Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers

Francisco Enrique Vicente Castro
Worcester Polytechnic Institute
fgcastro@cs.wpi.edu

Kathi Fisler
Brown University
kfisler@cs.brown.edu

ABSTRACT

Cognitive theories of how programmers produce code suggest that novices' approaches are primarily driven by the retrieval of previously-learned plans. These plans can be high-level, focusing on task decomposition and composition, or low-level, focusing on code implementations. These theories, however, do not describe *how* novices move between high-level tasks and low-level code, especially when faced with novel problems. Understanding these transitions can help concretely tease out *why and where* novices struggle and *how* they use their knowledge of plans and design techniques when they get stuck.

We studied this by conducting think-alouds with CS1 students at two universities as they solved multi-task programming problems with unfamiliar components. Our analysis paid particular attention to a series of design techniques that the students had been explicitly taught in their respective courses. We identified patterns of high- and low-level thinking that relate to students' success on the problems, and propose a concrete framework of high- and low-level work that summarizes the transitions that we observed.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **CS1**; • **Human-centered computing** → *User studies.*

KEYWORDS

Program design, plan composition, rainfall, design recipe

ACM Reference Format:
Francisco Enrique Vicente Castro and Kathi Fisler. 2020. Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366847

1 INTRODUCTION

Most theories of how novice programmers design solutions for programming problems draw from plan-based models of programming. *Plans* refer to the organization of *tasks* or of *code* that relate to the tasks of a problem [3, 15]. Problem tasks refer to problem components that need to be addressed in order to complete a solution; for example, a problem that asks to compute the average of a list of numbers would have the following tasks: (1) sum the list-values, (2) count the list-values, and finally (3) divide the sum by the count. Programmers either (a) retrieve plans from memory top-down or (b) develop plans bottom-up to generate new plans [13, 14].

These theories, however, do not describe how novices move between a problem's high-level tasks and the tasks' low-level code implementations. Prior work has shown that while novices think in terms of a problem's core tasks, they struggle with decomposition and composition issues around their solutions [2]. Our work explores this interplay between task- and code-level thinking among novice programmers within the context of two CS1 courses that teach a systematic design-process called the HTDP *Design Recipe* [6] (section 3) for solving programming problems. Our courses of study provide an interesting context: when students struggle with problems, they have design techniques to fall back on. These may have an impact on how students think around the problem's tasks and their code. We thus focus on the following research questions:

RQ 1. What patterns of movements between high- and low-level thinking do we see among our students?

RQ 2. How do students' patterns of task- and code-level thinking relate to their success on our programming problems?

RQ 3. When students get stuck on a problem, what do they do and how do they use the HTDP design recipe as part of getting unstuck?

2 RELATED WORK

Pirolli et al.'s studies on novices writing recursive programs found that programmers rely heavily on prior knowledge of solutions when developing new programs [11, 12]. Spohrer and Soloway's findings from their study of students' think-aloud protocols echo Pirolli's findings, suggesting that students use both prior knowledge of programming plans, as well as relevant non-programming plans to write code [16]. Spohrer and Soloway also found that most of novice programmers' mistakes come from *plan composition*: novices struggle with composing program plans to form working solutions. Rist built on these theories by proposing a model that describes two states novice programmers enter when solving a programming problem [13, 14]. Novices enter a *plan-retrieval* state when they have a solution to a similar programming problem in memory: they retrieve the solution and reproduce the code in *top-down* fashion. When they don't have a solution to retrieve, novices enter a *plan-creation* state wherein they identify a core computation called a *focus* (typically a core computation in the problem such as summing in an averaging problem) and implement code for the focus, building on this focus *bottom-up* until a working solution is developed. None of these theories describe how novices navigate between tasks and code, especially when faced with programming problems that have

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366847

both familiar and unfamiliar components. In this context, novices would need to navigate their use of prior plans and new tasks, both of which they need to build code for. This is the dynamic we explore in this work.

More recent studies have also explored how students engaged in plan-composition when explicitly taught planning strategies, such as de Raadt et al.'s work on teaching students a "strategy guide" for integrating plans [5]. Muller et al. talks about pattern-oriented instruction in which students are taught to attach labels to programming patterns and to look for common patterns across problems [10]. Our work additionally looks at how students navigate between tasks and code in the context of having a multi-step *design recipe* for approaching problems, particularly in situations when they get stuck in their programming process, perhaps due to not having plans or patterns to retrieve for novel problem-tasks.

