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

Francisco Enrique Vicente Gorosin Castro

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Computer Science

May 2020

APPROVED:

Professor Kathryn Fisler
Primary Advisor
Worcester Polytechnic Institute and Brown University

Professor Daniel Dougherty
Committee Member
Worcester Polytechnic Institute

Professor Jacob Whitehill
Committee Member
Worcester Polytechnic Institute

Professor Brian Dorn
Committee Member
University of Nebraska Omaha

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

Francisco Enrique Vicente Gorosin Castro

A Dissertation
submitted for the degree of
Doctor of Philosophy
in
Computer Science
at the
Worcester Polytechnic Institute
May 2020

Abstract

Studies assessing novice programming proficiency have often found that many students coming out of introductory-level programming courses still struggle with programming. To address this, some researchers have attempted to find and develop ways to better help students succeed in learning to program. This dissertation research contributes to this area by studying the programming processes of students trained through a specific program design curriculum, *How to Design Programs* (HTDP). HTDP is an introductory-level curriculum for teaching program design that teaches a unique systematic process called the *design recipe* that leverages the structure of input data to design programs. The design recipe explicitly scaffolds learners through the program design process by asking students to produce intermediate artifacts that represent a given problem in different ways up to a program solution to the problem. Although HTDP is used in several higher-education institutions and some K-12 programs, *how* HTDP-trained students design programs towards problems, particularly ones with multiple task-components, has not been thoroughly studied.

The overarching goal of this dissertation is to gain an understanding and insight into how students use the techniques put forth by the design recipe towards designing solutions for programming problems. I conducted a series of exploratory user studies with HTDP-trained student cohorts from HTDP course instances across two different universities to collect and analyze students' programming process data *in situ*. I synthesized findings from each study towards an overall *conceptual framework*, which serves as a data-grounded *theory* that captures several facets of HTDP-trained students' program design process. The main contribution of this work is this theory, which describes: (1) the program design-related skills that students used and the levels of complexity at which they applied these skills, (2) how students' use of design skills evolve during a course, (3) the interactions between program design skills and course contexts that influenced how students applied their skills, and (4) the programming process patterns by which students approached the programming problems we gave and how these approaches relate towards students' success with the problems. Using insights from the theory, I describe recommendations toward pedagogical practices for teaching HTDP-based courses, as well as broader reflections towards teaching introductory CS.

Publications included in this thesis

1. [22] **Francisco Enrique Vicente Castro** and Kathi Fisler. 2020. Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), Association for Computing Machinery, Portland, OR, USA, 487–493. DOI: https://doi.org/10.1145/3328778.3366847

2. [20] **Francisco Enrique Vicente Castro** and Kathi Fisler. 2017. Designing a Multi-faceted SOLO Taxonomy to Track Program Design Skills Through an Entire Course. In Proceedings of the 17th Koli Calling Conference on Computing Education Research (Koli Calling '17), ACM, New York, NY, USA, 10–19. DOI: https://doi.org/10.1145/3141880.3141891 (**Best Paper Award**)

3. [23] **Francisco Enrique Vicente Castro**, Shriram Krishnamurthi, and Kathi Fisler. 2017. The Impact of a Single Lecture on Program Plans in First-year CS. In Proceedings of the 17th Koli Calling Conference on Computing Education Research (Koli Calling '17), ACM, New York, NY, USA, 118–122. DOI: https://doi.org/10.1145/3141880.3141897

4. [52] Kathi Fisler and **Francisco Enrique Vicente Castro**. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17), ACM, New York, NY, USA, 12–20. DOI: https://doi.org/10.1145/3105726.3106183

5. [19] **Francisco Enrique Vicente Castro** and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16), ACM, New York, NY, USA, 205–210. DOI: https://doi.org/10.1145/2839509.2844574

Other publications and manuscripts during candidature

Published

1. [21] **Francisco Enrique Vicente Castro** and Kathi Fisler. 2019. Balancing Act: A Theory on the Interactions Between High-Level Task-thinking and Low-Level Implementation-thinking of Novice Programmers. In Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19), ACM, New York, NY, USA, 295–295. DOI: https://doi.org/10.1145/3291279.3341204

2. [26] **Francisco Enrique Vicente G. Castro**. 2018. Towards a Theory of HtDP-based Program-Design Learning. In Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER '18), ACM, New York, NY, USA, 260–261. DOI: https://doi.org/10.1145/3230977.3231020

3. [101] Jun Rangie C. Obispo, **Francisco Enrique Vicente G. Castro**, and Ma. Mercedes T. Rodrigo. 2018. Incidence of Einstellung Effect among Programming Students and its Relationship with Achievement. In Proceedings of Information and Computing Education Conference (ICE 2018), Cebu City, Philippines.

4. [25] **Francisco Enrique Vicente G. Castro**. 2016. Pedagogy and Measurement of Program Planning Skills. In Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16), ACM, New York, NY, USA, 273–274. DOI: https://doi.org/10.1145/2960310.2960344

5. [24] **Francisco Enrique Vicente G. Castro**. 2015. Investigating Novice Programmers' Plan Composition Strategies. In Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15), ACM, New York, NY, USA, 249–250. DOI: https://doi.org/10.1145/2787622.2787735

6. [18] **Francisco Enrique Vicente Castro**, Seth Adjei, Tyler Colombo, and Neil Heffernan. 2015. Building Models to Predict Hint-or-Attempt Actions of Students. In Proceedings of the 8th International Conference on Educational Data Mining (Madrid, Spain. 26-29 June 2015). International Educational Data Mining Society, 476-479. http://www.educationaldatamining.org/EDM2015/proceedings/short476-479.pdf

Under Review

1. Matthew Micciolo, Erin Ottmar, Avery Harrison, Hannah Smith, **Francisco Castro**, and Ivon Arroyo. (Under review). The Wearable Learning Cloud Platform: A Computational Thinking Tool Supporting Game Design and Active Play. Technology, Knowledge and Learning.

2. Ivon Arroyo, Hannah Smith, Avery Harrison, **Francisco Castro**, Esther Agbaji, Richard Valente, Matthew Micciolo and Erin Ottmar. (Under review). Wearable Tutors Augment Learning in the Embodied Mathematics Classroom.

3. Luisa Perez Lacera, **Francisco Castro**, and Ivon Arroyo. (Under review). Exploring the Impact of Using an Intelligent Tutoring System for Math Problem-solving on Students' Grit.

Research involving human subjects

All research in this dissertation involving human subjects have been approved by the WPI Institutional Review Committee (IRB). All WPI IRB approval letters are in Appendix F.2.

Acknowledgments

My journey through my doctoral program was made richer by all the people who have helped me grow and persevere, given me support, and generously offered their presence for me.

I am truly grateful to my advisor, Kathi Fisler, who took a chance on me and mentored me throughout my PhD program. I have learned a great deal from your mentorship: from reading and understanding research papers insightfully, to designing my studies and experiments, to writing research papers, to communicating my research to different audiences, as well as challenging my own work. I am thankful for your patience as we worked together to analyze and make sense of the hundreds and hundreds of pages of data, transcripts, and programming code that we've had to work on. Kathi has taught me so much about what it means to do research and I am honored to have been one of her students.

My special thanks go to Kayla DesPortes and Sebastian Dziallas. I first met both of these wonderful people at ICER 2015 and they eventually became some of my closest friends in grad school and beyond. I am grateful to both of you, not only for our very valuable friendship, but also for being my mentors when I first started doing qualitative research. I am deeply grateful for the many times we talked, supported each other throughout grad school, and now for our growing friendship beyond grad school.

I am also thankful to my many colleagues in Finland, who have also become very good friends: Juho Leinonen, Arto Hellas, Otto Seppälä, Rodrigo Duran, and Antti Leinonen. I appreciate the many discussions we've had on computing education research and I am thankful for welcoming me into your communities. Thank you for showing me around Helsinki during my conference trips to Finland. Arto and Rodrigo, in particular, have always supported me in the research that I do, and I am truly thankful for the support I've always received from you.

The WPI Computer Science Department staff: Nicole Caligiuri, Refie Cane, Tricia Desmarais, John Leveillee, Mike Voorhis, and Chris Caron, deserve my many thanks. They have helped me in so many ways in navigating the ins and outs of the department: from technical support, to reimbursements, to room reservations for my research studies, to the administrative gymnastics of funding. I appreciate all the generous help you've given me during my time at WPI.

Throughout my time in my doctoral program, I've moved around so many labs and met so many people who have become good friends. I'm thankful to my Fuller 314 labmates Han Jiang, Anand Ramakrishnan, Arkar Min Aung, Zeqian Li, and Ashvini Varatharaj. I always appreciated your company in the lab, the many times we spent making fun of each other during research breaks, complaining (about other things), our dinner nights, and especially, your support.

Thank you to my Learning Sciences and Technologies colleagues: Ivon Arroyo, Avery Harrison, Hannah Smith, Olivia Bogs, Luisa Perez Lacera, Erin Ottmar, and Naomi Wixon. Thank you so much for welcoming me into the research group. I've always appreciated and enjoyed being part of our research meetings and our collaborations. I'm so happy to work with you all on our projects and I'm very excited about the many cool projects we have lined up!

I also thank my research group and colleagues at Brown University: Jack Wrenn, Yanyan Ren,

Preston Tunnel-Wilson, Justin Pombrio, and Tasha Danas. I appreciate all our conversations, both about our research and outside of research, as well as being my support group while at Brown.

I'm so happy to have become a part of the Fossil Lab in my last year at WPI. The members of Fossil have been a continuous source of support, inspiration, and entertainment. Thank you to Sam Ogden, Jean-Baptiste Truong, ML Tlachac, Tongwei Ren, and Julian Lanson for your constant support, discussions over absolutely random things (sometimes research things, sometimes over drinks), and game nights. It is always refreshing and fun to hang out with all of you.

Several other colleagues have made my stay at WPI very memorable. My time as a member of the WPI CS Department Graduate Student Council with Marc Green has been awesome. Thank you, Marc, for all our council work together and I'm glad we've become good friends even beyond grad school. A big thanks also to Seth Adjei, with whom I wrote my first first-author paper as a doctoral student at WPI. You taught me so much about a field I knew little of back then, and we even got to publish a paper out of our ideas. Gillian Smith has been a very solid supporter and I truly appreciate all of our talks together. Thank you to my statistics professor, Michael Johnson, who has always generously shared with me his class material whenever I needed a refresher on my stats and who always encouraged me to move forward and finish my degree. I had a great time learning from Janice Gobert in our educational psychology class. Janice's work always leaves me inspired and in awe.

I am grateful to Glynis Hamel who patiently accommodated all my repeated requests for study participants from her classes, and for answering all my questions about the CS1 courses at WPI. Thank you as well to Alan Mislove, who generously helped me with logistics when I needed to conduct my research studies at Northeastern University. I hope to someday meet and thank you in person for this generosity. Thank you to my dissertation committee members, Brian Dorn, Dan Dougherty, and Jake Whitehill, for your input and advice on my work. I appreciate the advice and perspectives you brought to the table, which has helped me improve my work.

My friends and colleagues in the Philippines have also been a great source of support. Thank you to Didith Rodrigo for jump-starting my involvement in computing education research and for introducing me to Kathi. I've always been proud to say that I've worked with you and your work continues to inspire me to be an even better scientist. My friend Sarah Mercado has cheered for me throughout this process and I am grateful for the support, even across time zones.

This dissertation would not have been possible without the generous support of my family. My mom, Melanie Gorosin, always celebrated my successes from across the world and I am always grateful for her support. Lauren, Dave, Tyler, and Kristin have always believed in me and cheered me on. Additionally, Lauren's gift of a computer monitor for my dual-monitor setup at home has been instrumental in helping with my research and dissertation productivity. Finally, no amount of thanks can ever express how grateful I am for my husband, Jeffrey Castro-Norwood. Your unwavering support, celebration of all my successes, and continued encouragement motivated me throughout my PhD program. Thank you for your generous and loving presence throughout the whole process.

Financial support

This research was supported by the following US National Science Foundation grants:

1. Award number 1116539: SHF: Small: User Studies to Improve Novice Programming
2. Award number 1500039: SaTC-EDU: EAGER: Enhancing Cybersecurity Education through Peer Review

My presentations and attendance to the ACM International Computing Education Research Doctoral Consortiums were funded, in part, by the following travel grants:

1. ICER '18 Doctoral Consortium Travel Grant
2. ICER '16 Doctoral Consortium Travel Grant
3. ICER '15 Doctoral Consortium Travel Grant

My research with the WPI Advanced Learning Technologies Laboratory was supported by US National Science Foundation grant number 1917947: Developing Computational Thinking by Creating Multiplayer Physically Active Math Games.

Keywords

program design, plan composition, planning, rainfall, design recipe, solo taxonomy, cs1, qualitative methods, think-aloud, novice programmers, functional programming, computing education, cognition

To everyone who believed in me,
to everyone who had someone believe in them,
and to everyone who believed in someone's dreams and hopes,
may we all move the world towards a better future.

Contents

Abstract ii

Contents **ix**

List of Figures **xv**

List of Tables **xviii**

List of Abbreviations and Symbols **xxi**

1 Introduction **1**

1.1 How to Design Programs 2

1.2 Studying Program Design Within the HTDP Context 6

1.2.1 What Do We Know About Research on HTDP So Far? 7

1.2.2 Research Goal 9

1.2.3 Dissertation Research Questions 10

2 Related Work **13**

2.1 Models of Program Design in Introductory CS 13

2.1.1 Top-down Program Design 13

2.1.2 Program Plans 16

2.1.3 Bottom-up Program Design 16

2.1.4 Interplay of HTDP and Models of Design and Planning 17

2.2 Explicit Instruction 20

2.2.1 Explicit Instruction in non-CS contexts 20

2.2.2 Explicit Instruction in CS Pedagogy 21

2.2.3 HTDP: Explicit Design Strategy via a Step-by-step Process 23

2.3 Models of Skill Evolution in Introductory CS 23

3 Interplay of Bottom-Up and Datatype-Driven Design **27**

3.1 An Exploratory Study 28

3.2 The Adding Machine Problem 29

3.3 Study Design and Data Collection 30

3.3.1 The HTDP Course Instance . . . 30
3.3.2 Participants . . . 31
3.3.3 Logistics . . . 31
3.4 Analysis . . . 32
3.4.1 Coding the programming session videos . . . 32
3.4.2 Coding the survey responses . . . 33
3.5 Results and Interpretation . . . 33
3.5.1 Writing Focal Code After Templates . . . 33
3.5.2 Difficulties with Plan Composition . . . 34
3.5.3 Use of Advanced Techniques . . . 38
3.5.4 Findings from the Survey . . . 38
3.6 Discussion . . . 40
3.6.1 Students Work Through Core Problem Tasks . . . 40
3.6.2 Students Lacked Schemas and Struggled with Plan Compositions . . . 40
3.6.3 Students Decompose Problems On-the-fly . . . 41
3.6.4 Tension Between Rist's Focal Expansion Model and HTDP . . . 41
3.7 Status of Dissertation Research Questions . . . 42

4 Evolution of Program Design Skills 45
4.1 Building on the Adding Machine Study . . . 46
4.2 Study Design and Data Collection . . . 46
4.2.1 Study Logistics . . . 46
4.2.2 Problem Selection Rationale . . . 47
4.2.3 Participants . . . 49
4.3 Developing an Analysis Framework . . . 50
4.3.1 Identifying Skills and Skill Progressions . . . 50
4.3.2 Calibrating the SOLO Levels . . . 56
4.4 Assessing the Taxonomy With Other Student Data . . . 57
4.4.1 Assessing Our Multi-Strand Approach . . . 58
4.5 Assessing Students' Design Progression with the Taxonomy . . . 59
4.6 Designing Problem Progressions Around the Taxonomy . . . 60
4.7 Validation Study with Experienced Instructors . . . 60
4.7.1 Validation Rationale . . . 61
4.7.2 Data Collection and Logistics . . . 61
4.7.3 Analysis and Coding . . . 62
4.7.4 Validation Study: Results and Discussion . . . 65
4.8 Discussion . . . 69
4.8.1 Meaningful alignment of skills through syntax vs. semantics . . . 69
4.8.2 Using a SOLO taxonomy for longitudinal skill assessments . . . 70

4.8.3 Constructing a data-grounded theory for program design 70
4.8.4 Other factors that may affect program design 70
4.9 Status of Dissertation Research Questions 71

5 Navigating Schemas **75**
5.1 Exploring How Students Navigate Multiple Schemas 75
5.2 Study Design 76
5.2.1 Participants 76
5.2.2 Analysis: Narrative Construction 76
5.3 Programming Process Narratives 77
5.3.1 WPI2-STUD3 77
5.3.2 WPI2-STUD6 79
5.3.3 WPI2-STUD7 80
5.3.4 WPI2-STUD11 81
5.4 Analysis and Discussion 82
5.4.1 What drove students to use the accumulator pattern? 83
5.4.2 Interactions between pattern use and task-level thinking 83
5.4.3 The Complexity of the Rainfall Problem 85
5.4.4 The Need for Finer-grained Analyses of Course Contexts 86
5.5 Status of Dissertation Research Questions 87

6 Task-level and Code-level Thinking **89**
6.1 Exploring the Interplay of Task- and Code-level Thinking 90
6.2 Study Design and Data Collection 90
6.2.1 The HTDP Course Instances 90
6.2.2 Participants 91
6.2.3 Logistics 91
6.2.4 The Study Problems 93
6.3 Analysis and Discussion 94
6.3.1 Framework: Task- and Code-level Thinking 94
6.3.2 RQ1: Movements Between Tasks and Code 97
6.3.3 RQ2: Success on Programming Problems 98
6.3.4 RQ 3: How Did Students Get Unstuck? 101
6.3.5 Analyzing the Course Contexts 104
6.4 Insights and Takeaways 111
6.4.1 Insights on Teaching the Design Recipe 111
6.4.2 Insights on Task- and Code-level Thinking 111
6.5 Status of Dissertation Research Questions 112

7 Exploratory Study on Planning **117**

7.1 Study Context: Cognitive Foundations of Planning . . . 118
7.2 Study Design . . . 119
7.2.1 The Host Courses . . . 120
7.2.2 Pre-Assessment . . . 121
7.2.3 The Lecture Intervention . . . 123
7.2.4 Post-Assessment . . . 123
7.3 Analysis and Findings . . . 124
7.3.1 Coding Analysis of the Data . . . 124
7.3.2 The view from CRS-BROWNU . . . 125
7.3.3 The view from CRS-WPI . . . 127
7.3.4 Preference Ranking of Own Post Solutions . . . 129
7.3.5 Changes in Solution Structures . . . 130
7.4 Threats to Validity . . . 131
7.5 Discussion . . . 132

8 Discussion and Conclusions 135
8.1 Answering the Dissertation Research Questions . . . 136
8.1.1 Students' program design skills . . . 136
8.1.2 Interactions between program design skills . . . 138
8.1.3 Evolution of students' program design skills . . . 141
8.1.4 Approaches to solving multi-task problems . . . 142
8.2 What did we learn about teaching program design with HTDP? . . . 143
8.2.1 What did we learn about how students use HTDP? . . . 143
8.2.2 Make problem decomposition an explicit *early* step of the design recipe . . . 145
8.2.3 Focus on teaching *how* to use the design recipe steps . . . 147
8.3 On teaching program design in general . . . 152
8.4 Bricolage, Planning, and the HTDP Design Recipe . . . 157
8.4.1 Productive bricolage among HTDP-trained students . . . 157
8.4.2 Haphazard bricolage among HTDP-trained students . . . 158
8.4.3 Supporting productive bricolage through HTDP . . . 159
8.5 Threats to validity . . . 160
8.5.1 Improving the contextual grounding for our taxonomy design . . . 160
8.5.2 Problem context limitations . . . 161
8.5.3 Computing intercoder reliability for our skills taxonomy . . . 162
8.5.4 Mitigating short-term learning gains . . . 162
8.5.5 Teasing out workload factors of our study sessions . . . 162
8.6 Open Questions and Future Work . . . 163
8.6.1 Further validation of the SOLO-based program design skills taxonomy . . . 163
8.6.2 Using our skills taxonomy as a framework for designing assessments . . . 164

8.6.3 Student performance with new instructional activities 164
8.6.4 Impact of programming language on students' planning 165

Bibliography **167**

A Solutions to the Study Problems **181**
A.1 Rainfall: Clean-first 182
A.2 Rainfall: Process-multiple 183
A.3 Rainfall: Single-traversal 184
A.4 Max-Temps: Reshape-first 185
A.5 Max-Temps: Collect-first (accumulator-style) 186
A.6 Adding Machine: Reshape-first 187
A.7 Adding Machine: Accumulator-style 188

B HTDP and Focal Expansion Model Study: Instruments and Data **189**
B.1 SnagIt setup web page 190
B.2 Adding Machine problem statement web page 191
B.3 SnagIt video save web page 192
B.4 Code and programming video submission web page 193
B.5 Adding Machine post-programming survey 194
B.6 Post-programming survey question 1 responses 195
B.7 Post-programming survey question 2 responses 197
B.8 Post-programming survey question 3 responses 198
B.9 Post-programming survey question 4 responses 200

C Program Design Skills Evolution Study: Instruments and Data **203**
C.1 Interview Questions 204
C.1.1 Code-writing exercises and reviewing homework solutions 204
C.1.2 Solution comparison or ranking multiple solutions 204
C.1.3 Class, course content, curriculum 205
C.1.4 Programming language 205
C.2 Full Homework Problem Sets 206
C.2.1 Homework 3 problems 206
C.2.2 Homework 5 problems 207
C.3 Study Recruitment Survey 209
C.4 Student Survey Responses 210
C.5 Instructor Recruitment Survey 212
C.6 Instructor recruitment survey responses 213
C.7 Instructor worksheet 214
C.7.1 Page 1: Instructions and skill descriptions 214

C.7.2 Page 2: Skill ratings and descriptions . . . 215
C.8 Instructor worksheet responses . . . 216
C.8.1 INSTRUCTOR1 . . . 216
C.8.2 INSTRUCTOR2 . . . 217
C.8.3 INSTRUCTOR3 . . . 218
C.8.4 INSTRUCTOR4 . . . 219
C.8.5 INSTRUCTOR5 . . . 220
C.8.6 INSTRUCTOR6 . . . 221
C.8.7 INSTRUCTOR7 . . . 222

D Study on Task- and Code-level Thinking: Instruments and Data 223
D.1 WPI Study Recruitment Survey . . . 224
D.2 WPI Student Survey Responses . . . 225
D.3 NEU Study Recruitment Survey . . . 226
D.4 NEU Student Survey Responses . . . 227

E Exploratory Study on Planning: Instruments and Data 229
E.1 CRS-BROWNU Pre-Assessment Problems . . . 230
E.1.1 Programming Problems . . . 230
E.1.2 Ranking Problems . . . 231
E.2 CRS-WPI Pre-Assessment Problems . . . 235
E.3 CRS-BROWNU and CRS-WPI Post-Assessment Problems . . . 236

