutions to see if they are being solved in the expected way.

5.6.4 General Efficacy

We also analyze the effect of introducing Faded Parsons Problems beyond teaching programming patterns. We want to understand how practice with each exercise interface affects students' ability to successfully complete subsequent `writing` exercises with similar solutions (Code Writing Transfer). Open-ended code writing tasks are a well-established measure of mastery in many Computer Science courses. Results can be seen in Table 5.3.

Code Writing Transfer

Both `parsons` and `tracing` exercises are a meaningfully different type of practice for students compared to `writing`. As `writing` is a well-established goal of programming expertise in introductory Computer Science courses, we evaluate students' success on `writing` questions.

In Study 1, the Code Writing Transfer rate is highest for `parsons` (85.3%), followed by `writing` (74.7%) and then `tracing` (55.3%). However, in Study 2, we find no significant difference between `parsons` (27.7%) and `writing` (28.7%). The poor Code Writing Transfer from `tracing` motivated its exclusion from Study 2.

5.6.5 Student Perception of Faded Parsons Problems

The Subjective Study was designed to gain insight into student perception of `parsons`. For this, we restrict our analysis to participants that effort-completed all Study 1 exercises, as they had reasonable exposure to both `writing` and `parsons`. All quotes are from open-ended survey responses.

Student Preference for `parsons`

The Subjective Study included 7 exercises where students, after reading the problem statement, could choose to solve an exercise as `parsons` or `writing` and then were asked to explain their choice. We compare to a distribution of students choosing randomly between the two, and find students were much more likely to choose `parsons` (77.6%) over `writing` (22.4%). The primary reason for choosing `parsons` was their perceived difficulty, further analyzed below. Separately, 22 students chose `parsons`in order to focus on the structure of the solution and "allowing them to think like the instructor" (P96), making this the next most frequent explanation. However, 5 students chose `writing` for this same reason, preferring the "additional freedom" (P135) of "starting from scratch" (P114) to create "a more intuitive solution" (P85).

Perceived Difficulty

In the Subjective Study, participants were asked to fill out a survey including a Likert-scale question about whether `writing` was easier to solve, `parsons` was easier to solve, or both are about the same. We find that 26 students (57%) thought `parsons` were easier, 12 students (26%) thought they were similarly difficult, and only 8 students (7%) thought `writing` were easier. Many students echoed this perception in open-ended questions, with 35 explaining their choice of `parsons` as choosing the easier exercise interface. However, 6 chose `writing` for this same reason, thinking they would get "more of a real practice" (P160) when forced "to start thinking on their own" (P137).

Actual Difficulty

We also analyze the rate at which students effort-completed exercises from Study 1 and Study 2 without solving them correctly. Previous work suggests that other variations of Parsons Problems are easier than Code Writing exercises (Ericson, Margulieux, et al., 2017), but Faded Parsons Problems have not yet been assessed for relative difficulty. In Study 1, we analyze exercises from the third week, where we instantiated all 9 *(pattern, exercise interface)* combinations, since the exercise interfaces equally represent each pattern. In Study 2, we analyze the exercises from the exposure phase, comparing treatment to control.

We find that `parsons` and `writing` provide similar difficulty to students. In Study 1 there is no significant difference between `writing` (29.3%) and `parsons` (35.3%). In Study 2, we again find no difference between `writing` (53.5%) and `parsons` (51.2%). These results conflict with the student perception that `parsons` are easier than `writing`.

5.6.6 Synthesis Of Results

This work has found strong evidence that Faded Parsons Problems are an effective exercise for exposing students to patterns and having them correctly transfer them to subsequent exercises. By contrast, open-ended Code Writing exercises – a widely used, popular approach to programming exercises – often do not expose students to the intended patterns even when a well-documented instructor solution is provided after the fact. Practice with Faded Parsons Problems also shows similar transfer to general programming success in subsequent Code Writing exercises over similar content. Though Code Tracing exercises are fairly effective at having students transfer patterns as well, they are ineffective at transferring to general programming success in subsequent Code Writing exercises. Code Tracing exercises offer clear evidence for how practicing with exercises can support students in practicing certain programming skills while insufficiently practicing others. Faded Parsons Problems appear to offer a good balance between Code Tracing and Code Writing exercises, teaching both patterns and general coding skills as well as either of those two interfaces. Furthermore, students had a preference to work with Faded Parsons Problems over Code Writing exercises, thinking of Faded Parsons Problems as easier problems despite their similar actual difficulty.

5.7 Limitations and Future Work

Notwithstanding the evidence in favor of integrating Faded Parsons Problems into CS1 courses, several questions remain open. We have not compared Faded Parsons Problems systematically to other Parsons Problem variants or to Code Skeleton questions, in which lines are already arranged but contain blanks, nor have we systematically studied the effects of what and how much code is blanked out in fading the scaffolding. We also do not suggest that all Code Writing exercises should be replaced with Faded Parsons Problems; indeed, we found one pattern, OOP1, for which Code Writing exercises provided more effective practice. Finally, we studied a limited number of patterns chosen based on the instructor's existing curriculum rather than on any systematic survey of the importance of different patterns.

The need to be minimally disruptive to the existing curriculum imposed additional constraints that may affect validity of results. First, the study exercises were offered as extra credit rather than required, so self-selection may have biased the study population towards more highly motivated students. Second, Study 2's problems required students to learn more material in addition to the patterns, and students had only 2 exercises in which to learn the patterns, compared to 5 in Study 1. Finally, to remain consistent with the instructor's existing behavior of grouping similar problems together in assignments, both studies intentionally exposed students to the same patterns in consecutive exercises, which might boost students' ability to recognize and apply patterns in the pattern acquisition phase of each study.

On the other hand, the positive results do suggest exploring other uses of scaffolding. For example, instead of giving students instructor solutions after homework assignments were due, what if students instead got points for "unlocking" them by solving Faded Parsons Problems or Code Tracing exercises? In addition, students preferred Faded Parsons Problems over code-writing in part because Faded Parsons Problems were perceived as easier, even though our results suggest otherwise; we have not studied the potentially positive effect of this perception on student self-efficacy.

5.8 Conclusion

The studies we describe provide clear evidence that Faded Parsons Problems are particularly effective at teaching programming patterns in CS1 courses compared to code-tracing and code-writing exercises, without detracting from the ability to *transfer* this knowledge to code-writing exercises. Because Faded Parsons Problems can be created easily from existing code-writing exercises, they can be introduced into existing curricula with minimal disruption. Because they provide immediate feedback, they are particularly valuable for promoting mastery learning in online instruction, where immediate manual feedback may be impossible. Because students piece together and complete an instructor-designed solution, students are more likely to be exposed to a high-quality solution than they would be in constructing their own solution from scratch, which could provide opportunities beyond patterns. These benefits, combined with students' stated preference for Faded Parsons Problems over code-writing exercises, provide strong evidence in favor of integrating such problems widely into introductory programming courses.

Chapter 6

A Design Framework for Creating Reconstruction Exercises that Teach Software Architecture Patterns

The previous study showed exciting results for a CS1 context. While I think there is still a great opportunity to explore how upper-division students could be supported by Faded Parsons Problems, I felt I could make a bigger contribution by digging more into *why* Faded Parsons Problems are effective. By connecting qualitative data with related work, I identified a set of design goals that could help understand what made Faded Parsons Problems effective. To probe these design goals, I then designed two new exercises targeted at students in an upper-division Software Engineering course focused on web development. These exercises were evaluated through an in-lab, qualitative study with 12 students of the course. The study suggests promise, both for the design goals and the specific exercises. In particular, I report on the effectiveness of these exercises as a teaching tool, the appropriateness of each design goal, and the biggest opportunities to further improve these particular exercises.[1]

6.1 Introduction

Despite the increasing need for effective programmers (Scaffidi et al., 2005; Barr et al., 2011; *U.S. Department of Labor, Occupational Outlook Handbook, Software Developers* 2019), developing and maturing programming ability continues to be a significant challenge in part because programming is a complex task consisting of many skills and types of knowledge (du Boulay, 1986; Jenkins, 2002; Cutts et al., 2012; Valstar et al., 2019).

One skill required by professional programmers is *problem decomposition* (or planning), or the ability to take a complex goal and break it down into smaller, more manageable goals. As programmers decompose a problem, they must evaluate countless different boundaries at which to break down the problem. To ensure that each smaller goal is manageable,

[1]At the time of writing, this work is currently under review at a conference.

programmers rely on *patterns*. Patterns may be a way of achieving a specific goal (e.g., counter variables in loops, recursion with base cases), or a way to organize responsibilities (e.g., singletons, Model-View-Controller (MVC) web architectures).

Experts have been found to have access to more patterns than novices (Wiedenbeck et al., 1993; Robins et al., 2003; Ko et al., 2008; McCauley et al., 2008; O'Dell, 2017). However, the programming assessments we use in courses often do not give feedback on students' use of patterns or problem decomposition. Autograders and linters are two commonly used tools in CS classes. Autograders typically focus on the output of programs, while linters typically focus on low-level style (e.g., variable naming conventions). Neither of these tools focus on the algorithms being used by the student nor any thinking they did before they began writing code. That is, our current programming assessments do not provide students an opportunity to engage in deliberate practice with problem decomposition and patterns, which has been found to be a key predictor of continued learning in a given field (Ericsson et al., 1993).

In this work, we explore how to create exercises that provide students this much-needed deliberate practice. Specifically, we explore how we can offer this practice to upper-division students in a Software Engineering course, as their problems often span multiple functions, files, and programming languages. We identify a set of 5 design goals to create exercises aimed at scaffolding students to provide deliberate practice. We then create two new exercises focused on web architecture patterns, Subgoal Decomposition and Data Flow. In these exercises, students reconstruct many intentionally-designed solutions. We run a qualitative study with 12 upper-division students in a Software Engineering course to evaluate these exercises. Our study explores the appropriateness of our design goals, the effectiveness of these exercises as a teaching tool, and the biggest opportunities to further improve these exercises.

Our contributions are as follows:

- We identify a set of 5 design goals.
- We create two new exercises from those design goals, focused on teaching upper-division students MVC architecture and how to use third party services authentication in a web application.
- We probe students attitudes towards these exercises, finding significant potential for these design goals and exercises.

6.2 Related Work

6.2.1 Problem Decomposition

Soloway (1986) describes programming as a "design discipline," where programs need not only produce an output but provide a trail of why they were designed in a certain way. This trail is communicated through using common plans, and is constructed by someone who is able to iteratively break down goals repeatedly into subgoals until they arrive at these common plans. Basu et al. (2015), in work focused on improving automated assessment, posit that “solving technical problems” can be broken down into three stages: problem understanding, problem planning, and implementation, which should be completed in that order. That is, students and programmers should have confidently planned a solution before working on implementing it.

Past work substantiates the benefits of problem decomposition, or top-down thinking. Davies (1991) analyzed code generation of expert and novice programmers. Though they found a mix of top-down and opportunistic programming in both groups, experts spent more time at the start using a top-down approach. Song et al. (2014) found similar results when interviewing undergraduate first years, seniors, and experts, noting that experts used more problem decomposition, allowing them to take a breadth-first approach to solving the problem. Burton-Jones et al. (2008) explored how decomposition quality affected novices’ problem solving performance, finding that being given higher quality decompositions improved student performance.

