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 gathered various data from students, including programming samples and transcripts from interview and think-aloud sessions. As we coded the data, a progression resembling a SOLO taxonomy appeared to emerge bottom-up. As we refined this with top-down perspective from our curricular goals, we ended up with a novel multi-faceted SOLO taxonomy to track students' progress. We also identified data that don't fit a SOLO progression, yet reflect relevant traits and habits about design. In applying our framework, we learned several lessons about defining SOLO taxonomies and study protocols that leverage them. The major contributions of this work are (1) a taxonomy with separate but inter-related progressions for different design skills (that is applied to an entire course rather than a single assignment), along with (2) various methodological lessons about applying and designing assessments around SOLO taxonomies in this context.

CCS CONCEPTS

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

KEYWORDS

CS1, SOLO taxonomy, program design, qualitative methods

1 INTRODUCTION

Introductory computing courses are often designed towards a common learning outcome: students should be able to develop viable programs to solve at least small-scale problems. While different courses may use different programming languages and problem domains, the underlying program-design skills are fairly common, and include selecting appropriate language constructs, composing code fragments for multiple problem tasks, and checking whether the resulting program satisfies the original problem constraints.

Design skills evolve throughout a course, fostered through repeated application of language constructs and development of program schemas. We would expect students to approach program 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 assessments that explore design skills from various perspectives, but also rubrics for summarizing design skills across assessments.

This paper reports on the first phase of a study into the evolution of students' design skills in an introductory CS curriculum. We interviewed students about their design practices every two weeks during a 7-week CS1 course. Two sessions reviewed students' homework submissions, while the third asked students to solve the Rainfall problem. We then used open coding on the transcripts to develop a rubric for assessing the evolution of program-design skills. What emerged was a multi-strand SOLO taxonomy. We also identified several factors that are not amenable to a SOLO progression, but suggest potential impact on students' design decisions. After developing progressions for individual skill strands, we calibrated across the strands to give some coherence to each SOLO-level. Given the number of component skills that "program design" encompasses, we believe a multi-strand taxonomy offers a more nuanced view of how design skills develop than more recent efforts to use a single taxonomy to assess student work along a single dimension.

The main focus of this paper is the design of our multi-strand SOLO taxonomy, and what we learned from applying it to assess student data (program solutions and interview/think-aloud data) across an entire course. Lessons about student design thinking that emerged from using this taxonomy will be the focus of a different paper. Our main contributions here are:

(1) proposing multi-strand SOLO taxonomies
(2) a concrete multi-strand taxonomy that captures aspects of program design beyond coding and algorithm selection
(3) two models for applying the taxonomy across an entire course
(4) methodological lessons about designing such taxonomies and designing assessments around those taxonomies

2 RELATED WORK

Biggs and Collis proposed the *Structure of Observed Learning Outcomes* (SOLO) taxonomy model in 1982 [2]. These taxonomies capture the complexity of learning outcomes, looking at which aspects of an overall task students have mastered. Each taxonomy progresses through five levels of complexity:

- *Pre-structural*: little to no understanding of the topic
- *Uni-structural*: understand one aspect of the task
- *Multi-structural*: understand several aspects of the task, but the aspects are understood independently of one another
- *Relational*: understand several aspects of the task and how they inter-operate

Koli Calling 2017, November 16–19, 2017, Koli, Finland

ACM ISBN 978-1-4503-5301-4/17/11…$15.00
https://doi.org/10.1145/3141880.3141891

- *Extended Abstract*: can generalize understanding of aspects to a new domain

A single taxonomy is intended to detail a progression of outcomes within a single conceptual task. Biggs and Collis proposed the model as assisting in both assessment of student learning and in creating "constructive alignment" between assessments and curricula.

Within computer science, SOLO taxonomies appear to have first gained traction in the mid 2000s in the work of CSEd researchers in Australia and New Zealand. Papers that use SOLO in CS education generally focus on assessment of student work on a single assessment: researchers identify a skill that they plan to assess, articulate a SOLO taxonomy relative to that skill, apply the taxonomy to student work on an assessment (exam or exercise), and report student performance relative to the taxonomy. The papers present the final SOLO taxonomy, without discussing how it was designed. Such papers include Whalley et al. [23] (code reading, comprehension, and summarization), Lister et al. [10] (code comprehension), Shuhidan et al. [17] (writing code to calculate max/min integers), Ginat et al. [6] (algorithm design), and Izu et al. [7] (code design).

Our work builds on this model in several key ways. Our SOLO taxonomy is designed to help us track and assess students' progress across multiple assessments in a (full) course, not just a single assessment. As the course naturally targets several skills, our taxonomy has multiple strands, which we aligned at key points (see section 5.2). In addition, we detail the process through which we designed our taxonomy: it results from a combination of bottom-up and top-down analysis of our data, which grounds our taxonomy in our larger project from multiple perspectives.

Our taxonomy topics overlap those of Ginat et al. [6] and Izu et al. [7]. Ginat's emphasis is on selection and composition of high-level design patterns. Task decomposition and code composition are one of the aspects in our taxonomy, but we frame the progression differently. For example, Ginat's unistructural level captures solutions that translate a specification into a straightforward use of a single design pattern. Ours instead assumes that each problem could have multiple tasks, and calls a solution unistructural if it maps some single subtask into code (potentially ignoring other subtasks). By the relational level, Ginat expects students to compose plans through interleaving, whereas our taxonomy expects some composition of code for separate tasks. In other writing, Ginat has expressed a preference for interleaved code on the grounds of efficiency [11]. Our curriculum, in contrast, does not emphasize efficiency, but teaches students to think about readability and maintainability as code is adapted to different contexts. Thus our two projects have different values regarding composition skills, which reflect in the different definitions of our SOLO levels.