Historical Context of this Work. This work follows our existing line of work on understanding the programming processes of CS1 students taught with the HTDP curriculum. In an earlier exploratory study [2], we analyzed video captures of students' IDE as they worked on a multi-task problem and found that while students primarily work in terms of problem-tasks, they struggled to decompose and compose the code around these tasks. Students also used familiar code patterns verbatim without adjusting the patterns to the need of the tasks. That study, however, lacked richer data on which observations can be drawn about *how* students were thinking around the problem, *why* they struggled in their process, and *how* they tried to get unstuck. In a follow-up study [7], we collected think-aloud data as students solved the Rainfall problem [15] and found that students who connected specific parts of their solutions to specific tasks *and* maintained those connections throughout their process produced more correct code. This is a key finding in this work as well; additionally, our work considers what happens around task- and code-level thinking when students get a problem with entire tasks that they have not seen before.

3 COURSE CURRICULUM

How to Design Programs [6] is an introductory computing curriculum that teaches students a multi-step *Design Recipe* (DR) for designing programs based on the structure of the input data. Given a programming problem, students are taught to work through a progression of steps:

(1) **Data Definitions**: Identify and define the structure of the input data.
(2) **Examples of Data**: Write examples of the input data (as executable code).
(3) **Signature and Purpose Statement**: Write the name, input types, and output type (the *signature*) for a function that will solve the problem and a brief summary of the function's goal (the *purpose statement*).
(4) **Input—Output Examples**: Write concrete examples (as executable *test cases*) of what output the program should produce on specific inputs.
(5) **Function Template**: Using the *data definition* as a reference, write a skeleton of the function body (the *template*) that fully traverses the input data. The template is specific only to the *type* of the input data, not to computations within a given problem, allowing the same template to be reused across multiple functions on the same type.
(6) **Function Definition**: Fill in the template with problem-specific details.
(7) **Testing**: Run the function on the test cases, refining the function and adding tests as necessary.

```
; RECIPE STEP 1: THE DATA DEFINITION
; A list-of-string is
; - empty, or
; - (cons string list-of-string)

; RECIPE STEP 2: EXAMPLES OF DATA
(define animals (list "bunny" "dog" "kitty"))

; RECIPE STEP 4: THE TEST CASES
; check-expect captures a test case, with both
; the expression to run and its expected answer.
(check-expect (has-dog? empty) false)
(check-expect (has-dog? animals) true)

; RECIPE STEP 5: THE TEMPLATE
; The ellipses get filled with details from the
; specific problem upon reaching STEP 6.
; (define (func alos)
;   (cond [(empty? alos) ...]
;         [(cons? alos)
;          ... (first alos)
;          ... (func (rest alos)) ...]))

; RECIPE STEP 3: THE SIGNATURE AND PURPOSE
; has-dog? : list-of-string -> boolean
; Produces true if "dog" is in the list

; RECIPE STEP 6: THE FINAL FUNCTION
(define (has-dog? alos)
  (cond [(empty? alos) 0]
        [(cons? alos)
         (or (string=? (first alos) "dog")
             (has-dog? (rest alos)))]))
```

Figure 1: A design recipe walk-through on a problem to find the string "dog" in a list of strings. Racket uses semicolons for single-line comments; `cond` is a construct for multi-armed if-statements. Racket naming conventions use hyphens to separate words, rather than camel casing.

Figure 1 shows a complete walk-through of the DR steps on a sample problem; the top-to-bottom ordering of the code reflects how a typical student submission would look. We present code in Racket (a variant of Scheme), as that is the language used in the courses we studied and the HTDP textbook [6]. An HTDP course applies the recipe to increasingly rich data structures as the course progresses: from atomic data (e.g. numbers and strings), to compound data (structs), to lists of atomic data and structs, and binary and n-ary trees. Additional topics covered in HTDP courses may include higher-order functions, accumulator-patterns (building partial results in parameters), and stateful variables. The DR steps provide a form of scaffolding [1] that leads a student from a prose-based problem statement to a working program. The progression from data definitions to examples to code move the student through different representations of the problem, providing a form of concreteness fading [8] as students progress towards a symbolic-form solution.

