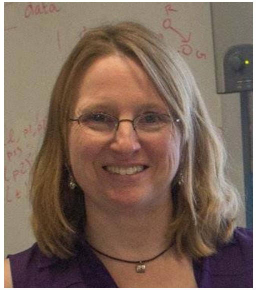
Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers



Francis Castro
@_franciscastro_

WPI

Kathi Fisler
@KathiFisler

BROWN

Paper and slides:
bit.ly/francis-sigcse2020

Email:
fgcastro@cs.wpi.edu

SIGCSE 2020
Portland, Oregon

Use a ‘**for**’ loop and ‘**if**’ . . .

Given a list of numbers, produce the average of the non-negative numbers that occur before -999

Get the **non-negatives** first, then **sum** and **count** . . .

Thinking in **code**

Thinking in **tasks**

Research tells us that: Students **retrieve** prior **code** and/or **task** knowledge

---

- How do students **move** between these two levels while programming?
- How do these movements relate to their **success** on our programming problems?
- How do students approach –
  - familiar problems?
  - novel problems?
- What do they do when they get stuck?
  (How do they use design techniques they’re taught?)

How do students move between these two levels while programming?

We gave students **problems with varying degrees of novelty**

Problems

Rainfall – compose known tasks/subproblems in new ways

Given a list of numbers, produce the **average** of the **non-negative** numbers that occur **before -999**

Example: **rainfall** ([**1**, **1**, -3, **4**, -999, 20]) is 2

Tasks

- Sum
- Count
- Average [new composition]
- Ignore negatives
- Terminate [new task]

Max-Temps – solve and compose new tasks/subproblems

Given a list of numbers, return the **max** values in each **sublist** as separated by a delimiter (e.g., '**d**')

Example: **maxtemps** ([**3**, **5**, d, **2**, d, **7**, **5**, **3**]) is [5, 2, 7]

- Find sublists [new task]
- Find max
- Build results
- (Reshape input) [new task]

[**3**, **5**, d, **2**, d, **7**, **5**, **3**]

↓

[**[3, 5]**, **[2]**, **[7, 5, 3]**]

Students — **Think-alouds with students** from two universities; both schools used the **same curriculum** (design recipe) with some variations in topic orderings

```
STEP 1: DESCRIBE THE SHAPE OF THE INPUT

   A list-of-number is
   - empty, or
   - (cons number list-of-number)

STEP 2: WRITE EXAMPLES OF THE INPUT

   (define even-nums (list 4  2  6))
   (define odd-nums  (list 5  1  27))
   (define one-num   (list 142))

STEP 3: DESCRIBE THE PROPOSED FUNCTION

   sum-nums : list-of-numbers -> number
   Produces the sum of numbers in the list

STEP 4: ILLUSTRATE THE FUNCTION'S PURPOSE W/ EXAMPLES (TEST CASES)

   (check-expect (sum-nums even-nums) 12)
   (check-expect (sum-nums odd-nums) 33)

STEP 5: WRITE A FUNCTION TEMPLATE BASED ON THE INPUT SHAPE (STEP 1)

   (define (fxn-name list-input)
     (cond [(empty? list-input) ... ]
           [(cons? list-input) ... (first list-input)
                            (fxn-name (rest list-input))]))

STEP 6: FILL IN THE DETAILS

   (define (sum-nums nums-list)
     (cond [(empty? nums-list) 0 ]
           [(cons? nums-list) (+ (first nums-list)
                                 (sum-nums (rest nums-list)))]))

STEP 7: TEST AND REFINE
```

Methods — We audio-recorded and transcribed the sessions, then **coded** for how students went about each problem, capturing (among others):

- when they talked in terms of tasks/code
- **tasks** students identified or planned out
- how individual tasks were **implemented**
- overall **approach** of their final solution

Key observations: How do they move between tasks and code?

Code-focused students jump immediately into writing code

- No high-level solution plan – only think about problem-level tasks **on-the-fly**
- Get **stuck**

One-way students start with a high-level plan –

- – but revert to a **code-focused** behavior
- Don't return to thinking about their plan or problem-level tasks and get **stuck** later on

Cyclic students often talked about problem-level tasks in the context of a **high-level plan**

- Concrete descriptions of tasks' code implementations
- **Compositions** of code is guided by their descriptions of **relationships between tasks**
- Mostly succeeded on the problems

crs2-student4:
"I'm thinking [the] best way to approach [Rainfall], you take your list of 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"

Key observations: Why do students get stuck?