Izu et al. (who respond to Ginat et al.) focused on code design rather than pattern selection and integration [7]. Their taxonomy is in terms of combining building blocks, which may or may not arise from previously-learned general patterns. Our taxonomy has a strand on writing functions, and like Izu et al., we look at how students combine code fragments into solutions. Section 5.2 shows, however, that our taxonomy shifts from syntactic to semantic understanding at the boundary of the multistructural and relational levels, whereas Izu's remains syntactic. Izu et al. also refer to solving "the given task", whereas for us, identifying individual tasks in the first place (and mapping them to code) is a key learning objective. In general, we believe that having a multi-strand taxonomy lets us explore inter-related design skills of decomposing problems, mapping subtasks to code, and composing code. Both Ginat and Izu conflate or fix some of these issues. In addition, neither considers testing, which we consider an equally-important design skill.

Thompson's SOLO taxonomy for grading programming assignments (which was also used to design exam questions) captures multiple components of activity, but he weaves these into a single strand (that requires certain behaviors from each component) [20]. For example, the multistructural level captured students who were "making the standard in more than one aspect of the project", where aspects included code not crashing, code meeting a baseline of required features, and following programming and user-interface standards. We prefer our model of multiple strands, as it lets us capture when students have made different levels of progress on different components (whereas Thompson effectively reduces a students' level to the least level across all components). We are not aware of other works that define a family of inter-related SOLO taxonomies, each tied to the same underlying curriculum.

Some of our interview questions were inspired by work on retrieval of program schemas [12, 19], plan composition [18], and novice planning behavior [15, 16]. While these works do not suggest progressions within design practice, they do suggest factors that may influence program design. Our SOLO strand on task decomposition, in particular, reflects insights from this work.

3 COURSE: *HOW TO DESIGN PROGRAMS*

How to Design Programs (henceforth HTDP) is an introductory computing curriculum that has been adopted in higher education institutions and some K-12 programs [1, 5, 13]. HTDP uses a unique pedagogy for teaching program design through a multi-step process (called the *design recipe*) for designing programs based on the structure of the input data [4]. Given a programming problem, students are taught to work through a progression of steps:

(1) Identify and define the structure of the input data.
(2) Write (as executable code) examples of the input data.
(3) Write the name, input types, and output type (the *contract*) for a function that will solve the problem and a one-line summary of the program's goal (the *purpose*).
(4) Write (as executable code) concrete *examples* (or *test cases*) of what the program should produce on specific inputs.
(5) Write a skeleton of the body of the function that will fully traverse the input data (the *template*). The template is specific only to the type of input data, not to the computation being performed in the given problem.
(6) Fill in the template with problem-specific details.
(7) Run the function on the tests, adding tests as necessary.

Figure 1 shows a complete example of the HTDP steps. The code is written in Racket (a variant of Scheme), the language used in the HTDP textbook [4]. The curriculum is particularly well-suited to functional languages (though some instructors have adapted it to imperative settings). It emphasizes data structuring (through tuples, lists, and trees) more than variations on control flow (beyond recursion). While other curricula foster skills such as the selection of appropriate language constructs, code composition, and checking

```
; Problem: Design a function count-votes that consumes a list of names and a specific name,
; and counts how many times the specific name appears in the list of names.

;;; The Data Definition (RECIPE STEP 1)
; A list-of-string is
; - empty, or
; - (cons string list-of-string)

;;; Examples of Data (RECIPE STEP 2)
(define names (cons "pedro" (cons "pedro" (cons "ming" empty))))

;;; The Template (RECIPE STEP 5)
; Traverses the main input type, ignoring other problem-specific parameters.
;   This allows the same template to be reused across multiple functions on the same type.
; The ellipses get filled with details from the specific problem, when the student gets to step 6.
#|
(define (los-func alos)
  (cond [(empty? alos) ...]
        [(cons? alos) ... (first alos)
                      ... (los-func (rest alos)) ...]))
|#

;;; The contract and purpose (RECIPE STEP 3) and final function (RECIPE STEP 6)
; count-votes : list-of-string string -> number
; produces number of times the given string appears in the given list
(define (count-votes alos for-name)
  (cond [(empty? alos) 0]
        [(cons? alos) (cond [(string=? (first alos) for-name)
                             (+ 1 (count-votes (rest alos) for-name))]
                            [else (count-votes (rest alos) for-name)])]))

;;; Test Cases / Examples of Function (RECIPE STEP 4) -- written prior to writing code
(check-expect (count-votes empty "ann") 0)
(check-expect (count-votes names "ann") 0)
(check-expect (count-votes names "pedro") 2)
```

Figure 1: The HTDP recipe steps on a problem to count how many times a given string appears in a list. Semicolon is a comment character; vertical bars with hash signs create block comments. Racket naming conventions use hyphens to separate words, rather than camel casing. `cond` is the construct for a multi-armed `if` statement. `cons` is an operator for building lists from an element and an existing list. `check-expect` captures a test case, with both the expression to run and its expected answer.

program behavior against problem constraints (i.e. testing), these are embodied either explicitly (e.g. writing tests in *Step 4*) or implicitly (e.g. necessary construct selection and code composition in *Step 6*) in the HTDP process. We are interested in identifying what skills students draw on or manifest when using the HTDP process.

