Development of a Data-Grounded Theory of Program Design in HTDP

How to Design Programs

(What we learned from about ~180 hours of watching and listening to students as they program)

Francisco Castro

Advisor: Kathi Fisler, PhD

Computing Education Research (CER) Areas

My dissertation

How to explicitly teach program design strategies?

Muller ac.

Name: *Maximum Value*

Initial state: collection of values.

Goal: maximal value in the collection

Algorithm:

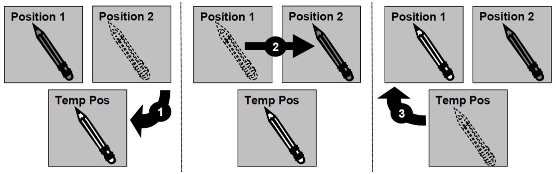
```
Initialize   Max to First_value
While there are more items do
    Assign next element to Next_Element
    If Next_Element > Max then
        Assign Next_Element to Max
```

+ 9-item guideline for constructing problems that use the patterns

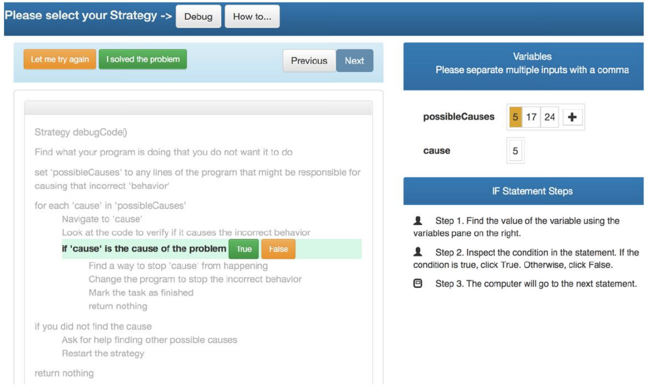
Ko ac.

```
# If you need help finding the problem, ask for help.
Find what your program is doing that you do not want it to do
# Write the line number inside of the program
# and separate with commas.
SET 'possibleCauses' to any lines of the program that
    might be responsible for causing that incorrect 'behavior'
FOR EACH 'cause' IN 'possibleCauses'
    Navigate to 'cause'
    # Ask for help if you need guidance on how.
    Look at the code to verify if it causes the incorrect behavior
    IF 'cause' is the cause of the problem
        # If you need help finding the problem, ask for help.
        Find a way to stop 'cause' from happening
        # Ask for help if you need guidance on how.
        Change the program to stop the incorrect behavior
        # Ask for help if you need guidance on how.
        Mark the task as finished
        RETURN nothing
IF you did not find the cause
    Ask for help finding other possible causes
    Restart the strategy
RETURN nothing
```

De Raadt ac.




```
#include <stdio.h>

int main() {
     int firstPosition  = 5; // First position containing value to swap
     int secondPosition = 6; // Second position containing value to swap
     int tempPosition;       // Temporary position for swap

     // Output the numbers after the swap
     printf("Before Swap...\n");
     printf("First: %i, Second: %i\n", firstPosition, secondPosition);

     // Swap the two numbers in a triangular swap
     // 1. Copy the value from the second position to temp
     tempPosition = secondPosition;

     // 2. Copy the value from the first position to the second
     secondPosition = firstPosition;

     // 3. Copy the value from the temp position to the first
     firstPosition = tempPosition;

     // Output the numbers after the swap
     printf("After Swap...\n");
     printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```



STEP 1: DESCRIBE THE SHAPE OF THE INPUT

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

THE DESIGN RECIPE

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/ INPUT-OUTPUT EXAMPLES

```
(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

How to Design Programs (**HTDP**) teaches an *explicit design process*, but has not been studied in terms of *how* students use it *in situ*

- How do students allocate tasks? (traversals/accums/etc)
- Need to study how students use design recipe to identify how to teach it in a way that's helpful to students

Early HTDP work

- Felleisen ac.
  Introduced HTDP and accompanying tools
- Bienusa ac., Crestani ac., Sperber ac.
  Germany: Experience reports of designing a CS1 curriculum
- Fisler, Fisler ac.
  High-level solution structures, errors that HTDP students produced
- Ren ac.
  HTDP for categorizing students' office hours questions
- Wrenn ac.
  Tool for providing feedback on examples written (pretty cool)