F Additional Files, Figures, and Entries 241
F.1 NEU Design Recipe Course Web Page . . . 242
F.2 WPI IRB Approval Letters . . . 243

List of Figures

1.1 The HTDP design recipe steps on a problem to sum a list of numbers. 5

2.1 Top-down design observed by Jeffries *et al.* [67] from experts and novices 14

3.1 Screen view for the Adding Machine problem statement 31
3.2 (a) Sample coding sequence and (b) the actual Racket program code for student WPI1-STUD1 at the 9-minute mark 33
3.3 Code from student WPI1-STUD18 interleaved function calls within one function without decomposition 35
3.4 Code from student WPI1-STUD1 pulled identified tasks into a separate function 35
3.5 Attempts to write reshaping code by students (a) WPI1-STUD6 , (b) WPI1-STUD8 , and (c) WPI1-STUD14 38

4.1 Open-coding student transcripts facilitated through thematic card sorting 51
4.2 Process summary for iteratively developing the multi-faceted SOLO taxonomy. Discussions between authors during each iteration refined the themes and descriptions for the taxonomy levels. 51
4.3 Developing a skill strand on pattern-use from instructor and student data 65

5.1 WPI2-STUD3 's final Rainfall solution 78
5.2 WPI2-STUD6 's final Rainfall solution 80
5.3 WPI2-STUD7 's final Rainfall solution 81
5.4 WPI2-STUD11 's final Rainfall solution 82

6.1 A snippet of the narrative coding for student NEU1-STUD1 96
6.2 WPI3-STUD2 's Rainfall solution 101
6.3 Problem descriptions from the first homework of the host courses 106
6.4 Homework 1 text of the host courses that remind students to apply the design recipe . . . 106
6.5 NEU Homework 7 text on writing functions for lists 107
6.6 WPI Homework 2 text on writing functions for lists of strings 107
6.7 WPI Lab 2 text on writing functions for lists 108
6.8 WPI lecture notes on defining accumulator-style functions 108

6.9 Accumulator function examples with one accumulator parameter returned in the base case that WPI students were shown in lecture . . . 109
6.10 A context-preserving accumulator function that does not return the accumulator in the base case that WPI students were shown in lecture . . . 109

7.1 Structures of *Adding Machine* solutions in CRS-BROWNU from the pre-assessment . . . 125
7.2 Structures of *Earthquake Monitor* solutions in CRS-BROWNU from the post-assessment . 126
7.3 Criteria CRS-BROWNU students cited while ranking solutions to *Rainfall* (pre), *Shopping Cart* (pre), and *Earthquake Monitor* (post) . . . 126
7.4 Structures of *Earthquake Monitor* solutions from the post-assessment . . . 128
7.5 Structures of *Data Smoothing* solutions from the post-assessment . . . 128
7.6 Comparing individual CRS-BROWNU students' *Adding Machine* structures (pre) to that of their preferred *Earthquake Monitor* solution (post) . . . 129
7.7 CRS-WPI students' preferred *Earthquake Monitor* structures (post) . . . 130

A.1 Rainfall solution using the *Clean-first* approach . . . 182
A.2 Rainfall solution using the *Process-multiple* approach . . . 183
A.3 Rainfall solution using the *Single-traversal* approach . . . 184
A.4 Max-Temps solution using the *Reshape-first* approach . . . 185
A.5 Max-Temps solution using the *Collect-first* approach (accumulator-style) . . . 186
A.6 Adding Machine solution using the *Reshape-first* approach . . . 187
A.7 Adding Machine solution using an accumulator-style approach . . . 188

B.1 Instructions for setting up SnagIt to capture programming sessions. . . . 190
B.2 Adding Machine problem statement used for the study. . . . 191
B.3 Instructions for saving the SnagIt programming video capture (SnagIt12 version). . . . 192
B.4 Instructions for submitting the Adding Machine code and SnagIt programming video. . . 193
B.5 The survey students filled out after working on the Adding Machine problem (administered through the WPI Qualtrics [64] distribution). . . . 194

C.1 The survey that volunteer students filled out to express interest in participating in the study (administered through the WPI Qualtrics [64] distribution). . . . 209
C.2 The survey that volunteer HTDP instructors filled out for participation in the study. . . . 212
C.3 Page 1 of the instructor worksheet. This page provides instructions on how to complete the worksheet and the descriptions of the skills to be scored. . . . 214
C.4 Page 2 of the instructor worksheet. Instructors fill this page in with their ratings of a student's design skills and their justification for their ratings. . . . 215

D.1 The survey that volunteer WPI students filled out to express interest in participating in study 5 (administered through the WPI Qualtrics [64] distribution). . . . 224

D.2 The survey that volunteer NEU students filled out to express interest in participating in study 5 (administered through the WPI Qualtrics [64] distribution). 226

F.1 The NEU course web page on the design recipe 242
F.2 WPI IRB approval letter for the original study 243
F.3 WPI IRB approval letter for modifications to the original study 244

List of Tables

3.1 First tasks coded by students, with location 34

3.2 Tasks students wrote code for and their location (*AM: within the Adding Machine function, HLP: within a helper function, BOTH: within both the Adding Machine function and a helper function*), with students' final course grade (*NR* is WPI's version of a non-passing grade); bottom sub-table indicate counts of task appearances in locations 36

3.3 Status of helper functions students wrote 37

4.1 Topics and activities for each study session. 47

4.2 Emergent themes from open-coding student transcripts 52

4.3 Multi-strand SOLO taxonomy of observed program design skills. We omit the *Extended abstract* level as none of our students reached that level for this study 53

4.4 Analysis of Student Skill Progressions. The abbreviated headings correspond to the 4 design skills in the taxonomy: MTE = *Methodical choice of tests and examples*, CFB = *Composing expressions within function bodies*, DTC = *Decomposing tasks and composing solutions*, and LRF = *Leveraging multiple representations of functions*. 57

4.5 Students assigned to each instructor; 2 students are assigned to each instructor, indicated by check-marks 62

4.6 Analysis of instructor responses 64

4.7 The SOLO taxonomy for the skill strand on the *Meaningful Use of Patterns* 66

4.8 Re-coded analysis of instructor responses with the MUP skill 67

5.1 Participant overview. The first exam was in course week 3. 77

6.1 Topic sequences for the host courses in this study 92

6.2 WPI students implementing solution approaches for Rainfall and Max-Temps, grouped by task–code movement patterns. [C/F] indicates whether students' final code were [C]lose (minor errors on some tasks) or [F]ar from a correct solution (missing tasks, major implementation errors). 99

6.3 NEU students implementing solution approaches for Rainfall and Max-Temps, grouped by task–code movement patterns. [C/F] indicates whether students' final code were [C]lose (minor errors on some tasks) or [F]ar from a correct solution (missing tasks, major implementation errors). 100

7.1 Student populations in the study 121
7.2 Criteria raised by CRS-BROWNU students across pre- and post-assessment rankings . . . 127

B.1 Student responses (verbatim) to post-programming survey question 1. 195
B.2 Student responses (verbatim) to post-programming survey question 2. 197
B.3 Student responses (verbatim) to post-programming survey question 3. 198
B.4 Student responses (verbatim) to post-programming survey question 4. 200

C.1 Student responses to the participant recruitment survey (verbatim) 210
C.2 Validation study recruitment survey responses of the 7 participating HTDP instructors . . 213

D.1 WPI student responses to the participant recruitment survey (verbatim) 225
D.2 NEU student responses to the participant recruitment survey (verbatim) 227

List of Abbreviations and Symbols

Abbreviations	
HTDP	How to Design Programs
SOLO	Structure of Observed Learning Outcomes
WPI	Worcester Polytechnic Institute
NEU	Northeastern University
DRQ	Dissertation Research Question
STUDY-RQ	Research question (specific to a study)
MTE	Methodical choice of Tests and Examples
CFB	Composing expressions within Function Bodies
DTC	Decomposing Tasks and Composing solutions
LRF	Leveraging Multiple Representations of Functions
MUP	Meaningful Use of Patterns

Chapter 1

Introduction

Problems in introductory computing courses are well-documented, ranging from high attrition rates to students coming out of introductory courses with inadequate programming proficiency as shown by generally dismal performance in assessments [84, 89, 121]. Researchers have found various factors that contribute to these problems: some students finish their introductory courses without even at least a mastery of the language constructs they've learned [89]; some concentrate on (code) implementation activities and skip planning or design activities when solving programming problems [84, 89]; a pervading notion that "CS is hard", leading to lack of interest and motivation [70, 71]; the experience of students of CS classroom environments as unwelcoming, leading to feelings of isolation and disconnect from the domain [54, 71]; and ineffective learning contexts due to curricula or tools that focus heavily on language constructs and lack explicit instruction of program design techniques and strategies [35, 50, 127], to name some . Program design (among others) has been, and continues to be, a significant challenge in computing education [73, 90, 122, 126], and researchers continue to study ways to create learning contexts that utilize effective methods, tools, and assessments that improve learning, engagement, diversity, and instruction.

Many computing education researchers and practitioners study novice programmers and introductory-level computing courses, with the goal of finding and developing ways to better help students succeed in learning programming. Some of these efforts are focused towards designing tools to aid in teaching programming [76, 109], studying motivational constructs such as self-efficacy and their links to success in learning to program [33, 83, 111], understanding how to improve retention rates in introductory-CS courses [61, 70, 74, 104], or designing activities to use in programming classes [65], among others. A smaller subset of researchers study how students *design* programs, or how to effectively teach students *program design*[1]. Researchers in this subset have studied ways to explicitly teach program design techniques and how explicit instruction affects student programming performance. This is done often by augmenting or redesigning curricula or programming activities with additional material on design techniques. For example, de Raadt studied how to design a curriculum to include the explicit instruction of programming strategies for introductory programming courses [35]. Keen and

[1] We say *program design* to mean a *systematic* approach to creating programs through *planning* [48].

Mammen [68] report on the outcomes of explicitly teaching students program decomposition through a course-length programming project. Some work discuss introducing students to design patterns in courses teaching with an object-oriented approach [63, 113, 143]. Other work in this area touch on other aspects of program design, such as tools that enhance error messages to aid debugging [8], or programming environments that constrain program structuring to a puzzle-like mechanism to do away with syntax errors and focus on semantic logic, as in the case of work on blocks-based languages [138]. My work differs from these in that I study how students use a specific design process, the HTDP *design recipe*, that involves the systematic use of a specific set of design techniques. I discuss the design recipe and the design techniques involved in the following subsection.

1.1 How to Design Programs

How to Design Programs (henceforth HTDP) is an introductory computing curriculum [48] that has been adopted in higher education institutions and some K-12 programs [12, 50, 112]. 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 [48].

Given a programming problem, students are taught to work through a progression of steps:

1. **Data definitions:** Identify and define the structure of the input data.

2. **Examples of Data:** Write examples of the input data (as executable code).

3. **Signature and Purpose Statement:** Write the name, input types, and output type (the *signature* or *contract*) for a function that will solve the problem and a brief summary of the function's goal (the *purpose statement*).

4. **Input–Output Examples:** Write concrete examples (as executable code) of what the program should produce on specific inputs, written previously in step 2 (the *test cases*).

5. **Function Template:** Using the data definitions as a reference, write a skeleton of the function body (the *template*) that fully traverses the input data. The template is specific only to the *type* of the input data, not to the computations within a given problem, allowing the same template to be reused across multiple functions on the same type.

6. **Function Definition:** Fill in the template with problem-specific details.

7. **Testing:** Run the function on the test cases, adding tests as necessary and refining the function(s).

We illustrate the use of the HTDP design recipe steps in the following example. The code in the example is written in Racket (a variant of Scheme). In the code snippets, semicolons (;) denote single-line comments and hash signs with vertical bars (`#| ... |#`) denote block comments. Racket naming conventions use hyphens to separate words (e.g. `function-for-something`) rather than camel-casing

(e.g. `functionForSomething`).

Example Problem: Design a function `sum-nums` to sum a list of numbers. If the input list is empty, simply return 0.

Recipe step 1 | Data definitions: *Identify and define the structure of the input data.*

The input in this example is a list of numbers that can either be: (1) an empty list or (2) a non-empty list composed of a number and a list of numbers. Here, `cons` is an operator for building lists from an element and an existing list. Note that the data definition shows the recursive structure of the list input data (in the non-empty list part).

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

Recipe step 2 | Examples of Data: *Write examples of the input data (as executable code).*

Here, students write concrete examples of the input data. Students are taught to think about different examples of input for the specific problem domain, as well as to leverage the structure of the input data defined in Step 1 when writing examples (*e.g. What does an input of an empty list look like? What would different instances of non-empty lists look like?*).

```
(define no-element empty)
(define one-element (cons 8 empty))
(define even-nums (cons 12 (cons 4 (cons 6 empty))))
(define odd-nums (cons 11 (cons 5 (cons 3 (cons 7 empty)))))
```

Recipe step 3 | Signature and Purpose Statement: *Write the name, input types, and output type (the signature or contract) for a function that will solve the problem and a brief summary of the function's goal (the purpose statement).*

Students begin to articulate a proposed program or function by *naming* the function and specifying its classes of input data and its class of output data through the *type signature/contract*. The *purpose statement* is a brief description of the program's goal. The central idea in this step is that it invites students to summarize the key high-level details from the given problem statement.

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

Recipe step 4 | Input–Output Examples: *Write concrete examples (as executable test cases) of what the program should produce on specific inputs, written previously in step 2.*

Students write examples of function calls on specific inputs and their respective outputs, building on the input data examples they have written in step 2, as well as using the type signature they described in step 3 as a guide when identifying the input and output types for the examples. In the following code snippet, `check-expect` captures a test case, with both the expression to run and its expected answer.

```
(check-expect (sum-nums no-element) 0)
(check-expect (sum-nums one-element) 8)
(check-expect (sum-nums even-nums) 22)
(check-expect (sum-nums odd-nums) 26)
```

Recipe step 5 | Function Template: *Using the data definition(s) as a reference, write a skeleton of the function body (the template) that fully traverses the input data. The template is specific only to the type of the input data, not to the computations within a given problem, allowing the same template to be reused across multiple functions on the same type.*

Here, students begin to write their program by first writing a function template for the program. A template is essentially skeleton code that provides a traversal of the input and reflects the *shape* of the input. This example illustrates the use of a template for list-type data. In the following code snippet, `cond` is the construct for a multi-armed `if`-statement and the ellipses are "holes" in the template which get filled in later with problem-specific details.

```
#|
(define (list-function list-input)
  (cond [(empty? list-input) ... ]
        [(cons? list-input) ... (first list-input)
                                (list-function (rest list-input)) ... ]))
|#
```

In this step, students write a template for any combination of tupled and recursive data. This specific example illustrates the use of a *list template*, which has a conditional that checks whether the list is empty. If it is, the list template simply contains a hole for the function's result in that case. Otherwise, the list must have both a first element and the subsequent elements (the `rest` or the *tail* of the list); the template therefore includes a recursive call on the `rest` of the list. The template has holes in place of concrete code for combining the result of processing the first element, with the result from the recursive call.

Recipe step 6 | Function Definition: *Fill in the template with problem-specific details.*

Students build on the template by replacing the template names with appropriate function names (using the signature written in step 3 as a reference) and filling in the ellipses with problem-specific operations. Here, students can use the information that they have defined,

written down, or articulated from prior steps, as well as retrieve functions and operations previously-learned from prior lessons, to guide them in selecting and combining appropriate operations to carry out the necessary problem computations.

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

Recipe step 7 | Testing: *Run the function on the test cases (Step 4), adding tests as necessary and refining the function(s).*

The last step invites students to iteratively refine their program to debug potential errors encountered in the development process and to also examine where their program may fail, by adding test cases that may cover interesting cases and refining their program further to cover new cases discovered.

The worked example for the HTDP steps to solve the problem of summing a list of numbers is summarized in Figure 1.1, which illustrates what a typical submission from an HTDP-trained student might look like.

```
; STEP 1: DATA DEFINITION
; A list-of-number
; - empty
; - (cons number list-of-number)

; STEP 2: EXAMPLES OF DATA
(define one-element (cons 8 empty))
(define even-nums (cons 12 (cons 4 (cons 6 empty))))
(define odd-nums (cons 5 (cons 3 (cons 7 empty))))

; STEP 3: SIGNATURE AND PURPOSE STATEMENT
; sum-nums : list-of-numbers -> number
; Produces the sum of all numbers in the list

; STEP 4: INPUT-OUTPUT EXAMPLES
(check-expect (sum-nums one-element) 8)
(check-expect (sum-nums even-nums) 22)
(check-expect (sum-nums odd-nums) 26)

; STEP 5: FUNCTION TEMPLATE
#|
(define (list-function list-input)
  (cond [(empty? list-input) ... ]
        [(cons? list-input) ... (first list-input)
                            (list-function (rest list-input)) ... ]))
|#

; STEP 6: FUNCTION DEFINITION
(define (sum-nums nums-list)
  (cond [(empty? nums-list) 0]
        [(cons? nums-list) (+ (first nums-list)
                              (sum-nums (rest nums-list)))]))
```

Figure 1.1: The HTDP design recipe steps on a problem to sum a list of numbers.

As illustrated, the steps alternate between thinking abstractly (data types, contracts, templates) and concretely (examples of data, examples of program behavior, and completed function code) within the context of a problem. 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 or the output, the student likely hasn't yet understood the problem. In particular, the HTDP template is a programming pattern that is especially suited to problems with single tasks[2], *i.e.* a single computation applied over a single datatype; the worked example for summing a list of numbers (Figure 1.1) illustrates this. Additionally, the multiple levels of abstraction reflected in the steps—from contracts, to examples of function behavior (test cases), 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 processes in the HTDP context.

Due to its emphasis on recursively-defined data structures, 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). 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 reiterated throughout this progression. After trees, the curriculum discusses higher-order functions (*e.g.* maps and filters), functions that accumulate partial results in parameters (*i.e.* accumulators), and introduces stateful variables.

1.2 Studying Program Design Within the HTDP Context

My work is an analysis of how HTDP-trained students use the design practices put forth by the curriculum when solving programming problems. For example, an aspect I focus on is students' use of datatype-driven templates[3] and how students adapt these templates to programming problems that do not directly fit the template, such as problems with multiple task components. As we've mentioned in Section 1.1, the HTDP template is particularly suited to problems with single tasks. Problems with multiple tasks, however, need design decisions that require planning around template-appropriate functions, such as through problem decomposition, by way of multiple template instances that each handle a single problem task/computation; we illustrate this in the following code snippet with an example for the problem of averaging a list of numbers: the overall *average/divide* task is decomposed over the *sum* and *count* tasks, and each task is delegated into their own template functions (we assume returning a zero for an empty list).

[2]Problem *tasks* refer to problem components that need to be addressed in order to complete a solution; for example, a problem that asks to compute the average of a list of numbers would have the following tasks: (1) sum the list-values, (2) count the list-values, and finally (3) divide the sum by the count.

[3]In particular, the template for list-type data, as the problems we used in our studies are list-based

```
(define (average list-input)
  (cond [(empty? list-input) 0]
        [(cons? list-input) (/ (sum list-input)
                               (count list-input))]))

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

(define (count list-input)
  (cond [(empty? list-input) 0]
        [(cons? list-input) (+ 1 (count (rest list-input)))]))
```

Other ways that the tasks for average may be allocated to code (in terms of the concepts covered in HTDP-based courses) include using additional parameters (termed *accumulators*) to track task-related values (*i.e.* tracking *sum* and *count*) or by using higher-order functions like `fold`. Appendix A further illustrate the use of these constructs for an averaging problem. Students may also leverage information from the intermediate artifacts (*e.g.* data definitions, data examples, input–output examples, signatures, purpose statements) produced at various design recipe steps to help them plan their programs. For example: do students write and analyze examples to figure out the task-components of the problems they're solving? Do they use data definitions to capture the characteristics of a problem's input data? We're interested in understanding how students apply or use these design techniques in their program design process. I engage with this research problem by adapting methods from human-factors research: my data comes from semi-structured interviews, think-aloud protocols, and field observations, collected from students enrolled in HTDP-taught introductory-level (CS1[4]) courses as they solve multi-task programming problems (in one case, I use video recordings of students' work to explore their process). I analyze these data qualitatively through protocol analysis, card sorts, and grounded theory methods, for example, to capture and synthesize observations.

1.2.1 What Do We Know About Research on HTDP So Far?

In an entry for the Communications of the ACM [59], Mark Guzdial, a computing education researcher, wrote:

> *Kathi Fisler beat the Rainfall Problem in 2014. [...] But we're still not really sure why. Is it because of her curriculum, because she's using functional programming, because of the data structures she's using, or because she's teaching higher-level functions? We don't really know what makes programming so hard, and we don't yet have enough theory to explain why it works when we get it right.*

[4] An introductory-level CS course at the college level in the United States is typically termed *CS1*. All student cohorts in the studies for this dissertation are from CS1-level courses.

Guzdial was referring to Fisler's 2014 ICER[5] paper where Fisler described the high-level solution structures that HTDP-trained students produced towards the multi-task Rainfall problem[6], the language constructs students used to implement their solutions, and the errors that students had committed. She also raised some questions about what potentially could have contributed to students' success (or the failures of some) on the problem and suggested looking at possible factors such as: extent of coverage of HTDP, planning around design principles (*e.g.* cleaning data, reducing traversals), use of test cases to identify sources of errors, and use of higher-order functions. In my own discussions with Fisler (my doctoral advisor), we questioned the claim of "beating the Rainfall problem" because we did not have concrete explanations about *why* the HTDP students in her study were successful with the problem (or otherwise). She had only shown what students came up with as solutions, but not *how* they had approached the infamous Rainfall problem (*e.g. Did they, in fact, use the design techniques put forth by the curriculum they had learned in? How do students' processes look like in light of the techniques that they were taught?*); she did not have the data to come up with these explanations. She thus could not make concrete connections between the curriculum's claimed benefits and the outputs of her studied student-cohorts.

Earlier research reports of HTDP have mostly focused on introducing HTDP's design method into the field of computing education. For example, Felleisen *et al.*'s article [50] introduced the three "components" of their work, which include a program design method (the *design recipe*), a series of sublanguages of Scheme, and support software (the DrRacket development environment [44]). They also report on some preliminary findings: teachers who were trained in using the design recipe anecdotally reported finding value in the curriculum in improving students' general problem solving skills and that female students preferred an HTDP-based programming course over one based on a conventional AP curriculum. A set of papers from researchers in Germany report designing an introductory-CS curriculum *based* on HTDP, but mostly focus on their experience of designing such a curriculum [12, 32, 128]. One of these papers report some observations with students' use of HTDP-inspired design practices, though more as a set of (potentially anecdotal) experience reports rather than as planned studies towards exploring these specifically: some students did not use the design recipe in exams even when they said that they "might come in handy", some did not find signatures useful, and some were "shocked" to find that the design recipe helped them when they used it [12]. Ramsey also reports a "personal, qualitative case study" [112], citing observations of students' use of the design recipe techniques: students were careless about writing signatures and purpose statements; some students learned to value writing input–output examples/test cases, but some were also haphazard about writing them (*e.g.* writing multiple examples that essentially illustrated the same test scenario, just with different values); and many students didn't use templates to structure their functions (although it was not clear whether this led students to write ineffective code). Ramsey also proposed some pedagogical practices towards improving the teaching of HTDP-based courses, notably: focusing on

[5] ACM International Computing Education Research conference

[6] The problem essentially asks students to compute the average of a sequence of numbers that appear before a sentinel value; we describe this problem and its viable solutions further in Section 4.2.2. This problem is infamous: decades of research have repeatedly shown that CS1-level students fail to solve the problem [59, 126].

teaching students how to review their work (an explicit "review and refactor" step) and developing a clear, principled way of assessing students' work.