Though problem decompositions show promising correlations, many students struggle to use this skill effectively. Lahtinen et al. (2005) conducted a survey of over 500 students and teachers on difficulties in learning and teaching programming, finding that “design[ing] a program to solve a certain task” is one of the largest challenges. Kwon (2017) studied novices creating pseudocode to solve problems, finding that students struggled in building the strategic understanding necessary to create useful plans that could then be translated into programs. PlanIT! (Milliken et al., 2021), Green (Alphonce et al., 2005), and CIMEL ITS (Moritz et al., 2005) are tools that work to integrate plans and code with each other into a single development environment. The goal of these tools is to improve problem solving ability, reduce the friction to creating plans, and support students in more directly connecting plans and code. Shi et al. (2019) motivated their work form a more indirect perspective, designing Pyrus, an interface where students work together with limited knowledge to construct a program. By constraining students’ actions in the solving exercises, they found that students spent more time planning their solutions. Cunningham et al. (2021) design a set of exercises focused on purpose-first programming, where participants progress from selecting and ordering English plan goals to filling in small code snippets. They find that participants from a secondary data-oriented programming course could complete scaffolded code writing and debugging activities after only 30 minutes of instruction.

Our work builds on the motivation that problem decomposition is a valuable skill that is often under-practiced in courses today. When writing code, while nothing prohibits students from creating a plan for their program, students are also free to minimally plan their solution and jump straight into implementation. Our exercises are designed as a sequence to go from high levels of abstraction to lower ones. Additionally, our work anchors heavily on the use of subgoals as an intermediate representation before introducing code. This adds an explicit layer between the high-level goal of the problem statement and the low-level implementation details of code.

6.2.2 Programming Patterns

To decompose problems, students must recognize opportunities into how to decompose a goal into *more manageable* subgoals. A key component to evaluating whether a subgoal is manageable is recognizing if a subgoal matches a known pattern. Patterns (or plans, schemas, templates) are higher-level, reusable abstractions of code (Spohrer et al., 1986; Wiedenbeck et al., 1993; Marshall, 1995; Astrachan et al., 1998; Clancy et al., 1999; Muller et al., 2007; Xie et al., 2019; Milliken et al., 2021). Patterns may be grounded in specific syntax or be more abstract; they may be a way of achieving a specific goal or a way to organize responsibilities. Studies have found evidence that experts have access to more patterns than novices, allowing experts to be more effective programmers (Wiedenbeck et al., 1993; Robins et al., 2003; Ko et al., 2008; McCauley et al., 2008; O'Dell, 2017).

Castro et al. (2016) analyzed CS1 students responses to scaffolded code writing exercises. They found that students run into challenges when they don't have a pattern from a similar problem. Instead of decomposing the problem, they will start from a small code fragment as a focus and expand outwards as they construct their program. Ko et al. (2008) built the Whyline debugger, a tool that allows developers to ask why did and why didn't questions about a programs output. They found the tool improved debugging efficiency of novices by helping them trace through code in a framework leveraging unfamiliar patterns. McCauley et al. (2008) reviewed literature on debugging, and found that experts are more likely than novices to take a breadth-first approach in understanding a system when tackling bugs. Unlike novices, these experts are able to leverage patterns they've seen to understand the structure of the system and begin to identify hypotheses for issues.

Though programmers develop patterns over time, more can be done to effectively teach patterns in CS classes. Muller et al. (2007) propose Pattern-Oriented Instruction in introductory Computer Science classes, where patterns are explicitly incorporated into a course's instruction. They found that this explicit change to instruction led to novices improving their problem decomposition and solution construction skills. Xie et al. (2019) also designed a curriculum focused on explicit instruction of templates. They find some evidence that this new template-oriented curriculum improved student coding ability. Chunking from cognitive theory highlights one way students are currently acquiring patterns without this explicit instruction. It suggests that students, as they view examples with identifiable similarities, construct and store more complex patterns as single cognitive "chunks" which they can retrieve later (Chase et al., 1973; Marshall, 1995; Castro et al., 2016).

Weinman, Fox, and M. A. Hearst (2021) builds off this theory, finding that when students work on a series of problems with syntactically similar solutions, they are more likely to acquire patterns and correctly apply them to future code writing exercises. Specifically, they used Faded Parsons Problems, an exercise which requires students to unscramble and complete lines of code to construct a syntactically and logically correct program. By providing students with instructor-defined lines of code, Weinman et al. were able to ensure students were constructing solutions that matched the intended syntax-based patterns.

Our work builds on the motivation that patterns are both important and could be better taught in our courses. This work prioritizes exploring how to improve learning through assigned programming exercises over curricula changes. This work differs, however, in its population and content. This work focuses on upper-division students that have deeper programming experience than CS1 students and work on larger problems with more abstract patterns. Additionally, instead of solely focusing on goal-based syntactic patterns, this work also focuses on organizational patterns (i.e., the MVC framework).

6.3 Design Goals: Reconstruct Many Intentionally-Designed Solutions

Writing programs requires programmers to draw on a range of skills and different types of knowledge. While code writing exercises are a valuable opportunity to practice all of those skills in conjunction, we want to create exercises that provide opportunities to deliberately practice a subset of those skills; specifically, problem decomposition and leveraging architectural patterns. Faded Parsons Problems, in which students unscramble and complete lines of code, have been found effective at helping students acquire syntactic patterns that span a few lines (Weinman, Fox, and M. A. Hearst, 2021). However, Faded Parsons Problems have students engage with individual lines of code, which is reasonable only for shorter programs, such as those used in an introductory CS course. In this section, we present a set of design goals motivated by educational theories and attributes of Faded Parsons Problems. By extracting these design goals, researchers and instructors can create new exercises that are relevant in a range of domains beyond CS1. We present a set of 5 design goals, which can be collectively captured as creating opportunities for students to *reconstruct many intentionally-designed solutions*.

6.3.1 Construct: Provide Opportunity to Practice

Formative assessments like homework are critical to the classroom. Exercises like code writing involve "active learning" – having students reflect on ideas and how to use them. Active learning is a powerful pedagogical tool (Michael, 2006). When assigning an exercise for a course, it is important to examine what is given to students and what they must actively reflect on. Code tracing exercises, where students predict the output of a program, are effective at having introductory students engage with the notional machine (Boulay et al., 1981; Griffin, 2016). However, it is less effective at teaching students to construct programs, as the correct logic is already given to them. Code writing exercises, in contrast, force students to actively engage with every part of introductory programming, as they must construct code unaided.

Parsons Problems have been found to correlate more closely with code writing exercises than code tracing for assessment or transfer of general code writing ability (Denny et al., 2008; Ericson, Margulieux, et al., 2017; Weinman, Fox, and M. A. Hearst, 2021). Though

Parsons Problems provide code snippets, students must still reason how the lines connect to correctly construct the logical flow of the program. Exercises should allow students to actively practice the concepts they are studying by *constructing* some component of the solution.

6.3.2 Reconstruct: Scaffold Vocabulary and Details

To balance the design goal above, exercises must not be overwhelming to students by giving too many things to practice on. Programming is a complex task consisting of many different skills and types of knowledge (du Boulay, 1986; Jenkins, 2002; Cutts et al., 2012). Cognitive Load theory poses that students have limited mental capacities while working on problems, and if overtaxed they will be unable to reflect on and store their learning from completing exercises (Sweller, 1988; Renkl et al., 2002). One way to reduce cognitive load is to provide more structure or material to students.

Parsons Problems scaffold syntactic knowledge by providing lines of code. Traditional Parson Problems adjust this scaffolding by using syntactic or semantic distractors as lines of code that will not be used in the solution. Faded Parsons Problems adjust this scaffolding by omitting instructor-selected code fragments within the lines of code. These can be particularly beneficial when asking students to practice with new syntax and concepts, as students are able to recognize them before being able to compose with them.

What the exercise scaffolds must be carefully selected to balance with the previous design goal. Exercises should provide students with pieces that are not a learning goal of the exercise. This allows students to focus their learning on what isn't given to them as they *reconstruct* a solution from those pieces.

6.3.3 Reconstruct: Scaffold Process

In programming courses, it is common to grade students primarily on the completeness of the final program. However, though it can be difficult to grade (Ju et al., 2018), instructors also want students to practice different processes (e.g., top-down problem decomposition, test-driven development, pair programming).

Parsons Problems lightly follow this design goal, in that a top-down thinking approach of data or control flow can make them easier to solve. However, these processes are often less apparent in introductory Computer Science courses, where full assignments can consist of 3-10 lines of code. In upper-division courses, it is common for the solution to an assignment to contain hundreds of lines of code in multiple languages and files. In these more complex domains, exercises should be designed such that students follow good processes as they *reconstruct* their solutions.

6.3.4 Intentionally-Designed Solutions: Guide Students to One Solution

There are many ways to construct a program that achieves a specific goal. For example, for loops can be replaced by while loops, or an algorithm using iteration can also be solved with recursion. However, sometimes one approach is much simpler to reason through than the other. Or an instructor might want to intentionally have students grapple with a newer approach. At a more granular level, instructors may want to expose students to better-styled or more compact solutions. In code writing exercises, any expectations to have students solve a problem with particular syntax or approaches are limited to the adjusting the problem description.

Parsons Problems, by design, often have only one solution. Because the code fragments are provided to students, an instructor can guarantee that students solve the problem in a certain way. This allows instructors to guarantee that they are exposing students to well-structured solutions that cover certain learning goals (Weinman, Fox, and M. A. Hearst, 2021). Exercises should expose students to *intentionally-designed solutions* by the instructor.

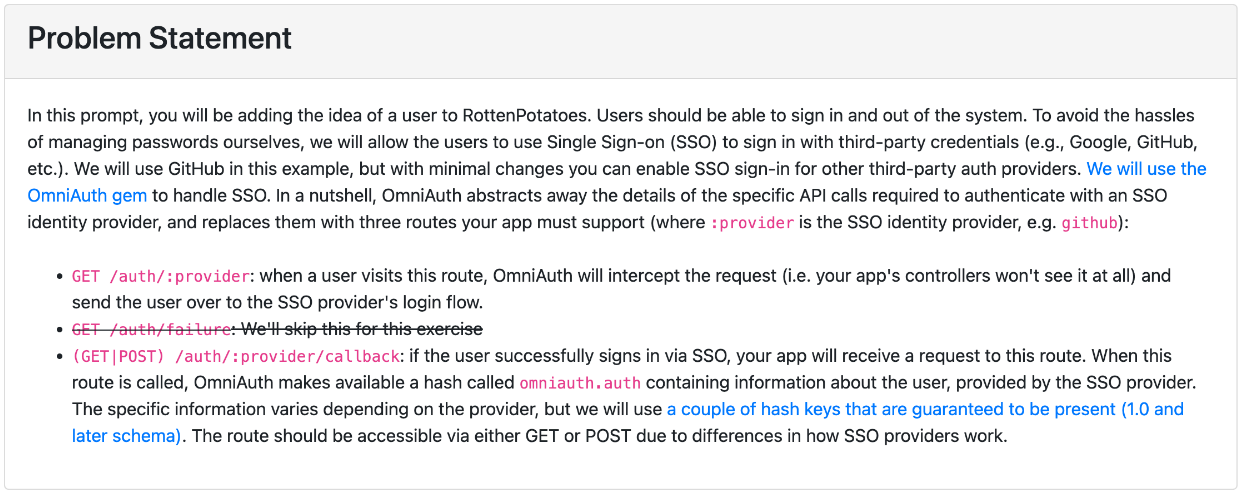
6.3.5 Many solutions: Present Examples Efficiently

Constructivism posits that people learn by comparing new experiences to what they already know, identifying similarities and differences, and synthesizing that into new knowledge (Fosnot, 2013). Indeed, programmers become more effective over time by seeing more programs and developing generalized “schema” they can apply to future problems (Rist, 1991). In introductory courses, problem sets often consist of several problems that build off each other and can each be solved in minutes.

Ericson, Margulieux, et al. (2017) found that Parsons problems take only 70% of the time as code writing exercises in introductory Computer Science courses. This could, for example, allow teachers to assign 7 Parsons Problems and have it take as long as 5 code writing exercises, giving students two more examples to build new schema. Exercises should efficiently expose students to *many solutions* to help them build generalizable knowledge.