STEP 1: DESCRIBE THE SHAPE OF THE INPUT

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

THE DESIGN RECIPE

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/ INPUT-OUTPUT EXAMPLES

```
(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

How to Design Programs (**HTDP**) teaches an *explicit design process*, but has not been studied in terms of *how* students use it *in situ*

- How do students allocate tasks? (traversals/accums/etc)
- Need to study how students use design recipe to identify how to teach it in a way that's helpful to students

---

How do HTDP-trained students use the *design recipe* to solve *multi-task* programming problems?

DRQ1. What program design skills do HTDP-trained students exhibit when developing solutions for multi-task programming problems?

DRQ2. What interactions do we observe between students' program design skills and how do these contribute to their development of solutions for multi-task programming problems?

DRQ3. How do HTDP-trained students' use of program design skills evolve during a CS1-level course?

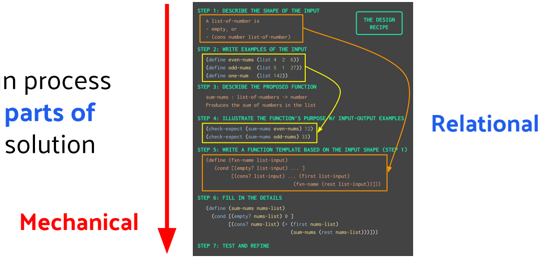
DRQ4. How do HTDP-trained students approach multi-task programming problems with novel components?

Dissertation Research Overview

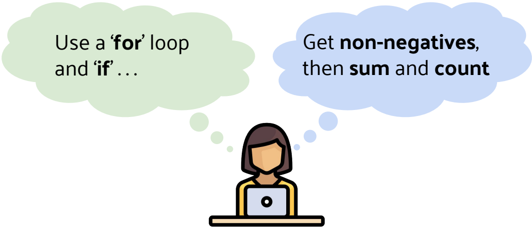
Research Question	Data
DRQ1. What program design skills do HTDP-trained students exhibit when developing solutions for multi-task programming problems?	• Video captures of programming sessions while solving multi-task problems • Interview, think-aloud, code submissions, scratch work, field observations
DRQ2. What interactions do we observe between students' program design skills and how do these contribute to their development of solutions for multi-task programming problems?	• Interview, think-aloud, code submissions, scratch work, field observations
DRQ3. How do HTDP-trained students' use of program design skills evolve during a CS1-level course?	• Interview, think-aloud, code submissions, scratch work, field observations from multiple points within a CS1 course
DRQ4. How do HTDP-trained students approach multi-task programming problems with novel components?	• Video captures of programming sessions while solving multi-task problems • Interview, think-aloud, code submissions, scratch work, field observations on two problems of varying degrees of novelty

Overall Takeaways – What we learned

1. Students engage in their program design process either **mechanically** or by **relating** how **parts of their process** contribute to their overall solution

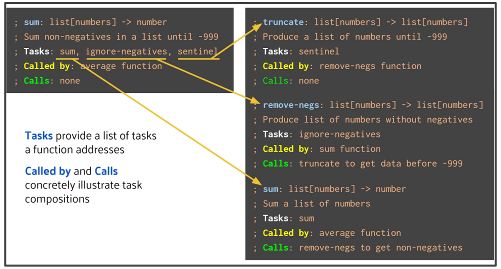


2. How HTDP students **move between task- and code-level thinking** indicate their success in designing solutions



3. **Problem decomposition** is a critical program design skill and needs to be explicitly made a part of the design process

4. Students may benefit from instructional activities and problems explicitly aimed at moving them **beyond a mechanical use of** their program design **skills**



5. Studying how students use the design skills put forth by a curriculum must **consider** both the **curriculum and instructional activities** used to teach the curriculum

6. Assessing students' program design skills needs to consider both the **level and consistency** at which they are applying them