More recent work by Fisler *et al.* [53] proposed problems that required students to produce programs that addressed data-processing problems, as well as asked students to evaluate the programs they produced (this study was published around the same time I was running my own studies, so some of the problems in this work are used in my work as well). They suggested that more modern planning studies should also consider how to teach *problem decomposition*, and the factors that influence how students structure their code, such as curricula, pedagogy, and linguistic features. Ren *et al.*'s work [115] used the design recipe (among other additional factors such as reviewing past homework) as a framework towards assessing the use of teaching assistant (TA) office hours, finding that the design recipe can be used to categorize students' questions during TA hours. Wrenn and Krishnamurthi took a tool route and developed *Examplar* [146], a tool that provides students with feedback on the examples they wrote (independent of how much code they've written). They found that with the use of the tool, students' quality of test suites generally improved, but they were inconclusive about whether the tool helped students improve their own solutions.

Overall, much of the current work around HTDP are in their early stages, although the findings seem promising. Fisler has generally found that HTDP-trained students produced working solutions for programming problems, although it remains unclear how students' use of the design recipe techniques contributes to this success. Others have reported more concerning issues around students' use of design techniques: some students don't even use them or if they do, were haphazard and undirected in their use. The methods I employ in my studies (think-alouds, semi-structured interviews, field observations) enable me to dig deeper into students' design processes and develop descriptive frameworks that are grounded in what students actually do during their program design sessions and can be used to explain the factors that contribute to students' design processes.

1.2.2 Research Goal

My work is an exploration of how CS1-level students design programs towards multi-task problems that require processing lists, with a specific focus on their use of the HTDP design recipe.

My overall goal for my dissertation is to develop a *conceptual framework* that captures and describes *students' use of the* HTDP *design recipe in their overall program design process*. In other words, I am creating *an explanatory model (i.e. a system of ideas and constructs) of how novice programmers work as they program, when they have been taught the* HTDP *design recipe in their formal classroom instruction* on program design. At the completion of my dissertation project, the resulting conceptual framework should provide a data-grounded *theory* on how the design techniques put forth by the HTDP curriculum are used by students "in the wild" (*i.e.* when tasked to design a solution for a programming problem on their own).

The conceptual framework is envisioned to describe students' use of the HTDP design techniques, the techniques' interactions with other aspects of students' program design process (aspects such as

students' task- and code-level thinking of problems), and how these interactions contribute or relate to students' success in developing code for multi-task programming problems. This dissertation is an initial effort towards creating a theoretical grounding to HTDP in how students use the HTDP design recipe, in the hopes of refining how the curriculum is taught in practice, or informing the design of learning or teaching artifacts (*e.g.* learning activities, tools, assessments) that are based on, or informed by, the curriculum.

Research that helps us understand how students use the HTDP design recipe in practice is valuable because this informs us how well the design recipe meets our expected/intended goals for it, what we are missing in the design of the curriculum, and how we might improve on the teaching or design of the curriculum. Insights drawn from this research will be helpful for educators who use HTDP or draw on HTDP's design practices to design CS courses or curricula, or educators in general who are designing their own CS curricula and assessments by, for example, enabling educators to assess instructional material to concretely identify content that can support the development of particular design skills and offering insight on how educators could teach these content well.

1.2.3 Dissertation Research Questions

At a high-level, this dissertation project focuses on understanding how HTDP-trained students use the design practices put forth by the HTDP design recipe, particularly when solving programming problems with multiple task components. These problems invite students to make design decisions which may be informed by their use of the design recipe practices as they analyze the problem towards crafting a solution. We started this project with the broad research question:

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

As we conducted our studies, we iteratively refined this broad question based on our synthesis of our findings, resulting in the following specific research questions. Each research question has a *research activity* that illustrates how to address the research question, and the data to gather and analyze to investigate them.

Research Question	Data
DRQ1. What program design skills do HTDP-trained students exhibit when developing solutions for multi-task programming problems? *Research activity:* Identify program design skills and practices observed in students' program design process	• Video captures of student programming sessions while solving a multi-task programming problem • Student interview and think-aloud transcripts, written code solution, scratch work, and field observation notes on multi-task 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? *Research activity:* Describe (1) relationships between program design skills and aspects and (2) their role in students' design processes	• Student interview and think-aloud transcripts, written code solution, scratch work, and field observation notes on multi-task problems with familiar components
DRQ3. How do HTDP-trained students' use of program design skills evolve during a CS1-level course? *Research activity:* Describe how students' use and understanding of program design skills and aspects evolve	• Student interview and think-aloud transcripts, written code solution, scratch work, and field observation notes from study sessions at multiple points within a CS1 course
DRQ4. How do HTDP-trained students approach multi-task programming problems with novel components? *Research activity:* Describe how students apply program design skills and aspects in novel problems	• Video captures of student programming sessions while solving a multi-task programming problem • Student interview and think-aloud transcripts, written code solution, scratch work, and field observation notes on two multi-task programming problems, each of varying degrees of novelty

Chapter 2

Related Work

My work covers 3 main themes around the learning and pedagogy of introductory-level program design in computer science, specifically:

1. Models of how novice programmers design programs, particularly in introductory-level computer science (CS1) courses
2. Explicit instruction of program design strategies and techniques and their integration into curricula
3. Models of the evolution of program design-related skills

We discuss each of these themes by describing existing work within each theme and how HTDP relates to each theme.

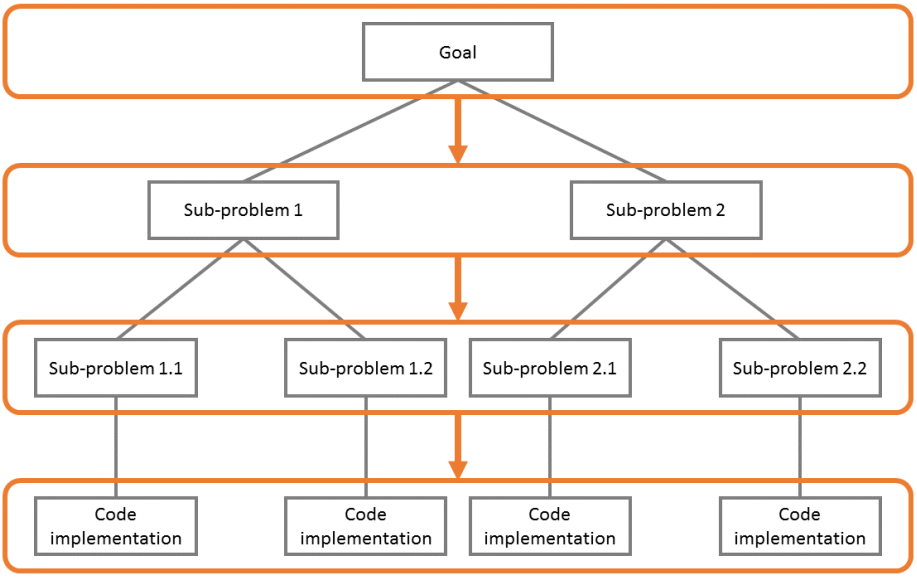
2.1 Models of Program Design in Introductory CS

2.1.1 Top-down Program Design

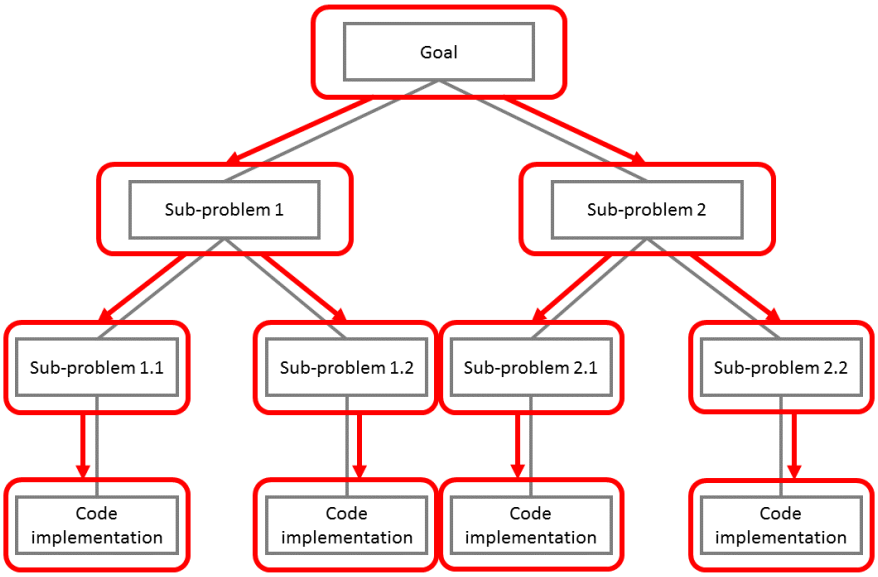
Early research on program design mostly focuses on comparing differences in programming knowledge structures between novice programmers and expert programmers [34], with a perspective of program design as a generally *top-down* process [117]. Top-down program design conceptualizes the development of a program similarly to a tree structure, starting with an abstract idea of a solution at the top of the tree, which is progressively refined and expanded in greater detail (usually breaking it down into constituent components) until a concrete solution is implemented in program code.

Jeffries *et al.* [67] have pointed to both novices and experts exhibiting a top-down strategy when designing programs, but with differences in how this is carried out (Figure 2.1). Top-down design by experts is done by decomposing problems in a *breadth-first* pattern with a similar level of abstraction at each iteration until a problem is reduced into manageable operations. Novices likewise decompose problems, but in a *depth-first* pattern as they fixate on an identified subproblem and focus on implementing an immediate solution to the subproblem before moving on to other subproblems. These

differences in design strategies between novices and experts are attributed to several factors such as a lack of programming experience among novices, leading to a more limited repertoire of programming knowledge versus experts, and novices having ineffective strategies for retrieving programming knowledge, in contrast with experts' years of experience, which enable the automatic recall of relevant knowledge.



(a) Experts exhibited a top-down, breadth-first pattern



(b) Novices exhibited a top-down, depth-first pattern

Figure 2.1: Top-down design observed by Jeffries *et al.* [67] from experts and novices

Interaction of Top-down Design with Knowledge Representation and Domain Knowledge

Schneiderman and Mayer [124] also observe the use of top-down design by expert programmers. They describe experts as developing a complex multi-level body of knowledge stored in long-term memory. One part of this knowledge is *semantic knowledge*, which refers to general programming

concepts independent of specific programming languages. They also have *syntactic knowledge* that is more precise and detailed than semantic knowledge, focusing on details concerning the formatting of language constructs. They report that in experts' processes of encoding or converting a program to an internal semantics or "chunking" [92], experts recognize the collective function of groups of statements to form comprehensible chunks, instead of absorbing a program merely on the basis of a low-level representation (*e.g.* individual language constructs).

Novices, on the other hand, organize knowledge syntactically using knowledge structures that are mostly based on surface features; elements of this organization are usually not related to each other in a strongly organized manner [1, 2]. This difference between novices and experts in the level of knowledge representation explains the difference in their top-down design strategy. Novices mostly have low-level, mostly concrete representations of programming knowledge. This imposes limitations on their use of abstract design and constrains them to design at detailed levels; in turn, this leads to their use of a more localized, depth-first strategy. Experts, due to the development of expertise through experience, develop functionally-based schemata and have representations at both abstract and concrete levels, which support their use of breadth-first design [1, 2, 46, 117].

Support for the top-down and depth-first design patterns by novices is also shown in the works of Anderson, Farrell, and Sauers [6] and Pirolli and Anderson [106] in their studies of novices learning to write recursive programs. They describe novices' early efforts towards writing recursive programs as heavily guided by analogies using examples of recursive functions [6, 105, 106]. Novices rely heavily on known solutions and primarily spend time looking at examples when developing new programs [106, 107], modifying and adapting already-learned solutions to fit the context of new problems. They also explain that novices' problem-solving is data-limited (*i.e.* they have limited knowledge) and additionally, that a major limiting factor in their learning is working memory capacity [5]. As novices grow in experience writing programs, they develop more efficient ways to chunk information.

Expert programmers, however, do not always exhibit a top-down, breadth-first design pattern, as argued by Adelson and Soloway [3]. They studied novice and expert programmers designing in familiar and unfamiliar domains. They observed that when designing in familiar domains, experts start with an abstract solution which they expand systematically through increasingly more detailed models, showing top-down, breadth-first design (similar to Jeffries *et al.*'s [67] findings). This changes however, when they start to design in unfamiliar or new domains. In this situation, the experts revert to a more localized process, expanding and simulating partial solutions, showing a depth-first pattern, similar to novices.

The breakdown of top-down design among expert programmers was also observed by Guindon *et al.* [58]. They found evidence of experts exhibiting a process of successive refinement when working on familiar problems. When working on unfamiliar problems, however, they employed a more *exploratory* and *serendipitous* design process, characterized by creating solution pieces at different levels of detail without a clear decomposition of the problem or clear connections between the pieces, which is similar to previously-observed behaviors of how novices program in prior work.

2.1.2 Program Plans

The top-down model of program design draws on the concept of programming knowledge retrieval. Differences in expertise spell out differences in the repertoire of programming knowledge and the level of abstraction or representation of this knowledge, which in turn affects design strategies exhibited by programmers (*e.g.* breadth-first vs. depth-first).

Soloway describes the knowledge used to write programs as being embodied in *plans* [127]. *Plans* are organizations of groups of code that work together to achieve a specified goal [127, 129]. Plans may be composed of sub-plans to achieve multiple sub-goals. Depending on a programmer's experience or level of expertise, plans may be stored at different levels of abstraction. Literature on programming knowledge representation often use the terms plans, schemas, (language-independent) *templates*, and *canned solutions* interchangeably. The concept of a plan draws from cognitive psychology on the idea of "chunking" knowledge into schemata, or the mental organization of knowledge [92, 127]. Plans can be viewed as having both *deep structure* and *surface structure*. The deep structure of a plan describes the actions and the dependencies between the actions that collectively achieve the overall goal. The surface structure of a plan refers to the artifact implementation of the plan (*e.g.* language constructs), without regard to the semantic dependencies that describe the workings of the plan [120, 129].

Most of what causes major difficulties in programming among novices is their inability to compose and coordinate components of a program due to them having mostly a surface structure knowledge of plans [120, 129]. Experts, on top of their knowledge of the syntax and semantics of language constructs, have built libraries of stereotypical solutions to problems and strategies for coordinating and composing these solutions. A hallmark of expertise thus is the ability to view a current problem in terms of old problems and strategically transfer solutions from old to new problems. According to Soloway, when programmers do not have the relevant plans or plan knowledge to solve a problem, they simply flounder and use knowledge or techniques inappropriately [127].

2.1.3 Bottom-up Program Design

Rist [117, 119] expanded on existing models of program design to account for situations in which (novice) programmers lacked knowledge or similar solutions on which to build. His model describes two paths programmers take when writing code:

1. *Plan Retrieval*: When a programmer knows a viable schema or a solution to a similar problem, they will retrieve it (from memory) and implement the code in a top-down fashion, with smaller-scale modifications to address problem subtleties.

2. *Plan Creation*: When a programmer has no schema in memory or does not know a solution to a problem, they identify a core computation called a *focus* or *focal computation*. A focus is usually a major computation required in the problem. For example, in a problem requiring the computations of the average of a sequence of numbers, there are three potential focal computations: one for each of the required tasks of *summing* the elements, *counting* the elements,

and *dividing* the sum by the count. Rist's model does not address which focal a programmer would handle first, rather, it claims that an expression related to one of the tasks would be written and expanded upon first. Programmers would start by writing code to implement the focus, and then expands or builds around that code bottom-up until a working solution is achieved, or integrates the new code into the rest of the program.

When a known solution applies to a more difficult problem, the model predicts that students switch to bottom-up creation mode after retrieval. As a programmer's experience increases, they make heavier use of retrieval. Rist developed this model from watching students produce code in Pascal for problems such as calculating the volume of a box-like house or sorting weights into ascending order. The essence of the model lies in: (a) new plan creation starting from a focal computation, and (b) the construction of code being either top-down or bottom-up, depending on whether a plan is being created or retrieved.

Rist also described how the decomposition of problems take place [117, 119]. When the overall plan is too complex and matching schema cannot be retrieved for it, Rist describes that the plan must be decomposed and that this decomposition may be any of the following two types (or potentially a combination of these):

1. *Decomposition based on "slots" in a schema*:
   This involves the identification of component tasks in a schema, for example, identifying the "sum", "count", and "divide" tasks for an "averaging" goal. Once the component tasks are identified, each task-component is filled with detailed plans one at a time.

2. *Decomposition based on the generalization of plans or the repetition of events*:
   A plan for summing a large set of input can be decomposed into a simpler goal of summing two initial values. The simpler goal has a simpler plan that may be retrieved, providing a focus which can then be extended to achieve the larger goal. For example, retrieving the simple sum plan (adding two values) and then extending it by adding code for a loop to repeat the sum plan achieves the overall goal of summing a set of input.

In general, the implementation of the components of an overall plan is dependent on the availability of schema. In cases where plan components have no retrievable schemas, a programmer will initiate the development of code through a pattern of bottom-up development from an identifiable computation. For plan components with retrievable schemas, programmers will generally follow a top-down, forward development after retrieval of applicable schemas (from memory). The plan components are then extended by adding code either for expansion or to merge or combine plans together. The knowledge retrieved by programmers can be abstract or specific.

2.1.4 Interplay of HTDP and Models of Design and Planning

HTDP both draws on and challenges the existing models of program design we discussed in prior sections. We assume a novice programmer for points made in this section.

An implicit assumption in the top-down model is that after identifying relevant information from a programming problem, the programmer maps the information to some implementation. Note here, that the "relevant information" may be for an entire problem, for example, when a problem is small enough (such as summing a set of numbers), or when the problem is big or has several task-components, but the programmer has already seen the problem—or components of it—and thus has knowledge of how to implement the problem. The information may also just be for familiar sub-components of a problem. The mapping from information to implementation is easy when there are retrievable schemas, in which case, these schemas are implemented and combined in some way (depending on problem constraints and requirements) to achieve a solution. When there are no retrievable schemas, the mapping is primarily done bottom-up, by way of operations for tasks and then extending and combining these towards a solution.

HTDP challenges this direct mapping from problem to implementation. As a programmer goes through each step (as outlined in Section 1.1), information about the problem is also made more concrete in step-by-step fashion. Students first describe a problem's input data by articulating its structure and identifying concrete examples of the input. At this point of the process, the programmer is not dealing with defining a solution yet. Rather, the programmer is concretizing an important piece of information from the problem, the input data, which is processed by the (proposed) solution later on.

The next steps start to deal with describing a solution by first describing an expected behavior, independent of an implementation that mechanizes this behavior. This is done by first writing out a contract, which names the program (or function) and describes the type of the input and the type of the output. This is followed by writing a purpose statement, which invites the student to summarize an expected behavior relative to the program's goal. These steps provide a representation of a proposed program; there is no implementation yet, but these steps ask the student to concretely capture or describe an expectation of the program's behavior relative to the possible space of inputs derived from the problem or the input examples identified in an earlier step. This representation of the program's behavior is further developed through the next step of writing input-output pairs, or test cases. In this step, the student is asked to describe concrete output for specific inputs, both from the examples of input identified earlier and potentially, additional concrete inputs the student may identify.

The next steps are when the student actually starts implementing a concrete solution in code. Using the identified input type as a reference, the student retrieves and writes a *template* for that input type (top-down retrieval of the template), which serves as a program skeleton. This template provides a traversal of the data, and this traversal provides a *context* in which operations on the data can be defined. Note before this template step, information about the problem has been distilled into concrete instances: the type and structure of the input data have been identified, concrete examples of the input are described, the expected type of the output has been identified, and the expected behavior has been articulated through input and output pairs. These are all concrete information that has become available (in addition to the usually more abstract problem statement) and can be leveraged by the student for the next step of building the implementation.

The student can leverage the concrete information at hand to identify operations that will transform the input into the expected output. While the template provides the data traversal mechanism, at this point, the student needs to develop an understanding of the computations that need to be addressed within the traversal of the data. In other words, a student, when combining operations with the template, has to be able to understand how an output is built up from repeated application of an operation over the data. We discuss the use of the template in further detail later.

The HTDP process serves as a scaffold and also aids design in two ways:

1. It has been designed to help novices create a more concrete understanding of the problem-space by systematically concretizing information about a problem.

   Here, HTDP uses the concept of top-down design, but instead of in terms of an increasingly more detailed implementation of a program (as the original top-down design prescribes), provides increasingly more detailed and explicitly described information about the problem. Concrete information helps provide more actionable insight about the problem. Identifying the type of the input, for example, tells the student what the structure or *shape* of the code should be (*e.g. Does it require a list traversal? Does it require a tree traversal?*). Students are also directed to identify the space of input, which can concretely illustrate tasks or special behavior that may otherwise have remained hidden in an abstract problem statement (*e.g. Are all the elements in the input relevant to the problem? Does a computation need to be applied to all of the elements in the input?*). The use of explicit strategies and examples has been cited by early studies [127] as a critical need for effective instruction and in more recent research has been proven to be an effective method of program design instruction [35] (we discuss this more in Section 2.2).

2. It provides several representations of a program at various levels of abstraction: a general description of the program (contract and purpose), a model of the behavior of a program (input-output examples or tests), and a datatype-driven program structure (template).

   One of the motivations for our research is what we can learn about how students use concrete information and the relationships between the program representations put forth by the design recipe to inform and build their solutions. Findings around this can be used to inform the design of instruction and interventions for programming instruction, for example, tools and material that help students design better tests, or tools for visualizing relationships between program components (the design of these, however, is beyond the scope of this work).

The use of the template also essentially challenges bottom-up creation, or at least delays it. The bottom-up model predicts that a student will identify and implement a problem-related operation (related to a problem task) and then build on that focus towards an overall solution. HTDP however, provides the student at least a retrievable schema through the template, which the student fills with relevant operations. In the traditional bottom-up model, the identified operation provides the context on which to build the program; in HTDP, the template, in essence, the structure of the data, provides a context for the operations. This way, HTDP seems to embody both top-down and bottom-up

patterns in its approach. Another motivation for our work is understanding how this concept of the datatype structure providing programming context helps students in planning programs. This is an interesting question as it challenges the traditional practice of matching constructs to problems and provides curricular and instructional designers an alternative approach that can be used in introductory computing curricula.