4 STUDY DESIGN AND DATA COLLECTION

We chose two programming problems with multiple subtasks and had students from HTDP-based CS1 courses at two universities (both selective, private, and in the USA) work on each one. The first problem (Rainfall) consisted of subtasks that students had previously coded in other contexts, but not previously composed in this way. The second problem (Max-Temps) was harder, involving subtasks that students had not previously seen but that were solvable (though challenging) using the HTDP process as covered to date in each course. Problem details are provided later in this section.

The* HTDP *Course Instances[1]. The two HTDP courses in our study varied in topic orderings and concept emphasis. We collected data from COURSE1 in Spring 2018 and from COURSE2 in Fall 2018. At the time we ran our studies, each course had covered the basic design recipe, structures, lists, trees, and higher-order functions. COURSE1 spent the week prior the study covering higher-order functions (`map`, `filter`) and was covering accumulator-style programming *during* the week of the study sessions. COURSE2, in contrast, had spent at least three weeks with higher-order functions (`map`, `filter`, `fold`) before the study and was scheduled to cover accumulator-style programming after the scheduled study sessions; study sessions for COURSE1 were done about a week before final exams, sessions for COURSE2 about 2 weeks before final exams. COURSE1 ran at intense pace for 7 weeks, COURSE2 ran on a 14-week semester.

Participants. Instructors of both courses publicized the study to their students. Interested students provided information on their intended major, prior programming experience, and an estimate of their current course grade on a volunteer survey[1]. A total of 13 COURSE1 students and 84 COURSE2 students signed up for the study. From each participant pool, we selected students based on their availability to participate in study sessions (sessions were 2 hours total per student) and their self-reported course performance (A, B, C, D), resulting in 12 COURSE1 and 10 COURSE2 participants. The following table summarizes the number of students self-reporting each grade, for each course; (M/F) indicates self-reported gender.

Course	Self-estimated course grade			
	A	B	C	D
COURSE1	2(M), 3(F)	1(M), 4(F)	-	2(F)
COURSE2	3(M), 1(F)	4(M)	2(F)	-

Logistics. Each participant did two 1-hour sessions, the first on Rainfall and the second on Max-Temps. Within a session, students had 30 minutes to work on the problem in think-aloud fashion. A retrospective interview followed, during which students described their process and responded to interviewer observations from the think-aloud [17]. Students worked on their own computer (they could open notes they deemed relevant to the problem) and used the course's standard programming environment. We audio recorded all sessions and collected students' solutions and scratch work. The first author conducted all of the sessions. Neither author was on the staff for either course, or even affiliated with the COURSE2 university. Each participant received USD 20 per session.

[1]The course syllabi and survey are at: https://github.com/franciscastro/sigcse-2020

The Study Problems. The exact wordings for each problem follow, along with its most common solution approaches and our rationale for including it in the study.

Rainfall. *Design a program called* `rainfall` *that consumes a list of numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). If you cannot compute the average for whatever reason, return -1.*

Common solution structures:

(1) **Clean-first:** Produce an intermediate data structure of non-negative values truncated at the sentinel; sum and count the cleaned data, check for zero-division, and finally compute the average.
(2) **Process-multiple:** Traverse the input twice, once to sum and once to count, ignoring negatives and halting at the sentinel (or the empty-list); then check for zero-division and compute the average.
(3) **Single-traversal:** Traverse the input once, updating sum and count on each nonnegative input, then check for zero-division and compute the average upon reaching the sentinel (or the empty-list).

Rainfall provides an interesting context for our study: students have done most of the Rainfall tasks separately — summing, counting, and filtering lists (based on some criterion); they have not done list-problems that terminated at a specific value (rather than the end of the list) that may or may not appear. A main challenge in Rainfall is composition, as the problem is not itself structurally recursive (i.e. each of sum and count are straightforward applications of the HTDP template, but the rainfall function needs to decompose these computations into their own subtasks) and students have not composed the task-components together in a solution.

Max-Temps. *Imagine that we have lists containing a combination of numbers and the string "new-day". The numbers represent temperature readings as taken by a sensor or weather monitoring device. The "new-day" string is sent at the start of each new calendar day. Design a program* `max-temps` *that takes one of these lists and returns a list of numbers representing the max temperature for each day. If the input list is empty, return empty.*

Common solution structures:

(1) **Reshape-first:** Reshape the input into a list-of-lists that omit the delimiters, then recur over the outer list to compute the max of each inner-list.
(2) **Collect-first:** Collect sublist elements until the delimiter, then find the max of the collected elements before moving to the next sublist.
(3) **Process-until:** Find the max between consecutive list elements as the list is traversed. When a delimiter is found, concatenate the max onto the result of processing the rest of the input.