6.4 Designing New Exercises

We created two exercises for this study. These exercises were created for a Software Engineering class that uses Rails to teach web programming. Before designing the exercises, the first author interviewed two professors (one of whom is an author) and two TAs of the Software Engineering course. Instructors want students to practice working on all parts of a web application (e.g., databases, testing, configuration files, server-side logic, client-side logic). The current assignments provide significant instructional scaffolding, breaking down the problem for students. However, instructors and TAs noticed that students often created solutions that, while passing the functionality requirements, were not well structured or ad-


Problem Statement

In this prompt, you will be adding the idea of a user to RottenPotatoes. Users should be able to sign in and out of the system. To avoid the hassles of managing passwords ourselves, we will allow the users to use Single Sign-on (SSO) to sign in with third-party credentials (e.g., Google, GitHub, etc.). We will use GitHub in this example, but with minimal changes you can enable SSO sign-in for other third-party auth providers. We will use the OmniAuth gem to handle SSO. In a nutshell, OmniAuth abstracts away the details of the specific API calls required to authenticate with an SSO identity provider, and replaces them with three routes your app must support (where `:provider` is the SSO identity provider, e.g. `github`):

- `GET /auth/:provider`: when a user visits this route, OmniAuth will intercept the request (i.e. your app's controllers won't see it at all) and send the user over to the SSO provider's login flow.
- ~~`GET /auth/failure`: We'll skip this for this exercise~~
- `(GET|POST) /auth/:provider/callback`: if the user successfully signs in via SSO, your app will receive a request to this route. When this route is called, OmniAuth makes available a hash called `omniauth.auth` containing information about the user, provided by the SSO provider. The specific information varies depending on the provider, but we will use a couple of hash keys that are guaranteed to be present (1.0 and later schema). The route should be accessible via either GET or POST due to differences in how SSO providers work.

Figure 6.1: **Upper-Division problem prompt.** In this study, students worked on adding 3rd party authentication to an existing app.

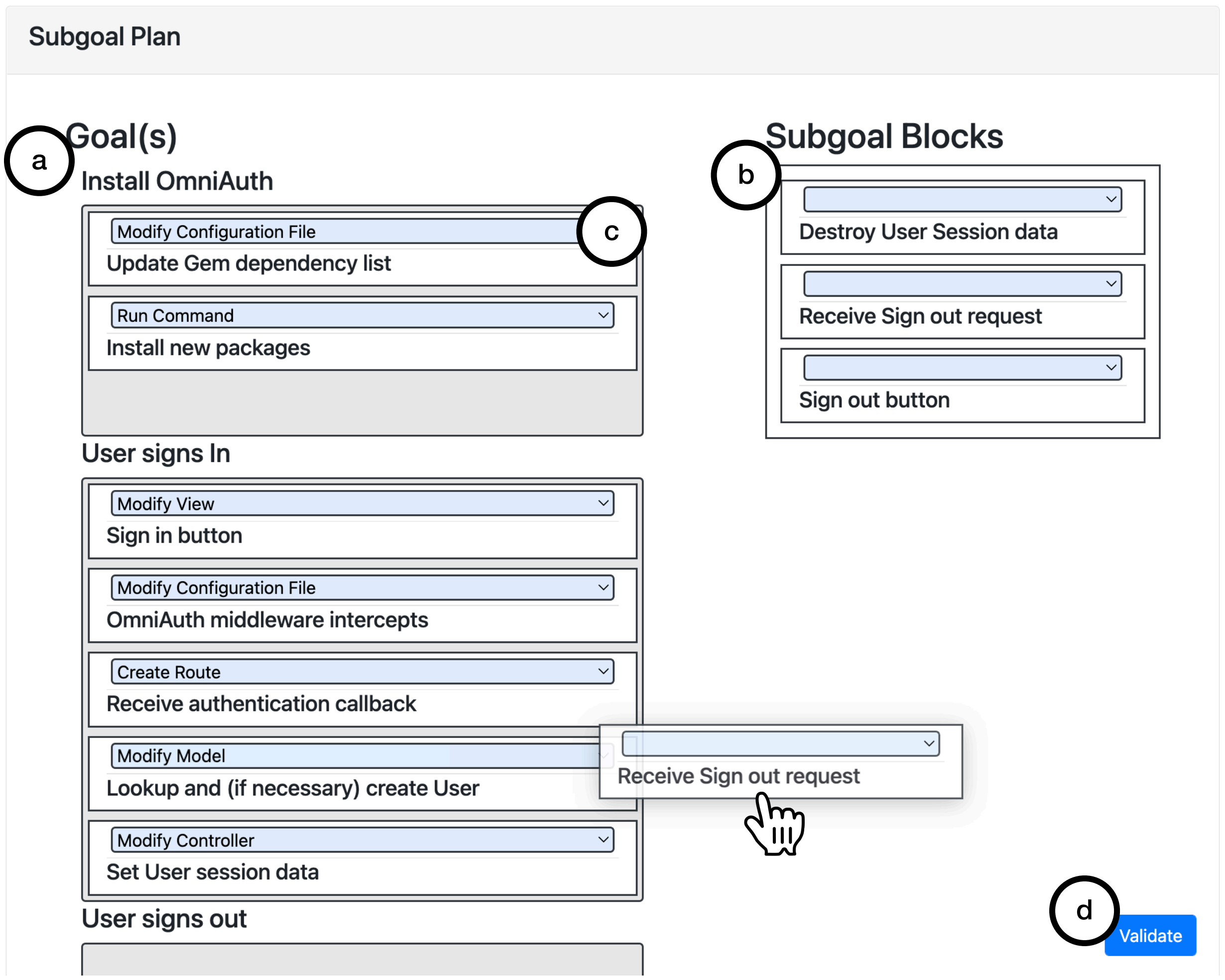
herent to MVC responsibilities. In office hours, TAs noticed that students often struggled with knowing what components to look in when debugging their own programs. Based on these interviews, and the design goals above, we created a sequence of exercises to help students focus on these problems whose goal was to be at the right level of abstraction to build architectural patterns.

Instructors had noted that students struggled with understanding how asynchronous requests should work in the context of their applications. Instructors had already created a draft assignment covering this topic by asking students to add third party authentication to their application using a public library (Figure 6.1).

6.4.1 Exercise 1: Subgoal Decomposition

The first exercise can be seen in Figure 6.2. In this exercise, students are asked to arrange subgoals into the correct goal, and order them based on execution flow. Additionally, students must specify where in the Rails MVC framework they would modify code to implement the subgoal.

This exercise provides students an opportunity to *practice* how the subgoals work together to achieve a goal as well as their understanding of boundaries in the Rails MVC framework. It *scaffolds vocabulary* by providing subgoals to students, giving them a set of smaller problems that can be combined to achieve a larger goal. While it provides examples of well-separated subgoals, the learning goal is for students to understand how these subgoals connect and how the Rails MVC framework helps define them. It *guides students to one solution* because, at the level of abstraction of these subgoals, there is a single correct solution based on execution


Subgoal Plan

Goal(s)

Install OmniAuth

Modify Configuration File
Update Gem dependency list

Run Command
Install new packages

User signs In

Modify View
Sign in button

Modify Configuration File
OmniAuth middleware intercepts

Create Route
Receive authentication callback

Modify Model
Lookup and (if necessary) create User

Modify Controller
Set User session data

User signs out

Subgoal Blocks

Destroy User Session data

Receive Sign out request

Sign out button

Receive Sign out request

Figure 6.2: **A mostly solved Subgoal Decomposition exercise.** (a) Students are given a list of goals, each of which begin empty. (b) Students drag Subgoal Blocks from the right into the appropriate goal on the left, ordered within each goal by execution flow. (c) Students also specify where in the Rails MVC framework they would modify code. (d) Students can validate their answer at any point.

Table 6.1: **Connecting Design Goals to Exercises.**

	Subgoal Decomposition	**Data Flow**	**Faded Parsons Problems**
Practice	Rearranging subgoals, Framework placement	Data creation, Connectors	Rearranging lines of code
Vocabulary	Subgoals, Framework	Syntax	Syntax
Process	Subgoal layer	Subgoal layer	Not necessary
One Solution	Given subgoals, Focus on execution	Given data, Given connectors	Given lines of code
Efficiency	Section 6.6.2	Section 6.6.2	Previous research (Ericson, Margulieux, et al.,

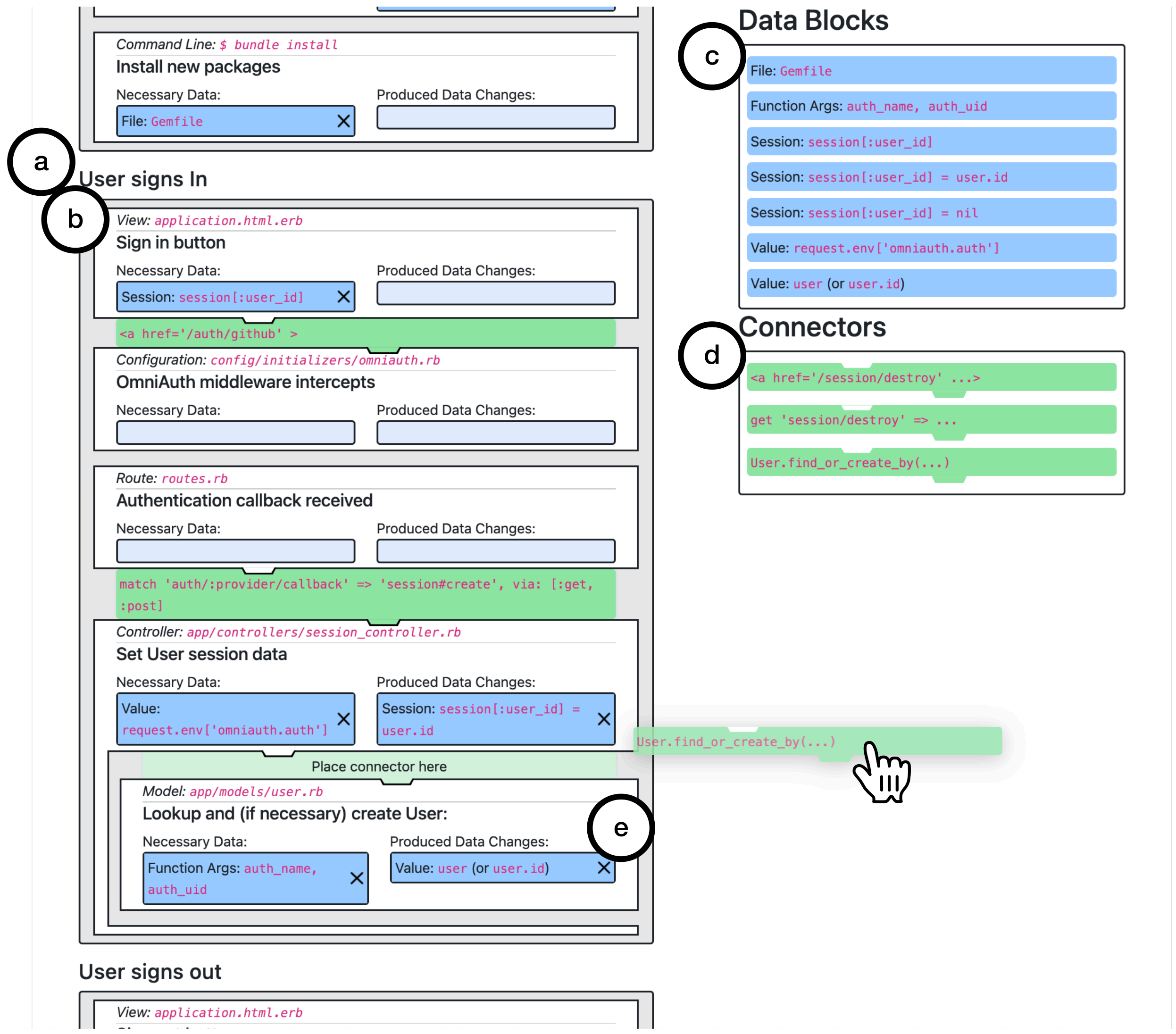
flow. Finally, it is designed to *present examples efficiently* by being significantly faster than writing code, but that can only be validated through user observation (Section 6.6).