2.2 Explicit Instruction

A pedagogical practice gaining increased attention in CS education is explicit instruction. De Raadt *et al.* [38] note that novice programmers have traditionally been expected to practice problem-solving skills implicitly in their programming, developed by repeated attempts on practice problems. Hermans and Smit argue that explicit direct instruction is more effective than exploratory approaches, even in programming education [62]. Prior studies have revealed the presence of problem-solving strategies used by experts, yet similar strategies (or ones that are appropriate for students who are just starting to learn to program) are generally not explicitly incorporated into introductory programming curricula [35, 127]. Programming strategies refer to the understanding of how to apply programming knowledge appropriately to solve problems. Traditional programming instruction and texts focus on providing programming knowledge which consists of language constructs and facilities, and the rules on how to combine them. A systematic framework or methodology for choosing appropriate solutions or for building strategies to do this, however, has mostly been absent, even when implicit problem-solving instruction has been shown to result in poor learning. While there are studies showing that implicit learning may improve learner performance (perhaps due to repeated practice), it does not create understanding of the underlying systems used [35].

2.2.1 Explicit Instruction in non-CS contexts

The idea of explicit instruction and its reported beneficial affordances is not new. Researchers from other fields have shown differences in performance between learners in implicit and explicit instruction contexts. A study by Biederman and Shiffrar [11] has shown a quantitative difference between teaching explicit and implicit approaches in chick-sexing (determining the sex of day-old chicks). Their results showed that an experimental group of inexperienced sexers outperformed a control group. The control group underwent the traditional training in chick-sexing, which involves standing alongside an expert instructor, making observations during the sexing process, and finally attempting the task through trial and error. The experimental group studied an instruction sheet with explicitly described key visual aspects from an expert professional sexer.

A study on language learning by Reber [114] showed that implicit-only learning did not promote the understanding of underlying systems of knowledge (*i.e.* the conceptual knowledge of the languages being learned), even when language patterns (*e.g.* syntax and grammar rules) may be recognized. In his study, an experimental group was shown sequences generated from an artificial grammar, without being

shown the rules used to build the sequences, while a control group was shown randomly generated sequences. In a post-test where both groups were shown a new set of sequences, the experimental group was able to identify grammatically correct sequences but were unable to express an understanding of the rules for the sequences. Berry and Dienes [42] shared a similar finding in their study on simulated transport systems: participants who learned the workings of a transport system through implicit instruction were subsequently able to operate the system, but were unable to show any understanding of the underlying rules of the system.

2.2.2 Explicit Instruction in CS Pedagogy

Bailie [7], working on the idea that the process of planning can be facilitated through decomposition, ran a study that involved having students explicitly write a collection of small functions ("modules") as a treatment and then have them practice building programs that make use of the already-existing functions in a posttest. While the work admits the absence of a control group to compare results with, preliminary results show some performance improvement among students in the posttest as compared to the pretest (the pretest and posttest had the same setup, with different problems of comparable difficulty). Bailie attributes the improvement to the explicit writing and availability of the functions.

De Raadt *et al.* [37] tested a curriculum with programming strategies explicitly incorporated in course materials, exercises, and assessments. The work considered the dimension of instruction delivery. Implicit instruction expects students to undertake new learning or extend prior learning without being given the full context for what they are to learn and how. For example, when taught looping constructs, students are expected to generate different types of loop implementations such as sentinel-controlled loops or using it to repeat computations. Explicit instruction openly describes the subject to be learned and how to go about learning the subject, and is usually documented in some form. They wanted to explore (a) if explicit strategy instruction can be successfully incorporated in an introductory programming curriculum and (b) the effects of incorporating explicit instruction in an introductory programming curriculum. Explicit instruction of strategies in this work involved (1) naming programming strategies, (2) explaining the benefits of specific programming strategies, and (3) providing examples of the applications of the strategies.

The researchers compared the performance of students in an experimental group who learned through an introductory CS curriculum with explicit programming strategy instruction, with students in a control curriculum (implicit instruction). They found that while the added content required additional lecture time and decreased the time for students' practical work, students still completed exercises within scheduled time. The researchers attributed this to the instruction easing the burden on students when working on exercises. Even with explicit instruction, some of their students still forgot or neglected the plans they learned, but (in subsequent interviews) seemed to express more confidence in their own solutions, compared to students in the control group. Students from the experimental group also demonstrated the use of, or attempted to use, planning vocabulary in their explanations, but would still resort to using syntactic descriptions of code. They also observed that students who

underwent explicit instruction were more likely to understand and use the strategies they learned than their counterparts in the control group.

Muller *et al.* [95] described guidelines for pattern-based instruction that can be used for creating sets of problems around specific "design patterns". The "patterns" described in this work seemed to draw on object-oriented design and imperative programming (with emphasis on initialization, control flow, and variables) and focused on problems such as counting, accumulation, computations of extreme values, searching, criterion-matching, and identifying most frequent elements, to name some. Their guidelines consisted of 9 points:

1. **Representative example:** Provide a simple concrete example to illustrate the pattern
2. **Pattern definition:** Define a pattern abstracted from analogous problems by describing the pattern's components
3. **Pattern name:** Define a name for the pattern that captures and illustrates its essence
4. **Similar patterns and similar problems:** Identify similarities and differences to related patterns and problems these solve
5. **Comparison of solutions:** Identify and compare alternative solutions to a given problem with a focus on differences in algorithm efficiency
6. **Typical uses:** Identify representative contexts in which the pattern is commonly utilized
7. **Common mistakes and difficulties:** Identify common misuses and difficulties with the pattern
8. **Pattern composing:** Identify problems with solutions that require compositions of several patterns, or multiple uses of the same pattern
9. **Entry and turning point:** Practice modifying solution patterns for similar problems to satisfy varying constraints

The main idea around these guidelines was to introduce patterns to students through sets of problems. Recognizing similarities and differences between problems and their solutions may encourage the reuse of ideas from previously solved problems. Their idea was that looking for common characteristics between problems supports the understanding of the behavior of the solutions for these problems and the analysis of similar solutions. They called their approach *pattern oriented instruction*: their approach involved attaching labels to algorithmic patterns and presenting various problems to students (designed with the 9 guidelines described above), while encouraging students to look for common patterns across problems. They found that their students were successful at applying the patterns at the end of their courses [94].

2.2.3 HTDP: Explicit Design Strategy via a Step-by-step Process

The current trend of research around explicit instruction in CS pedagogy involves teaching mappings of problem types to implementation-focused strategies. My work frames the idea of a "strategy" differently. Explicit strategies in prior work [35,95] mostly leverage the use of programming constructs and language features and focus on the application and merging of these constructs to satisfy problem goals and constraints. We instead look at the step-by-step HTDP design process (*i.e.* the design recipe), and its data-centric focus, as the "explicit strategy" for designing programs. The design pattern at the heart of HTDP, the program template, is based on datatypes and is driven by data definitions (recipe step 1) rather than language-construct-centric strategies.

Drawing on the idea that experts are characterized with having a vast repertoire of programming knowledge and strategies [67], prior work on explicit strategy instruction work towards increasing a novice's collection of programming strategies. This is useful for novices when they are able to achieve a direct mapping between problems and their known strategies [117, 119]. The retrieval of schemas may fail when a mapping cannot be done from a novel problem to a schema. Without a direct mapping, it brings a novice back to Rist's focal expansion model, even when their solution-tinkering would be on a more abstract level of chunks of strategy implementations.

Instead of working on direct mappings between problems and schemas of solution implementations, HTDP's design recipe works towards illuminating more information about the problem domain as a novice works through the recipe steps. The main retrievable schema here is the template, which is retrievable mainly with the knowledge of a problem's input *type* (topics in the latter parts of HTDP-based courses add more schema for functions that map, filter, or fold computations). The concrete information provided by artifacts developed in each recipe step helps inform the expansion of the template towards a solution. This presents interesting questions to explore around (1) whether students recognize relationships between the different components of the HTDP design process and (2) how they use these relationships to inform and structure their solutions.

2.3 Models of Skill Evolution in Introductory CS

One of our goals in this work is to understand how HTDP-trained students' use (and understanding) of program design skills evolve within an introductory CS course (CS1). We describe in this section, a model called the SOLO taxonomy, which describes the increasing complexity of how concepts are learned or understood. We also discuss the extent to which the SOLO model has been used in CSEd research.

Biggs and Collis proposed the Structure of Observed Learning Outcomes (SOLO) taxonomy model in 1982 [13]. This taxonomy captures 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
- **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.* [141] (code reading, comprehension, and summarization), Lister *et al.* [86] (code comprehension), Shuhidan *et al.* [125] (writing code to calculate max/min integers), Ginat *et al.* [55] (algorithm design), and Izu *et al.* [66] (code design).

Ginat's emphasis is on selection and composition of high-level design patterns [55]. Ginat's unistructural level captures solutions that translate a specification into a straightforward use of a single design pattern. By the relational level, Ginat expects students to compose plans through the interleaving of code. Ginat has generally expressed a preference for interleaved code on the grounds of efficiency [94]. The HTDP curriculum, in contrast, does not emphasize efficiency, but teaches students to think about readability and maintainability as code is adapted to different contexts. Their project and the HTDP-based courses we study thus have different values regarding composition skills.

Izu *et al.* (who respond to Ginat *et al.*) focused on code design rather than pattern selection and integration [66]. Their taxonomy is in terms of combining building blocks, which may or may not arise from previously-learned general patterns. Izu *et al.* looked at how students combined code fragments into solutions, although this combination focuses on the syntactic merging of code (akin to Ginat's interleaving of code) and it is not clear how or whether a semantic understanding of this code merging plays a role in students' understanding of combining code. Both Ginat and Izu conflate issues around decomposing problems and composing code. In addition, neither considers testing, which is considered an equally-important design skill in our work.

Thompson's SOLO taxonomy for grading programming assignments (which he also used to design exam questions) captures multiple components of activity. He weaves these into a single programming skill progression (that requires certain behaviors from each component) [135]. 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.

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.

Chapter 3

Study: Interplay of HTDP with Rist's Focal Expansion Theory

Background and Context: Most models of how novices program suggest that they use previously learned examples or solutions as starting points for new programs. Rist's *focal expansion* model, in particular, suggests that novices program bottom-up from a statement or expression that captures the essence of an identified program task. In contrast, HTDP aims to be more systematic by teaching students to first write scaffolding code that exploits the structure of input data as part of an overall design process towards generating programming solutions.

Objective: We sought to explore the interplay of bottom-up programming and datatype-driven design, using Rist's model and HTDP as concrete instances of each, to gain insight into how Rist's idea of bottom-up programming play out in the context of data-traversal schemas (Section 3.1).

Method: We gave HTDP-trained students a multi-task programming problem that required program structures they had not yet learned. We video-recorded their programming sessions and coded the videos by looking for how students used each of focal-expressions and HTDP scaffolds in attempting to solve the problem. Students filled out a survey that asked them about their thought-processes as they wrote their solutions for the programming problem (Sections 3.3 and 3.4).

Findings: We found that students largely worked through core problem tasks, either through task-related focal operations or as entire functions, but generally struggled with problem decomposition and solution composition (Sections 3.5 and 3.6).

A version of this chapter is published in the following venue:
[19] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16), 205–210. DOI: https://doi.org/10.1145/2839509.2844574

3.1 An Exploratory Study

Most models of how novices program suggest that novices draw on prior knowledge of learned examples or solutions as starting points for new programs [106, 107, 130]. An interesting point of exploration, thus, is when novices face a new problem that is sufficiently different that previously-learned examples don't apply. In particular, Rist's focal expansion model [117] (Section 2.1.3) suggests that novices enter a "creation" state when they do not have existing plan knowledge for a given problem: they start from a code fragment called the *focus*, which they expand in a bottom-up pattern towards a working solution. In contrast, HTDP suggests a design recipe to follow to systematically elicit problem details, written as concrete artifacts that can be used to inform the development of a solution to get beyond a blank page when starting a programming problem. Step 5 of the design recipe directs students to write scaffolding code (the template) that exploits the structure of the input data, as described by the data definitions produced in a prior recipe step (step 1).

The HTDP templates are at the heart of the difference between HTDP and Rist's model: in essence, the templates defer entry into Rists's creation state by providing a retrievable schema (based solely on the type of input to a function), which provides the *context* for writing code for computations. This process should direct students to place focal computations either in template holes or in auxiliary (*i.e.* helper) functions. Additionally, the artifacts produced in prior design recipe steps (*e.g.* data definitions, examples, signatures, explanations, test-cases) should help inform the novice on how to appropriately fill in the template. In Rist's model, an identified focal code provides the context for building a solution instead. Seeing what HTDP-trained students do after writing templates should give insights into whether and how Rist's model, and the general idea of bottom-up programming, play out in the context of data-traversal schemas.

In this first study, we sought to explore and understand the interplay of bottom-up programming and datatype-driven design, using Rist's model and HTDP as concrete instances of each. We gave HTDP-trained students a programming problem over a familiar input datatype (a list of numbers), but which required programming techniques they had not yet seen, for example: reshaping a list input with an underlying structure into a list of lists, recurring on a modified suffix of the list, or detecting a sentinel pattern to terminate a computation. This combination should essentially push students into Rist's creation state (perhaps after retrieving the list template). We sought to address the following research questions in this study:

STUDY-RQ1. When do HTDP-trained students use templates?

STUDY-RQ2. How does Rist's idea of focal computations manifest in HTDP programs?

STUDY-RQ3. How and when do HTDP students integrate focal computations into existing code?

Our questions attempt to avoid bias in favor of either Rist's model or HTDP's claimed benefits. While we expected students to follow the HTDP design recipe process (as this was a key part of the course we studied), we did not assume that students had internalized that process enough to actually do so. This is an exploratory study, asking whether: (a) Rist's focal-expansion theory applies to a

programming process that is directed by the HTDP design recipe, and (b) the design recipe provides useful scaffolding to students on problems that require significantly different programming techniques than what they have already seen.

3.2 The Adding Machine Problem

In this study, we used a programming problem called *Adding Machine*[1]. The exact wording for the problem follows, along with viable solution approaches:

> *Design a program called* `adding-machine` *that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.*
>
> Example:
> `(adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))` should produce `(list 3 7 10)`

The Adding Machine problem involves four tasks:

1. Ignoring data after the double-zero sentinel pattern
2. Identifying sublists separated by single-zero delimiters
3. Summing the elements in each sublist
4. Building the output list from the sums of the sublists

Viable approaches for solving Adding Machine include:

1. **Reshape the data first.**
   The sublists in the input are embedded in a flat list of elements delimited by zeros. The input could be *reshaped* into a list of lists that omits the zeros. For example, the input `(list 1 2 0 7 0 5 4 1 0 0 6)` could be reshaped as `(list (list 1 2) (list 7) (list 5 4 1))`. Separate functions could recur over the outer list to compute the sum of each inner-list.

2. **Accumulate sums in a parameter.**
   The recursive function could take an additional parameter for the sum of the current sublist. When a 0 is detected at the front of the input list, this parameter would be concatenated onto the result of processing the rest of the input list with the sum parameter (re-)initialized to 0.

3. **Recur on a new (modified) list containing the sublist sum.**
   The first position of the list can be used to store the running sublist sum. For example, the call `(adding-machine (list 1 2 0 5))` would generate the call `(adding-machine (list 3 0 5))`. Special care is required, however, if a sublist can sum to 0.

[1] We thank Mike Clancy for pointing us to the Adding Machine problem.

4. **Recur on a new list that skips the first sublist.**
   A function could recur on the suffix of the list without the first sublist, using a separate function to produce the sum of the prefix corresponding to the first sublist.

Detecting the consecutive-zero sentinel pattern adds a bit of complexity, as solutions must check both the length of the remaining input (an input that doesn't contain the `0 0` pattern might have only one element) and the values of the first two elements. Solutions can either truncate the input data at the double-zero pattern in a separate pre-traversal, or integrate checking for the pattern into the core computation.

Adding Machine is a reasonable problem choice for exploring the interactions between Rist's focal expansion model and the HTDP design recipe. At the point at which we collected data (Section 3.3), lists of numbers would have been a familiar datatype to the students: many will have already internalized the schema for flat lists. Each of the high-level solution approaches outlined previously, however, uses more advanced programming concepts that the students have not yet seen before in the course: parameters that accumulate data (accumulators) or recurring on something other than the `rest` of the list. While basic mastery of lists would suffice to reshape the input data, students would not be exposed to the *idea* of doing so until much later in the course. Thus, if HTDP-trained students are given this problem after a couple of weeks of programming with lists, they will have a schema that appears to apply (the basic list template), but no experience with solutions that draw on the more advanced concepts used in the Adding Machine solution approaches. We expect that many students should end up in the *plan-creation* state while working on the problem, even if they initially retrieve the basic template for lists.

3.3 Study Design and Data Collection

3.3.1 The HTDP Course Instance

We collected data in the Spring 2015 offering of *CS1101: Introduction to Program Design*[2], taught using HTDP in the Racket programming language [110]. We conducted the study five weeks into the 7-week academic term, after students had roughly 16 lectures (50-minutes each), 4 labs (50-minutes each), and 4 multi-exercise programming assignments for homework. Before this point, the course had covered defining and calling functions, composing functions, conditionals, recursive functions over lists, and the HTDP design process (as described in Section 1.1). The students had written several functions over lists of numbers, strings, and records prior to doing the study. The course had not yet covered trees, accumulating results of computations in additional parameters (accumulators), or recursive calls on an argument other than the `rest` of the list.

[2]We thank WPI Professor Joe Beck for letting us collect data in his course.

3.3.2 Participants

We ran the study during a weekly lab session. In total, 138 students submitted data; we randomly sampled 25 students to analyze in this study. In terms of final course grades, the sampled population earned 5 As, 13 Bs, 3 Cs, 3 fails, and 1 incomplete (Table 3.2). We thus had a good mix of students relative to mastery of the material and likelihood of needing help.

3.3.3 Logistics

Students were given roughly 40 minutes to work on Adding Machine during a weekly lab session. Each student used the SnagIt video-capture tool [134, 145] (pre-installed in the lab computers) to record all activity within the window for the course IDE (DrRacket [44]). Students uploaded both the video and their final source-code file at the end of the lab session. The problem statement for Adding Machine, instructions for using SnagIt to capture their programming activity, and instructions for submitting their code were provided online. The web page for the Adding Machine problem statement is shown in Figure 3.1; we provide the full screenshots of the web pages relevant to this study in Appendix B.

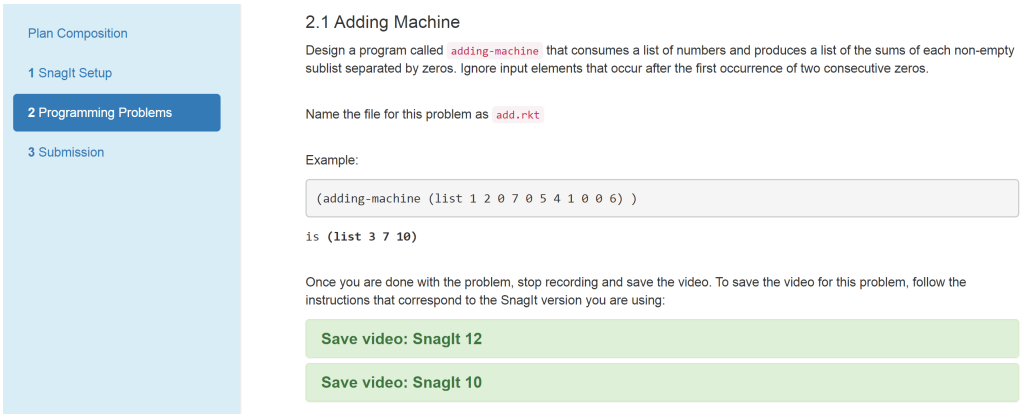


Figure 3.1: Screen view for the Adding Machine problem statement

After submitting their video captures and code, we also asked students to fill out an online survey (see Appendix B.5 for the Qualtrics survey instrument used) that invites them to recall their thought-processes as they wrote their solutions for the Adding Machine problem. The survey asked students the following questions:

1. How did you get started with the problem? Provide 1 to 2 sentences describing the process of how you got started with the problem.

2. At any point during the programming session, were there instances when you looked at notes? If yes, what did you have to look up?

3. Were there any points in the problem when the design recipe was useful? If yes, describe in a sentence or two when you used it.

4. Was there a time when you were trying to use the design recipe but felt you didn't know what to do next? If yes, describe in a sentence when that happened.

3.4 Analysis

3.4.1 Coding the programming session videos

We analyzed the videos by qualitatively coding the video events. Our coding focused on recording the following events:

1. **Template use**
   Students wrote an HTDP-prescribed template for a function.

2. **Task-specific computation**
   Students wrote code related to one of the problem's four problem tasks. We recorded the actual code written, the task it belongs to, and the function in which students put the code. The tasks were recorded with the following labels:

   - **singlezero**: Handling single-zero delimiters
   - **doublezero**: Handling the double-zero sentinel pattern
   - **sumelts**: Summing the elements of a sublist
   - **buildsumlist**: Building the list of sublist sums
   - **listoflist**: Reshaping the main input into a list of lists, excluding the zero delimiters

3. **Writing input-output examples/test-cases**
   Students wrote input-output examples/test cases for a specific function. We recorded the function name and number of tests written for the function before the next event occurred.

4. **Other**
   Students made edits that did not fall into one of the above categories.

The coding procedure involved watching the programming videos of each sampled student and recording events, based on the event list above, in the order that they appeared in the video. We illustrate the result of this process in the example coding summary in Figure 3.2, which shows (a) the coding summary and (b) the corresponding state of the Racket program; the program shows the code as of step 3 in the summary. In the summary, AM refers to the Adding Machine function. The student wrote another function named `findzero`, replaced with the alias, `helper1`, in the summary. Names for helper functions were replaced with aliases with the format `helper<number>` to facilitate consistency in the coding as the students would sometimes change the names of helper functions as they programmed their solutions. We produced a summary such as in Figure 3.2a for each of the 25 sampled students.

```
1 Test AM 3
2 Template-list AM
3 AM buildsumlist (cons (helper1 first-L)
                        (AM rest-L))
4 Template-list helper1
5 helper1 sumelts (+ first-L (helper1 rest-L))
6 helper1 singlezero (= 0 first-L (AM rest-L))
7 AM singlezero (= 0 first-L)
8 Test helper1 3
```

(a) Coding summary

```
;; ListofNumber->ListofNumber
;; adds together elements of a sublist and returns
    them as a list
(check-expect
     (adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))
     (list 3 7 10))
(check-expect (adding-machine empty) empty)
(check-expect
     (adding-machine (list 5 15 22 0 7 0 8 1))
     (list 42 7 9))

(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (findzero (first lon))
                    (adding-machine (rest lon)))]))
```

(b) Corresponding code state at step 3

Figure 3.2: (a) Sample coding sequence and (b) the actual Racket program code for student WPI1-STUD1 at the 9-minute mark

3.4.2 Coding the survey responses

We open-coded [28, 69] the survey responses to look for themes within each survey question, as they relate to how students got started with their solution, what they looked up in their notes (if ever), whether and how they found the design recipe useful, and what they did when they got stuck.