Max-Temps is more complex than Rainfall. Its viable solutions require tasks students have not coded—or even seen—in class. *Reshape-first* requires restructuring the input into a list-of-lists. While nested lists are new to students, the design recipe templates handle them

in similar ways to other nested data structures that students had used. *Process-until* requires keeping track of the current sublist's max either in an additional parameter (which COURSE1 students *may* have seen, given the study timing) or by modifying the head of the input mid-traversal. Finally, while functions over lists typically recur on the tail of the list, Max-Temps solutions may require recurring on a modified suffix (e.g., the one after the first sublist).

5 ANALYSIS AND DISCUSSION

Our research questions (section 1) revolve around understanding students' transitions between task- and code-level thinking as they solve problems, as well as how they navigate with their design techniques to get unstuck when struggling with the problems.

5.1 Framework: Task- and Code-level Thinking

Our work aims to capture *how* students think around problem-tasks and code-implementations, and their use of the DR within this dynamic. To do this, we randomly selected half the students from each cohort, from each self-estimated course grade: 6 COURSE1 (2 As, 2 Bs, 2 Ds) and 5 COURSE2 (2 As, 1 B, 2 Cs) students. We then coded [4, 9] their transcripts, field notes, and code by constructing qualitative narratives, similar to Whalley and Kasto's descriptive accounts of students' programming [17], tagging student comments based on the following guide questions (summarized for space):

(1) How do they think through the tasks before starting to retrieve code patterns?
(2) How do they interleave thinking about tasks and code?
(3) What tasks or code patterns did they get stuck in?
(4) Do they return to thinking about tasks once they start implementing them in code, whether or not they get stuck?
(5) How do they try to get unstuck?
(6) When do they use the *design recipe* or its components?
(7) How do they perceive the role of the *design recipe*?

What emerged from our coding was a descriptive conceptual framework of what students did pertaining to implementation- and task-level thinking. We describe each of these levels in turn.

Task-level thinking concerns the identification and description of a problem's task-components and involves the following actions:

- *Identifying and describing tasks*: Describing tasks in terms of their role in the overall problem; novices may elicit tasks from the problem-statement or relevant plans they retrieve
- *Describing relationships between tasks*: Describing how tasks relate to each other, such as how tasks' outputs relate to other tasks' inputs, or the ordering of the tasks (perhaps informed by their input-output relationships)
- *Plan-retrieval for familiar tasks*: Retrieving plans for familiar problem-tasks that novices have in memory

Implementation-level thinking is concerned with concrete code or code patterns students write to actualize the tasks and involves the following actions:

- *Implementing code*: Writing the code that novices retrieve, or modifying code to fit it within the problem context
- *Composing code*: Putting together relevant code in a way that solves or actualizes components of a solution
- *Plan-retrieval of task-relevant code*: Retrieving code patterns for familiar tasks that novices have in memory; may come as built-in operations and functions, or entire code structures such as DR templates or previously-implemented code

The following excerpt illustrates a COURSE2 student touching on both high-level tasks and the low-level code-patterns he retrieved. He describes an overall plan for Rainfall: he concretely *describes* the tasks he identified from the problem, *relates* tasks to each other by describing the ordering of the tasks and the output of some tasks, and *retrieves* code patterns for familiar tasks:

> COURSE2-STUD4: *I'm thinking [the] best way to approach [Rainfall], you take your list of numbers, you get all the numbers before minus 999, create a new list from that [then] take out all the non-negative numbers and then [do]* `foldr` *with the average.* `Foldr` *to find the sum and then divide that by the length*

We used this conceptual framework in coding and describing how the rest of the students navigated the program-design process. We used the dot-bulleted items to tag student comments as related to task- or implementation-level. We then used our guide questions to construct qualitative narratives that explain relationships between actions (e.g. how descriptions of task-relationships led to code compositions), citing code changes captured in field notes and students' own code and scratch work as supporting observations.

5.2 RQ1: Movements Between Tasks and Code

Our first *RQ* asks about patterns of student movement between high-level (HL) tasks and low-level (LL) code as they solved our programming problems. We found three main patterns of task-code transitions in our data:[2]

Cyclic. This pattern is characterized by a back-and-forth movement between task- and code-level descriptions of the components of a solution. *Cyclic* students concretely describe problem-tasks (HL) and describe code they will use to implement those tasks (LL). The composition of their code (LL) is guided by the concrete relationships they establish between tasks (HL), for example, by describing how the output of one task is used as input for another. Their descriptions of tasks are often within the context of an overall plan for a solution (e.g. *truncate at the sentinel first, then remove negatives, then compute average*); the connections they make between tasks fill the gaps between tasks, making a plan more complete.