6.4.2 Exercise 2: Data Flow

The second exercise can be seen in Figure 6.3. This exercise grounds the abstract subgoals into more specific implementation details, focusing on how data flows throughout the subgoals to eventually achieve the desired effect. In this exercise, students are asked to correctly place connectors, the code snippets that indicate to Rails that control flow should continue to the next subgoal. Additionally, for each subgoal, they must specify any data the that is *necessary* for the subgoal to execute or *produced* by the subgoal to be used later. These data represent variables such as function arguments, instance variables, and file changes.

This exercise provides students an opportunity to *practice* how data is read, written, and updated to achieve each goal. It *scaffolds vocabulary* by providing concrete code snippets to students. While it provides examples of relevant syntax, other exercises are better suited to teach students to generate their own syntax. It *guides students to one solution* because the implementation details are self-contained enough to be unique; for example, a subgoal might be responsible for taking a unique id and returning a user object, but the exercise does not specify *how* that look-up is achieved. Additionally, the Rails MVC framework is opinionated about where in the framework certain data transformations occur; for example, access to the User database should be done in the Model. Finally, this exercise is also designed to *present examples efficiently*, but it must be validated through user observation.

Though not explored in this study, Faded Parsons Problems could be used to continue the progression towards lower layers of abstraction, having students work on Faded Parsons Problems for each subgoal after completing the Data Flow exercises. Our new exercises could not only be used effectively in conjunction with Faded Parsons Problems, but are also examples of generalizing the effective attributes of Faded Parsons Problems to more complex domains. Table 6.1 provides a summary of how all three exercises connect to our design goals.



Command Line: `$ bundle install`
Install new packages
Necessary Data: File: `Gemfile` ✕
Produced Data Changes:

User signs In

View: `application.html.erb`
Sign in button
Necessary Data: Session: `session[:user_id]` ✕
Produced Data Changes:

```
<a href='/auth/github' >
```

Configuration: `config/initializers/omniauth.rb`
OmniAuth middleware intercepts
Necessary Data:
Produced Data Changes:

Route: `routes.rb`
Authentication callback received
Necessary Data:
Produced Data Changes:

```
match 'auth/:provider/callback' => 'session#create', via: [:get,
:post]
```

Controller: `app/controllers/session_controller.rb`
Set User session data
Necessary Data: Value: `request.env['omniauth.auth']` ✕
Produced Data Changes: Session: `session[:user_id] = user.id` ✕

Place connector here

```
User.find_or_create_by(...)
```

Model: `app/models/user.rb`
Lookup and (if necessary) create User:
Necessary Data: Function Args: `auth_name, auth_uid` ✕
Produced Data Changes: Value: `user` (or `user.id`) ✕

User signs out

View: `application.html.erb`

Data Blocks

File: `Gemfile`
Function Args: `auth_name, auth_uid`
Session: `session[:user_id]`
Session: `session[:user_id] = user.id`
Session: `session[:user_id] = nil`
Value: `request.env['omniauth.auth']`
Value: `user` (or `user.id`)

Connectors

```
<a href='/session/destroy' ...>
get 'session/destroy' => ...
User.find_or_create_by(...)
```

Figure 6.3: **A mostly solved Data Flow exercise.** (a) Students are given the same list of goals and subgoals as the previous exercise. (b) Each subgoal specifies the file and location in the framework where its code belongs. There are containers in which students can drop Data Blocks to specify necessary data and produced data changes by each subgoal. (c) Students must correctly drag Data Blocks into the correct subgoals. Data Blocks can be used multiple times. (d) Students must also drag Connectors between subgoals where appropriate. (e) Students can remove any incorrectly placed Data Blocks by pressing an X button. (not pictured) Students can validate their answer at any point.

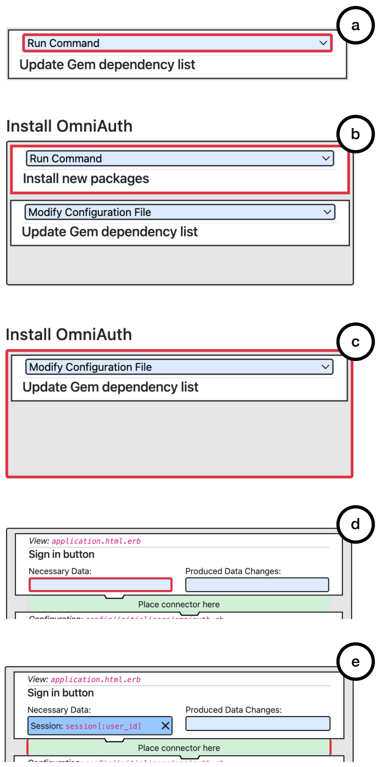


Figure 6.4: **Different types of feedback students receive on these exercises.** Only the first mistake was highlighted. (a) Subgoal Decomposition, when a subgoal was placed in an incorrect part of the framework. (b) Subgoal Decomposition, when a Subgoal was placed in the incorrect goal or incorrect order within a goal. (c) Subgoal Decomposition, when a goal is incomplete. (d) Data Flow, when a subgoal has an incorrect Data Block placed (or missing) in a subgoal. (e) Data Flow, when a subgoal has an incorrect Connector placed or missing between subgoals.

6.4.3 Feedback Mechanism

In this study, we designed the exercises so that students could receive feedback at any point as they worked toward their solution. All feedback types can be seen in Figure 6.4. When students requested feedback, the first incorrect component was highlighted in red. The Subgoal Decomposition exercise provided feedback at three levels of granularity: framework location, subgoal placement and order, and goal completeness. The Data Flow exercise provided feedback to highlight incorrect Data Blocks and incorrect Connectors. We discuss opportunities to improve this feedback mechanism in Section 6.6.3.

6.4.4 Creating Content

The previous subsections describe the exercises designed for this study and how they are motivated by the design goals. However, they do not describe how to create the content for these exercises. To create a problem, the instructor must specify: *goals*, *subgoals*, a *framework* in which subgoals can be placed, *connectors* between each subgoal, and the key *data* that is created and used to achieve a functional solution.

To create this content, we began with a problem prompt for an in-development exercise that had not yet been implemented in the course (Figure 6.1), along with a complete solution to the problem. We removed the code- and file-specific scaffolding from the prompt. To create *goals*, we considered series of actions the programmer makes for configuration (e.g., installing libraries) and any action the user could take to start a process (e.g., clicking on a button). To create *subgoals*, we used all file and function boundaries in the correct solution to create the list of subgoals; that is, each subgoal referenced code in no more than one file, and no function was represented by more than one subgoal. To create the *framework* options, we leveraged terminology from the Rails MVC framework. To create the *connectors*, we extracted the code snippets from the reference solution responsible for code execution to flow to the next subgoal. To create *data* choices, we extracted side effects (e.g., instance variables, file changes), function parameters, and return values. Finally, we removed a handful data changes which were deemed to be out of scope (e.g., Gemfile.lock).

6.5 Evaluation

We ran a study to explore the following research questions:

- What was the efficacy for learning of these exercises? (Section 6.6.1)
- What is the impact of each design goal? (Section 6.6.2)
- How might we further improve these exercises? (Section 6.6.3)

We invited all students enrolled in the upper-division Software Engineering course at a large US research university. None of the researchers were instructors of the course that semester. We recruited 12 participants (P1-P12, 8 men and 4 women). Participants were compensated $15/hour for their time.

The study was structured as follows (see Figure 6.5). Participants began each 90-minute session by signing a consent form and were then interviewed in a semi-structured manner to gain understanding about two questions: (i) the students' experiences in the course so far and (ii) their experiences with current code writing assignments. Participants then read a problem statement and worked on that same problem through three exercises: (i) create as complete a problem decomposition as possible without any tooling; (ii) solve the problem as a Subgoal Decomposition exercise (Fig. 6.2); (ii) solve the problem as a Data Flow exercise (Fig. 6.3). After each of these three exercises, participants were asked three questions to

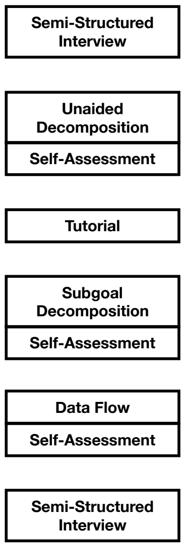


Figure 6.5: **Study description**. The series of tasks participants worked through. Participants had 10 minutes for the unaided decomposition exercise, and 25 minutes combined for the Subgoal Decomposition and Data Flow exercise.

self-assess their confidence in their understanding. Participants had 10 minutes for the initial problem decomposition, and 25 minutes combined to work through the second two exercises. Additionally, participants worked through a tutorial for the two new exercises before working on either of them. After completing the exercises, participants concluded by sharing their thoughts in a semi-structured interview reflecting on their experiences working with these new exercises.

The study took place remotely using Zoom. Participants shared audio and their screen while working through the tutorial and three exercises, and optionally shared their video. The participants took part in this study on a single monitor using a web application. They used their own machines and selected their own environments to participate in the study. The study sessions were recorded to be revisited for analysis.

6.6 Results

In the following sections, we revisit our research questions. First, we explore the efficacy of the exercises as a teaching tool in their current form. Then, we explore how well the exercises achieved the design goals and the impact of each design goal. Finally, we highlight room for improvement for the exercises. To answer our research questions, the first author used grounded theory to gather themes from participant answers to the semi-structured interview questions. We also analyzed the screen recordings of them solving the exercises, and aggregated their self-assessed learning scores.

Though we did not directly measure transfer learning in this study, participants self-reported that our new exercises were educational, particularly in understanding the "big picture" of an approach and the MVC architecture pattern. Additionally, all participants wanted our new exercises in the course, primarily to be used alongside their current code writing exercises.

We find that our new exercises achieved the desired design goals and that each design goal added value. Participants found that being given subgoals supported building their understanding of how to divide the problem into different parts. Though code writing exercises are effective at teaching Rails syntax, participants found that our exercises taught the structure of the solution because it guided them through decomposing the problem through two layers of abstraction. However, the Data Flow exercises required too much detail for a couple of the participants that wanted to shift to a bottom-up approach to make sense of the solution. Although some participants felt that being forced to a single solution is contrary to the open-ended nature of programming, they found it was valuable when working at the level of abstraction of subgoals. Finally, students were able to solve these exercises an order of magnitude faster than code writing exercises, and self-reported that these exercises would save them time if they completed them before attempting to write code.

Participants indicated that these exercises could be made more effective by revisiting the Data Blocks to make them more clear and by improving the feedback mechanism.

6.6.1 Exercise Efficacy

In this section, we explore the efficacy of these exercises in their current form.

Student Learning

Due to the design of this study, we cannot conclude with a pre-/post-test or a controlled study what students actually learned. However, there are many indications suggesting that these exercises complement learning gaps from code writing exercises, which we discuss below.

When asked about their experiences in the course at the start of the study, 7 participants commented that they struggle to keep the "big picture", or the overall task, in mind when working on code writing exercises. In contrast, only one participant felt they can keep the "big picture" in mind. Participants reflected that the current scaffolding structure in code writing assignments assignments make it easiest to work on the problem in small parts that have well-defined boundaries from the instructor directly. Similarly, 4 participants expressed that they don't think at all about the architecture of their solution while working on code writing exercises. Two participants expressed that they try to keep MVC in mind, but felt that code writing assignments make that difficult. Participant comments suggest that the current assignments are effective at teaching, but are limited at teaching problem decomposition or the boundaries of responsibility suggested by MVC architectures.

Table 6.2: **Summary results from self-rated questions after each exercise.** We report the time estimate as the average number of hours students reported. We report the Likert-scale questions on solution confidence and code completeness in two ways: the average Likert score on a 5-point scale, and the percentage of participants that rated higher than the previous exercise minus the percentage of those that rated lower.