7. Developing a data-grounded theory for HTDP provides a **language for explaining what** students do, **why** they do them, and **how** these affect their design work

Timeline of Studies

2015 [Castro and Fisler, 2016]

Study 1: Exploring how HTDP students design for new problems

Students worked on Adding-Machine during a weekly lab

[1 2 0 7 0 5 4 0 0 6]

[3 7 9]

- Video-captured IDE activity
- Retrospective survey (how started, use of DR, got stuck, notes)

Preliminary observations of students' design processes

Data wasn't rich enough

2016 [Castro and Fisler, 2017]

Study 2: Exploring students' design work throughout a CS1 course

Longitudinal study – conducted studies with students at multiple points during CS1

- Interview sessions on homework problems + solution comparison
- Think-aloud session on **Rainfall** problem

[2 -5 0 -3 4 -999 20 6] → 2

Coding through a Grounded Theory analysis of qualitative data

Framework of program design skills

Need to validate the framework with other HTDP experts

2017

Study 3: Validating the SOLO skills framework with HTDP instructors

Recruited HTDP instructors from different institutions

- Assessed students based on the skills identified in the skills taxonomy
- Explain ratings
- Describe other factors they looked for when assessing students

Thematic coding of instructor responses

Timeline of Studies

2017 [Fisler and Castro, 2017]

2018 — 2019 [Castro and Fisler, 2020]

Study 4: Exploring how students navigate schemas to design solutions

Developed design process narratives from think-aloud data

- Discussions of solution structure
- Discussions of problem tasks
- Reasoning around edits
- Selection of schemas

Study 5: Multi-university study exploring how students move between task-level and code-level thinking on multi-task problems

Sessions: Think-aloud + retrospective interviews on multi-task problems

- **Rainfall**

  `[ 2 -5 0 -3 4 -999 20 6 ]` → `2`

- **Max-Temps**

  `[ 40 42 d 50 d 52 56 53 ]` → `[ 42 50 56 ]`

- Grounded theory-based analysis of think-alouds, field observations, code
- Developed design process narratives

Deeper analysis of Study 2 data

Explore findings on students from a different university

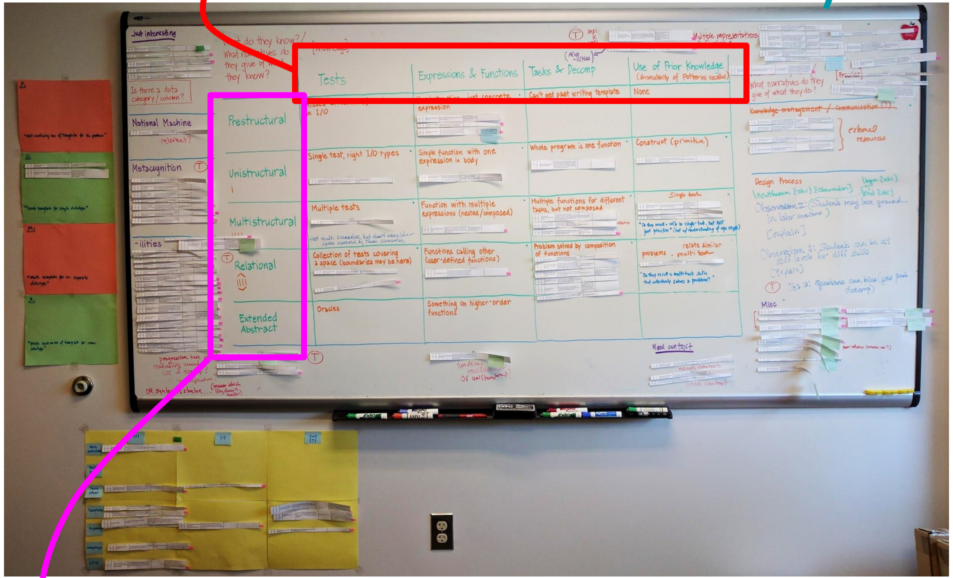
Use our skills framework as analytical lens for new data

Making sense of our data

Themes

- program design skills
- other factors (e.g. quality attributes, value judgments)

Coding the data through a Grounded Theory-based analysis



SOLO level	Methodical choice of tests and examples	Composing expressions within function bodies	Decomposing tasks and composing solutions	Leveraging multiple representations of functions
Prestructural	Does not know how to write tests/test expressions; misses the structure of tests, ie input and output	Does not know how to define a function	Does not identify relevant tasks for a problem; does not know how to translate elements of a problem statement into relevant tasks	Just dives in and writes code; uses only a single representation