Students struggle to describe **how tasks connect** to each other

Max-Temps example:

- Reshaping data – extract and track the sublists

[**3**, **5**, d, **2**, d, **7**, **5**, **3**] → [**[3, 5]**, **[2]**, **[7, 5, 3]**]

- Students can't figure out how to keep track of sublists (list-of-lists)

crs1-student1: *"I think what would be the best if I split it up into lists and then worked through each list individually but I'm not quite sure how to [store the lists]"*

- Fragmented plans: they do not know –
  - What reshaping function produces
  - What data to use as input to process a reshaped input

Key observations: Why do students get stuck?

Students fail to identify the **limitations of retrieved patterns** in the context of the tasks

Rainfall example:

- Mechanical use of the list template

```
(define (average nums-list)
  (cond [(empty? nums-list) empty ]
        [(cons? nums-list)
         (/ (+ (first nums-list) (average (rest nums-list)))
            (+ 1 (average (rest nums-list))))]))
```

- Makes sense (based on problem statement), but overuses the template

Given a list of numbers, produce the average of the non-negative numbers that occur before -999

- Did not think about how average's task-components impact the use of the template code
  - Need to modify template? How?

average (list input, but no traversal)

sum (traversal task) **count (traversal task)**

```
(define (average nums-list)
  (cond [(empty? nums-list) -1 ]
        [(cons? nums-list)
         (/ (sum nums-list) (count nums-list))]))
```

Key observations: Why do students get stuck?

Students are mechanically producing data definitions (step 1) they'd seen before

```
A Newday is one of
- "new-day"
- Number

(define (nd-temp nd)
   (cond [(string=? nd "new-day") ... ]
         [(number? nd) ... ]))
```

- Fine for a single element... but not the input list

```
A Day is one of
- empty
- (cons number string)
```

- Used regular list template (with errors)
- Suggests mechanical writing of data-definitions

- Instructor interviews show that students had only seen a limited repertoire of data

Recommendation: Have students do a wider variety of data design activities

```
A list-of-element is
- empty, or
- (cons string list-of-element), or
- (cons number list-of-element), or

                    ↓

(define (fxn-name input)
   (cond [(empty? input) ... ]
         [(string? (first input)) ... ]
         [(number? (first input)) ... ]))
```

Key observations: How did students try to get unstuck?

We hoped students would fall back on appropriate design recipe steps when stuck…

- **Mechanical use** – they started with them, but **did not go back to them** when they got stuck
- Missed opportunities:

```
(define (rainfall lon)
  (cond [(empty? lon) 0 ]
        [(cons? lon)
         (if (not-999? (first lon))
             (/ (+ (first lon) (rainfall (rest lon)))
                (length (filter not-999? lon)))
             0)]))
```

crs2-student9 – What's happening here?

- Mechanical use of template
- Did not decompose the code around the tasks
- Struggled to figure out what to do with -999

Expand examples to identify tasks/decompositions

```
(rainfall (list 1  2  3))    ->  2

(/ (sum (list 1  2  3))
   (count (list 1  2  3)))   ->  2
```

Using examples may show base-case role of -999

```
(rainfall (list 3  1  -7  2  -999  4))  ->  2
(rainfall (list 3  1  -7  2  -999))     ->  2
(rainfall (list 3  1  -7  2))           ->  2
```

Takeaways

Students with most success

- Concretely described **task-relationships** in the context of an **overall solution plan**
- Used insight from task-relationships to **guide the composition of their code**

Students who struggled

- **Primarily worked in code** without context of an overall solution plan

Teaching design practices

- Not enough to teach how to use design techniques to plan solutions in advance
- Students need to do a wider variety of data design activities
- Students also need to be taught **how to go back to design techniques** when stuck

- Example activities:
  - Give code with errors and use design recipe steps to reason about causes of errors
  - Expand examples to identify tasks/decompositions

Takeaways

Teaching design practices

- Teach **problem-level decomposition** explicitly – guide code compositions with insights from task-relationships

- Have students do more activities around identifying and planning around tasks without being expected to write code

- Example activities:
  - Multi-task problems: Have students identify and write tasks and concretely describe how tasks relate to each other (e.g. use type signatures)
  - Show how a decomposition of a problem into tasks in advance lends to smaller, (template) functions

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

Speaker:

Francis Castro

@_franciscastro_

WPI

Paper and slides:
bit.ly/francis-sigcse2020

Email:
fgcastro@cs.wpi.edu

Email or tweet me for questions!