The steps alternate between thinking abstractly (data types, contracts, templates) and concretely (examples of data, examples of program behavior, and completed function code). Each step builds on at least one previous step. The recipe thus scaffolds the process of program design, while also serving as a diagnostic for instructors: if a student is struggling to write a function but can't describe the input, the student likely hasn't yet understood the question. Additionally, the multiple levels of abstraction reflected in the steps—from contracts, to examples of function behavior (tests), to the structure of the input data (templates), to the completed code, gradually provides students more detail for a program solution as they work through the process. Understanding what relationships students see between these different levels of abstraction is one of our goals in studying design process development in HTDP.

An HTDP course shows how to apply the recipe to increasingly rich data structures: it starts with programs over atomic data (numbers, strings, images), then progresses to compound data (structs/records), lists of atomic data, lists of structs, binary trees, and n-ary trees (mutual recursion). All design steps, including testing and template design, are re-iterated throughout this progression. After trees, the curriculum discusses higher-order functions (maps and filters), functions that accumulate partial results in parameters, and introduces stateful variables. Additional topics are covered in semester-length HTDP courses; the course used in this study runs on a shorter calendar, and covers only the listed topics.

4 DATA COLLECTION

Our goal was to study how students evolved in their program-design skills, as framed by the HTDP design recipe and planning literature, over the duration of the course. We were also interested in identifying underlying factors that might impact how effectively students were using the recipe.

Table 1: Topics and activities for each study session.

Session 1	
Topic	Homework on lists of structs (sum cost of ads for a political candidate)
Activities	(1) Interview on homework solution (2) Compare alternative solutions
Session 2	
Topic	Homework on n-ary trees (check oxygen levels on system of rivers)
Activities	(1) Interview on homework solution (2) Compare alternative solutions
Session 3	
Topic	Open coding - Planning (Rainfall)
Activities	(1) Think-aloud while writing code from scratch (2) Interview on Rainfall solution

4.1 Logistics

We collected data from study sessions conducted with volunteer CS1 students. Three sessions were conducted individually with each of the volunteers. In sessions 1 and 2, we used an interview protocol to draw out students' knowledge and ideas about their program design process by asking students to describe how they had approached a specific homework problem (which they had recently submitted). We also showed students an alternative solution to what they submitted for homework and asked them to discuss the differences (alternatives might move some functionality to a helper function, or use an `or` statement in place of an `if` statement that returned booleans, for example). In session 3, we gave them a problem to try writing from scratch while talking aloud. After solving the problem, we had them reflect on their work, similar to the activity in sessions 1 and 2 asking students to describe their approach (but without having them compare their work to an alternative implementation). The design of session 3 — a think-aloud followed by a retrospective interview, was partly adapted from Whalley and Kasto's design [22] to help clarify interviewer observations during the think-aloud portion in addition to the interview questions used in sessions 1 and 2. Table 1 summarizes the topic and activities done in each session.[1] Appendix A lists the questions through which we asked students to reflect on their approaches and to compare solutions.

One author conducted all of the sessions, each lasting roughly an hour. Sessions occurred every two weeks starting after the first exam (courses at WPI run at intense pace for 7 weeks). Neither author was instructor for the course. Sessions were individual for each participant. Each received USD 15 per session and an additional USD 20 upon completing the study. We audio recorded all sessions and collected students' solutions and scratch paper.

4.2 Problem Selection Rationale

When selecting problems to include in each session, we wanted problems that (a) would reflect the various design practices taught in HtDP, while (b) having at least two identifiable subtasks (so that there were plausible alternative solutions to discuss). For the first two sessions, we discussed problems that involved traversing a data structure and building up an answer based on data from each element (each list element or each tree node). In the first session, a function that computed part of the per-element data had been assigned as an earlier problem on the same homework. In the second session, students were left to consider the per-node task on their own, without scaffolding from prior problems. Rainfall [18] (session 3) was different in having multiple traversal-based subtasks (counting, summing, and eliminating some elements): the default traversal patterns students had learned did not apply naively to Rainfall, thus letting us see how students employed their design skills when decomposing richer problems.

The discussions of alternative solutions to problems in sessions 1 and 2 were designed to let us gauge what students notice about solutions: did they talk only about code, or did they see tasks and structure within code? We were curious about which additional criteria (such as efficiency, readability, or shapes of code) students might bring to their design work. Different alternatives for the same problem would cluster subtasks differently: a function to check whether any list element met a criterion, for example, could either check the criterion while traversing the list, or could extract elements that met the criterion then check whether the list is empty. Students had been exposed to functions similar to each subtask in earlier problems, so this design let us explore what they had picked up from that prior exposure.

The problems also embodied different suggestions towards testing: the first session problem looked for a specific name, which suggests covering data with and without the name; the second had three possible outputs, each of which should be covered. Rainfall had some tests implicit in the problem description (such as those involving negative numbers), but testing for Rainfall is more subtle as the position of negative numbers within the list can be important.

4.3 Participants

Participants came from a CS1 course at WPI, a selective STEM-focused university in the USA. We requested volunteers before the first exam. Interested students provided information on their intended major, whether they would take CS2 the following term, their prior programming experience, programming languages they had used, and a self-evaluation of their performance in the course so far. We separately got their first exam grades from the instructor.

From an initial pool of 15 volunteers, we recruited 13 for the study (one dropped out before the study began and another wasn't planning to take CS2—the full study extended over both courses). We had six males and seven females. In terms of first-exam performance, 6 students received a grade of A (3 male, 3 female), 3 received a B (2 male, 1 female), and 4 received a C (1 male, 3 female).