Unistructural	Able to write tests; descriptions of test cases do not explain the purpose of the test(s); does not express the idea of varying test scenarios	Able to define functions in a simple context: uses primitive operations on primitive types in a function body	Able to identify relevant tasks but no reflections of separate tasks when talking about the code	Blindly follows the design recipe; sees each function representation as independent of others
Multistructural	Able to write multiple tests; describes the purpose of individual tests but does not articulate any relationship between or collective purpose for the tests	Able to define functions whose bodies contain nested non-primitive expressions or function calls, but does not articulate the semantics of how the results of calling a function return to the calling context	Able to identify relevant tasks; articulates the delegation of tasks into separate functions/ expressions but fails to articulate how to effectively compose the tasks in a way that solves the problem	Articulates a sense of the function representations talking about or referring to the same computation
Relational	Able to write tests; identifies a collective purpose for the tests, i.e. boundaries, edge cases, test space coverage, but limited within the context of the problem	Able to define functions whose bodies contain nested non-primitive expressions or function calls and is able to articulate the semantics of how the results of calling a function return to the calling context	Able to identify relevant tasks; articulates the delegation of tasks into separate functions/ expressions and can articulate how to effectively compose the tasks in a way that solves the problem	Articulates a mechanism through which function representations are related, e.g. template uses types to drive the code structure, execution of a program connects to a test space, etc.

SOLO levels

Structure of **O**bserved **L**earning **O**utcomes

- Prestructural
- Unistructural
- Multistructural
- Relational
- Extended Abstract

Increasing levels of conceptual complexity

Best paper
Koli Calling 2017

Designing a Multi-Faceted SOLO Taxonomy to Track Program Design Skills Through an Entire Course

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

Kathi Fisler
Brown University and WPI
kfisler@cs.brown.edu

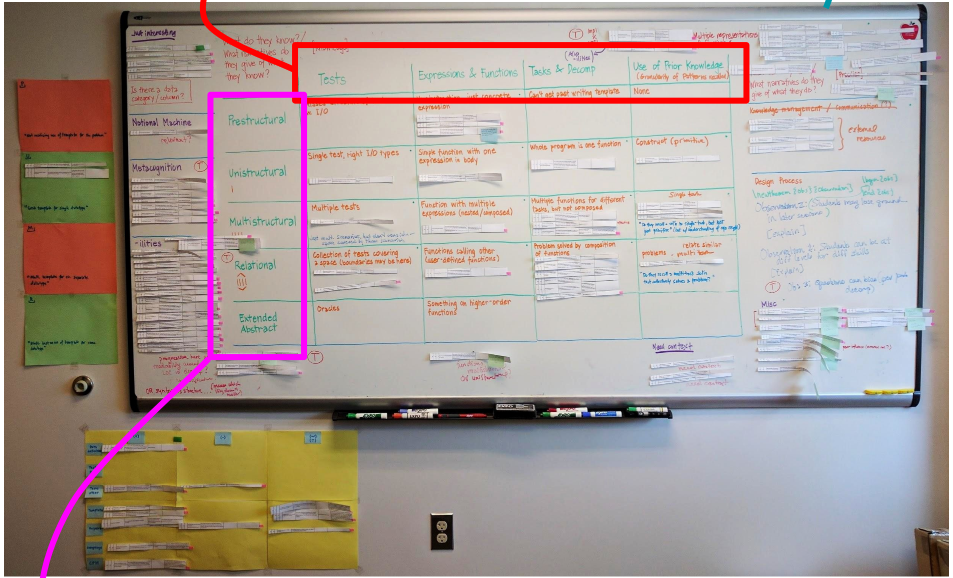
ABSTRACT

This paper explores how to assess students' skills in program design and how those skills evolve across an entire CS1 course. We ...

design differently, and perhaps more systematically, as they gain in experience and confidence. Understanding how program-design skills evolve in novice learners provides valuable input to those who design curricula and pedagogy. Such understanding requires both

Making sense of our data

Themes
- program design skills
- other factors (e.g. quality attributes, value judgments)



Coding the data through a Grounded Theory-based analysis

SOLO level	Decomposing tasks and composing solutions
Prestructural	Does not identify relevant tasks
Unistructural	Identify tasks but no logical separation
Multistructural	Decomposed tasks, no relational composition
Relational	Decomposition into tasks and concrete relationships between tasks

SOLO levels

Structure of Observed Learning Outcomes

- Prestructural
- Unistructural
- Multistructural
- Relational
- Extended Abstract

Increasing levels of conceptual complexity