	Unaided Decomposition	**Subgoal Decomposition**	**Data Flow**
Avg. Time Estimate (hrs)	4.10	3.14	2.68
Avg. Solution Confidence	3.29	4.08	4.31
Avg. Code Completeness	3.48	3.82	4.30
Solution Confidence (% up - % down)	–	75%	42%
Code Completeness (% up - % down)	–	67%	75%

First, we explore student perception on whether our exercises were teaching anything. After each of the three exercises (unaided decomposition, Subgoal Decomposition, and Data Flow), we asked participants 3 questions to self-assess their learning: If they were given the same problem prompt as a code writing assignment, how long would it take them; On a scale of 1 to 5, how confident were they in their understanding of how to solve the problem; On a scale of 1 to 5, how complete was their understanding of the code they would need to write. For the second and third question, participants sometimes made statements such as "still a 4, but a higher 4 than before". As such, for these questions we report both the average numeric score as well as whether their scores increased or decreased between exercises.

A summary of results can be seen in Table 6.2. Responses to all three questions suggest that students felt they were getting a better understanding of how a program could solve the problem as they worked on our exercises. Their average time estimate to write code dropped .95 hours after working on the Subgoal Decomposition exercise, and then dropped a further .47 hours after working on the Data Flow exercise. Their confidence in how to solve the problem went up for most participants after working on the Subgoal Decomposition exercise, then up further after working on the Data Flow exercise. Similarly, the completeness of their understanding of the code went up for most participants after working on the Subgoal Decomposition exercise, then up further after working on the Data Flow exercise. The Subgoal Decomposition exercise was always the first exercise and 5 participants were unable to complete the Data Flow exercise in the given time, so the relative efficacy of the two exercises is not compared by this study.

Though students self-reports suggest they were learning, it does not provide insight into *what* they were learning. After working on our exercises, we asked participants what was most and least effective about each exercise. Many participants volunteered their perception on what these exercises encourage them to focus on, and therefore what they learn. Below, we report on the counts of each topic participant mentioned.

10 participants mentioned that these exercises gave them much more clarity on the "big

picture" and "flow" of the problem. Some highlighted how much more difficult that is when working with code, as the sheer amount of code belonging to a problem makes it infeasible to keep the full context in mind. In particular, some critiqued the way textbooks organize code by file; while this helps highlight implementation details, it makes it very difficult to follow the execution flow through an application. Code writing assignments have been scaffolded to allow students to solve them incrementally, but this also means students often solve each subproblem with no idea how it will eventually create their desired final product. One participant disagreed, expressing that code writing is the best way to gain an understanding of the bigger picture because our exercises were too disconnected from actual code implementations.

9 participants mentioned that these exercises helped them improve their understanding of MVC architecture by grounding the concepts in a concrete example. Because these exercises require selecting to which component each subgoal belongs, the exercises highlight "Rails' particular conventions" [P5] and the "cononical" [P7] way of using them. Some participants mentioned using their knowledge of the framework to select the appropriate data blocks in the Data Flow exercise, for example that session variables are not accessible from the model. These exercises require students to connect their understanding of the MVC framework to specific subgoals and code snippets.

Finally, several participants highlighted the effect of these exercises on learning Rails syntax. However, they were conflicted on whether the effect was positive or negative. Some felt that the Data Flow exercise was an effective way to gain more comfort with Rails syntax and different ways to use it. Others, however, felt that code writing forces students to look up syntax and correct their own mistakes, therefore providing a better opportunity to learn syntax. Despite this ambivalence, participants in both groups agreed that these exercises were effective at reinforcing the bridge between syntax and more abstract concepts like MVC or 3rd party verification.

Student Affect Towards These Exercises

We participants how, if at all, they would want these exercises integrated into the current course. All 12 participants expressed an interest in including these exercises in the course. 10 participants wanted the exercises to be integrated as a part of the code writing exercises, most thinking they should be done as a pre-assignment. Though most participants valued both exercises, 2 mentioned wanting to only use Subgoal Decomposition exercises. 4 participants wanted these exercises to accompany existing textbook and lecture material as a light-weight opportunity to check their understanding of the material. 3 participants wanted these exercises to be used as quiz questions as a very effective way to assess their application knowledge. This suggests that students found value in these exercises compared to the current learning mechanisms they have available.

However, participants felt that these exercises should not fully replace code writing exercises. Of the 10 that wanted to include in code writing, only 2 suggested that these exercises should replace code writing. Participants felt that "coding is essential" for practice [P8, P10].

Coding build a certain “confidence” [P4] and “toughness” [P8] that is critical to training for eventual jobs. Despite the fact that these exercises do not provide traditional coding practice, participants still found them quite valuable. These exercises “get you 80% [of the way] there efficiently” [P2] and reduce the painful and time-consuming task of applying concepts for the first time in coding assignments. This suggests these exercises could be used as a type of scaffolding that could be removed to eventually get students much more comfortable with unaided code writing at the scale of these problems.

6.6.2 Revisiting the Design Goals

Provide Opportunity to Practice

As mentioned in Section 6.6.1, participants felt these exercises helped them practice their understanding of the big picture of the solution and MVC architecture. More generally, students expressed appreciation for the interactivity of these exercises by comparing them to textbooks and lectures. Textbooks and lectures can be “very descriptive” [P10] of high level ideas and concepts compared to exercises. However, only one of the 12 participants recalled that the problem they worked on in this study had already been covered in a required reading and lecture a few weeks prior. Students often don’t have “a clear idea of what goes where inside the code” [P2] as they try to apply these concepts. In fact, participants felt that textbooks and lecture are poor ways to understand code. They “don’t have the mental concentration to read through all the code line-by-line” [P1], find code snippets “hard to follow” [P3], and often skip over it and hope that they’ll “look back at [it] when [they] need it” [P4]. 11 participants appreciated the interactivity of our exercises, though one participant thought our exercises would be most effective as static content. These are some of the reasons participants in our study were excited about these exercises that allow them to practice.

Scaffold Vocabulary and Details

This design goal was motivated by an opportunity to reduce cognitive load for students. Participants confirmed that current code writing assignments can be overwhelming. Participants reported code writing assignments as demotivating, often taking much more time than is justified by how much they learn. They are overloaded by the number of files they must navigate and they often end up “doing things in files that [they’re] not supposed to” [P10]. Even when they find the correct location, they feel “messing up...is catastrophic” [P9] due to the complex connections between different components of the application.

The exercises we presented to participants reduced their perception of cognitive load. These exercises hid superfluous files that were not directly relevant to the solution, giving participants a confident sense of which what files they would need to modify [P9, P12]. Participants found the focus on subgoals and then individual code snippets to be “much easier to understand and digest” [P1]. The exercises made it easier to think through the problem itself by “dividing everything clearly into different parts” [P3]. Participants found value in the reduced cognitive load of working on these exercises, though we saw other benefits as well.

These exercises allowed students to maintain a top-down thinking approach by providing them subgoals to work with. Before working on our exercises, participants were first asked to decompose the problem themselves and explain it. Participants were able to decompose the problem verbally and through writing. The Subgoal Decomposition exercise consisted of 10 subgoals. The first author analyzed each utterance and written artifact, marked the single closest subgoal the participant could be referring to, and then analyzed which of these 10 subgoals were referenced by each participant in their unaided decomposition. 3 of the 12 participants hit the limit of their ability to decompose the problem in under 10 minutes; that is, without additional scaffolding, they would have been unable to make more sense of the problem without writing code. On average, participants addressed 35% of the subgoals, with 20% of the subgoals explicitly given in the problem prompt. This suggests that students in this course need some sort of scaffolding, such as the one provided by this design goal, to be able to cleanly decompose problems.

Participants also found value in the concrete code snippets provided in the Data Flow problems. When coming across unfamiliar code, several participants referred back to documentation or Google to understand certain syntax. One particularly appreciated this, as "syntax can be annoying to piece together" [P7] and take "a lot of time...distracting from core stuff" [P11] when learning a new language or library. Even with the code snippets, these problems "skipped over superfluous information of the code" [P2] that are often found in textbook examples for completeness, while still requiring students to do more work to visualize how to actually implement the code.

Though students generally responded positively to this design goal, some raised concerns that it might have provided too much scaffolding. We further discuss the difficulty of these exercises in Section 6.6.3.

Scaffold Process

A learning goal of this course is to have students gain comfort with every part of the web stack. As such, many assignments in the course ask students to add or test some functionality, running commands and writing code across several files and different languages. In code writing assignments, students navigate freely between the high-level problem statement and implementation details. As such, the primary scaffolding instructors can provide is to give low-level instructions around specific changes and break down the problem into smaller parts, depriving students the opportunity to practice understanding and decomposing the high-level problem statement. We designed this sequence of exercises to scaffold the process of problem decomposition for students, first breaking the problem into distinct subgoals and then using code snippets to define what happens within and between each subgoal.

These exercises "allowed" [P2] and "forced" [P3] students to "sort out" [P8] the whole problem first, instead of focusing narrowly on low-level components to fill in. "The granularity [students are] thinking of [got] more specific with each iteration" [P4]. By not providing access to a code editor, these exercises force students to decompose the problem. Participants mentioned that they usually don't "think too much before typing anything" [P6] and use a "code first ask questions later" [P4] approach. In the unaided decomposition exercises, we found that 6 of the 12 participants would have started writing code in under 10 minutes of reading the problem, despite still not having an understanding of all the subgoals that would be necessary to solve it.

Most participants found value in this forced decomposition. Instead of jumping rapidly between writing code and thinking back to the big picture, these exercises support first getting "an overall view of what [students] are learning right now" [P3]. P5 mentioned getting stuck "sitting there for hours on end trying to figure out what's happening"when coding, but with these exercises , "if [they were] lost, [they could] find out why". Participants also highlighted the benefits of being able to focus on the whole problem because this exercise allowed them to keep it all in their head. P7 would use these exercises to "checkpoint" their thinking as they worked on the problem. P11 would use the exercises to "remember subtleties of some steps that [they] didn't recognize before."

Despite generally positive feedback, three participants raised concerns about the amount of process that was scaffolded; specifically, they were frustrated by the level of detail expected by the Data Flow exercise. P4 and P6 found it "disorienting" to think through that level of implementation without being able to use code to make better sense of the subgoals in a bottom-up manner. Another participant, P9, was concerned the Data Flow exercises gave away too much code of the final solution, though most participants felt there was still significant work to be done to implement the solution even after solving the Data Flow exercise.

Though the Data Flow exercise may have gone into too much detail, participants felt this sequencing of exercises helped them better understand how different components work together to obtain a goal. Instead of "just learning a little bit of Ruby" [P5] through the "fill in this thing" [P4] approach of current code writing assignments, these exercises help "actually understand the MVC framework" [P5] and how it applies to the problem. The exercises "help reinforce relevant ideas" [P3] to "get the structure" [P11] of a solution. Both exercises are critical to these insights; the Subgoal Decomposition exercise gives a "high-level view of what's first, what's after, and what needs to be done" [P4], while the Data Flow exercises cements "the actual things that will make it work" [P4] without giving away the solution.

Guide Students to One Solution

By design, these exercises guide students to specific solutions. Participants found this helped them "construct knowledge of the language" [P3] and learn "Rails conventions" [P5]. These exercises pushed some participants to come up with better organized solutions than they would otherwise, giving them examples of "a canonical way of doing things" [P7]. For some participants, these solutions forced participants into a "different way of thinking about how to solve the problem" [P9], exposing participants to new solution strategies. At the syntax level, several participants looked up syntax that was effective in this problem but unfamiliar despite being covered earlier in the course.