For this phase of the project, we used data from a sample of 6 students to develop the rubric for assessing students' design skills. We randomly selected 2 student volunteers from each of the first-exam grade bins (for a total of 18 session transcripts). All 6 students were freshmen (3 male, 3 female), 5 majoring in computer science and one in bioinformatics. Of the 6 randomly selected students, 1 self-reported having no programming experience prior to CS1. In terms of self-evaluation of their course performance, 1 reported understanding the topics very well (with an easy time working on

[1] Full homework questions available at: https://github.com/franciscastro/koli-2017

assignments), 4 understood the topics well enough (assignments were a bit challenging), and 1 found both the topics and course assignments fairly challenging.

5 DEVELOPING AN ANALYSIS FRAMEWORK

To assess the development of student design skills, we needed to identify which skills students draw on, based on their narratives during the sessions. Developing a rubric for scoring students' design skills was thus the main task for this first phase of the project.

5.1 Identifying Skills and Skill Progressions

After the first session, we open-coded [3, 8] the student transcripts from *Session 1* as the other sessions were ongoing. To facilitate this, we literally cut transcript printouts into phrases and iteratively used card sorting[2] to cluster student comments into themes. Given the questions that we asked students about their work (appendix A), which reflected both HTDP and planning literature, we expected certain themes to arise in students' responses (e.g., testing, working with templates, use of learned schemas). Some themes emerged independently of the curriculum, while one arose as we sought to align the emerging codes with the curriculum. Table 2 summarizes the resulting themes.

Comments on some themes suggested a *progression within a core skill* (Table 2.a) resembling increasing levels of conceptual complexity akin to SOLO levels. To capture these observed progressions, we defined a SOLO taxonomy for each of these themes by mapping the comments within each theme to a corresponding SOLO level. This produced the multi-strand SOLO taxonomy; each theme identified became a *skill* strand in the overall taxonomy. This was likewise done iteratively to help us refine the concrete definitions of each skill's SOLO level. In one case (*leverage multiple representations of functions*), a SOLO taxonomy arose more top-down, as we tried to make sense of a collection of seemingly related comments within the context of the overall curriculum. We also found other themes that had comments of varying depth, but no core skill that could give rise to a taxonomy (Table 2.b — we discuss these in section 9). Figure 2 summarizes our iterative open-coding and taxonomy development process. The actual taxonomies for SOLO-amenable themes appear in table 3; we discuss each theme in turn.

Methodical Choice of Tests and Examples

Our study questions (appendix A) asked students about their choice of test cases and examples of data. The students talked about the *kinds* of tests and data examples they were writing, as well as their *reasoning* around why they chose them. Some examples of our observations include instances when students would simply enumerate one test after another without identifying an inherent purpose for their choices. There were also instances when students *justified* why some tests and examples were interesting cases for the problem context. The progression around this skill describes the extent to which students are writing tests to cover a given problem space; for example, the possible input, output, and interesting corner cases. Here are two sample answers, both from session 1:

[2]Card sorting is a method originally employed by psychologists to study how people organize knowledge [24]; it has become a popular user-centered design method for discovering optimal organizations of information for websites or software [14, 24].

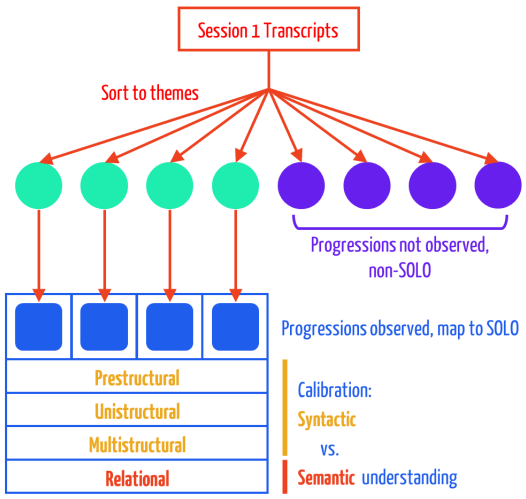

Figure 2: Process summary for developing the multi-faceted SOLO taxonomy. The entire process was done iteratively, with discussions between authors during each iteration refining the themes and descriptions for the taxonomy levels.

> ST1: *I don't think there was any specific reason [for choosing] these [tests]. Oh, one of these is political and one of these isn't. That's why. (question was about political ads)*
>
> ST6: *[This program] didn't really have any bounds, like it didn't have an if greater than, if less than [...] I did one for each condition, so if there's empty, I satisfied that with this test case. I did [a list that matches] in the first (element). I realize now I probably should've done another one where [the first list item] isn't matching the name.*

While ST6 talked about the *space of tests* in the context of the problem (seeing a collective purpose for the tests), ST1 spoke only about *individual* purposes of tests. Our progression for testing captures the depth at which students see tests collectively.

Writing and Evaluating Function Bodies

Students described how they wrote their functions in response to our question about the approach their code takes to solving the problem. The main distinction among comments lay in whether students described their code *syntactically* or with an understanding of the underlying *semantics*. Differences in semantic understanding is reflected in the following samples:

> ST1: *If [the list is not empty], then you go through the* `if` *statement and it checks to see if the ad's political. And if that's true, then it adds the cost of the ad to just the thing, the output, and it'll go back to the list and look at the next value and put it back to the beginning, and if it's not political, then it'll just keep going to the rest of the list until it reaches empty.*

ST1 fails to concretely articulate mechanisms around the helper function (extracts a boolean value from the data structure) and the results of the recursive call. Additional prompting further revealed a knowledge gap in the use of *selectors* in the student's function:

Table 2: Emergent themes from open-coding student transcripts.