Best paper Koli Calling 2017

Designing a Multi-Faceted SOLO Taxonomy to Track Program Design Skills Through an Entire Course

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

Kathi Fisler
Brown University and WPI
kfisler@cs.brown.edu

ABSTRACT

This paper explores how to assess students' skills in program design and how those skills evolve across an entire CS1 course. We

design differently, and perhaps more systematically, as they gain in experience and confidence. Understanding how program-design skills evolve in novice learners provides valuable input to those who design curricula and pedagogy. Such understanding requires both

Making sense of our data

Analyze and categorize new student data

Session	MTE	CDF	DTC	LRF
1	R	M	U	U
2	R	R	R	U
3	R	R	R	M

Students evolve in different skills at **different paces**

Session	MTE	CDF	DTC	LRF
1	U	M	U	U
2	R	R	M	U
3	U	M	M	U

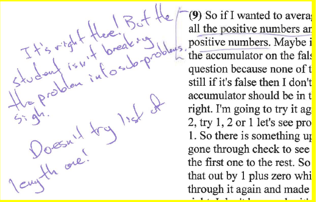
Students show **non-monotonic progression** of skills

(Program design skills)

SOLO level	MTE: Methodical choice of tests and examples	CDF: Composing expressions within function bodies	DTC: Decomposing tasks and composing solutions	LRF: Leveraging multiple representations of functions
Prestructural	Does not know how to write tests/test expressions; misses the structure of tests, ie input and output	Does not know how to define a function	Does not identify relevant tasks for a problem; does not know how to translate elements of a problem statement into relevant tasks	Just dives in and writes code; uses only a single representation
Unistructural	Able to write tests; descriptions of test cases do not explain the purpose of the test(s); does not express the idea of varying test scenarios	Able to define functions in a simple context: uses primitive operations on primitive types in a function body	Able to identify relevant tasks but no reflections of separate tasks when talking about the code	Blindly follows the design recipe; sees each function representation as independent of others
Multistructural	Able to write multiple tests; describes the purpose of individual tests but does not articulate any relationship between or collective purpose for the tests	Able to define functions whose bodies contain nested non-primitive expressions or function calls, but does not articulate the semantics of how the results of calling a function return to the calling context	Able to identify relevant tasks; articulates the delegation of tasks into separate functions/ expressions but fails to articulate how to effectively compose the tasks in a way that solves the problem	Articulates a sense of the function representations talking about or referring to the same computation
Relational	Able to write tests; identifies a collective purpose for the tests, i.e. boundaries, edge cases, test space coverage, but limited within the context of the problem	Able to define functions whose bodies contain nested non-primitive expressions or function calls and is able to articulate the semantics of how the results of calling a function return to the calling context	Able to identify relevant tasks; articulates the delegation of tasks into separate functions/ expressions and can articulate how to effectively compose the tasks in a way that solves the problem	Articulates a mechanism through which function representations are related, e.g. template uses types to drive the code structure, execution of a program connects to a test space, etc.