3.5 Results and Interpretation

None of the sampled students produced working solutions for Adding Machine, even when we observed them use HTDP list templates, develop focals, and attempt to decompose the problem. *Plan composition* was the main hurdle, particularly when students tried to reuse the list template code inappropriately in multiple smaller-scale plans. All but one student (24 of 25) used the list template for the list-of-numbers input (WPI1-STUD7 did not use the list template). From there, students took many approaches.

3.5.1 Writing Focal Code After Templates

We hypothesized that students would enter Rist's creation state after retrieving the list template. As such, we looked at what code students wrote immediately after writing the list template and where they put the code, checking whether it captured focal computations. Table 3.1 shows that all but 3 students (WPI1-STUD14 , WPI1-STUD19 , WPI1-STUD20) wrote expressions that took on specific problem tasks. 19 of the students put this new code into the template for the Adding Machine function. This matches the focal-expansion theory, as well as HTDP pedagogy. Most template holes get filled by focal-like expressions, though some decomposition through helper functions also goes there.

What isn't clear, however, is *whether* students had entered creation mode. Summing a list is a standard HTDP programming problem; as such, students may have retrieved the summing code. The single- and double-zero tasks are about the termination of computations; as such, they resemble base-cases of recursive functions (even though the usual base-case of a recursive function on a list handles the empty-list case). Students may have retrieved the pattern of terminating a traversal and adapted it to recognize patterns of zeros.

Table 3.1: First tasks coded by students, with location

Adding Machine Task	Within the Adding Machine function	Within a helper function
Summing a sublist (sumelts)	Within list template: 1. WPI1-STUD3 2. WPI1-STUD10 3. WPI1-STUD11 4. WPI1-STUD13 5. WPI1-STUD17 6. WPI1-STUD21	0
Building the output list (buildsumlist)	Within list template: 1. WPI1-STUD1 2. WPI1-STUD16 3. WPI1-STUD23	0
Handling single-zero delimiters (singlezero)	Within list template: 1. WPI1-STUD4 2. WPI1-STUD5 3. WPI1-STUD6 4. WPI1-STUD8 5. WPI1-STUD9 6. WPI1-STUD18 7. WPI1-STUD22 8. WPI1-STUD24 9. WPI1-STUD25	0
	Not within list template: 1. WPI1-STUD7	
Handling double-zero sentinel pattern (doublezero)	Within list template: 1. WPI1-STUD2	Within list template: 1. WPI1-STUD12 2. WPI1-STUD15
None/Other	1. WPI1-STUD14 2. WPI1-STUD19 3. WPI1-STUD20	

Overall, 21 students who started with templates immediately filled in template holes with focal expressions for a specific problem task. Two students (WPI1-STUD12 , WPI1-STUD15) began to decompose the problem by creating a helper function: in both cases to handle the double-zero task. Of the remaining 4, one wrote a focal computation for handling the single-zero task within a non-template function (WPI1-STUD7), and 3 wrote something not clearly linked to a problem task (WPI1-STUD14 , WPI1-STUD19 , WPI1-STUD20).

3.5.2 Difficulties with Plan Composition

Whether students retrieved or created plans for the problem tasks, they still had to compose them into an overall program. Here, students displayed significant difficulties. We illustrate these challenges with sample code from our data in our discussions below. Table 3.2 also summarizes the tasks that students wrote and their location in their code.

The solution in Figure 3.3 attempts to integrate the plans for the sum and double-zero tasks by sharing template code. The template base-case (the `empty?` check) returns a 0 as in the *sum* plan, while the “new" base case for *double-zero* returns a list (the output type of the overall function). In attempting to share the recursive call, which would be syntactically identical in both the *sum* and *truncate-at-00* plans, the student created an *inconsistency* in the output type of the program. Only 5 students wrote

```
(define (adding-machine lon)
  (cond
    [(empty? lon) 0]
    [else
      (if (and (= (first lon) 0) (= (second lon) 0))
          (list 0)
          (+ (first lon)
             (adding-machine (rest lon))))]))
```

Figure 3.3: Code from student WPI1-STUD18 interleaved function calls within one function without decomposition

both the sum and double-zero tasks (Table 3.2: students superscripted with [00+]); 3 of these put these tasks in the same function (WPI1-STUD2 , WPI1-STUD5 , WPI1-STUD8). Six students put both the single-zero and double-zero tasks in the same function (Table 3.2: students superscripted with [0-00]), missing that each terminates a different other task (processing a sublist and identifying input to process, respectively).

In the solution in Figure 3.4, the student tried to decompose the problem via a helper function. The output task (building the output list of sums) stayed in the main template (the `adding-machine` function), while the sum and single-zero tasks moved into the helper. This approach was on the right track, but had two key errors (aside from the missing double-zero plan): the single-zero detection needed to be a base-case (and return 0) in the helper, and the recursive call in the main `adding-machine` function needed to take the suffix of the list without the first sublist as input (rather than the entire `rest` of the list). Despite these flaws, this student at least had a largely consistent view of the output type of each function (the erroneous single-zero base case answer notwithstanding). This reflects an understanding that one use of template code can return only one type of output (missed in the solution in Figure 3.3).

```
(define (adding-machine lon)
  (cond [(empty? lon) empty]
        [else (cons (adder lon)
                    (adding-machine (rest lon)))]))

(define (adder lon)
  (cond [(empty? lon) 0]
        [(= 0 (first lon))
              (adding-machine (rest lon))
        [else (+ (first lon) (adder (rest lon)))]))
```

Figure 3.4: Code from student WPI1-STUD1 pulled identified tasks into a separate function

Eleven students (Table 3.2: students who had either **HLP** or **BOTH** entries under **sumelts**) moved the sum-sublist task into a helper function, as in the solution in Figure 3.4. None modified the portion of the list passed on the recursive call, instead using the recursive call verbatim from the template. Overall, 16 students created a helper function (Table 3.2: students with at least one instance of either **HLP** or **BOTH** in any of the tasks) that took a list of numbers as input and included some program tasks.

Even when students realized to create helpers, they often failed to effectively decompose the problem around those helpers: 12 of 25 students created helpers that they never called from their

Table 3.2: Tasks students wrote code for and their location (*AM: within the Adding Machine function, HLP: within a helper function, BOTH: within both the Adding Machine function and a helper function*), with students' final course grade (*NR* is WPI's version of a non-passing grade); bottom sub-table indicate counts of task appearances in locations

Student	**Tasks**				**Final course grade**
	single-zero	**double-zero**	**sumelts**	**buildsumlist**	
WPI1-STUD1	BOTH	-	HLP	AM	A
WPI1-STUD2 [00+]	-	AM	AM	AM	A
WPI1-STUD3	-	-	AM	-	NR
WPI1-STUD4 [00+], [0-00]	BOTH	AM	HLP	AM	C
WPI1-STUD5 [00+], [0-00]	BOTH	AM	BOTH	AM	B
WPI1-STUD6	BOTH	-	HLP	-	B
WPI1-STUD7	BOTH	-	BOTH	-	B
WPI1-STUD8	BOTH	-	BOTH	AM	A
WPI1-STUD9	AM	-	BOTH	-	B
WPI1-STUD10	-	-	AM	-	NR
WPI1-STUD11	AM	-	BOTH	-	B
WPI1-STUD12 [00+], [0-00]	HLP	BOTH	HLP	-	A
WPI1-STUD13	AM	-	AM	-	NR
WPI1-STUD14	BOTH	-	AM	-	A
WPI1-STUD15 [0-00]	HLP	HLP	-	-	B
WPI1-STUD16	HLP	-	-	AM	B
WPI1-STUD17	AM	-	AM	-	B
WPI1-STUD18 [00+], [0-00]	AM	AM	AM	-	B
WPI1-STUD19	-	-	-	-	C
WPI1-STUD20	-	-	-	-	B
WPI1-STUD21	AM	-	AM	-	I
WPI1-STUD22 [0-00]	AM	BOTH	-	AM	C
WPI1-STUD23	BOTH	-	HLP	AM	B
WPI1-STUD24	BOTH	-	BOTH	-	B
WPI1-STUD25	BOTH	-	AM	-	B
Counts of task appearances in locations					
Just in main (Adding Machine function)	7	4	9	8	
Just in helper function	3	1	5	0	
Both in main and helper function	10	2	6	0	

main function (Table 3.3: students with a '**U**' entry). These helpers attempted combinations of the single-zero, double-zero, and sum tasks. This seems a different manifestation of thinking through focals: rather than integrate a focal computation into an existing function (their original template), students tried to put them in separate functions. This is not unreasonable, as each of these three tasks involves traversing a list, and students had been taught to use recursive functions to traverse lists. Of the students who created helper functions, 8 used templates in writing all helper functions while 2 more did so sometimes (Table 3.3), again suggesting a strong HTDP influence. These observations

Table 3.3: Status of helper functions students wrote

Student	Helpers in templates / Total no. of helpers	Composition of helpers into main: [C]omposed, [U]ncomposed	No. of uncomposed helpers and tasks involved
WPI1-STUD1	1 / 1	C	-
WPI1-STUD2	-	-	-
WPI1-STUD3	-	-	-
WPI1-STUD4	1 / 2	C	-
WPI1-STUD5	0 / 1	U	1 - singlezero, sumelts
WPI1-STUD6	4 / 4	U	4 - singlezero, sumelts, listoflist
WPI1-STUD7	0 / 2	U	1 - singlezero
WPI1-STUD8	2 / 2	U	2 - singlezero, sumelts
WPI1-STUD9	1 / 1	C	-
WPI1-STUD10	-	-	-
WPI1-STUD11	1 / 1	U	1 - sumelts
WPI1-STUD12	0 / 3	U	1 - doublezero, singlezero
WPI1-STUD13	-	-	-
WPI1-STUD14	1 / 1	U	1 - singlezero
WPI1-STUD15	0 / 1	U	1 - doublezero, singlezero
WPI1-STUD16	0 / 2	U	1 - singlezero
WPI1-STUD17	-	-	-
WPI1-STUD18	-	-	-
WPI1-STUD19	-	-	-
WPI1-STUD20	-	-	-
WPI1-STUD21	-	-	-
WPI1-STUD22	1 / 1	U	1 - doublezero
WPI1-STUD23	2 / 3	U	1 - singlezero
WPI1-STUD24	0 / 1	C	-
WPI1-STUD25	1 / 1	U	1 - singlezero
	No. of students who wrote helpers: **16**	No. of students with helpers uncomposed into main: **12**	

suggest that upon entering the creation state (after setting up the templates), students resort to building their functions with a characteristic tinkering behavior [9] by patching up the holes in the template with familiar constructs and function calls, even when these result in output inconsistencies and essentially, plan composition problems.

Several students put the same tasks in both the main and helper functions (Table 3.2: last row of sub-table). Task-replication seems to depart from Rist's focal model, which suggests that students would write the focal computations once within their existing code, then build around them. This again reflects students' difficulties in decomposing the problem around the tasks.

3.5.3 Use of Advanced Techniques

Both accumulating intermediate data in parameters and reshaping the data into a list of sublists are advanced patterns that students had not seen in the course (and thus could not have retrieved). Only 1 student attempted to add a parameter for the running sublist sum (WPI1-STUD6). Only 3 attempted to reshape the data, but none did so successfully (Figure 3.5: WPI1-STUD6 , WPI1-STUD8 , WPI1-STUD14). The coding sequences for the latter suggests that as soon as students pulled out tasks to attempt to reshape the data, they proceeded into tinkering in and around these helpers. Students who tried this step (which would have been valuable had it worked) were clearly not thinking through focals, as data reshaping is not a computational task suggested in the problem statement, even though the problem hints at the flattened state of the data.

```
;;make-sub: list[number] list[number] -> list[list[number]]
;;breaks the given list into sub-lists with breaks at 0
(define (make-sub lon compiled)
  (cond[(empty? lon) compiled]
       [(not (= (first lon) 0))
        (cons (first lon) compiled)]
       [else (append compiled (make-sub (rest lon)))]))
```

(a)

```
;; take a lon, when there is a zero, make new list
;; if an element is zero, make new list (end last list)

(define (make-sub-lists lon)
  (cond [(empty? lon) empty]
        [else (if (= 0 (first lon))
                  (list (make-sub-lists (rest lon)))
                  (append (list (first lon)) (make-sub-lists (rest lon))))]))

(make-sub-lists (list 1 2 0 5 6 7))
```

(b)

```
;; ListOfNumbers -> ListOfListOfNumbers
;; The function consumes a ListOfNumbers and produces a ListOfListOfNumbers without the zeros.

(check-expect (list-splitter (list 1 2 0 7 0 5 4 1 0 0 6)) (list (list 1 2) (list 7) (list 5 4 1)))
(define (list-splitter lon)
  (cond
    [(empty? lon) empty]
    [else (list
     (cond
       [(= (first lon) 0) empty]
       [(= (+ (first lon) (first (rest lon))) 0) 500]
       [else (cons (first lon) (list-splitter (rest lon)))]))]))
```

(c)

Figure 3.5: Attempts to write reshaping code by students (a) WPI1-STUD6 , (b) WPI1-STUD8 , and (c) WPI1-STUD14

3.5.4 Findings from the Survey

We had hoped that the survey responses would augment our observations from our analysis of the programming video data. Our analysis of the survey responses, however, revealed that most of the responses were too vague. We thus could not use the survey responses to add meaningful insights towards our findings from our video analysis. Nonetheless, we summarize the survey responses in the

following discussions and present some responses that seem to support our findings from our video analysis. The full set of short-response answers to the survey are in Appendix B.

Question 1: Describe the process of how you got started with the problem

When asked how they started with the problem, most of the participants mentioned starting by following *some* of the design recipe steps, but did not elaborate why they did so or whether or how they connected one recipe step to another. Only two students said they started by writing data definitions, five wrote signatures and/or purpose statements, nine wrote input-output examples/test cases, and ten wrote list templates. One specifically noted that they only used the list template as a starting point, and then defaulted to using a trial-and-error approach:

> WPI1-STUD23 : *I started with the template for a list function, but ended up abandoning that approach. Im* [sic] *not a fan of the templates other than just a very very basic starting point. I never adhere strictly to the template. I just use trial and error and hope that my logic works.*

Six other students mentioned jumping directly into code; this and WPI1-STUD23 's response suggests that some students may not have absorbed the idea that the design recipe may be used as a guided, systematic approach in designing programs, rather than going on an undirected path towards a solution. It also suggests that students' use of the design recipe is primarily mechanical. Three students mentioned thinking about using helper functions, but were not specific about what the roles of these helpers might be in their solution. Four students thought about looking at their notes for examples or previously-seen problems, but were also not specific about what they looked for. Only one student (WPI1-STUD6) described writing a function that reshaped the input into a list of lists.

Question 2: What did you have to look up in your notes?

12 students said that they looked at their notes at some point. Three mentioned searching their notes for templates to use. Six searched their notes or the Racket documentation for built-ins to use and three looked for previous examples or problems, but none of these students were specific about exactly what they looked for or why.

Question 3: Describe when the design recipe was useful.

For this survey question, most students simply wrote about which design recipe step they wrote (*e.g.* writing templates, signature/purpose statements, test-cases, *etc.*), but did not elaborate on how helpful these steps were to them. Four mentioned using the design recipe to "setup" or "layout" their functions, but were not specific about whether they were referring to setting up their code with templates, or their overall plan for a solution. One mentioned that they realized to write the signature and purpose statement after having already started their code, but did not explain further what prompted the realization.

Question 4: Describe a time when you got stuck even when using the design recipe.

14 students said that they felt stuck even when using the design recipe. The responses to this question were overly varied and we could not find even a high-level categorization with which to group the responses. Some of these responses mentioned: stopping the use of design recipe (no explanation why); that the template "*didn't work very well*", which may suggest a recognition of the limits of the template relative to the needs of certain problem tasks; not being sure when to use a helper function; or could not figure out how to "*combine certain parts of a list*".

3.6 Discussion

This study was motivated by an apparent underlying tension between HTDP templates and Rist's model of bottom-up programming (*focal expansion*). Rist's work suggests a cognitive process that students follow on new programming problems: write a core computation for a problem task (the *focus*), then augment the program to produce the data needed for the focal computation. The potential tension with HTDP lies in the sub-expressions that templates introduce: these provide a *context* into which students will place focal computations. That context could either help or interfere with students' thinking as they integrate focals into a larger program. The following subsections summarize our observations from our data.

3.6.1 Students Work Through Core Problem Tasks

Our data suggests that students largely work through problem tasks: they write task-related focal computations on the front elements of the input list, or create new functions for problem tasks. They often appear to retrieve plans, in the form of individual recursive functions for individual tasks (such as summing a list). As such, our students often introduced focals as entire functions, not small expressions (as in Rist). This process of retrieving and reusing familiar plans may have deferred students' entry into creation mode, shifting more burden to plan composition as students attempt to figure out how to compose the code for their retrieved (and implemented) plans.

3.6.2 Students Lacked Schemas and Struggled with Plan Compositions

Our students struggled to compose plans: some failed to adjust the portion of the list being recurred over, others tried to perform multiple tasks with different output types (*e.g.* summing a list and building lists) within the same recursive traversal (rather than accumulating the sum or creating a helper that dealt with a separate ask). In both cases, our students used template expressions verbatim, rather than adjust them to the need of the computation at hand. Given that they had not seen programs that adjusted template code, this behavior is not particularly surprising. More generally, they lacked schemas for the more advanced programming patterns (such as accumulators and reshaping lists), which seems to have affected their ability to compose the plans they were attempting to write, instead retrieving insufficient plans.

This does not necessarily mean that the templates, and the context they provide, interfere more than help students. The recursive calls are still needed, even if on slightly different inputs in some (but not all) cases (*e.g.* an adjusted suffix of the list to recur on). The students in our study had not learned when to decompose problems into multiple instances of templates, or when to adjust certain parts of the template (such as adding a parameter for accumulating values, modifying the terminating case, or adjusting the list to recur on), and the templates failed to help them discover or resolve the issue. Given prior work on the importance of retrievable schemas, HTDP's templates fit squarely within known results on how people program.

3.6.3 Students Decompose Problems On-the-fly

Arguably, one key issue is that students are decomposing the problem *on the fly around the code they have already written*. This arises whether students use HTDP (which prescribes a context for code to live in: the template) or bottom-up programming (in which students' existing code provides the context). If the prior context isn't well-suited to the problem at hand, students will struggle with composition. We hypothesize that decomposing the *problem* up front, into tasks that can be composed cleanly into a solution, should make the actual coding less error prone.

The question then is: can we teach students to effectively decompose problems? Both Rist's work and ours show that students think in terms of core problem tasks. Decomposing problems (rather than code) is about grouping the tasks of a problem into chunks that can reasonably be handled together. What if we could teach students to use concrete examples (which HTDP-trained students write in steps 2 and 4 of the design recipe) to work out problem decompositions? In the case of Adding Machine, for example, a student could start with

```
(adding-machine (list 1 2 0 7 0 5 4 1 0 0 6))
```

then write that this should produce the same answer as

```
(list (+ 1 2) (+ 7) (+ 5 4 1)).
```

Realizing this might suggest specific functions that a student could write to transform the first expression into the second. Something systematic such as this seems preferable to expecting students to just keep experimenting until their bottom-up process hits on a workable solution. Just as students currently internalize schemas for writing code, we might expect they can learn to internalize schemas for decomposing problems through concrete examples.

3.6.4 Tension Between Rist's Focal Expansion Model and HTDP

We suspect that some tension between Rist's model and HTDP arises from differences between functional and imperative programming. Deciding what Rist's model might mean within functional programming took us considerable discussion. Rist's description of "bottom up" references code organization typical of imperative programs: variable declaration and initialization at the top, computation

in the middle, and results and output on the bottom. Functional programs are organized differently: variables are initialized when functions are called, and outputs are typically composed from nested expressions within the middle of a function.

For this study, we chose to interpret "bottom up" as "write contextual code after the focus". Functional programs also tend to decompose problems into several functions, whereas imperative solutions for CS1-level programs often live within a single procedure. This changes the decomposition patterns that students need for problems such as Adding Machine. This suggests that cognitive behavior in creation mode may differ based on the affordances of the programming language at hand.

3.7 Status of Dissertation Research Questions

Our findings from this study provide some answers towards two of our research questions:

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

1. **Students work through core problem tasks.**
   Students wrote code for tasks they *elicited* from the problem statement. Some students also *identified* problem-relevant tasks that are not explicit in the problem statement itself, such as reshaping the input, which may be inspired by aspects of the problem: in the case of Adding Machine, this could be the flattened state of the input.

2. **Students retrieve code-level plans.**
   Students retrieved code structures that they have seen or used. Students retrieved the list template when writing functions that operated on lists and also seemed to retrieve entire functions for familiar tasks, such as the summing function.

3. **Students apply some of the HTDP-prescribed design practices.**
   Students applied *some* of the design recipe steps as they wrote their solutions, specifically: writing signatures, purpose statements, input–output examples/test-cases, and templates.

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

When solving multi-task programming problems, our observations suggest that the students in this study are primarily driven by the following approaches:

1. **Students decompose the problem on-the-fly around code they have already written.**

2. **Students retrieve and use code-level plans verbatim without adjusting them to the need of the computation at hand.**

3. **Students use HTDP-prescribed design practices mechanically.**

Each of these approaches contributed to the students' failures to produce correct solutions for the multi-task Adding Machine problem. *On-the-fly problem decomposition* and the *verbatim use of code-level plans* often went hand-in-hand: when students decompose the problem on-the-fly around code they have already written, they seem to fail to factor in the limits of the code they are composing into their current implementation. Instead, they compose plans (such as templates and functions) into their code *verbatim* without considering the impact of these compositions (*e.g.* output inconsistencies within functions) or the changes that may need to be applied onto a plan to correctly implement a computation (*e.g.* adjusting the recursive calls). The latter could partly be explained by the students' lack of schemas for the programming techniques (*e.g.* accumulators and reshaping lists) they needed to implement some of the tasks; they were thus limited to retrieving the plans they were familiar with, which were not sufficient for the needs of the tasks.

From our limited observations from the video and survey data, how the students used the design recipe to approach the problem seemed to not be helpful to them in overcoming the hurdles they encountered; their use of the design recipe appeared to be primarily *mechanical*. For example, some students wrote only some of the steps, some wrote only a couple of input–output examples that mostly mirrored the example provided in the problem statement and thus did not explore the potential space of inputs that could have informed the design of their functions, and some started with the template and abandoned its use later on, opting for a "*trial-and-error*" approach. None of the students discussed (in the posttest survey) about how the design recipe steps informed the design of their solutions (or at least, none of them talked about it concretely, if any). We hypothesized that if students used concrete examples to work out problem decompositions (as in the example in Section 3.6.3), doing so may help them identify tasks or functions that could help solve the problem.