(a) SOLO-amenable themes: these became the core skill strands in the multi-strand SOLO taxonomy

Theme	Description
Methodical choice of tests and examples	Knowledge of writing tests; understanding the individual/collective purposes of the tests
Writing and evaluating function bodies	Knowledge of writing functions and the composition of expressions (i.e. built-in/user-defined functions) to build function bodies (i.e. code-level perspective of programs)
Decomposing tasks and composing solutions	Knowledge of identifying tasks in a given problem, the decompositions of a program into relevant tasks, and the composition of solutions to tasks (problem-level perspective of programs)
Leverage multiple representations of functions	Knowledge of the various representations of functions and how they interact, through the components of the HTDP design recipe

(b) Non-SOLO themes

Theme	Description
Quality attributes ("ilities")	Qualities, properties, or criteria that is expected of or characterizes code or coding practice (e.g. readability and maintainability of code)
Knowledge recalled	References to knowledge used in programming; this could be course knowledge (i.e. learned from the course) or pre-course knowledge (i.e. learned prior to taking CS1)
Metacognition	References to one's cognitive processes or metacognitive behaviors such as self-regulation
Value judgments	Value judgments towards aspects of the programming process, experience, or learning

Table 3: Multi-strand SOLO taxonomy. We omit *extended abstract* as none of our students reached that level in this study.

SOLO level	Methodical choice of tests and examples	Writing and evaluating function bodies	Decomposing tasks and composing solutions	Leverage multiple representations of functions
Prestructural	Does not know how to write tests; misses the input/output structure of tests	Does not know how to define a function	Does not identify relevant tasks for a problem	Just dives in and writes code; uses only a single representation
Unistructural	Able to write tests; descriptions of tests 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; articulates 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 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 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.

ST1: *we didn't think we could pull out the one value from the [data structure] from the original function. We had to move that into a helper function, or else it wouldn't work.*

Contrast this with ST3's comment, which concretely explains how the return value of the helper function relates to the calling function:

ST3: *if we are [given] a list, then we need to process it with a helper function. [my function] checks if the politician's name is equal to the [input string]. And essentially [if it is] we want to add the cost of that politician's ad to the rest, keep it as a rolling sum. [...] it's going to add the air cost of the first [list element] to the rest of the list. And we call the function on itself, so it would go through the entire list.*

Decomposing Tasks and Composing Solutions

None of our questions directly asked about how students decomposed the problem into subtasks, but students typically commented on individual problem tasks in the context of the code. For example:

ST6: *so the first thing I did with the list of names, I run it through this program [...] which takes this name and this list and it gives me another list of ads containing only [ads that match the name]. So, that list of ads is then acted on by this [other] function [...] which takes a list of ad [...] and produces the number of the total cost of that list.*

While the narrative resembled *writing and evaluating function bodies* in discussing code, it differed in the kind of abstraction it employed; instead of a focus on *language-specific* components of a program, ST6's narrative focused on *tasks* captured around the functions, as well as the *compositions* of those tasks. The articulation of how tasks are *composed* is critical: it establishes the logical relationship between identified tasks and how tasks can be effectively put together to produce output. The alignment of tasks and code structure thus became a core skill for a SOLO taxonomy.

The following excerpt from ST2 (when comparing two solutions for a problem) shows a lower-level variation of this:

ST2: *I notice that you don't have a helper function for this one, it's just like all in one function. [...] And then you also have an* `or` *statement, but like within the* `or` *statement, you also have like a* `string=?`*, but I have a helper function for that, so I think that's like that only main difference.*

While the student described the presence of a helper function, she does not identify either an explicit task that connects to the helper functions, or an explicit purpose for the helper. This was more a sense of decomposition *for the sake of decomposition*, and less about the delegation of identified tasks to helpers.

Leverage Multiple Representations of Functions

Given the design recipe, we were not surprised to see comments on HTDP templates. At first, we were unsure what to do with them: students spoke of templates with different depth, but a unifying core skill was not immediately apparent. Only after reviewing many unclustered comments from a top-down perspective based on HTDP did a unifying skill emerge: how students worked across the multiple representations of functions inherent in the design recipe.

The design recipe steps exploit and relate multiple representations of functions: students describe a function through its input and output types (a.k.a. domain and range), samples of input/output pairs (test cases), and the symbolic code that captures the detailed implementation. Ideally, the design recipe helps students learn how to leverage these different representations, as well as the template that bridges the types and the symbolic code, to help think through how to develop a function.

Some students, such as ST3 in the excerpt below, worked through the recipe representations mechanically, while others, such as ST6, conveyed relationships among the representations:

ST3: *because it's processing a list of ad, [we used] a* `cond` *statement [...] because earlier when we defined the list of ad, we said it had to be either* `empty` *or it had to be a* `cons` *statement* (a list).

ST6: *I realize I was writing the* `check-expect`*s (tests) to satisfy the function that I wrote rather than writing the function I wrote to satisfy the* `check-expect`*s which I think sometimes you can write a bad program and then just have the* `check-expect`*s satisfy that program.*

ST6's higher-level reflection about tests *driving* function design suggests a more cohesive understanding of the knowledge and use of HTDP components. Most template-related comments thus clustered under a SOLO progression about interactions between the information from different representations. Without reflecting on the practices of the curriculum top-down, we are not convinced we would have identified this progression just from the data.