Code-focused. These students primarily jump into writing code for the tasks they identify. They identify tasks *on-the-fly* as they program, often without concretely describing how the tasks relate to each other. They focus on retrieving and implementing code for a task at-hand, then add code for whichever tasks they shift their focus to next. Their descriptions of plans are often *fragmented*; they have a list of tasks they identified, but no concrete descriptions of the connections between those tasks.

One-way. Students who exhibit this pattern often identified a high-level plan for a solution early on in their process, then focused on implementing code without going back to their high-level plan. Their process often shows characteristics observed from *code-focused* students: they have fragmented descriptions of plans later

[2]Illustrative narratives are at: https://github.com/franciscastro/sigcse-2020

Table 1: Number of students implementing various solution approaches, grouped by Task-Code movement patterns. C/F in brackets indicate whether students' final code were [C]lose (minor errors on some tasks) or [F]ar from a correct solution (missing tasks, major implementation errors).

(a) Rainfall solution approaches: CleanF - *Clean-first*, PMult - *Process-multiple*, STrav - *Single-traversal*, NCP - *No clear plan.*

Pattern	Approach	# Students	
		Course1	Course2
Cyclic	CleanF	4 [C]	5 [C]
	PMult	2 [C]	1 [C]
	STrav	1 [C]	-
Code-focused	CleanF	2 [F]	-
	STrav	1 [F]	3 [F]
	NCP	-	1 [F]
One-way	STrav	2 [F]	-

(b) Max-Temps solution approaches: ReshapeF - *Reshape-first*, CollectF - *Collect-first*, ProcessU - *Process-until*, NCP - *No clear plan.*

Pattern	Approach	# Students	
		Course1	Course2
Cyclic	ReshapeF	-	1[C], 1 [F]
	CollectF	3 [C]	2 [C]
Code-focused	ReshapeF	-	2 [F]
	CollectF	2 [F]	-
	ProcessU	5 [F]	1 [F]
	NCP	-	1 [F]
One-Way	ReshapeF	-	2 [F]
	CollectF	2 [F]	-

on as they fail to maintain the high-level insight of connections between tasks that they described initially.

5.3 RQ2: Success on Programming Problems

RQ 2 asks how students' movements between tasks and code relate to their success on our programming problems. Table 1 shows the number of students who attempted each solution approach per problem and whether their code was close to a working solution.

Observation 1. *Students who followed the cyclic pattern generally developed more correct solutions than those with other patterns.*

Transcripts of *cyclic* students in both problems showed that they concretely described the tasks they implemented, capturing both the *role* of each task and how the tasks *connected* to each other. The descriptions of these connections were critical in informing the composition of their code. In Rainfall, at least half the students in each cohort exhibited a *cyclic* movement between tasks and code and were generally close to a correct solution. None of the *code-focused* or *one-way* students developed correct solutions. Their transcripts reveal that while they identified problem-tasks, they struggled to concretely relate these tasks, which seemed to influence their ability to implement these tasks in code. Others revealed a dissonance between their high-level plan and the code-pattern they retrieved, failing to recognize the limitations of their retrieved patterns in the context of the tasks they identified. The number of *cyclic* students decreased in Max-Temps: of six students who were *cyclic* in Rainfall, three moved to *code-focused* and three to *one-way* in Max-Temps. Their data also show that they struggled to concretely describe connections between tasks they identified.

Observation 2. *Some students struggle to capture identified task-level relationships at the code level.*

A prevalent factor in why students get stuck, particularly in Max-Temps, is that they fail to concretely describe how tasks connect to each other. For example, students who attempt the *Reshape-first* approach can't figure out how to keep track of the sublists: the missing relational glue here is the data structure to keep track of the sublists (i.e. a list-of-lists). Without articulating the data structure, students do not know what a reshaping function should produce and what code constructs to use to implement reshaping. They also do not know what data to use as input for functions that process the reshaped input. The interviews and an inspection of the course syllabi reveal that students have not done problems involving list-of-lists or restructuring flat lists into nested lists; thus, they do not have patterns to retrieve for reshaping the data. Similarly, students struggled to figure out how to track data for *Process-until* (tracking the current max) and *Collect-first* (tracking the current sublist). All of these approaches also require some form of recursion over a modified suffix of the list, but students lack a previously-learned schema for modifying the suffix of the list to recur on.