Chapter 4

Study: Understanding How Program Design Skills Evolve During a CS1 Course

Background and Context: Our prior study (Chapter 3) lacked richer data on which we could draw more nuanced observations around HTDP-trained students' thought-processes when designing solutions for multi-task programming problems. We thus shifted our methodology towards collecting qualitative student data *in situ* to capture students' design processes at a more granular level.
Objective: Our main goals for this study are to explore students' design processes when solving multi-task problems (building on our prior study), as well as to understand how their program design skills evolve through a CS1 course (Section 4.1).
Method: We conducted a longitudinal study by interviewing students about their design practices every two weeks during a CS1 course. Two sessions reviewed students' homework submissions, while the third asked students to solve the Rainfall problem while thinking-aloud. We then coded their data to develop a framework for assessing the evolution of program design skills (Section 4.2).
Findings: We developed (1) a multi-strand SOLO taxonomy that captured students' performance levels within a set of design skills and (2) a collection of factors that students raise when discussing designs of programs (Section 4.3). We also conducted an initial validation of the taxonomy by using it to categorize data from students beyond the pool used to develop the taxonomy, as well as checked whether our taxonomy aligned with how other HTDP instructors assessed student design skills (Section 4.7).

A version of this chapter is published in the following venue:
[20] Francisco Enrique Vicente Castro and Kathi Fisler. 2017. Designing a Multi-faceted SOLO Taxonomy to Track Program Design Skills Through an Entire Course. In Proceedings of the 17th Koli Calling Conference on Computing Education Research (Koli Calling '17), ACM, New York, NY, USA, 10–19. DOI: https://doi.org/10.1145/3141880.3141891

4.1 Building on the Adding Machine Study

One of the challenges from the previous study (Chapter 3) was that it lacked richer data on which we could draw observations about *how* students were thinking around the multi-task programming problem (*i.e.* Adding Machine) they were solving, *why* they struggled during their programming process, or *how* they tried to get unstuck. Our data limited our interpretations to *what* we saw the students do and did not enable us to identify *why* they did so, or *how* they went about their process *cognitively*. While the survey responses provided *some* supporting data towards our interpretations, these did not add any more meaningful insight about how the students were thinking about the problem or their use of the HTDP design recipe.

We thus decided to shift our methodology for this study to one that enabled us to collect data on what students did, alongside what they were thinking. One of the main goals of this project is to understand students' thought-processes around how they approached multi-task programming problems when they have been taught a systematic process of doing so through the design recipe. We turned towards using think-alouds, which has been used in early computing education studies [3, 14, 117], and continue to be used in more recent studies to access students' thought-processes *in situ*, or while engaged in computing-related activities [41, 97, 132, 133]. Additionally, we decided to track students' progression over multiple study sessions throughout an entire CS1 course, instead of just one. We envisioned that this would give us some insight towards how students' thinking around the program design techniques they are taught might evolve over time; for example, we wanted to see whether students' use of the design recipe became more insightful over time, or if they remained mechanical in its use.

4.2 Study Design and 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 a CS1 course. We were also interested in identifying underlying factors that might impact how effectively students were using the recipe.

4.2.1 Study 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 specific homework problems that 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 multi-task programming problem to try writing from scratch while thinking aloud. After solving the problem, we had them reflect on their work, similar to the activity in sessions 1 and 2 asking

Table 4.1: 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

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 research design [140] to help clarify researcher observations during the think-aloud portion in addition to the interview questions used in sessions 1 and 2. Table 4.1 summarizes the topic and activities done in each session. We discuss the specific problems used for each session and our rationale for selecting the problems in the following subsection (Section 4.2.2). Appendix C.1 lists the questions through which we asked students to reflect on their approaches and to compare solutions.

I conducted all of the sessions, each lasting roughly an hour. Sessions occurred every two weeks starting after the first exam[1]. The instructor for the course was not part of the research team. 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 work.

4.2.2 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 (*i.e.* 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. Following are the exact wordings for the specific programming problems used in each of the first and second sessions wherein students were asked to describe their approach to solving

[1]Courses at WPI run at intense pace for 7 weeks

each problem; Appendix C.2 shows the full homework problem sets from which these problems were drawn.

- **Session 1 interview problem: Campaign air cost**
  Write a function `campaign-air-cost` that consumes a list of ads and the name of a politician and produces the total air-cost of all political ads for that politician.

- **Session 2 interview problem: Dissolved oxygen (DO) warning**
  Cold-water fish such as trout and salmon are more vulnerable to low DO levels than are warm-water fish. Although they can survive for a short time in water with a DO level of less than 5 mg/L, they will die in water below 3 mg/L. Develop a function `cold-water-fish-warning` that consumes a river system and produces a string. If all rivers in the system have DO levels at 5 mg/L or above, the string produced is "OK". If any of the rivers in the system have a DO level below 3 mg/L, the function produces the string "Deadly". Otherwise, the string produced is "Marginal". You'll need to use helpers here. Think about where you'll want to use the templates.

Rainfall [127] (session 3) was different in having multiple traversal-based subtasks (*e.g.* 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 exact wording for Rainfall follow, along with its most common solution approaches.

> **Rainfall**: *Design a program called* `rainfall` *that consumes a list of numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). If you cannot compute the average for whatever reason, return -1.*
>
> Example: `(rainfall (list 1 -2 5 -999 8))` should produce 3.

Common solution structures for solving Rainfall include:

1. **Clean first**
   Produce an intermediate data structure of non-negative values truncated at the sentinel; sum and count the cleaned data, check for zero-division, and finally compute the average.

2. **Process-multiple**
   Traverse the input twice, once to sum and once to count, ignoring negatives and halting at the sentinel (or the empty-list); then check for zero-division and compute the average.

3. **Single-traversal**
   Traverse the input once, updating sum and count on each non-negative input, then check for zero-division and compute the average upon reaching the sentinel (or the empty-list).

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 or driving 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. Session 2 used the same problem as the interview problem to talk about alternative solutions, while session 1 used the following problem (from the same homework problem set as the campaign air cost problem):

- **Session 1 alternative solutions problem: Find ads**
  Write a function `any-ads-for?` that consumes a list of ads and a String representing a product name or politician's name, and produces a Boolean. The function returns true if the list contains any ads for the given product or politician.

The problems also embodied different suggestions towards testing: the first session problem, *Campaign air cost*, looked for a specific name, which suggests covering data with and without the name; the second, *Dissolved oxygen warning*, 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.

4.2.3 Participants

The participants came from the Fall 2016 offering of *CS1101: Introduction to Program Design*, taught using HTDP in Racket. We requested volunteers before the first exam. In a recruitment survey (Appendix C.3), 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[2]). We had six males and seven females (all

[2]The full study extended over both courses. The study described in this chapter is the first phase. The second phase was conducted with students as they took CS2 in the following school term with the goal of understanding how design practices transferred to succeeding programming courses; the second phase is beyond the scope of this dissertation.

self-identified). 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).

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 assignments), 4 understood the topics well enough (assignments were a bit challenging), and 1 found both the topics and course assignments fairly challenging. Appendix C.4 shows the full student survey responses.

4.3 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 study.

4.3.1 Identifying Skills and Skill Progressions

After the first session, we open-coded [28, 69] 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[3] to cluster student comments into themes (Figure 4.1). Given the questions that we asked students about their work (Appendix C.1), 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 4.2 summarizes the resulting themes.

Comments on some themes suggested a progression within a core skill (Table 4.2a) resembling increasing levels of conceptual complexity akin to SOLO levels [13]. 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 (*leveraging 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 4.2b — we discuss these in Section 4.8.4). Figure 4.2 summarizes our iterative open-coding and taxonomy development process. The actual taxonomies for SOLO-amenable themes appear in Table 4.3; we discuss each theme in turn.

[3] Psychologists originally employed *card sorting* to study how people organize knowledge [144]; it has become a popular user-centered design method for discovering optimal organizations of information for websites or software [116, 144].

Figure 4.1: Open-coding student transcripts facilitated through thematic card sorting

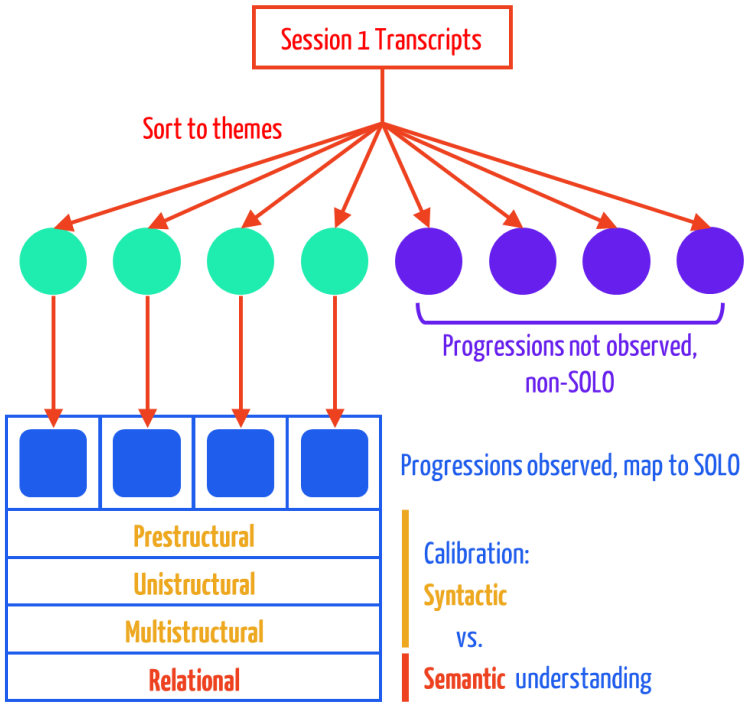


Figure 4.2: Process summary for iteratively developing the multi-faceted SOLO taxonomy. Discussions between authors during each iteration refined the themes and descriptions for the taxonomy levels.

Table 4.2: 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
Composing expressions within function bodies	Knowledge of writing functions and the composition of expressions (*i.e.* built-in/user-defined functions) to build function bodies (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)
Leveraging 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 (eg readability or 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 judgements towards aspects of the programming process, experience, or learning

Table 4.3: Multi-strand SOLO taxonomy of observed program design skills. We omit the *Extended abstract* level as none of our students reached that level for this study

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.

Skill strand: Methodical Choice of Tests and Examples

Our study questions (Appendix C.1) 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:

WPI2-STUD3 : *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)

WPI2-STUD13 : *[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 WPI2-STUD13 talked about the space of tests in the context of the problem (seeing a collective purpose for the tests), WPI2-STUD3 spoke only about individual purposes of tests. Our progression for testing captures the depth at which students see tests collectively.

Skill strand: Composing Expressions Within 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:

WPI2-STUD3 : *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.*

WPI2-STUD3 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:

WPI2-STUD3 : *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 WPI2-STUD6 's comment, which concretely explains how the return value of the helper function relates to the calling function:

WPI2-STUD6 : *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.*

Skill strand: 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:

WPI2-STUD13 : *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 *composing expressions within 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, WPI2-STUD13 '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 WPI2-STUD4 (when comparing two solutions for a problem) shows a lower-level variation of this:

WPI2-STUD4 : *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 function, 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.

Skill strand: Leveraging 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 (examples and 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 WPI2-STUD6 in the excerpt below, worked through the recipe representations mechanically, while others, such as WPI2-STUD13 , conveyed relationships among the representations:

> WPI2-STUD6 : *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).*

> WPI2-STUD13 : *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.*

WPI2-STUD13 '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.

4.3.2 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 [66], which does not require semantic understanding to reach a relational

Table 4.4: Analysis of Student Skill Progressions. The abbreviated headings correspond to the 4 design skills in the taxonomy: MTE = *Methodical choice of tests and examples*, CFB = *Composing expressions within function bodies*, DTC = *Decomposing tasks and composing solutions*, and LRF = *Leveraging multiple representations of functions*.

Student	Session	MTE	CFB	DTC	LRF
WPI2-STUD3	1	U	M	U	U
	2	R	R	M	U
	3	U	M	M	U
WPI2-STUD4	1	R	M	U	M
	2	M	M	-	U
	3	U	R	M	M
WPI2-STUD6	1	R	R	R	R
	2	R	R	R	-
	3	R	R	R	R
WPI2-STUD7	1	M	R	M	M
	2	R	R	U	M
	3	M	R	M	M
WPI2-STUD11	1	R	M	U	U
	2	R	R	R	U
	3	R	R	R	M
WPI2-STUD13	1	R	R	R	R
	2	M	R	R	M
	3	R	R	R	M

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.

4.4 Assessing the Taxonomy With Other Student Data

The taxonomy in Table 4.3 arose from our trying to make sense of *isolated* comments that students made during session 1. We did not 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 4.4 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 4.3.

My doctoral advisor (Kathi Fisler) and I 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 and found that the taxonomy applied similarly to those transcripts.

4.4.1 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 4.4.1. *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, WPI2-STUD3 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 4.4.2. *Skill strands vary in the nature of mechanical application and requirement of abstract-level thinking.*

Composing expressions within function bodies (**CFB**) 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 *composing expressions within 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 [117, 119], 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 *composing expressions within 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 [66].

One 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.

4.5 Assessing Students' Design Progression with the Taxonomy

We developed this taxonomy as part of a larger project (see footnote in Section 4.2.3) 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 level of problem complexity.

This study was of the second type. We can thus examine Table 4.4 for insights into how students' skills evolved across our study problems.

Table 4.4 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. Another is that the study problems themselves (not to mention the interview questions) might bias students towards answers that appear to justify a level. We discuss the latter concern in the following section.

4.6 Designing Problem Progressions Around the Taxonomy

Reflecting on the data in Table 4.4 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). In the course that our participants were in, 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. The strand on *methodical choice of tests and examples* conflates these issues, but the strand on *leveraging multiple representations of functions* can help tease these issues apart, as the *relational* level requires students to make insightful connections across the different design practices towards informing the design or structure of their solution. This capability to tease out *mechanical* versus *intentional* use of design practices is another advantage that our multi-strand taxonomy provides. Additionally, assessment designers should create problem sets that also tease apart these differences.

Overall, the data in Table 4.4 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 [85, 139], 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.

4.7 Validation Study with Experienced Instructors

Our initial effort towards validating our taxonomy (Section 4.4) involved checking the taxonomy against the work of students beyond the sample used to develop the taxonomy. Our primary goal for this validation study is to further validate the taxonomy with other HTDP instructors. We then refine the taxonomy descriptions based on findings from the instructor validation.

4.7.1 Validation Rationale

One of our goals with our program design skills taxonomy is to validate it with experts, specifically, with other HTDP instructors. We wanted to check whether our taxonomy captures the same things that other HTDP instructors look for when assessing how HTDP-trained students applied their design skills.

In explaining our approach towards the validation of our work, I describe our use of "validation" in this study, and whenever we use the terminology throughout this document. I've approached this dissertation qualitatively, and more specifically, from an interpretivist paradigm, meaning that our goal with our studies is to generate an *understanding* of our object of study (*i.e.* how HTDP-trained students approach multi-task programming problems) through narratives whose *meanings* and *explanations* are emergent from, and grounded on, the data we collect and the processes we observe (for a more in-depth discussion of qualitative research, the interpretive paradigm, and grounded theory, we defer to the work of Bhattacharya [10], Charmaz [28], Glaser and Strauss [56], Leung [77], Lewis [78], and Whittemore *et al.* [142]). Our taxonomy is a product of our iterative process of coding and triangulating data from the multiple programming artifacts that we collected from the students and developing and presenting thick descriptions of our observed constructs.

Validation in qualitative research can be approached through a variety of methods (Liao and Hitchcock provide an extensive list [79]). We approach our validation study primarily through three methods: *triangulation* (converging evidence from multiple data sources), *audit trails* (comprehensive documentation of all our procedures and data), and a form of *expert checking* (using outside experts to assess our student data). In particular, we use the data we collect from expert checking to understand how our descriptions of the program design skills that we observed (and their various levels of conceptual complexity) align with what actual HTDP instructors (the experts) look for when assessing student design work. We triangulate across the instructors' explanations, comments, and transcript annotations, as well as detail our data collection and analysis process to develop a comprehensive audit trail. Findings from this validation study could lend evidence towards the taxonomy's usability as a skill-assessment rubric by other instructors, even potentially by those who don't teach with HTDP (since the skills in the taxonomy aren't unique to HTDP). The use of the taxonomy as a skill-assessment rubric in actual CS1 courses is beyond the scope of this work; we discuss future work on the potential use of the taxonomy and the use of intercoder reliability to evaluate its usability as an assessment scheme in Section 8.5.3. Lastly, as we used the taxonomy, we noted that there are cases wherein we found it difficult to use the taxonomy to capture why some students failed to correctly solve Rainfall. We discuss this in more detail in Section 4.7.4.

4.7.2 Data Collection and Logistics

We sent a call-for-participants in a mailing list for HTDP instructors to invite volunteers for our validation study. Interested instructors filled out a survey (Appendix C.5) where they provided information on their experience in teaching HTDP (number of years teaching HTDP courses, levels of education taught, HTDP topics taught). Ten instructors initially volunteered for the validation study.

Validation instruments

We sent each of the ten volunteer instructors two sets of validation materials; each set had one student think-aloud transcript (with the accompanying code submission and scratch work) and one worksheet. We designed a worksheet for the instructors to use for rating HTDP-trained students' program design skills. We pilot-tested the worksheets with three past HTDP teaching assistants (who were also part of our research group) and used the pilot-testers' feedback to revise the worksheets (the full worksheet is in Appendix C.7). We selected three students from the pool of students that we used to design our SOLO taxonomy (Section 4.2.3), based on their first-exam performance (WPI2-STUD11 - A, WPI2-STUD6 - B, and WPI2-STUD3 - C) and assigned each instructor two students. We distributed the assignment of students to the instructors so that we could get an almost equal number of responses for each student. Table 4.5 lists the assignment of students to instructors.

The instructors were asked to read the transcripts and code of each student assigned to them and use the worksheets to assign a score from a five-point scale (0: *None*, 1: *Little to none*, 2: *Fragmented, but present*, 3: *Strong*, 4: *Applies beyond current problem*) to each of the four design skills in the taxonomy, with justification. In the worksheet, the instructors were given a description of the skills, but not the full taxonomy (*i.e.* the SOLO levels of conceptual complexity). The instructors were also asked to describe skills or factors they would have considered in their rating that were not covered by the four design skills. From our initial pool of 10 instructors, 7 instructors followed-up and submitted back their filled-out worksheets. The recruitment survey responses of the 7 instructors who followed-up are in Appendix C.6. All the instructors who submitted responses indicated having taught HTDP for more than 3 years; four attended an HTDP workshop in the past and 2 are self-taught.

Table 4.5: Students assigned to each instructor; 2 students are assigned to each instructor, indicated by check-marks

Instructor	WPI2-STUD11	WPI2-STUD6	WPI2-STUD3
INSTRUCTOR1		✓	✓
INSTRUCTOR2	✓		✓
INSTRUCTOR3	✓		✓
INSTRUCTOR4	✓	✓	
INSTRUCTOR5	✓	✓	
INSTRUCTOR6	✓	✓	
INSTRUCTOR7		✓	✓
Total	5	5	4

4.7.3 Analysis and Coding

We wanted to understand whether our skills taxonomy captured the same skills or factors that HTDP instructors looked for when assessing students' design skills. We thus coded the instructors' responses to identify whether or not their rating descriptions for each skill were covered by, or similar to, our taxonomy descriptions. We iteratively analyzed the rating descriptions by marking the descriptions with the following codes:

- **C**: Covered - descriptions don't deviate from the current description of the skill
- **U**: Uncovered - Descriptions are not within scope of the skill, but specific about a concept that is currently *not* in the overall taxonomy
- **P**: Partial - Descriptions are about or related to the skill, but not clear about the skill level being talked about (*i.e.* vague, but related description)
- **M**: Mismatch - Descriptions match another skill's descriptions
- **ND**: No data - No data/description/justification provided

For example, INSTRUCTOR5 's description of WPI2-STUD11 's application of the *decomposing tasks and composing solutions* skill below is coded as *C*, citing her delegation of tasks into separate functions and their appropriate composition to solve Rainfall:

> INSTRUCTOR5 : *The student has correctly composed functions to solve subproblems. Three useful subproblems/tasks are identified: summing a list of numbers, shortening the input list to rainfall, and getting the length of a list of numbers. The functions to solve these subproblems are effectively used in the rainfall function.*

INSTRUCTOR3 's description (also for *task-decomposition*) for WPI2-STUD3 is marked as *C* with a *U* (with a note on what construct is uncovered). He comments on how the student did not decompose the problem in the first place and relates this to the student's lack of a clear sense of the limitations of the pattern she used (the student overused the list template):

> INSTRUCTOR3 : *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 hammer to hold, and whether it can do the whole job. Until you know the patterns well, you don't know their limitations.*

Table 4.6 shows the results of our analysis of the instructors' worksheet responses; the full verbatim responses are in Appendix C.8. Table 4.6's abbreviated headings for the design skills follow that of Table 4.4: MTE = *Methodical choice of tests and examples*, CFB = *Composing expressions within function bodies*, DTC = *Decomposing tasks and composing solutions*, and LRF = *Leveraging multiple representations of functions*. In our analysis, we did not focus on the numeric ratings (see Section 4.7.2) the instructors provided as the instructors may attribute their own meaning to the numeric ratings: for example, instructors may have different numeric ratings for the skills, but have similar justifications, or conversely, have similar numeric ratings with different justifications. Instead, we are interested in whether our skill (and skill level) descriptions capture what instructors look for when assessing

Table 4.6: Analysis of instructor responses

Instructor	Student	MTE	CFB	DTC	LRF
INSTRUCTOR1	WPI2-STUD3	C	C	P	P
	WPI2-STUD6	C	C	P	P
INSTRUCTOR2	WPI2-STUD3	C	U: Lacks understanding of accumulator pattern	P	C
	WPI2-STUD11	C	U: Conceptual understanding of language constructs	C U: Pattern selection in the context of a plan	C
INSTRUCTOR3	WPI2-STUD3	C	C U: Lacks insight on limits of template pattern used	C U: No clear sense of scope or boundaries of patterns learned	C
	WPI2-STUD11	C	U: Conceptual understanding of language constructs	C	C
INSTRUCTOR4	WPI2-STUD6	C	M: DTC (P)	P	P
	WPI2-STUD11	C	M: DTC (P)	ND	P
INSTRUCTOR5	WPI2-STUD6	C	C	C	ND
	WPI2-STUD11	C	C	C	U: Code style choice U: Conceptual understanding of language constructs
INSTRUCTOR6	WPI2-STUD6	C	C U: Mechanical use of accumulator pattern	C	C
	WPI2-STUD11	C	U: Code style choice	C	C
INSTRUCTOR7	WPI2-STUD3	C	P	P U: Used accumulator pattern	P
	WPI2-STUD6	C	P M: DTC (P)	P	P

students' design skills; our analysis thus focuses on a description-level comparison. Some cells have multiple codes attached to them: this means that the instructor's descriptions touched on multiple factors. For example, INSTRUCTOR3 's descriptions for *CFB* for WPI2-STUD3 were already covered by our taxonomy descriptions (*C*), but also mentioned something related to *DTC* (*M: DTC (C)*), and described an uncovered aspect that involves template patterns (*U: Lacks insight on limits of template pattern used*). *Mismatch* entries such as *M: DTC (P)* means that the description matched another skill's descriptions that would have been covered "*Partially*" (*P*) within the appropriate skill.