Two participants expressed concerns about being guided to a specific solution in general, since there are many ways to write good code [P5, P10]. But both participants expressed that, in this particular scenario, there is value. There are particularly good ways to decompose problems even with eventual differences in implementation details [P5], and languages like Rails are very convention-driven so it is helpful to practice those conventions [P10]. Particularly when staying at a level of abstraction above complete code solutions, these exercises allow students flexibility in their implementation while still providing a "good understanding of what goals should be met" [P9].

Present Examples Efficiently

To understand if these exercises were efficient compared to code writing, we analyze two measures from our study. First, we explore how long it took participants to work through these exercises. Participants reported spending an average of 6.6 hours on current code writing assignments in the course. All participants completed the Subgoal Decomposition exercise in under 18 minutes. 7 participants completed both the Subgoal Decomposition and Data Flow exercise in under 25 minutes, reporting spending an average of 4.6 hours on current code writing assignments. These two exercises are much faster than code writing exercises.

However, these exercises should not fully replace code writing exercises. Instead, we can explore whether these exercises make the eventual code writing task more efficient. After each of the three exercises, participants expressed how long they thought it would take them to write code to complete the task. The average time estimate began at 4.1 hours (after they decomposed the problem themselves unaided), then dropped to 3.15 hours after the Subgoal Decomposition exercise, then dropped further to 2.68 hours after the Data Flow exercise. Participants reported that spending at most 25 minutes on our exercises would save them 85 minutes in writing a full solution.

While our participants didn't want to spend more time on homework, they expressed a desire to work through more examples, suggesting that this efficiency is important. Students compare examples to help make sense of "why code belongs in one file vs. another" [P1]. Compared to textbook and lecture examples, these exercises clearly "show the actual flow" of the solution in a "realistic sense" [P5]. Examples allow students to practice what they're learning "in a more real-world setting" [P6], to go from concepts to grounded instances. Students need examples that they can understand (e.g., by working through themselves) and are at the right level of detail for patterns to emerge.

6.6.3 Exercise Improvements

In this section, we explore the themes that emerged from student feedback to highlight opportunities to improve these exercises.

Understanding Data Blocks

In the Data Flow exercises, participants had little trouble correctly placing connectors between subgoals. They thought the connectors were a "good way to visualize the flow of a program" [P10], particularly when compared to "jumping in and out of folders". The connectors were "crucial to understanding what needs to be accomplished" [P4] and helped participants "understand what in a higher level needs to happen and what actually happens inside of each subgoal" [P5].

However, correctly placing the data blocks presented more of a challenge. 4 participants noted that they were confused that some subgoals didn't require any data blocks. 3 participants were confused by the interaction of necessary and produced data in nested subgoals with those of their parent subgoal. P9 highlighted other confusions that might arise from students interpreting the data blocks. First, P9 internalized the subgoals as a stack of function calls, where each eventually returns data to the subgoal above it instead of passing it through to the subgoal below. Second, P9 misinterpreted the semantic reason behind why certain data blocks were required for certain subgoals. For example, the system needs access to the user session data to determine whether to show a sign in or sign out button, but P9 thought this data was necessary so that the system could continually pass the variable down to eventually update the user session data after a successful sign in or sign out. While this was caused by a misconception around side effects, it highlights that data blocks in their current form could be abstracting away too much context.

Despite these critiques, students still found data blocks to be a crucial part of the exercise. Data blocks "make it concrete" [P4] how different types of variables (e.g., instance, session) associate with different parts of the MVC framework [P10], help understand "what actually happens inside of each subgoal" [P5], and provide much-needed assistance to students in learning conventional Rails syntax [P7, P10]. Future work should explore how to improve the way data blocks are presented to students.

Difficulty of Problems

Two key factors contribute to the difficulty of Subgoal Decomposition and Data Flow exercises: the content that students have to work with (e.g., subgoals, code snippets), and the feedback they receive when checking their answers.

Participants expressed that the content made these exercises feel manageable, even for uncomfortably new concepts. In these exercises, "there is a clear answer and it's finite" [P5], so you don't end up "sitting there for hours on end trying to figure out what's happening" [P5] when stuck. Unlike code writing assignments, where at any point a student may need to start over from scratch and try a new approach, these exercises provide a finite space for students to explore. Some participants were able to leverage their understanding of the MVC frameworks or particular syntax to narrow down their options at each step [P1, P2, P4, P10, P11]. These exercises reinforce the connection between these foundational ideas. However, this also suggests that these exercises could lose some efficacy if students have such a strong foundation they can ignore the problem domain of the exercise, particularly for Data Flow problems. This is a similar challenge to students being able to nearly solve some Parsons Problems simply by using their understanding of data flow dependencies and control flow patterns.

To further explore whether these exercises provided the right level of content to students, we analyzed the mistakes students made when solving the exercises. The first author watched the recorded sessions and tallied every mistake made by each participant.

For the Subgoal Decomposition exercise, the top three mistakes mostly matched the least familiar concepts to students at that point in the course. 11 participants selected the incorrect location for where they would modify code to integrate OmniAuth middleware; students had not practiced with middleware before, and the code to set up middleware is written outside the traditional View-Route-Controller flow. 10 participants failed to correctly assign the location for creating new routes, placing it as part of either the view or controller; students had learned about routes, but had not created any new routes themselves as part of any assignment. 5 participants switched the order of updating the Gem dependency list and actually installing the new packages; students had learned about Gemfiles, but had not practiced this two-step process themselves. Interestingly, there were few mistakes on the arrangement of the asynchronous OmniAuth subgoals, even though this was a fairly unpracticed concept for students.

For the Data Flow exercise, all participants placed the connectors correctly. However, we do not report on the mistakes from the Data Flow exercises. Only 7 of the 12 participants finished the Data Flow exercise, so analysis of all participant mistakes would bias towards earlier in the exercise and analysis of only those 7 participants would bias towards participants that made fewer mistakes. Additionally, the feedback mechanism of this implementation led some participants to solve this in an unnaturally top-to-bottom manner. For example, of the 7 participants that completed the exercise, 5 skipped past the two Omniauth subgoals because, when checking their answer up to that point, the system highlighted that the next missing data block was below these subgoals.

Some participants felt the feedback provided when checking their answers was too granular, making thoughtless "guess-and-check" strategies too accessible for practical classroom use. In this experiment, at any point participants could request feedback that highlighted the first mistake in the current solution. P11 appreciated the level of feedback and only used it when they truly felt stuck, but many participants used feedback frequently to guide their problem solving process. In a classroom setting, students often "just want to get the work done as quickly as possible" [P10], even if they won't learn as much from the exercises.

Despite raising these concerns, only 2 participants had suggestions for how to improve the feedback mechanism. P7 suggested additional feedback, highlighting all mistakes instead of just the first one, so that students could focus on correcting their mistakes in any order. P9, on the other hand, suggested making the feedback one level less granular, as this could make it infeasible to brute-force an answer with frequent checks and perhaps guide students towards trying to understand why it was incorrect. In this course, code writing exercises give test-based feedback on the functionality of components as a whole. Future work could explore what an analogue might be for these exercises, where there is not code to run.

A separate approach to continue tuning the difficulty of these problems could be in the type of feedback provided. The current exercises highlights mistakes in the current submission. However, since these exercises consist of only pre-defined content, it would be possible for instructors to statically link certain types of mistakes to course material or documentation. For example, subgoals about installing a library could link to Rails documentation on the library management tool. This static linking of resources could help guide students to the concepts that instructors want to reinforce with these exercises.

6.7 Limitations and Future Work

This work suggests that these exercises could be effective in providing deliberate practice in problem decomposition and patterns for upper-division students. However, there are a few limitations to this work. First, participants could only compare our exercises to the scaffolded code writing exercises they used in the course, so we do not know how our exercises compare to other code writing exercises (e.g., month-long projects) or other exercises (e.g., Faded Parsons Problems). Second, we did not measure learning in a controlled way with pre- and post-tests. Third, we ran our study in a lab, not a classroom setting. Fourth, students all came from a single class and worked through a single problem. Though the problem prompt used in this study did come from an instructor's existing draft, we did not explore if students would have similar sentiments about these exercises for different problems in the course. Despite these limitations, the results of this chapter suggest that the design goals and exercises presented are worth further investigation to understand how well they generalize.

This work posited a set of design goals for exercises to teach a range of programming patterns and reports on student responses to the implementation of each design goal on two specific exercises. Future work could attempt to modify the exercises to better isolate the impact of each individual design goal. Additionally, the positive results suggest value in exploring similar exercises in other domains. For example, in Machine Learning, students might be learning to use PyTorch [2]. In the Subgoal Decomposition exercise, instead of focusing on the MVC framework for the location of subgoals, the exercise could require students to specify whether subgoals affect graph initialization or are used in graph execution, or which subgoals affect state in the model. In the Data Flow exercise, connectors might no longer be relevant. Or an entirely new exercise could be built based on the 5 design goals above, providing more insights into a generalized process for creating new exercises that address these design goals.

[2]https://pytorch.org/

6.8 Conclusion

We present and evaluate two new exercises to teach problem decomposition and architectural patterns to upper-division students. These exercises focus on topics that are difficult and inefficient to learn with traditional code writing exercises. These exercises also probe our set of design goals for teaching patterns more generally: reconstructing many intentionally-designed solutions. These design goals allow us to create exercises that provide deliberate practice to students for mastering higher-level patterns and processes.

Chapter 7

Conclusion

This work presents exercises designed to help Computer Science students learn foundational patterns. This work further motivates the opportunities to build new exercises to be used in our CS classrooms and reduce the reliance on Code Writing exercises as *the primary* opportunity for practice. Though testing the output of programs is an incredibly convenient way to test correctness, we sometimes overemphasize the relevance of the final program to a course's learning goals. This work attempts to address this dissonance by motivating exercises that require students to reconstruct many intentionally-designed solutions. These exercises scaffold away parts of programming, such as recalling syntax, to support students in focusing on the bigger picture of how a solution works. These exercises guide students to a single solution out of the infinite possibilities to help students not just achieve a target program functionality, but also understand specific ways to approach problems. These exercises give students guardrails, reducing the time they may spend confused and frustrated exploring a path to a solution that will not pan out.

More concretely, in this work I present three new exercises based on this motivation. I find support that Parsons Problems alone do not transfer learning to Code Writing for sufficiently advanced students (Chapters 3, 4). I motivate the design of Faded Parsons Problems as a variation of Parsons Problems which can be created from instructor solutions without also creating new distractor lines of code (Chapter 4). I evaluate Faded Parsons Problems in a large classroom study, finding them to be a very effective complement to Code Tracing and Code Writing exercises in a CS1 classroom (Chapter 5). I introduce two new exercises focused on supporting upper-division students learning higher-abstraction patterns, (Chapter 6). I confirm that the gap between reading and writing code extends to upper-division courses and that these new exercises begin to address that gap (Chapter 6). I introduce a set of design goals to motivate the success of these two new exercises and being to probe how students engage with each of these design goals (Chapters 2, 6).

This work leaves many open questions that I believe are fruitful for future work. I have only tested Subgoal Decomposition and Data Flow exercises in an in-lab setting with 12 students. While the students responded positively towards these exercises, they should be further improved before actually being integrated into a course. In addition to improving the exercises themselves, future research could more concretely understand what learning improvements come from having students work with these exercises in a classroom setting.

This work explored creating Faded Parsons Problems to help CS1 students acquire new patterns. However, others have researched Parsons Problems to more generally scaffold students between Code Tracing and Code Writing exercises (Ericson, Margulieux, et al., 2017; Zavala et al., 2017). Future research could explore using Faded Parsons Problems outside the context of teaching patterns. Faded Parsons Problems may be effective a step to further scaffold students between Parsons Problems and Code Writing exercises for general programming skills. Or they might be effective at teaching other foundational programming skills or ideas.