Observation 3. *Some students who got stuck failed to adapt patterns to new contexts.*

In Rainfall, students got stuck on handling the -999 sentinel. Students mentioned in the interviews that they have not worked on problems that terminated list-computations prematurely at a specific element rather than the end of the list (the empty-list). They get stuck because they fail to see the similarity of roles between -999 and the empty-list as base-cases.

Observation 4. *Some students who got stuck failed to identify the limitations of the pattern they retrieved.*

A prevalent mistake among students who got stuck in Rainfall is that they started mechanically from the list-template and wrote code for the average formula within the recursive-case of the template, as in student COURSE1-STUD2's code below:

```
(define (average alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon)
     (/ (+ (first alon) (average (rest alon)))
        (number (alon)))]))
```

From the perspective of the problem, this makes sense: the list-type input prompts the retrieval of the list-template, which students filled in with code for *average*. This, however, overuses the template. Students did not decompose the *average* code around the *sum* and *count* subtasks into their own recursive templates, missing that the average task itself is not a recursive computation, and thus requires a slight modification to the template. They did not concretely think about how the retrieved average formula's task-components impact the use of the template code. Courses explain that a single template function can only perform one traversal-based operation; our data suggests that this task-level decomposition needs more emphasis, and that students may need to be taught how to recognize apriori when a problem requires them to modify the schemas they know.

5.4 RQ 3: How Did Students Get Unstuck?

RQ 3 asks what students do when they get stuck on a problem. Our analysis, however, pointed to an obvious pattern: **when students got stuck, they *remained stuck***. We hoped students would fall back on appropriate design recipe steps to uncover gaps in their understanding of the problem or of the tasks. They didn't recognize to use the design recipe techniques to get unstuck. In general:

Observation 5. *Even when students started with the design recipe steps, they did not come back to them when they got stuck.*

Missed opportunities in Rainfall. Writing examples of the input or test cases would have helped students see the base-case role of -999; none of those who got stuck did this. The code below shows examples of data, which all produce the same rainfall average result:

```
; Examples of data
(define data1 (list 1 2 -7 3 -999 4))
(define data2 (list 1 2 -7 3 -999))
(define data3 (list 1 2 -7 3))
```

Students mostly wrote examples and test cases at the start of their process and often just copied the example given in the problem statement. When they wrote test cases at the latter parts of their process, it was only to check if their code ran. They rarely, if ever, wrote examples and test cases to concretely illustrate their current understanding of the problem or task at hand.

The average-formula problem is trickier: students could identify each of the task-components first, then follow the design recipe for each of the inner-tasks (i.e. *sum* and *count*). Doing so should lead students to simply call *sum* and *count* within *average*. This, however, requires a more explicit *task-level decomposition* step before working on the design recipe, to identify task-components that may require their own template-based traversals. None of the students who got stuck did this, instead modifying code in trial-and-error fashion.

Missed opportunities in Max-Temps. Some students wrote data definitions, motivated by the novelty of a list with both numeric and string elements. Their data definitions, however, were either incomplete or incorrect. For example, student COURSE2-STUD3 wrote a data definition for the elements of the input list, but missed writing a data definition for the actual list; this led her to use a template that was only good for a list element, but not the input list itself:

```
; A Newday is one of
; - "new-day"
; - Number
;
; (define (nd-temp nd)
;   (cond [(string=? nd "new-day")...]
;         [(number? nd) ...]))
```

The correct input data definition and template would have been:

```
; A list-of-element is
; - empty, or
; - (cons number list-of-element), or
; - (cons string list-of-element)
;
; (define (func input)
;   (cond [(empty? input) ...]
;         [(string? (first input)) ...]
;         [(number? (first input)) ...]))
```

Even when students got stuck writing their code using an incorrect template, they never went back to reexamine and correct their data definitions (or template). Students whose Max-Temps code was close to correct wrote their functions using the correct template, suggesting that identifying the correct template may be a step in the right direction in writing a correct Max-Temps solution.