4.7.4 Validation Study: Results and Discussion

Overall, the instructors' descriptions for the *MTE* skill do not deviate from our taxonomy's skill descriptions. Most of the instructors' descriptions for *CFB* were captured by our taxonomy's descriptions, though there are a few instances (three) of instructors also describing aspects that were more related to our descriptions in *DTC*. Two of these cases were from INSTRUCTOR4 who talked about using helper functions, which is partially related to *DTC*; the instance from INSTRUCTOR7 just vaguely mentioned "*decomposing the problem*" in their description. These descriptions were too vague and uninformative (hence the additional *(P)*-codings) to tease out why the instructors mentioned these in their justifications. Finally, our taxonomy also captured most of the instructors' descriptions for *DTC*, and almost half of the instructors' descriptions for *LRF* were also within our descriptions. We discuss the *U*-, *P*-, and *ND*-coded items in the following subsections.

Developing a Skill Strand on Pattern-use

Common across *CFB* and *DTC* are instances of *U*-coded descriptions relating to the use of *patterns*, for example, the use of the list template and accumulator patterns. The instructors' descriptions talked about the selection of appropriate patterns and recognizing the limitations of the selected patterns relative to a plan. As we discussed the refinement of our skills taxonomy based on our observations from the instructor descriptions, we recognized that the descriptions on the use of patterns could be added as a skill strand. Using the pattern-related descriptions as our framing, in addition to the SOLO levels of conceptual complexity, we developed a new skill strand bottom-up (Figure 4.3) by developing explanations towards the erroneous Rainfall cases from our student data. Our resulting SOLO taxonomy for the skill, *Meaningful Use of Patterns* (MUP), is in Table 4.7.

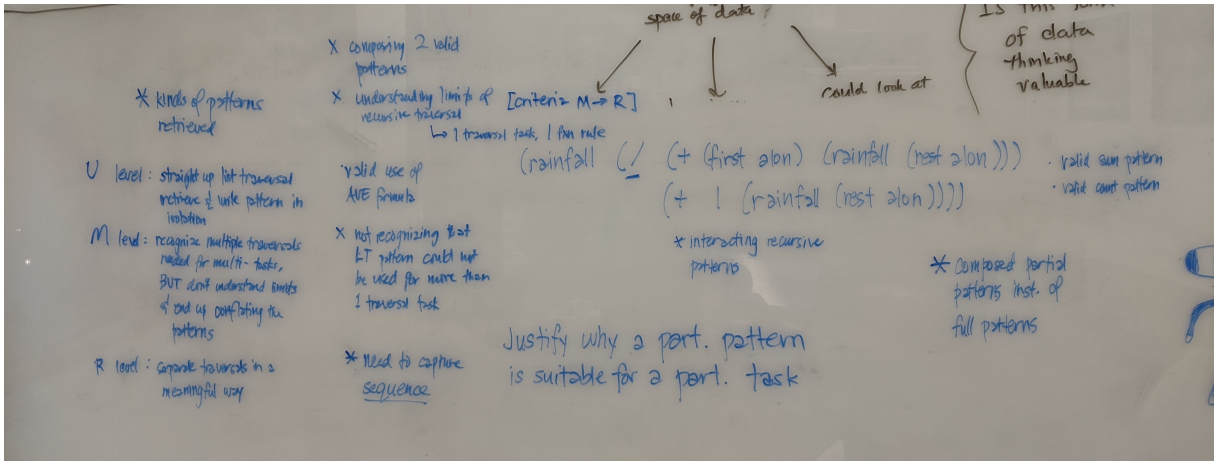


Figure 4.3: Developing a skill strand on pattern-use from instructor and student data

Table 4.7: The SOLO taxonomy for the skill strand on the *Meaningful Use of Patterns*

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)

As we've mentioned earlier, there were erroneous Rainfall cases for which we found it difficult to use the taxonomy to explain *why* the students failed to solve Rainfall. A specific case of this is with students who generated the following Rainfall code (or some similar variant):

```
(define (rainfall input)
  (cond [(empty? input) ... ]
        [(cons? input) (/ (+ (first input) (rainfall (rest input)))
                          (+ 1 (rainfall (rest input))))]))
```

Our observations suggested that the students who came up with this code were thinking about the problem's task-components *and* recognized that the tasks needed their own traversals (*relational*: *decomposing tasks and composing solutions*). Additionally, the function body itself is syntactically correct (*relational*: *composing expressions within function bodies*). While it is clear that the function is incorrect, neither of these two skills clearly explains why, or more so, why the student got stuck thinking along this line of work. This scenario is captured by the *multistructural* level for the *MUP* skill: students recognized the need for separate traversals for each task, yet fails to recognize the limit of the list template pattern relative to the tasks that they identified.

Having developed the new skill strand, we re-coded the instructor responses to determine how many of the previously *U*-coded (*Uncovered*) entries are now captured by the new skill. Table 4.8 shows our re-coded analysis. All of the previously *U*-coded entries relating to pattern-use have been replaced with a *C* code under the new *MUP* column for the new *meaningful use of patterns* skill; empty cells under this column simply denote that the skill was not mentioned by an instructor for a particular student, which shouldn't be surprising as this was a new skill that we had not asked about in the study worksheets. Comparing the re-coded analysis with the previous analysis, it was reasonable for the instructors to have entered their descriptions about code patterns under the *CFB* skill as this was the only skill in the taxonomy that talked about code. Prior descriptions about code patterns that were previously under *DTC* related how the selection of patterns influenced the composition of tasks; we discuss this further in our follow-up study in Chapter 5.

Table 4.8: Re-coded analysis of instructor responses with the MUP skill

Instructor	Student	MTE	CFB	DTC	LRF	MUP
INSTRUCTOR1	WPI2-STUD3	C	C	P	P	
	WPI2-STUD6	C	C	P	P	
INSTRUCTOR2	WPI2-STUD3	C	ND	P	C	C
	WPI2-STUD11	C	U: Conceptual understanding of language constructs	C	C	C
INSTRUCTOR3	WPI2-STUD3	C	C	C	C	C
	WPI2-STUD11	C	U: Conceptual understanding of language constructs	C	C	
INSTRUCTOR4	WPI2-STUD6	C	M: DTC (P)	P	P	
	WPI2-STUD11	C	M: DTC (P)	ND	P	
INSTRUCTOR5	WPI2-STUD6	C	C	C	ND	
	WPI2-STUD10	C	C	C	U: Code style choice U: Conceptual understanding of language constructs	
INSTRUCTOR6	WPI2-STUD6	C	C	C	C	C
	WPI2-STUD10	C	U: Code style choice	C	C	
INSTRUCTOR7	WPI2-STUD3	C	P	P	P	P
	WPI2-STUD6	C	P M: DTC (P)	P	P	

Other Design Factors Instructors Identified

Other design factors that instructors identified in their responses (remaining *U*-coded entries) include:

1. **Code style choices**: Factors pertaining to coding "best practices" that may be tied to *quality attributes* of code such as readability or conciseness. Instances mentioned by instructors include:
   a) Defining helper functions within a `local` vs. globally (INSTRUCTOR4)
   b) "Bad style" because of "too many" `cond` cases (INSTRUCTOR2)
   c) Avoiding magic numbers (INSTRUCTOR2)
   d) "Too much" deep nesting of function calls (INSTRUCTOR6)
   e) Calling a function multiple times instead of creating a local variable to store the result of the function call for reuse (INSTRUCTOR5)
2. **Metacognition or self-regulation**: Practices or skills related to metacognition or self-regulation. Several (vague) instances of this were mentioned by one instructor (INSTRUCTOR6):
   a) Communication skills

b) Organization skills

c) Proofreading and revision

3. **Conceptual understanding of language constructs**: Whether students knew what a built-in function was for. Instances of this primarily mentioned about a student not knowing or not being able to explain when to use the `list*` vs. `cons` (INSTRUCTOR2 , INSTRUCTOR3 , INSTRUCTOR5)

These design factors are the same ones we've found students describing or mentioning in their think-alouds or interviews (described in Non-SOLO themes: Table 4.2b). *Quality attributes* (*i.e. code style choices*) seems likely an instructor-specific or subjective criterion; for example, different instructors (and even students) may have different outlooks on what constitutes *readable* code (*e.g.* helper functions defined globally vs. locally). If this is to be part of a rubric for assessing student work, instructors must be specific about how they are rating for this particular aspect (and generally, other *quality attribute* aspects) so that students' skill ratings actually reflect how they are applying a skill, rather than a failure to subscribe to "best practice" rules in class.

Other Observations from the Instructor Data

We classified some instructor entries/responses as *ND* (No data) or *P* (Partial). An example of a (*P*)artial-coded response is: "*Wrote contract and purpose for main function, but not examples or test cases.*" (INSTRUCTOR1 on WPI2-STUD3 's application of the *LRF* skill). This is *P*-coded because while there was a mention of the use of specific design recipe components, there were no concrete explanations or descriptions of the extent to which the student connected these artifacts to each other or their relationship to the student's overall solution. We interpret this as a research instrument problem, similar to the response problems we encountered with our *Adding Machine* study survey in Chapter 3 (Section 3.5.4). If the validation study was instead conducted with a semi-structured interview protocol (even possibly a think-aloud), we expect that missing responses or problems with the lack of detail in some of the instructors' descriptions could have been addressed with additional prompts; this is one of the main reasons we shifted towards think-aloud and interview approaches, as we wanted deeper insight into our study participants' thought-processes (*in situ*).

Finally, there are a few *M*-coded entries of instructors whose descriptions did not seem to align with the skill they wrote them in. All three cases were from 2 instructors who wrote more *DTC*-related descriptions within *CFB*; these two instructors had other descriptions on other skills that were coded *P* because of the vagueness and incompleteness of their entries (one of them also did not write in any explanation under *DTC*). As it stands, these two instructors' data were not as useful to our overall analysis (including INSTRUCTOR1 whose data also had many *P*-codings) due to their lack of detail. On closer inspection, it's not evidently clear what their entries (see Appendix C.8) suggest about the *CFB* skill: on both student cases, INSTRUCTOR4 simply mentions the use of helper functions, while INSTRUCTOR7 simply mentions that the "*student should try more ways of decomposing the problem*". Both of these clearly relate to *DTC*, but given the lack of detail in their entries, we can't draw on any

deeper conclusions about what the impact of this is with our current descriptions for the *CFB* skill. If we look at the worksheet that the instructors were given (Appendix C.7), the description for the *composing expressions within function bodies* skill did not mention anything about helper functions (although it does mention the composition of expressions). These findings may point to two things: (1) the instrument may need to have a clearer distinction between the syntactic composition of helpers vs. the allocation of tasks to helpers, or (2) that the instructors' descriptions may actually just have been referring to the syntactic compositions of the students' helper functions, in which case (had the explanations been more detailed), the descriptions would have been coded as *C*overed.

Overall, these gaps in our validation study also means that our validation work through expert-checking with instructors was not as extensive as we would have liked, so whether the current state of our taxonomy indeed captures what HTDP instructors look for when assessing HTDP-trained students' program design work is still an open problem. We do find, however, that the instructors who provided detailed responses had explanations and descriptions that largely aligned with our own taxonomy descriptions, suggesting that our taxonomy is at least on the right track. In fact, it is these detailed descriptions that enabled us to capture and describe the new *MUP* skill, which we had not captured in our prior analyses. Future work on refining the taxonomy should consider the use of think-aloud or interview protocols with instructors to delve deeper into the instructors' reasoning or thought-processes as they assess student work. We discuss these threats to validity and open questions further in Sections 8.5 and 8.6.

4.8 Discussion

This study 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 of programs. The idea of a multi-strand taxonomy is one of the main contributions resulting from this study. A multi-strand taxonomy is valuable because it (1) captures inter-related nuances while respecting that (2) 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 is an interesting and important analysis for future work.

4.8.1 Meaningful alignment of skills through syntax vs. semantics

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. In our work, we utilized 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; this warrants further study to explore what these might look like for program design.

4.8.2 Using a SOLO taxonomy for longitudinal skill assessments

Our work also raises questions about how to use a SOLO taxonomy to assess progress over a longer period (*i.e.* throughout a course) 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.

4.8.3 Constructing a data-grounded theory for program design

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 *conceptual 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. We have also validated the taxonomy with other HTDP instructors, finding that most of their descriptions largely aligned with the taxonomy descriptions we defined. We also refined the taxonomy through the addition of a new skill strand on pattern-use that we were not able to previously capture in our initial development of the taxonomy. The gaps in our validation study (Section 4.7.4), however, suggests that the taxonomy still needs further work in the refinement of its definitions.

4.8.4 Other factors that may affect program design

Finally, we want to account for issues that students raised during the study sessions which did not lend themselves to SOLO-esque progressions. Table 4.2b summarizes these issues, which include concepts such as readability, efficiency, and value judgments about the design techniques covered in

the course; the HTDP instructors in our validation study also raised similar aspects. 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 to interpret drops in demonstrated skill levels across multiple assessments.

4.9 Status of Dissertation Research Questions

Our findings from this study provide new answers to some of our research questions, as well as updates towards our previous answers (Section 3.7):

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

> We found in our previous study that HTDP-trained students primarily demonstrated the following design practices and skills when solving a multi-task programming problem:
>
> 1. Students work through core problem tasks.
> 2. Students retrieve code-level plans.
> 3. Students apply *some* of the HTDP-prescribed design practices

Our collection of richer data *in situ* through think-alouds and semi-structured interviews with students allowed us to describe the extent to which students demonstrated the above design practices and skills:

1. **Working through core problem tasks**
   Some students start their programming process by describing problem-related tasks they identify and articulating an overall plan around those tasks; these students usually write their code in the context of their task-level plan. Some do so on-the-fly as they write code; some of these students seem to eventually focus solely on their code and lose sight of the tasks or their overall plan. When asked to describe their program, some students would describe the tasks embodied by their code and how these tasks inter-operated with each other; others fail to articulate the delegation of tasks into the different pieces of code they wrote, instead focusing on the code-specific mechanisms of their program. We captured the differences in complexity with which they described working on problem-tasks in our taxonomy's *decomposing tasks and composing solutions* skill strand.

2. **Retrieving and using code-level plans**
   When asked to describe their program, some students focused primarily on the code-specific mechanisms of their program. In general, all the students were able to correctly describe the low-level syntax structure and evaluation mechanisms of the expressions and function calls in their code. Student behaviors relating to the writing and composition of expressions to build

functions are captured in the *composing expressions within function bodies* skill strand.
When working with a multi-task problem such as Rainfall, most students retrieved the list template and populated the template with code for problem-tasks. Some, however, failed to recognize the limitations of the template pattern in the context of the traversal tasks they are working with, inappropriately conflating multiple traversal tasks in a single template. Some students who struggled to use an accumulator pattern for their functions seemed not to realize that tasks can be allocated to multiple accumulators. How students worked with the patterns they retrieved are captured in the taxonomy's *meaningful use of code patterns* skill strand.

3. **Applying HTDP-prescribed design practices**
Some students simply copied the example/test-case in the Rainfall problem statement, while some explored a broader range of inputs and attempted to write interesting test-case scenarios. A few explicitly mentioned designing their functions based on potential behaviors they identified from their suite of test-cases (as opposed to just writing multiple test-cases that illustrated essentially the same scenarios, just with different values). The different levels with which students wrote and used examples and test-cases in the design of their programs are described in the taxonomy's *methodical choice of tests and examples* skill strand.
Some students immediately jumped into writing their code (usually starting with the list template) and did not follow or use the design recipe at all. Some used some of the design recipe steps and articulated some relationship between the programming artifacts produced from each step, such as how test-cases relate to some expected behavior of the program or some part of the template code that implements a task. How students talked about the interactions between the different components of the design recipe is described in the *leveraging multiple representations of functions* skill strand.

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

- **Students applied the skills at different levels of conceptual complexity.**
We were able to capture, at a more nuanced detail, the gradations at which students were applying the program design skills that we observed them demonstrate. Our synthesis of the data suggested progressions within each skill resembling increasing levels of conceptual complexity akin to SOLO levels. We thus mapped our observations within each skill to a corresponding SOLO level, resulting in the set of SOLO-based skill taxonomies described in Table 4.3. In the case of the skill, *leveraging multiple representations of functions*, we developed this skill taxonomy more top-down as we synthesized design recipe-related observations in the context of the overall HTDP design recipe process.

- **Students evolve in different skills at different paces.**
We applied our multi-faceted skills taxonomy to assess students' use of their program design skills at multiple points during their CS1 course. We found that *students can be at different levels*

for different skills at a given time; they do not necessarily progress through skills simultaneously. This suggests that our skills framework captures the different ways in which different skills develop.

- **Some students show non-monotonic progression of skills.**
  Some students demonstrated some skills at a lower level than they have previously demonstrated. One reason for this might be that *students may not have internalized the skills they displayed at earlier sessions*. For example, students may not have explicitly internalized their use of certain skills as good practices (like writing test cases to cover a problem space), so they may not be consistent in applying them intentionally as they solve programming problems. It is also possible that *the problems in our study may have pushed students towards particular skill levels*. In study session 1, for example, students described their design process on a homework problem for which an earlier problem (on the same homework) provided a usable helper function. This could have prompted students to make comments at a higher *task-decomposition* skill level than they would have if they solved the problem unscaffolded. Lastly, the drops in skill level may reflect the *level of problem complexity at which students can apply their skills*.

Other lessons learned

While not necessarily skills or practices, we also found factors that students seem to raise/consider in their programming process; these factors could potentially affect the level at which students demonstrate the skills we identified:

1. **Quality attributes ("-ilities")**: Properties or criteria that characterize code or coding practice (*e.g.* readability of code: breaking down code to make it easier to read)

2. **Knowledge recalled**: References to knowledge recalled/used in programming that were either learned from the course or before the course (*e.g.* recalling primitives from other languages and looking for its equivalent in the current language)

3. **Metacognition**: References to one's cognitive processes or metacognitive behaviors (*e.g.* self-regulating behaviors such as rewriting templates to reinforce mastery)

4. **Value judgments**: Value judgments towards aspects of the programming process, experience, or learning (*e.g.* dislike towards writing tests because it feels repetitive; finding purpose statements helpful for knowing what a piece of code does)

Chapter 5

Study: Exploring How Novice Programmers Navigate Viable Schemas

Background and Context: Prior planning studies of Rainfall have not explicitly discussed the pedagogic contexts and schemas that inform how students plan for Rainfall, but have observed students produce a common high-level structure. In contrast, students in our HTDP-based courses are typically exposed to multiple viable solution structures for Rainfall. Our context thus provides an opportunity to explore how novices navigate multiple applicable schemas.

Objective: We focused on a subset of our data from our second study (Chapter 4) to explore how students navigate the schemas they have seen to solve the (multi-task) Rainfall problem. (Section 5.1)

Method: We constructed qualitative narratives that describe what drove students to select and switch the schemas they used, focusing on interactions between students' schema-use and how they think about the task-components of Rainfall. (Section 5.3)

Findings: Our findings highlight interactions between how students use and think about about the schemas they've seen and how they think about the task-components of Rainfall. We also describe how the class examples and activities instructors choose to teach patterns may potentially affect how students understand and use these schemas. (Section 5.4)

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

5.1 Exploring How Students Navigate Multiple Schemas

This study continues our exploratory work on understanding students' programming processes towards solving multi-task problems, in the context of HTDP. Here, we explore the following research question:

> **STUDY-RQ1.** When novice programmers have seen multiple schemas (as part of a class) that might apply to a problem, how does their solution emerge and evolve?

We approached this question by constructing qualitative narratives of four students' attempts at solving the Rainfall problem. We describe the factors that drove students to select, switch, and apply the program schemas they know.

Our analyses in this study focuses on the Rainfall problem [127], which has become a benchmark in computing education. Part of the motivation for this study is that Rainfall appears straightforward, while having non-trivial underlying complexity. Most existing work on the challenges of Rainfall was conducted in the context of imperative programming [122, 126, 127, 137]. Some researchers have studied Rainfall with students who used functional programming [52] (similar to what the students in our study use), but they have not reported specific challenges that arise when students attempt Rainfall in this context. Given that different programming languages have different idioms and affordances, a better understanding of how students solve—and struggle with—Rainfall in different pedagogic contexts and programming languages will enhance our understanding of this deceptively interesting programming problem.

The functional perspective is particularly interesting because students who learn functional programming are typically exposed to *multiple viable solution structures* for Rainfall. Studying how students approach Rainfall with functional programming thus provides an opportunity to explore how novice students navigate multiple applicable schemas, each of which they may only partly understand from CS1. Currently, none of the theories of how novice programmers construct their solution (discussed in Chapter 2) addresses what happens when novices have weaker knowledge of multiple viable schemas, or how novices switch schemas mid-process.

5.2 Study Design

5.2.1 Participants

We selected four students from the participant pool of our second study (Chapter 4); we selected these students to reflect variety in course performance (as of the first exam), prior experience, and the structure of students' final solutions. Table 5.1 gives an overview of the students we selected for this narrative-based study[1].

5.2.2 Analysis: Narrative Construction

My advisor, who is an experienced HTDP instructor (not the instructor for this instance of the course), constructed the narratives from the typed think-aloud and interview transcripts[2]. I reviewed the

[1] See Appendix C.4 for the complete pool of students from Study 2
[2] See Section 4.2 for the full data collection process for the think-aloud and interview sessions

Table 5.1: Participant overview. The first exam was in course week 3.

Student (Gender)	Prior Programming Experience	First Exam
WPI2-STUD3 (F)	C++, Online courses	71 (C)
WPI2-STUD6 (M)	Java, Python, Ruby, Self-study	87 (B)
WPI2-STUD7 (M)	Python, JavaScript, Java, HTML5, CSS, PHP, Self-study, AP class, High school class, Online courses	89 (B)
WPI2-STUD11 (F)	None	93 (A)

narratives for accuracy, based on the transcripts, code, and my field notes collected from Chapter 4. We divided the work this way so that the narratives would reflect the pedagogy and learning of HTDP and the students' programming process more than their personalities. My advisor does not know the identities of the students.