5.2 Calibrating the SOLO Levels

We later adjusted some of our SOLO-level definitions so that each skill drew a consistent boundary between syntactic and semantic understanding: syntactic understanding is at most *multistructural* within each skill; each *relational* level requires some semantic understanding of the corresponding concept. For example, in *methodical choice of tests and examples*, there is an increase in the sophistication of the mechanical application of testing from *prestructural* to *multistructural*. Initially, knowledge of how to write or use tests is absent (*prestructural*); then, at *unistructural*, there are instances of writing tests, yet no deeper understanding of the purpose of doing so (e.g. writing tests because the problem description *says* so — this shows testing merely as the idea of applying a construct without any meaningful intent); at *multistructural*, there is a recognition of the purpose of each test, but without a cohesive understanding of what the collection of tests achieve in the context of the problem. This cohesive understanding is achieved in the *relational* level, where the collection of tests and examples are understood in the context of satisfying, for example, the space of possible input-output pairs for the problem. The *relational* level for each skill establishes logical connections between the conceptual artifacts or schemas from prior levels. This distinguishes our taxonomy from others, such as Izu et al.'s [7], which does not require semantic understanding to reach a relational level. A principled alignment such as this seems an important step in developing a multi-strand SOLO taxonomy. Otherwise, separate taxonomies per strand would suffice.

6 ASSESSING THE TAXONOMY

The taxonomy in table 3 arose from our trying to make sense of *isolated* comments that students made during session 1. We did not

Table 4: Analysis of Student Skill Progressions. The abbreviated headings correspond to the 4 design skills in the taxonomy: MTE = *Methodical choice of tests and examples*, WFB = *Writing and evaluating function bodies*, DTC = *Decomposing tasks and composing solutions*, and LFR = *Leverage multiple representations of functions.*

Student	Session	MTE	WFB	DTC	LFR
ST1	1	U	M	U	U
	2	R	R	M	U
	3	U	M	M	U
ST2	1	R	M	U	M
	2	M	M	-	U
	3	U	R	M	M
ST3	1	R	R	R	R
	2	R	R	R	-
	3	R	R	R	R
ST4	1	M	R	M	M
	2	R	R	U	M
	3	M	R	M	M
ST5	1	R	M	U	U
	2	R	R	R	U
	3	R	R	R	M
ST6	1	R	R	R	R
	2	M	R	R	M
	3	R	R	R	M

look at transcripts from sessions after the first one while developing the taxonomy. We had also considered data from only 6 of the 13 students when developing the taxonomy. These raise a key question about the applicability of the taxonomy:

> *Does every session transcript from each student yield a meaningful SOLO rating in each of the taxonomy strands?*

This question captures one form of validity for our taxonomy. Our study protocol asked students about their approach to testing, so we expected every transcript to address testing. The other three strands, however, were not directly discussed, meaning that there was the potential for students to omit discussing those issues.

Table 4 shows the results of applying the taxonomy to the 18 transcripts in the original sample (6 students, 3 sessions each). The letters in each cell refer to SOLO levels ([**P**]restructural, [**U**]nistructural, [**M**]ultistructural, [**R**]elational); a dash (-) means the student never mentioned that strand. The three-letter column headings abbreviate the separate skill strands of the taxonomy from table 3.

Each author coded the 18 transcripts individually. We then discussed all of the table entries in detail, refining our interpretation of the SOLO levels as needed. As we discussed all scorings as a team, we did not compute inter-coder reliability. When a student made comments at different SOLO levels for the same skill strand (for a single session), we made a holistic judgment about the student's level, weighing frequency and depth of the comments at each level.

We also checked the taxonomy against the transcripts from the remaining 7 study participants; we omit data on these for sake of space, but the taxonomy applied similarly to those transcripts.

6.1 Assessing Our Multi-Strand Approach

Reflecting on the table—both its immediately visible patterns and our interpretations of those patterns—yielded observations about using multi-strand taxonomies to track design skills.

Observation 1. *Students can be at different levels for different skills at a given time.*

While the design skills are interrelated, our analysis suggests that students do not necessarily progress through them simultaneously. For example, ST1 exhibits knowledge of *decomposing tasks and composing solutions* at the *multistructural* level by session 3, but still struggles with relating design-recipe components. Her data suggests a mechanical use of the design recipe, without reflecting or leveraging recipe components to inform the design of her programs.

This is part of the argument supporting a multidimensional taxonomy: students improve in some skills while staying flat in others. Our taxonomy gives us a much more nuanced reading than previous taxonomies would that conflate multiple aspects.

Observation 2. *Skill strands vary in the nature of mechanical application and requirement of abstract-level thinking.*

Writing and evaluating function bodies (**WFB**) is the only strand in which no student was ever at the unistructural level. There are several plausible explanations for this. By the time we started interviews (week 3 of the course), students had already taken (and passed) the first exam, which covered programming over lists of atomic data. Correct solutions to both the exam and the homework that students completed prior to the first session would have required code that satisfied the *multistructural* criteria.

One can also argue that *writing and evaluating function bodies* is the most mechanical of the design skills, at least up through the *multistructural* level. Assuming cognitive theories about copying code schemas are correct [15, 16], then a student achieves multistructural performance simply by retrieving and reproducing an applicable schema (perhaps with the help of documentation or APIs). This requires less thought about the specific problem than does thinking about test coverage or task decomposition, and less synthesis about the design process than the *multiple-function-representations* strand. In *decomposing tasks and composing solutions*, for example, *multistructural* requires seeing features of a problem in "chunks" that manifest in code: this cannot be achieved by simple recall.