6 CONCLUSIONS AND TAKEAWAYS

We developed a conceptual framework that captures the task- and code-level thinking students engaged in as they solved our programming problems. Our findings suggest that students who concretely described relationships between tasks were able to move back-and-forth between tasks and code and had the most success with the given problems. Students who struggled failed to capture how the tasks in the problems interconnected, could not transfer patterns to new contexts, or overused the patterns they retrieved.

Insights on Teaching the Design Recipe. Our observations of how students used the design recipe suggests that they see the recipe as a process to *start with and follow*, but not as a set of techniques to *return to* when they get stuck. Discussions with the course instructors corroborated this hypothesis: lectures had not emphasized using the recipe steps when debugging. These suggest that the students may have built a *habit* of following the design recipe, but not necessarily an *insight* around how each recipe step is a technique towards building a concrete understanding of the problem-space. Students may need additional instruction that focuses on *how* to use the design recipe steps *mid-process* (not only to start with) when they get stuck. For example, students might be given code with errors and explicitly asked to reason about the causes of the errors using specific design recipe steps. Targeted exercises such as this might help students practice a more insightful use of the design techniques rather than just as a mechanical habit.

Insights on Task- and Code-level Thinking. Our observations confirm findings from prior studies [2] that students think in terms of a problem's core tasks. Where they struggle is in concretizing relationships between tasks, which affects their ability to compose tasks' code implementations, especially for tasks they have not seen before. Using techniques from the design recipe can help uncover some of these relationships. Some task-relationships required new patterns students have not seen, for example, doing a recursion on a modified list suffix or prefix, or keeping track of lists using another list. This suggests that just because students have seen an instance of a pattern (e.g. a list of numbers; recursion on the tail of the list), does not mean that students can generalize those patterns to other contexts (e.g. a list of lists; recursion on a modified suffix of the list). As instructors, we should be careful not to assume that students have understood the underlying *idea* behind a pattern, simply because we've shown them an instance of it. Focusing on enforcing students' understanding of patterns they're taught may help them navigate between tasks and code better in new situations.

Conclusions. Overall, our findings indicate that students need to be explicitly taught techniques for navigating back to high-level plans when they get stuck, and how to identify when current code has diverged from a plan and needs to be rethought. Both are significant open questions for computing education.

ACKNOWLEDGMENTS

We thank the instructors and their students for participating in our project. Work supported by US NSF grants 1116539 and 1500039.

REFERENCES

[1] Jerome Seymour Bruner. 1978. The role of dialogue in language acquisition. In *The Child's Conception of Language*, W.J.M. Levelt A. Sinclair, R.J. Jarvella (Ed.). Springer-Verlag, New York, 241–256.

[2] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. 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)*. ACM, New York, NY, USA, 205–210. https://doi.org/10.1145/2839509.2844574

[3] Francisco Enrique Vicente G. Castro. 2016. Pedagogy and Measurement of Program Planning Skills. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, New York, NY, USA, 273–274. https://doi.org/10.1145/2960310.2960344

[4] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. SAGE.

[5] Michael de Raadt, Richard Watson, and Mark Toleman. 2009. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95 (ACE '09)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 45–54. http://dl.acm.org/citation.cfm?id=1862712.1862723

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/

[7] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 12–20. https://doi.org/10.1145/3105726.3106183

[8] Emily R. Fyfe, Nicole M. McNeil, Ji Y. Son, and Robert L. Goldstone. 2014. Concreteness Fading in Mathematics and Science Instruction: a Systematic Review. *Educational Psychology Review* 26, 1 (2014), 9–25.

[9] Päivi Kinnunen and Beth Simon. 2012. Phenomenography and grounded theory as research methods in computing education research field. *Computer Science Education* 22, 2 (June 2012), 199–218.

[10] Orna Muller, David Ginat, and Bruria Haberman. 2007. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '07)*. ACM, 151–155.

[11] Peter L. Pirolli and John R. Anderson. 1985. The Role of Learning from Examples in the Acquisition of Recursive Programming Skills. *Canadian Journal of Psychology* 39 (1985).

[12] Peter L. Pirolli, John R. Anderson, and Robert G. Farrell. 1984. Learning to program recursion. In *Proceedings of the Sixth Annual Cognitive Science Meetings*. 277–280.

[13] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989).

[14] Robert S. Rist. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Hum.-Comput. Interact.* 6, 1 (Mar 1991), 1–46.

[15] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858.

[16] James C. Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *Proceedings of IJCAI*.

[17] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '14)*. ACM, 279–284.