This work introduced a set of design goals to create exercises that help students acquire new patterns. Faded Parsons Problems are effective in a CS1 context, and Subgoal Decomposition and Data Flow exercises show promise for web architecture. Future work could use these design goals and examples to create new exercises for other domains (e.g., Machine Learning, System Architecture, etc.). Future work could also further substantiate how relevant each of these design goals are to teaching patterns.

This work presented design goals to create exercises that could be individually added to existing curricula. Future work could tie expand this scope to cover a multi-step instructional process for teaching patterns. Such a process could be more strongly tied to particular pedagogical frameworks, such as constructivism (Fosnot, 2013) or the Knowledge Integration framework (Linn, 2005), motivating the exploration of a particular sequencing of different exercises.

Finally, this work is motivated by patterns being a key building block to becoming an expert programmer that is also difficult to learn with the current mainstream exercises. But, programming consists of many different skills and ideas. Future work should continue to explore other skills or ideas we should focus on deliberately practicing and how to design new exercises to support that deliberate practice. For example, from my interviews with professors, there is a clear opportunity for more deliberate practice with debugging.

I believe that we can build new exercises to better support students learning to become expert programmers. One way to do that is to provide more opportunities for deliberate practice of the overwhelming skills and ideas that must come together to write a program. The research presented in this work offers one viewpoint on a way to make progress towards that goal, and I hope it guides and inspires others to keep building better exercises.

Bibliography

[1] C. Alphonce and B. Martin. "Green: A Customizable UML Class Diagram Plug-in for Eclipse". In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 108–109. ISBN: 1595931937. DOI: 10.1145/1094855.1094887. URL: https://doi.org/10.1145/1094855.1094887.

[2] O. Astrachan, G. Mitchener, G. Berry, and L. Cox. "Design patterns: an essential component of CS curricula". In: *Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education - SIGCSE '98*. New York, NY, USA: ACM Press, 1998, pp. 153–160. DOI: 10.1145/273133.273182. URL: https://doi.org/10.1145/273133.273182.

[3] R. S. Baker, A. T. Corbett, K. R. Koedinger, and A. Z. Wagner. "Off-task behavior in the cognitive tutor classroom". In: *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*. ACM Press, 2004. DOI: 10.1145/985692.985741. URL: https://doi.org/10.1145/985692.985741.

[4] D. Barr, J. Harrison, and L. Conery. "Computational Thinking: A Digital Age Skill for Everyone." In: *Learning and leading with technology* 38 (2011), pp. 20–23.

[5] S. Basu, A. Wu, B. Hou, and J. DeNero. "Problems before solutions: Automated problem clarification at scale". In: *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. 2015, pp. 205–213.

[6] L. E. Berk and A. Winsler. *Scaffolding Children's Learning: Vygotsky and Early Childhood Education (Naeyc Research Into Practice Series, Vol. 7)*. National Association for the Education of Young Children, 1995. ISBN: 0935989684.

[7] B. S. Bloom, M. D. Englehart, E. J. Furst, W. H. Hill, D. R. Krathwohl, et al. *Taxonomy of educational objectives, handbook I: the cognitive domain. New York: David McKay Co*. 1956.

[8] B. du Boulay, T. O'Shea, and J. Monk. "The black box inside the glass box: presenting computing concepts to novices". In: *International Journal of man-machine studies* 14.3 (1981), pp. 237–249.

[9] S. Bryant, P. Romero, and B. du Boulay. "Pair programming and the mysterious role of the navigator". In: *International Journal of Human-Computer Studies* 66.7 (July 2008), pp. 519–529. DOI: `10.1016/j.ijhcs.2007.03.005`. URL: `https://doi.org/10.1016/j.ijhcs.2007.03.005`.

[10] A. Burton-Jones and P. Meso. "The effects of decomposition quality and multiple forms of information on novices' understanding of a domain from a conceptual model". In: *Journal of the Association for Information Systems* 9.12 (2008), p. 1.

[11] F. E. V. Castro and K. Fisler. "On the Interplay Between Bottom-Up and Datatype-Driven Program Design". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: Association for Computing Machinery, 2016, pp. 205–210. ISBN: 9781450336857. DOI: `10.1145/2839509.2844574`. URL: `https://doi.org/10.1145/2839509.2844574`.

[12] W. G. Chase and H. A. Simon. "Perception in chess". In: *Cognitive Psychology* 4.1 (Jan. 1973), pp. 55–81. DOI: `10.1016/0010-0285(73)90004-2`. URL: `https://doi.org/10.1016/0010-0285(73)90004-2`.

[13] N. Cheng and B. Harrington. "The Code Mangler: Evaluating Coding Ability Without Writing Any Code". In: *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education - SIGCSE '17*. New York, NY, USA: ACM Press, 2017, pp. 123–128. DOI: `10.1145/3017680.3017704`. URL: `https://doi.org/10.1145/3017680.3017704`.

[14] R. R. Choudhury, H. Yin, J. Moghadam, and A. Fox. "AutoStyle: Toward Coding Style Feedback At Scale". In: *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion - CSCW '16 Companion*. ACM Press, 2016. DOI: `10.1145/2818052.2874315`. URL: `https://doi.org/10.1145/2818052.2874315`.

[15] M. J. Clancy and M. C. Linn. "Patterns and pedagogy". In: *ACM SIGCSE Bulletin* 31.1 (Mar. 1999), pp. 37–42. DOI: `10.1145/384266.299673`. URL: `https://doi.org/10.1145/384266.299673`.

[16] K. Cunningham, B. J. Ericson, R. Agrawal Bejarano, and M. Guzdial. "Avoiding the Turing Tarpit: Learning Conversational Programming by Starting from Code's Purpose". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: `10.1145/3411764.3445571`. URL: `https://doi.org/10.1145/3411764.3445571`.

[17] Q. Cutts, S. Esper, M. Fecho, S. R. Foster, and B. Simon. "The abstraction transition taxonomy". In: *Proceedings of the ninth annual international conference on International computing education research - ICER '12*. ACM Press, 2012. DOI: `10.1145/2361276.2361290`. URL: `https://doi.org/10.1145/2361276.2361290`.

[18] S. P. Davies. “Characterizing the program design activity: Neither strictly top-down nor globally opportunistic”. In: *Behaviour & Information Technology* 10.3 (1991), pp. 173–190.

[19] P. Denny, A. Luxton-Reilly, and B. Simon. “Evaluating a new exam question”. In: *Proceeding of the fourth international workshop on Computing education research - ICER ’08*. ACM Press, 2008. DOI: `10.1145/1404520.1404532`. URL: `https://doi.org/10.1145/1404520.1404532`.

[20] B. du Boulay. “Some Difficulties of Learning to Program”. In: *Journal of Educational Computing Research* 2.1 (Feb. 1986), pp. 57–73. DOI: `10.2190/3lfx-9rrf-67t8-uvk9`. URL: `https://doi.org/10.2190/3lfx-9rrf-67t8-uvk9`.

[21] B. J. Ericson, J. D. Foley, and J. Rick. “Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER ’18*. ACM Press, 2018. DOI: `10.1145/3230977.3231000`. URL: `https://doi.org/10.1145/3230977.3231000`.

[22] B. J. Ericson, M. J. Guzdial, and B. B. Morrison. “Analysis of Interactive Features Designed to Enhance Learning in an Ebook”. In: *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER ’15*. ACM Press, 2015. DOI: `10.1145/2787622.2787731`. URL: `https://doi.org/10.1145/2787622.2787731`.

[23] B. J. Ericson, L. E. Margulieux, and J. Rick. “Solving parsons problems versus fixing and writing code”. In: *Proceedings of the 17th Koli Calling Conference on Computing Education Research - Koli Calling ’17*. ACM Press, 2017. DOI: `10.1145/3141880.3141895`. URL: `https://doi.org/10.1145/3141880.3141895`.

[24] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. “The role of deliberate practice in the acquisition of expert performance.” In: *Psychological review* 100.3 (1993), p. 363.

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

[26] J. L. Fleiss and J. Cohen. “The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability”. In: *Educational and psychological measurement* 33.3 (1973), pp. 613–619.

[27] C. T. Fosnot. *Constructivism: Theory, perspectives, and practice*. Teachers College Press, 2013.

[28] M. Friedman. “The use of ranks to avoid the assumption of normality implicit in the analysis of variance”. In: *Journal of the american statistical association* 32.200 (1937), pp. 675–701.

[29] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[30] S. Garner. "An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming". In: *Issues in Informing Science and Information Technology* 4 (2007), pp. 491–501. DOI: `10.28945/966`. URL: `https://doi.org/10.28945/966`.

[31] J. M. Griffin. "Learning by Taking Apart: Deconstructing Code by Reading, Tracing, and Debugging". In: *Proceedings of the 17th Annual Conference on Information Technology Education*. SIGITE '16. Boston, Massachusetts, USA: Association for Computing Machinery, 2016, pp. 148–153. ISBN: 9781450344524. DOI: `10.1145/2978192.2978231`. URL: `https://doi.org/10.1145/2978192.2978231`.

[32] P. J. Guo. "Online python tutor: embeddable web-based program visualization for cs education". In: *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*. New York, NY, USA: ACM Press, 2013, pp. 579–584. DOI: `10.1145/2445196.2445368`. URL: `https://doi.org/10.1145/2445196.2445368`.

[33] K. J. Harms, N. Rowlett, and C. Kelleher. "Enabling independent learning of programming concepts through programming completion puzzles". In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Oct. 2015. DOI: `10.1109/vlhcc.2015.7357226`. URL: `https://doi.org/10.1109/vlhcc.2015.7357226`.

[34] K. J. Harms, J. Chen, and C. L. Kelleher. "Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers". In: *Proceedings of the 2016 ACM Conference on International Computing Education Research - ICER '16*. ACM Press, 2016. DOI: `10.1145/2960310.2960314`. URL: `https://doi.org/10.1145/2960310.2960314`.

[35] S. G. Hart and L. E. Staveland. "Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research". In: *Advances in Psychology*. Elsevier, 1988, pp. 139–183. DOI: `10.1016/s0166-4115(08)62386-9`. URL: `https://doi.org/10.1016/s0166-4115(08)62386-9`.

[36] C. C. Haynes and B. J. Ericson. "Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: `10.1145/3411764.3445292`. URL: `https://doi.org/10.1145/3411764.3445292`.

[37] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi. "How do students solve parsons programming problems?" In: *Proceedings of the ninth annual international conference on International computing education research - ICER '12*. ACM Press, 2012. DOI: `10.1145/2361276.2361300`. URL: `https://doi.org/10.1145/2361276.2361300`.

[38] P. Ihantola, J. Helminen, and V. Karavirta. "How to study programming on mobile touch devices". In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research - Koli Calling '13*. ACM Press, 2013. DOI: 10.1145/2526968.2526974. URL: https://doi.org/10.1145/2526968.2526974.

[39] P. Ihantola and V. Karavirta. "Open source widget for parson's puzzles". In: *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10*. ACM Press, 2010. DOI: 10.1145/1822090.1822178. URL: https://doi.org/10.1145/1822090.1822178.

[40] T. Jenkins. "On the difficulty of learning to program". In: *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*. Vol. 4. Citeseer. Loughborough, Leicestershire, UK: Loughborough University, 2002, pp. 53–58.

[41] A. Ju and A. Fox. "TEAMSCOPE: Measuring Software Engineering Processes with Teamwork Telemetry". In: *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE 2018. Larnaca, Cyprus: Association for Computing Machinery, 2018, pp. 123–128. ISBN: 9781450357074. DOI: 10.1145/3197091.3197107. URL: https://doi.org/10.1145/3197091.3197107.

[42] A. Ko and B. Myers. "Debugging reinvented". In: *2008 ACM/IEEE 30th International Conference on Software Engineering*. 2008, pp. 301–310. DOI: 10.1145/1368088.1368130.

[43] K. Kwon. "Novice programmer's misconception of programming reflected on problem-solving plans". In: *International Journal of Computer Science Education in Schools* 1.4 (2017), pp. 14–24.