The progression from multistructural to relational in *writing and evaluating function bodies* does have some depth, as students must shift from working with nested expressions syntactically to doing so with semantic understanding. This goes beyond recall and reproduction of code patterns, and hence requires some real understanding. But shifts at the earlier levels don't seem to *require* more from the student than having learned richer code patterns to copy. A similar criticism applies to Izu et al.'s taxonomy [7].

One possible takeaway from this is that we (as a research community) should articulate the actual (cognitive) skills that underlie our progressions, and make sure new skills are actually required to progress through levels. Another is that we need to use research protocols that look beyond students' final solutions to include their thought processes. While we can accurately determine failure to achieve a higher level through solutions alone, evidence that witnesses a level can be more elusive with solutions alone.

7 ASSESSING STUDENTS' DESIGN PROGRESSION WITH THE TAXONOMY

We developed this taxonomy as part of a larger project to study how students' design skills evolve over a (sequence of) courses. We envision two broad uses of this taxonomy to this end:

- Fix problems that students will attempt at multiple points in a course, apply the taxonomy to gauge students' levels at each point, then check whether there is a linear progression (or at least no regression) in student skills over time.
- Give a sequence of increasingly difficult problems across the course, apply the taxonomy to gauge students' levels at each point, then examine whether students can scale their skills to new problems, or whether their skills break down at a certain problem complexity.

The study in which we gathered our data (section 4) was of the second type. We can thus examine Table 4 for insights into how students' skills evolved across our study problems.

Table 4 shows that students do sometimes lose ground in later sessions. There are several plausible reasons for this. Students may not have internalized the skills they seemed to display in an earlier session. For example, a student might have described test cases as covering a problem space in one week without explicitly internalizing this as good practice, so such comments don't arise in a later session. Finally, the study problems themselves (not to mention the interview questions) might bias students towards answers that appear to justify a level. This last concern is important enough to warrant its own discussion (section 8).

8 DESIGNING PROBLEM PROGRESSIONS AROUND THE TAXONOMY

Reflecting on the data in Table 4 illustrated ways in which our selection of study problems could impact where students end up on the taxonomy. For example, in session 1 we had students discuss a problem for which an earlier problem provided a useful helper function. This may have prompted some students to make comments that rated higher on *task-decomposition* than had students solved the problem unscaffolded (though we note from our data that some students were still unistructural despite this scaffolding).

Testing is another interesting case: several students received a lower testing rating in session 3 (open-ended Rainfall) than in session 2 (a graded homework). Testing is a significant factor in homework grades, leading students to include it in homework solutions. The lack of discussion of testing in the open-ended session, however, suggests that some students do not yet see testing as part of their design process, even though they can write good tests when asked (based on session 2). Put differently, students may have *skill* with a design technique, but not the *inclination* to apply that skill. Our taxonomy conflates these issues, leaving it to assessment designers to create problem sets that tease apart these differences.

Overall, the data in Table 4 humbled us about the subtleties of designing sequences of problems that would allow us to draw conclusions about students' design progress using our (or we suspect others') taxonomy. Problem statements should be reviewed for bias relative to taxonomy levels: do aspects of the problems steer students towards particular levels? Do other questions remove this bias, to help the instructor gain a clearer assessment of the students' skill level? The issue of designing problems that lend towards particular SOLO levels has been raised in other SOLO papers [9, 21], though these works mostly focused on categorizing students' code responses and don't tease out the more specific skills that drive students' development of their code.

9 DISCUSSION AND FUTURE WORK

Our work to date has yielded two artifacts: (a) a multi-strand SOLO taxonomy capturing different performance levels within a set of design skills, and (b) a collection of factors that students raise when discussing designs. The idea of a multi-strand taxonomy is one of the main contributions of this paper. A multi-strand taxonomy is valuable because it captures inter-related nuances while respecting that different skills develop in different ways. Exploring how and when a curriculum prepares students to work at the various levels of each skill strand drives home these nuances. A student *could* perform at a relational level in testing from very early in a course (even simple programs over numbers can have interesting boundary conditions), whereas the relational level in task decomposition requires more complex (multi-task) problems that would appear only later in a course. Contrasting when students *can* versus *do* achieve various SOLO levels will be part of the student performance analysis we are doing in ongoing work.

A multi-strand taxonomy needs to align or relate the strands in some way, otherwise it is no more than a collection of independent taxonomies organized into a table. This paper discussed one factor for aligning strands: all of our relational levels require students to display some understanding of the *semantics* underlying the corresponding strand concept, rather than just working with the skill *syntactically*. Given that this taxonomy deals with producing programs, syntax versus semantics is a useful concept around which to align strands. We suspect there are similar opportunities to align strands based on *cognitive* factors, though we are not yet sure what those would look like for program design.

Our work also raises questions about how to use a SOLO taxonomy to assess progress over a longer period than a single assessment (prior SOLO papers report only on single assessments). While one could give (essentially) the same problem multiple times and see whether students achieve higher performance levels, in our overall study, the problems we give the students either rise in complexity or remove some of the scaffolds present in earlier problems. Under this model, drops in SOLO level from one problem to another highlight the limits of students' skills. We suspect that some of the drops observed in our data reflect which design skills students have internalized, while others reflect the problem complexity at which students can apply the skills. We will continue to explore the meaning of drops in levels as we apply the taxonomy to more data.