Validating the framework with other HTDP instructors

Decomposing tasks and composing solutions	1	This student makes little or no attempt to decompose the problem, with a faint hint of "oh we need a helper" right at the end. It seems that this student doesn't yet have a clear sense of the scope or boundaries of the patterns that he/she is learning. I feel that a successful student will use patterns like tools in a toolbox, and say "oh, I need one of these and two of these, and then staple it together," where this student is still in the phase of trying to figure out which end of the cornerstone hammer to hold, and whether it can do the whole job. Until you know the patterns well, you don't know their limitations.



Refine our framework descriptions and identify a new design skill

SOLO level	Meaningful Use of Patterns
Prestructural	Does not know what code pattern to retrieve
Unistructural	Blindly retrieves and writes a list-traversal pattern (list template, accumulator), without insight about how the problem tasks fit the pattern
Multistructural	Recognizes the need for multiple traversals for multiple tasks, but doesn't recognize/understand the limits of the pattern relative to the tasks and inappropriately conflates the patterns used
Relational	Can separate traversal-tasks in a meaningful way through an appropriate assignment of tasks to patterns (multiple templates) or parts of patterns (multiple accumulators)

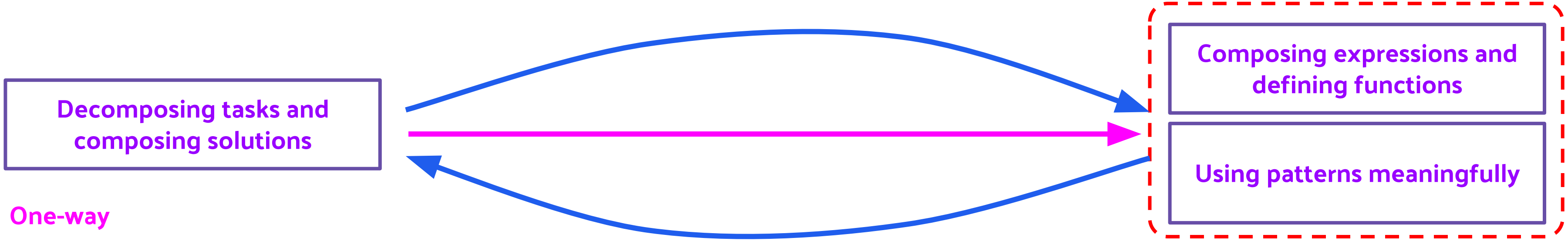
How students move between tasks and code matter

Cyclic

- Back-and-forth between tasks and code throughout their process
- **Consistently** apply skills at the **relational level**
  - Concretely describe task-relationships
  - Task-level plan guide the composition of code

Code-focused

- Jump immediately into writing code and **stay at the code-level**
- No overall task-level plan, no insight on how tasks inter-operate
- No insight about how tasks impact code



One-way

- Describe a task-level plan, but only at the **onset of their process**
- Regress to a code-focused process, failing to maintain connections between tasks, tasks and code

- Task-level planning is a critical skill
- Not enough to apply skills at the relational level, need to be **consistently relational**
- Lack of consistency may indicate fragility of skills and need for help

What did we learn about teaching program design with HTDP?

Teach task-level planning in advance

- Students decompose the problem at the **beginning** of their process
- Need to **make task-level planning a fundamental part** of the courses
  - "one function per task" wasn't enough

Focus on meaningful use of design recipe steps

- Difference between students who:
  - **reason about programs using the DR** vs.
  - **use the DR mechanically**
- Need to **teach DR beyond mechanical use**
  - course activities focused on **following** the DR to solve problems: **not enough**

- Activities that focus on how to **leverage design techniques for task-level planning**

Expanding examples to work out task decompositions