[44] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. "A Study of the Difficulties of Novice Programmers". In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05. Caparica, Portugal: Association for Computing Machinery, 2005, pp. 14–18. ISBN: 1595930248. DOI: 10.1145/1067445.1067453. URL: https://doi.org/10.1145/1067445.1067453.

[45] M. C. Linn. "The Knowledge Integration Perspective on Learning and Instruction". In: *The Cambridge Handbook of the Learning Sciences*. New York, NY, USA: Cambridge University Press, Apr. 2005, pp. 243–264. DOI: 10.1017/cbo9780511816833.016. URL: https://doi.org/10.1017/cbo9780511816833.016.

[46] M. C. Linn and M. J. Clancy. "The Case for Case Studies of Programming Problems". In: *Commun. ACM* 35.3 (Mar. 1992), pp. 121–132. ISSN: 0001-0782. DOI: 10.1145/131295.131301. URL: https://doi.org/10.1145/131295.131301.

[47] 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.* New York, NY, USA: ACM, May 2016, pp. 1449–1461. DOI: 10.1145/2858036.2858252. URL: https://doi.org/10.1145/2858036.2858252.

[48] M. Lopez, J. Whalley, P. Robbins, and R. Lister. "Relationships between reading, tracing and writing skills in introductory programming". In: *Proceeding of the fourth international workshop on Computing education research - ICER '08.* ACM Press, 2008. DOI: 10.1145/1404520.1404531. URL: https://doi.org/10.1145/1404520.1404531.

[49] S. P. Marshall. *Schemas in Problem Solving.* New York, NY, USA: Cambridge University Press, June 1995. DOI: 10.1017/cbo9780511527890. URL: https://doi.org/10.1017/cbo9780511527890.

[50] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander. "Debugging: a review of the literature from an educational perspective". In: *Computer Science Education* 18.2 (2008), pp. 67–92.

[51] T. J. McGill and S. E. Volet. "A Conceptual Framework for Analyzing Students' Knowledge of Programming". In: *Journal of Research on Computing in Education* 29.3 (Mar. 1997), pp. 276–297. DOI: 10.1080/08886504.1997.10782199. URL: https://doi.org/10.1080/08886504.1997.10782199.

[52] O. Meerbaum-Salant, M. Armoni, and M. (Ben-Ari. "Learning computer science concepts with Scratch". In: *Computer Science Education* 23.3 (Sept. 2013), pp. 239–264. DOI: 10.1080/08993408.2013.832022. URL: https://doi.org/10.1080/08993408.2013.832022.

[53] J. Michael. "Where's the evidence that active learning works?" In: *Advances in Physiology Education* 30.4 (Dec. 2006), pp. 159–167. DOI: 10.1152/advan.00053.2006. URL: https://doi.org/10.1152/advan.00053.2006.

[54] A. Milliken, W. Wang, V. Cateté, S. Martin, N. Gomes, Y. Dong, R. Harred, A. Isvik, T. Barnes, T. Price, and C. Martens. "PlanIT! A New Integrated Tool to Help Novices Design for Open-Ended Projects". In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education.* SIGCSE '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 232–238. ISBN: 9781450380621. DOI: 10.1145/3408877.3432552. URL: https://doi.org/10.1145/3408877.3432552.

[55] S. H. Moritz, F. Wei, S. M. Parvez, and G. D. Blank. "From Objects-First to Design-First with Multimedia and Intelligent Tutoring". In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education.* ITiCSE '05. Caparica, Portugal: Association for Computing Machinery, 2005, pp. 99–103. ISBN: 1595930248. DOI: 10.1145/1067445.1067475. URL: https://doi.org/10.1145/1067445.1067475.

[56] O. Muller, D. Ginat, and B. Haberman. "Pattern-oriented instruction and its influence on problem decomposition and solution construction". In: *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education - ITiCSE '07*. New York, NY, USA: ACM Press, 2007, pp. 151–155. DOI: `10.1145/1268784.1268830`. URL: `https://doi.org/10.1145/1268784.1268830`.

[57] D. H. O'Dell. "The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills." In: *Queue* 15.1 (2017), pp. 71–90.

[58] D. Parsons and P. Haden. "Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses". In: *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*. ACE '06. Hobart, Australia: Australian Computer Society, Inc., 2006, pp. 157–163. ISBN: 1920682341.

[59] D. Perkins and G. Salomon. "Transfer Of Learning". In: *International encyclopedia of education* 2 (1992), pp. 6452–6457.

[60] J. G. Politz, D. Patterson, S. Krishnamurthi, and K. Fisler. "CaptainTeach: Multi-Stage, in-Flow Peer Review for Programming Assignments". In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. Uppsala, Sweden: Association for Computing Machinery, 2014, pp. 267–272. ISBN: 9781450328333. DOI: `10.1145/2591708.2591738`. URL: `https://doi.org/10.1145/2591708.2591738`.

[61] A. Renkl, R. K. Atkinson, U. H. Maier, and R. Staley. "From example study to problem solving: Smooth transitions help learning". In: *The Journal of Experimental Education* 70.4 (2002), pp. 293–315.

[62] R. S. Rist. "Knowledge creation and retrieval in program design: A comparison of novice and intermediate student programmers". In: *Human-Computer Interaction* 6.1 (1991), pp. 1–46.

[63] M. Rizvi, T. Humphries, D. Major, M. Jones, and H. Lauzun. "A CS0 course using Scratch". In: *Journal of Computing Sciences in Colleges* 26.3 (2011), pp. 19–27.

[64] A. Robins, J. Rountree, and N. Rountree. "Learning and Teaching Programming: A Review and Discussion". In: *Computer Science Education* 13.2 (June 2003), pp. 137–172. DOI: `10.1076/csed.13.2.137.14200`. URL: `https://doi.org/10.1076/csed.13.2.137.14200`.

[65] J. Sajaniemi and M. Kuittinen. "An Experiment on Using Roles of Variables in Teaching Introductory Programming". In: *Computer Science Education* 15.1 (Mar. 2005), pp. 59–82. DOI: `10.1080/08993400500056563`. URL: `https://doi.org/10.1080/08993400500056563`.

[66] C. Scaffidi, M. Shaw, and B. Myers. "Estimating the Numbers of End Users and End User Programmers". In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. Hoboken, NJ, USA: IEEE, 2005, pp. 207–214. DOI: `10.1109/vlhcc.2005.34`. URL: `https://doi.org/10.1109/vlhcc.2005.34`.

[67] D. H. Schunk. "Social Cognitive Theory and Self-Regulated Learning". In: *Self-Regulated Learning and Academic Achievement*. Springer New York, 1989, pp. 83–110. DOI: 10.1007/978-1-4612-3618-4_4. URL: https://doi.org/10.1007/978-1-4612-3618-4_4.

[68] D. T. Seaton, Y. Bergner, I. Chuang, P. Mitros, and D. E. Pritchard. "Who does what in a massive open online course?" In: *Communications of the ACM* 57.4 (Apr. 2014), pp. 58–65. DOI: 10.1145/2500876. URL: https://doi.org/10.1145/2500876.

[69] J. Shi, A. Shah, G. Hedman, and E. O'Rourke. "Pyrus: Designing A Collaborative Programming Game to Promote Problem Solving Behaviors". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19*. New York, NY, USA: ACM Press, 2019, pp. 1–12. DOI: 10.1145/3290605.3300886. URL: https://doi.org/10.1145/3290605.3300886.

[70] E. Soloway. "Learning to program= learning to construct mechanisms and explanations". In: *Communications of the ACM* 29.9 (1986), pp. 850–858.

[71] T. Song and K. Becker. "Expert vs. novice: Problem decomposition/recomposition in engineering design". In: *2014 International Conference on Interactive Collaborative Learning (ICL)*. 2014, pp. 181–190. DOI: 10.1109/ICL.2014.7017768.

[72] J. C. Spohrer and E. Soloway. "Novice mistakes: are the folk wisdoms correct?" In: *Communications of the ACM* 29.7 (July 1986), pp. 624–632. DOI: 10.1145/6138.6145. URL: https://doi.org/10.1145/6138.6145.

[73] S. Suzuki. *Zen Mind, Beginner's Mind: Informal Talks on Zen Meditation and Practice*. Weatherhill, 1970. ISBN: 0834800799. URL: https://www.amazon.com/Zen-Mind-Beginners-Informal-Meditation/dp/0834800799.

[74] J. Sweller. "Cognitive Load During Problem Solving: Effects on Learning". In: *Cognitive Science* 12.2 (Apr. 1988), pp. 257–285. DOI: 10.1207/s15516709cog1202_4. URL: https://doi.org/10.1207/s15516709cog1202_4.

[75] *U.S. Department of Labor, Occupational Outlook Handbook, Software Developers*. (retrieved on April 22, 2020). Bureau of Labor Statistics. 2019. URL: https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm.

[76] S. Valstar, W. G. Griswold, and L. Porter. "The Relationship between Prerequisite Proficiency and Student Performance in an Upper-Division Computing Course". In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE '19. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 794–800. ISBN: 9781450358903. DOI: 10.1145/3287324.3287419. URL: https://doi.org/10.1145/3287324.3287419.

[77] N. Weinman, A. Fox, and M. Hearst. "Exploring Challenging Variations of Parsons Problems". In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. New York, NY, USA: ACM, Feb. 2020, p. 1349. DOI: 10.1145/3328778.3372639. URL: https://doi.org/10.1145/3328778.3372639.

[78] N. Weinman, A. Fox, and M. A. Hearst. “Improving Instruction of Programming Patterns with Faded Parsons Problems”. In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 2021, pp. 1–4.

[79] N. Weinman, B. Hsu, and A. Camacho. “Implementing a More Challenging Parsons Problem Interface for Teaching Computer Science”. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1417. ISBN: 9781450367936. URL: `https://doi.org/10.1145/3328778.3372548`.

[80] D. Weintrop and U. Wilensky. “Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms”. In: *ACM Transactions on Computing Education* 18.1 (Dec. 2017), pp. 1–25. DOI: `10.1145/3089799`. URL: `https://doi.org/10.1145/3089799`.

[81] S. Wiedenbeck, V. Fix, and J. Scholtz. “Characteristics of the mental representations of novice and expert programmers: an empirical study”. In: *International Journal of Man-Machine Studies* 39.5 (Nov. 1993), pp. 793–812. DOI: `10.1006/imms.1993.1084`. URL: `https://doi.org/10.1006/imms.1993.1084`.

[82] J. Wrenn and S. Krishnamurthi. “Executable Examples for Programming Problem Comprehension”. In: *Proceedings of the 2019 ACM Conference on International Computing Education Research*. ICER ’19. Toronto ON, Canada: Association for Computing Machinery, 2019, pp. 131–139. ISBN: 9781450361859. DOI: `10.1145/3291279.3339416`. URL: `https://doi.org/10.1145/3291279.3339416`.

[83] B. Xie, D. Loksa, G. L. Nelson, M. J. Davidson, D. Dong, H. Kwik, A. H. Tan, L. Hwa, M. Li, and A. J. Ko. “A theory of instruction for introductory programming skills”. In: *Computer Science Education* 29.2-3 (Jan. 2019), pp. 205–253. DOI: `10.1080/08993408.2019.1565235`. URL: `https://doi.org/10.1080/08993408.2019.1565235`.

[84] L. Zavala and B. Mendoza. “Precursor Skills to Writing Code”. In: *J. Comput. Sci. Coll.* 32.3 (Jan. 2017), pp. 149–156. ISSN: 1937-4771. URL: `http://dl.acm.org/citation.cfm?id=3015220.3015257`.

[85] R. Zhi, M. Chi, T. Barnes, and T. Price. “Evaluating the Effectiveness of Parsons Problems for Block-based Programming”. In: July 2019, pp. 51–59. ISBN: 978-1-4503-6185-9. DOI: `10.1145/3291279.3339419`.