Our SOLO taxonomy largely emerged from the data we gathered in the first session of our study, as we tried to organize and code comments from students' design interviews (we filled in some gaps based on our understanding of HTDP). Building our taxonomy from student data fundamentally makes our taxonomy descriptive rather than normative. The described progressions are not a prescriptive standard around how program design skills *should* evolve, however, it provides a framework with which to (1) evaluate *how* students

evolve in the identified skills in practice, (2) construct assessments that witness to various skill levels, and (3) evaluate curricula that teach these skills (while the taxonomy is influenced by HtDP, the skills identified are certainly not limited to HtDP or curricula that use functional programming). We have begun to validate the taxonomy, reporting here on the results of using it to categorize data from students beyond those from whose comments we derived the taxonomy. Another important form of validation will involve expert assessment: we plan to get experienced HtDP instructors from other institutions to rate the session transcripts without showing them the taxonomy, checking whether our taxonomy differentiates students in similar ways to human graders.

Finally, we want to account for issues that students raised during interviews which did not lend themselves to SOLO-esque progressions. Table 2.b summarizes these issues, which include concepts such as readability, efficiency, and value judgments about the design techniques covered in the course. Some of these issues could affect which SOLO level students demonstrate in some of our skill strands (a student who has a negative perception of testing, for example, seems less likely to take testing seriously enough to demonstrate a higher SOLO level on open-ended assessments). We expect that these non-SOLO factors will be important as we look to interpret drops in demonstrated skill levels across multiple assessments.

A INTERVIEW QUESTIONS

(Wording has been truncated slightly for space)
Code-writing exercises:

(1) Was the problem statement clear to you when you read it?
(2) Why did you choose your test cases? Do you think you've covered all possible scenarios with your tests?
(3) What did you think of doing first? Were you reminded of a construct in general or a general structure of solution that you thought would be useful?
(4) Have you previously seen problems that resemble this one?
(5) Did you feel stuck at any point while working on this problem?
(6) Describe the approach that your code takes to solving the problem. [If they just read the code, re-prompt]
(7) Were the program design techniques taught in class helpful to you when solving this problem?
(8) Did you use design techniques that weren't taught in class?
(9) Are there any constructs/commands of the programming language that you find difficult or confusing to use?
(10) What issues make programming constructs difficult to use - for example, the keyword used, the syntax, the examples given in class that uses it, the documentation for the construct, [etc]?

Solution comparison:

(1) What differences do you notice between the solutions?
(2) Identify strengths and weaknesses in each of the solutions.
(3) Given these solutions, which of these do you prefer and why?
(4) Is there a solution you find confusing or hard to understand?

ACKNOWLEDGMENTS

Kayla DesPortes and Sebastian Dziallas offered valuable advice on qualitative methods and analysis. Work supported by US National Science Foundation grant nos. 1116539 and 1500039.

REFERENCES

[1] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. 2008. HtDP and DMdA in the Battlefield: A Case Study in First-year Programming Instruction. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education (FDPE '08)*. ACM, 1–12.
[2] J. B. Biggs and K. Collis. 1982. *Evaluating the Quality of Learning: the SOLO taxonomy*. Academic Press.
[3] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. SAGE.
[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2001. *How to Design Programs*. MIT Press. http://www.htdp.org/
[5] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The TeachScheme! Project: Computing and Programming for Every Student. *Computer Science Education* 14, 1 (Jan. 2004), 55–77.
[6] David Ginat and Eti Menashe. 2015. SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE '15)*. ACM, 452–457.
[7] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 2016. A Study of Code Design Skills in Novice Programmers Using the SOLO Taxonomy. In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16)*. ACM, 251–259.
[8] 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.
[9] Raymond Lister, Tony Clear, Simon, Dennis J. Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, and Errol Thompson. 2010. Naturally Occurring Data As Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *SIGCSE Bull.* 41, 4 (Jan. 2010), 156–173.
[10] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. 2006. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education (ITICSE '06)*. ACM, 118–122.
[11] 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.
[12] 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).
[13] Norman Ramsey. 2014. On Teaching *How to Design Programs*: Observations from a Newcomer. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, 153–166.
[14] Carol Righi, Janice James, Michael Beasley, Donald L. Day, Jean E. Fox, Jennifer Gieber, Chris Howe, and Laconya Ruby. 2013. Card Sort Analysis Best Practices. *J. Usability Studies* 8, 3 (May 2013), 69–89.
[15] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* (1989).
[16] 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.
[17] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. 2009. A Taxonomic Study of Novice Programming Summative Assessment. In *Proceedings of the Australasian Computing Education Conference (ACE '09)*. Australian Computer Society, Inc., 147–156.
[18] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sept. 1986), 850–858.
[19] James C. Spohrer and Elliot Soloway. 1989. Simulating Student Programmers. In *Proceedings of IJCAI*.
[20] Errol Thompson. 2007. Holistic Assessment Criteria: Applying SOLO to Programming Projects. In *Proceedings of the Australasian Computing Education Conference (ACE '07)*. Australian Computer Society, Inc., 155–162.
[21] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. 2011. Salient Elements in Novice Solutions to Code Writing Problems. In *Proceedings of the Australasian Computing Education Conference (ACE '11)*. Australian Computer Society, Inc., 37–46.
[22] 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.
[23] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. 2006. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the Australasian Computing Education Conference (ACE '06)*. Australian Computer Society, Inc., 243–252.
[24] Jed R. Wood and Larry E. Wood. 2008. Card Sorting: Current Practices and Beyond. *J. Usability Studies* 4, 1 (Nov. 2008), 1–6.