```
(adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))    ->  (list 3 7 10)
(adding-machine (list (+ 1 2) (+ 7) (+ 5 4 1))) ->  (list 3 7 10)
```

```
(rainfall (list 3 -8 -1 2 -2 1 -999 5))                 ->  2
(rainfall (/ (sum (list 3 2 1)) (count (list 3 2 1)))) ->  2
```

STEP 1: DATA DEFINITION

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

STEP 2: EXAMPLES OF DATA

```
(define even-nums (list 4  2  6))
(define odd-nums  (list 5  1  27))
```

STEP 3: SIGNATURE & PURPOSE

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

STEP 4: INPUT-OUTPUT EXAMPLES

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

STEP 5: TEMPLATE BASED ON DATA DEFINITION

```
(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)))]))
```

What did we learn about teaching program design with HTDP?

Teach task-level planning in advance

Focus on meaningful use of design recipe steps

- Activities that focus on how to **leverage design techniques for task-level planning**

Expanding purpose statements: describe tasks and relationships between tasks

```
; sum: list[numbers] -> number
; Sum non-negatives in a list until -999
; Tasks: sum, ignore-negatives, sentinel
; Called by: average function
; Calls: none
```

```
; truncate: list[numbers] -> list[numbers]
; Produce a list of numbers until -999
; Tasks: sentinel
; Called by: remove-negs function
; Calls: none

; remove-negs: list[numbers] -> list[numbers]
; Produce list of numbers without negatives
; Tasks: ignore-negatives
; Called by: sum function
; Calls: truncate to get data before -999

; sum: list[numbers] -> number
; Sum a list of numbers
; Tasks: sum
; Called by: average function
; Calls: remove-negs to get non-negatives
```

Tasks provide a list of tasks a function addresses

Called by and **Calls** concretely illustrate task compositions

STEP 1: DATA DEFINITION

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

STEP 2: EXAMPLES OF DATA

```
(define even-nums (list 4  2  6))
(define odd-nums  (list 5  1  27))
```

STEP 3: SIGNATURE & PURPOSE

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

STEP 4: INPUT-OUTPUT EXAMPLES

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

STEP 5: TEMPLATE BASED ON DATA DEFINITION

```
(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)))]))
```

What did we learn about teaching program design with HTDP?

Engage students in design activities that involve more varied data contexts

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

Where students struggled

How to write data definitions, templates

Course problems – what we found

- Had only designed for data that used the basic list data definition, template
- **Not enough**: limited range of problems and activities constrained understanding of design techniques and patterns

Our study problems

- data-processing
- noisy
- significant elements
- underlying structure

DATA DEFINITION

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

→

TEMPLATE BASED ON DATA DEFINITION

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

Rainfall

[2 -5 0 -3 4 -999 20 6]

Max-Temps

[40 42 d 50 d 52 56 53]

Successful students

- described underlying concepts of patterns and techniques

Students who struggled

- discussed patterns and techniques only at syntax-level
- patterns are "fixed"

Need problems that –
- reinforce concepts underlying patterns
- practice use of techniques and patterns in novel contexts

Overall Takeaways – What we learned

1. Students engage in their program design process either **mechanically** or by **relating** how **parts of their process** contribute to their overall solution

2. How HTDP students **move between task- and code-level thinking** indicate their success in designing solutions

3. **Problem decomposition** is a critical program design skill and needs to be explicitly made a part of the design process

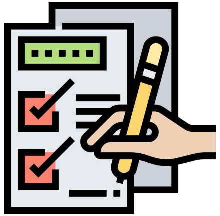
4. Students may benefit from instructional activities and problems explicitly aimed at moving them **beyond a mechanical use of** their program design **skills**

5. Studying how students use the design skills put forth by a curriculum must **consider** both the **curriculum and instructional activities** used to teach the curriculum

6. Assessing students' program design skills needs to consider both the **level and consistency** at which they are applying them

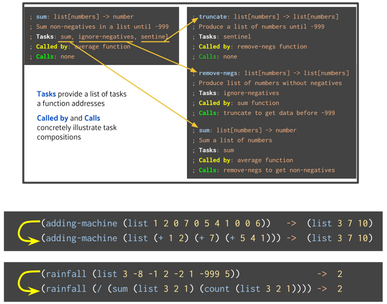
7. Developing a data-grounded theory for HTDP provides a **language for explaining what** students do, **why** they do them, and **how** these affect their design work

Future Directions



Further validation of the SOLO-based program design skills framework with other HTDP and non-HTDP CS1 courses

- Usability as a skill-assessment rubric?
- To what extent is our taxonomy curriculum-specific?



Study the **impact of recommended instructional activities** on how students perform on multi-task programming problems

- Do the activities move students from a mechanical to a relational use of design techniques and patterns?

Study the **impact of programming language on how students design** for multi-task programming problems

- Do students who learn in other programming languages struggle with our problems in similar ways as our students?
- What aspects of the languages have an impact on students' design work?