The narrative methodology used in this study is influenced in part by the narrative analysis method used by Whalley and Kasto in their investigation of novices' code writing strategies [140]. They developed descriptive accounts of how students used existing schema to write code from think-aloud and interview data. We also draw on ideas from grounded theory [69], in terms of the narrative reconstructions that describe how students varied in how they chose constructs, patterns, or techniques to build their Rainfall solutions.

In the analysis (*i.e.* narrative constructions), we marked comments pertaining to choice of schemas, choice of language constructs, discussion of design choices, mentions of problem tasks (whether or not they were reflected in code), and rationale for editing previously-written code. We also marked comments on how students perceived the Rainfall problem.

5.3 Programming Process Narratives

This section presents the narratives[3] of each participants' design process. We also summarize both the correctness and the structure of each final solution. Possible correctness values are *poor* (far from working), *fair* (in the right direction, but with many errors), and *almost correct* (very close and could have been fixed easily after some straightforward testing to show the bugs). We also show the final code produced by the students.

5.3.1 WPI2-STUD3

Correctness: Poor

Overall Structure: Accumulator, but role of parameter unclear

WPI2-STUD3 begins by writing the function name and input type. She proceeds to write the list-of-numbers template. Inside the non-empty list case, she inserts a conditional to check whether the first element of the list is positive.

[3]We follow Dziallas and Fincher [45] in calling these narratives, not case studies.

She thinks of using an accumulator in order to track the running sum of positive numbers. She goes back to her notes to check on how to write an accumulator function, then adds a local definition for a function with an accumulator parameter. She recalls that accumulator functions return the accumulator parameter in the base case; accordingly, she replaces the -1 she was originally returning in the base case with the accumulator parameter.

She notes that *"I can do the division at the end"*, then goes back to working on the running sum. If the first list element is negative, she calls the function recursively with the same parameter value. She returns to thinking about where to handle the division: *"I feel like the division should happen inside the function. So I don't want to be adding here ... I want to divide the rainfall - actually no wait I want to add the rainfall"* (at this point, she is wrestling with how to integrate the sum and average tasks within the same area of code).

WPI2-STUD3 notices that her current code never returns -1 (by inspection, not by running it): *"So now my issue is nothing will turn up -1 if the average can't be produced or if the list is just empty. So somehow I have to work that in there."* She decides to try running the code. She tries an input of all positive numbers, but gets back a negative average. She realizes this can't be right. She correctly articulates that an average is computed by dividing the sum by the count.

After this point, WPI2-STUD3 starts to thrash. She articulates a variety of possible edits involving -1 and the accumulator parameter. Her comments include statements like *"somehow I have to store the divided value into the accumulator or to make that produce at the end."*. She continues to try to reason out how her code works. She realizes that there are multiple subtasks: *"somehow I have to get the three of these things together without adding all three together"*. She seems to keep switching the task (addition, division, counting, or returning -1) to do around the recursive call on the rest of the list—her final code (Figure 5.1) reflects this confusion. Just before time is up, she thinks of using a separate helper function: *"So maybe I need to make a helper function where I just add them all up and then divide out later."*. Time runs out before she can try it in code.

During the interview after the coding session, WPI2-STUD3 remarks *"I thought accumulator would be useful because every time it finds another positive value [...] the average changes because the bottom number would keep getting bigger. So the accumulator would keep adjusting to that."* The student has associated some behavior with accumulators, but does not understand the pattern well enough to get close to a working solution.

```
;; rainfall: ListOfNumbers -> Number
;; consumes a List Of Numbers and it will produce the average
;; of the non-negative values in the list up to the first -999 (if it shows up)
;; if the average cannot be produced than the function will return -1

(define (rainfall  alon)
  (local [(define (rainfall alon acc)
            (cond [(empty? alon) acc]
                  [(cons? alon) (if (> (first alon) 0)
                                    (/  (rainfall (rest alon) (first alon)) (+ 1 acc))
                                    (rainfall (rest alon) acc))]))]
    (rainfall alon 0)))
```

Figure 5.1: WPI2-STUD3 's final Rainfall solution

5.3.2 WPI2-STUD6

Correctness: Fair (count of data inaccurate)

Overall Structure: Accumulator with parameter for running sum

WPI2-STUD6 begins by writing the function name, input type, and output type as he reads the problem. The student starts to follow the template by creating a conditional, articulating that the function should return 0 if the list is empty (this appears to be a pattern of habit, as the correct answer on empty input would have been -1). As the student is thinking out what to do when the list is non-empty, he articulates the algorithm for computing the average, and says *"we want to divide something by the length of [the input list]*", observing that only the non-negative values should be considered.

The student realizes he needs a helper function that sums the values in a list. The student articulates the type signature and writes a sum function following the HTDP template for lists of numbers. This function does not account for negative numbers or the -999 sentinel. The student then goes back to the original function and starts to handle the negatives, introducing a conditional that checks the sign and value of the first number on the list. When -999 is encountered, the student notes that the program should return the average (but doesn't completely fill in the needed code). As the student continues filling in the conditional, he starts to question whether the helper could be handled by built-in primitives.

The student finishes filling in the conditional and tries running the program, but discovers it goes into an infinite loop. At this point, the buggy program follows the pattern to recursively sum the positive numbers, returning the average when the -999 is encountered: one branch of the conditional within the recursion is implementing the sum task while another implements the average task (which can't work since this leaves no base case for the sum task). The student realizes that the code isn't *"storing the value"* of the running sum, and switches to an accumulator-based design, with a parameter to hold the running sum.

The student then begins a cycle of editing the code, testing it, having the tests fail, then editing again. As the student talks through the cycle, he begins looking for fragments of code to delete or modify: for example, he tries removing various branches of conditionals, including the one that terminates the recursion if the list becomes empty before reaching -999 (this branch never gets restored before time is called).

Next the student tries to figure out where to return -1: *"So this is still working but this is not working. So it's not producing -1. And so if the element's negative it's running the recursion on the rest of the list. Maybe - no. Maybe the [accumulator] could be set to something else other than just [the current accumulator value] or but I can't think of what it needs to be set to."* The student hits on the idea of a different helper function to handle the case in which all numbers in the original input list are negative. He proceeds to write a straight-up (correct) recursive function to check whether all numbers in a list are negative, then uses this to guard computation of the average once -999 is detected. That said, the student never got the tasks and their code mapping straight in his head. He kept modifying

the in-progress code with Rist-like focals, rather than thinking about how to decompose the problem.

The final code (Figure 5.2) contains two major errors: it does not handle input lists that lack the -999, and the average computation uses the wrong denominator (the length of the suffix that follows the -999, not the count of non-negative numbers before the -999).

```
;; rainfall: ListOfNumber -> Number
;; consumes a list of daily rainfall readings
;; and returns the average of the non-negative values in the list up to the first -999

(define (rainfall alon0)
  (local [(define (rainfall-accum alon acc)
            (cond [(= (first alon) -999)
                   (if (all-negative? alon)
                       -1
                       (/ acc (length alon)))]
                  [(positive? (first alon))
                   (rainfall-accum (rest alon) (+ (first alon) acc))]
                  [(negative? (first alon))
                   (rainfall-accum (rest alon) acc)]))]
    (rainfall-accum alon0 0)))

;; all-negative?: ListOfNumber -> Boolean
;; returns true if all of the elements in the list are negative

(define (all-negative? alon)
  (cond [(empty? alon) true]
        [(cons? alon) (and (not (positive? (first alon)))
                           (all-negative? (rest alon)))]))
```

Figure 5.2: WPI2-STUD6 's final Rainfall solution

5.3.3 WPI2-STUD7

Correctness: Fair (conflates sum and average tasks)
Overall Structure: Accumulator with filter (latter not integrated)

WPI2-STUD7 starts by writing the function signature and purpose. He begins to write the list template, filling in -1 as the answer in the empty-list case based on the problem statement. He wonders whether he should be using `local`, which is part of the standard pattern for writing functions with accumulators in the course. The student starts to write the inner accumulator function, again following the template. But this time, the student returns the accumulator value in the empty-list/base case. That is the standard usage pattern students have seen with accumulator functions to this point in the course. To this point, WPI2-STUD7 has not articulated what the accumulator variable represents; his work seems entirely syntactic.

The student talks about checking whether the first number in the list is negative, then about creating a helper function to compute the average; this comes up more as a side comment than as part of the flow of where this helper might get called from the overall Rainfall computation. The student realizes that the average computation will need both the running sum and the count of items, and thinks about how to obtain both values: *"it almost seems like I would use an accumulator to show how many times I've actually gotten through that. [...] so I guess we'll use another local"* (whether the student is suggesting another locally-defined accumulator function or another parameter within the existing accumulator is not clear at this point).

The student notes the requirement to stop at the first -999 and to ignore negatives. The student recognizes that `filter` could ignore the negative numbers, and would eliminate the need to check the sign of individual list elements during the accumulator function. The student writes a helper function that uses `filter` to remove all non-positive numbers from an input list. (The student does not, however, call this helper function from the accumulator function. The helper remains uncalled in the final code).

Next, the student adds a conditional to check for a value *less than* -999 (incorrect logic, changed in final). For the "then" branch, the student articulates calling the function recursively to process the rest of the list, while adding the new value to the accumulator. As shown in the final code (Figure 5.3), the student adds another parameter (`times`) to track the count of values. He tries to compute the average and use it as a new parameter value (he never articulates a clear role for this parameter). The else case of the conditional gets a recursive call to the function that takes the rest of the list and leaves the two accumulator parameters unchanged.

The student then enters a testing phase, running his code on a single test case. The test fails. The student correctly diagnoses that the execution never satisfies the -999 check and reverses the less-than computation in his conditional check. The student adjusts initial values for his accumulator parameters, but does not correctly trace the execution to isolate the actual errors in his code.

```
;; rainfall: ListOfNumber -> Number
;; Consumes a list of numbers representing daily rainfall readings.
;; Produce the average of the non-negative values in the list up to the first -999.


(define (rainfall alon0)
  (if (empty? alon0)
      -1
      (local [(define (rainfall alon acc times)
                (cond [(empty? alon) acc]
                      [(cons? alon)
                       (if (> (first alon) -999)
                           (rainfall (rest alon) (/ (+ (first alon) acc) times) (+ 1 times))
                           (rainfall (rest alon) acc times))]))]
        (rainfall alon0 0 1))))

(define (is-positive? num)
  (positive? num))

(define (positive-num? alon)
  (filter is-positive? alon))

(check-expect (rainfall (list 1 5 -2 0 -999 4)) 3)
```

Figure 5.3: WPI2-STUD7 's final Rainfall solution

5.3.4 WPI2-STUD11

Correctness: Almost correct (sans two `cond` cases reversed)

Overall Structure: Clean-first with accumulator (for cleaning)

WPI2-STUD11 begins by writing the function name, input type, and output type as she reads the problem statement. She proceeds to start writing the template, inserting -1 as the answer in the base case based on the problem statement. She instinctively questions whether the base case answer should instead be 0, but decides to follow the problem statement and see where it goes. She does not appear to

write the non-empty case of the template blindly, but instead talks through what might need to happen in this case.

She fairly quickly ponders whether she will need an accumulator, but she isn't entirely sure why this would be necessary. She thinks she should have a function that *"goes through each number in the list just to make sure it's not -999"*. She goes on to say that *"with every number that it passes that is not -999, it's gonna add those all up"*. So at this point, WPI2-STUD11 has decided to write a function that traverses the list and adds up all the relevant data.

WPI2-STUD11 begins to change course once she thinks about what to do upon finding the -999: *"so then I would need another helper function. Once it hit the -999, it would divide it by the [...] number of terms it went through but I don't know how I would do that yet"*. As she tries to write the base case of her accumulator function, she realizes that summing and the overall rainfall problem require different base-case answers: *"if it's empty, that would return either–it would return -1 for the rainfall purposes, but for this one I don't know if it would return 0 [or] -1"*. This prompts her to change her accumulator to instead build a list of the relevant (clean) data, with separate functions to compute the average of this list. Her final solution is a clean-first style, but with an accumulator in the function that cleans the data. During the reflection interview, she remarks how accomplished she feels for solving the problem.

```
;; a ListOfNumber is one of
;;   empty
;;   (cons Number ListOfNumber)


;; rainfall: ListOfNumber -> Number
;; produces the average of the non-negative values in the list
;; up to the first -999 (if it shows up in the given list of numbers)
(define (rainfall alon)
  (cond [(empty? alon) -1]
        [(cons? alon) (if (cons? (check-number alon))
                          (/ (add-number (check-number alon)) (length (check-number alon)))
                          -1)]))

;; add-number: ListOfNumber -> Number
;; produces the sum of the numbers in the given list
(define (add-number alon)
  (cond [(empty? alon) 0]
        [(cons? alon) (+ (first alon) (add-number (rest alon)))]))

;; check-number: ListOfNumber -> ListOfNumber
;; produces the sum of the non-negative values in the given list
(define (check-number alon0)
  (local ((define (check-number alon acc)
            (cond [(empty? alon) acc]
                  [(cons? alon) (cond [(> (first alon) 0) (check-number (rest alon) (list* (first alon) acc))]
                                      [(< (first alon) 0) (check-number (rest alon) acc)]
                                      [(= (first alon) -999) acc])])))
    (check-number alon0 empty)))


(check-expect (rainfall (list 1 3 -999)) 2)
(check-expect (rainfall (list -2 -1 -999)) -1)
(check-expect (rainfall empty) -1)
(check-expect (rainfall (list -999)) -1)
(check-expect (rainfall (list 1 3)) 2)
```

Figure 5.4: WPI2-STUD11 's final Rainfall solution

5.4 Analysis and Discussion

Our focus for this study was to understand how novice programmers solved Rainfall (a multi-task problem) when they have seen multiple viable schemas for solving the problem. As discussed in

Section 5.1, we have not found existing theories about how novices navigate or switch between multiple schemas. Observations from our data suggest possible elements of such theories.

5.4.1 What drove students to use the accumulator pattern?

All four students started saying they would use the list template and ended up using accumulators in some fashion. Whether the students perceived these as different patterns, or whether they view accumulators as a *variation* on the list template, is not evident in our transcripts. However, all four students commented on the typical base cases of these patterns, suggesting that they had internalized them separately.

The trigger to use accumulators differed across the students: WPI2-STUD3 and WPI2-STUD6 initially associated the accumulator with *tracking the sum* (though WPI2-STUD3 lost this association once she started to thrash); WPI2-STUD7 switched to an accumulator-style pattern *without a clear justification* and never stated a purpose for the accumulator (following the schema purely syntactically), though his final code suggests that he may have associated this with the idea of *tracking* some computation (tracking the count and the incorrect tracking of the average). WPI2-STUD11 explicitly ruled out an accumulator at first, then found it useful for tracking clean data. Use of accumulators could also likely be influenced by the timing of the study session involving Rainfall. At that time, the course had just covered accumulators: the pattern was fresh and it is possible that students may have assumed they *should* be using them. The lectures had shown the use of accumulators for summing a list of numbers.

5.4.2 Interactions between pattern use and task-level thinking

Observation 5.4.1. *Students who copy-and-paste the template (as* HTDP *recommends for beginners) get more stuck than those who recall the template and write it down "as they go".*

WPI2-STUD3 mechanically wrote down the list-of-number template before thinking about the details of Rainfall. The course teaches this practice, though once students have mastered the template, they tend to interleave writing the template with filling in the holes (particularly in easy spots, such as the base case). WPI2-STUD6 , WPI2-STUD7 , and WPI2-STUD11 all stated that they were going to use the list-of-number template, but they proceeded to work in "write as you go" fashion, which meant they started thinking about how they would fill in the holes around the recursive call to Rainfall before they committed to calling their function on the rest of the list. These students generally introduced an accumulator at this point, effectively switching their program schema mid-session. WPI2-STUD3 , in contrast, struggled more with the schema change and ended up farthest away from a working Rainfall solution.

Writing an incorrect schema could be correlated with several factors, including general programming skill, semantic understanding, and so on. Students may struggle to adapt schemas because their understanding of programming is syntactic more than semantic. If we can detect schema practices of students who struggle with programming, we might be able to offer targeted instruction in schema

selection and adaptation. The schema-switch seems to be associated with an *insightful thinking* about the schemas (patterns) they were using; we discuss this further in the subsequent observations.

Observation 5.4.2. *Students who articulated only the syntactic schema of accumulators, but not the underlying concept, struggled to adapt them to the needs of Rainfall.*

The mechanical use of accumulators was most evident in WPI2-STUD3 's process: while she associated the accumulator pattern with the sum task (an application of accumulators they've seen many times in the course), she does this completely mechanically. She copied the entire code for the accumulator-style sum, without thinking about how the other tasks impact the use of the accumulator pattern (*e.g.* she retains returning the accumulator in the base case), exhibiting a kind of functional fixedness [88] with the pattern.

As instructors, it is easy to assume that once students have seen the *idea* of a parameter that accumulates a running value, then they will add as many such parameters as a problem requires. This assumes that students understood the *underlying idea*, however, rather than simply absorbing the syntactic pattern. Students in our course had only seen examples with a single accumulator parameter, and in each of those programs, the value in that parameter was returned in the base case of the recursion. An accumulator-based Rainfall solution either needs two additional parameters (one for the running sum and one for the running count) or one additional parameter for the running list of clean data. Students had only seen examples that accumulated numbers up to this point in the course. Unless students understood the point of the accumulator, adapting to multiple parameters could be a significant challenge.

Interpreting this in an imperative context, it would be as if students had only ever seen programs with a single numeric variable, and did not immediately realize that they could have two variables. This is not a confusion that we have seen reported in other Rainfall studies. In functional programming, additional "variables" become additional parameters—perhaps that seems more complex to novices than additional standalone variables (which could be ignored while still allowing the program to run, whereas additional parameters need values or a syntax error results). Perhaps students in the imperative studies of Rainfall made different errors depending on whether they had seen programs with multiple variables. The point here is simply that different linguistic constructs have different affordances and pitfalls, and different courses prepare students for problems in subtle ways that we have likely overlooked in reporting studies. We need to understand our benchmark problems in multiple contexts to know what makes them challenging.

> WPI2-STUD3 : *I guess [the hardest part] was trying to figure out how to work in the -1 with the accumulator there because I didn't know where to put it because then all the examples we put the accumulator after empty because I guess all the answers were stored accumulator so it would produce the accumulator but in this one the answer wasn't stored in the accumulator.*

Observation 5.4.3. *Students who connected accumulator parameters or parts of their code to specific tasks, and maintained those connections through the schema switch, produced more correct code.*

The two students with clear roles for the accumulator were also the ones who more generally connected specific problem tasks to parts of their code. One of these was the only student who mentioned using `filter` to help deal with the negative numbers (though he never got that part integrated with his accumulator-based program for computing the average). These observations reinforce the idea that failure to decompose problems into tasks—not just failure to compose code—underlies student challenges with multi-task problems (others' work showing that students can handle similar problems when explicitly taught strategies or patterns supports this [39, 94]). Had someone suggested decomposing the problem into separate sum and count functions, we suspect the two weakest students might well have done better, since their transcripts showed they did have basic facility with the list template.

Observation 5.4.4. *Students had not understood that each sub-task that traverses a list needs its own function or accumulator parameter.*

Both HTDP and the host course explain that a single recursive function can perform only one traversal-based operation. The host course used for this study did not, however, reinforce this via assignments. Accumulator parameters enable a single function to track outputs of multiple tasks in a single traversal, but the course does not currently teach the explicit link between traversal-tasks and parameters. Our narratives show students struggling to integrate multiple traversal tasks (*e.g.* summing and counting) in a single function, even once they introduce accumulators. The connections between tasks, parameters, templates, and traversals are not (or have not been made) clear enough to these students, yet they seem critical to producing a correct Rainfall solution in any programming language.

Students similarly struggled to handle the sentinel. All prior problems in the course terminated a list recursion at the end of the list, not at a particular value. Most students recognized the sentinel as another base case for recursion, but they struggled to reconcile the return values in the empty-list and sentinel cases, especially in light of the -1. This is again a failure to separate tasks in their code. HTDP, in theory, is designed to help students think about how to create small programs for focused subtasks. For students who hit on problem decomposition, this works smoothly; for others, not so much. Our findings point towards the idea that the course curriculum needs a specific and directed emphasis on problem decomposition (in addition to the design recipe), for example, by teaching students how to identify and map tasks to each of functions or parameters/variables as part of the overall program design process.

5.4.3 The Complexity of the Rainfall Problem

Observation 5.4.5. *Students thought the problem was complex just from the problem statement.*

Our version of Rainfall[4] has more constraints and detail than Soloway's original phrasing [127], which read:

[4]See Section 4.2.2 for the full Rainfall problem statement, including common solution structures

> *Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.*

Later versions of the problem have included negative numbers, but even compared to those, our problem description has additional details such as: (a) -999 may never appear, (b) -999 may appear more than once, (c) an explicit instruction to return -1 if the average cannot be computed, and (d) use of the term "faulty readings" to contextualize the other negative numbers.

Prior versions of the problem typically omitted instructions on what to return if there is no data to average (detail (c)). We agree with Seppälä *et al.* [122] that this omission makes it hard to interpret students' mistakes. While throwing an error would be better than returning -1, the students in our study had not yet learned error-handling.

Details (a) and (b) regarding the sentinel are necessary because the input comes as a list rather than being entered interactively. Requiring the list to contain -999 actually complicates the problem for one who follows HTDP (or any datatype-based discipline) strictly. A list with a guaranteed sentinel would have a different data type definition (in which the base case is a list with the sentinel as the first element, not the empty list); this would lead to a different template. The current wording retains the schema that students already know. Taken together, however, all of these details have a price in terms of how students perceive the problem complexity:

> WPI2-STUD7 : *From what we've learned in class we generally use just simpler problems, and we rarely [...] put them all together. So when you are approached with a problem such as this, you almost struggle to figure out how to put it together 'cause you've never done it before. [...] [usually] it would be more in a Part A, Part B, Part C, Part D style.*

Future research should explore relationships between the level of detail in the problem statement, whether examples are provided [122], and when students perceive sub-tasks in more complex problems.

5.4.4 The Need for Finer-grained Analyses of Course Contexts

Our study data allowed us to ask a unique question in the context of Rainfall: how do novice students manage having seen multiple viable schemas for a programming problem? Students do not yet know the limitations of these schemas well (unlike experts). We would expect, then, to see students switching schemas or perhaps trying to merge them. We are not aware of theories of how novices switch schemas. We need to understand this, however, so we can teach students how to handle such situations more effectively.

Our data drive home the power—and hold—of previously-seen patterns for novice programmers. Instructors may think they are teaching a general approach (such as using an accumulator), but if students have only seen examples that use that approach in a single way (such as a single parameter that is returned as the final answer), they may struggle to adapt patterns to new situations. Approaches such as subgoal labeling [27] might help counter the syntactic power of a pattern. A key takeaway in this study is that the class examples instructors choose may inadvertently complicate problems like

Rainfall for students. If we want to know what makes Rainfall hard or easy, we need to consider the course context at a finer granularity than has been reported in previous studies.

Our students did not seem to perform as well as those in Fisler's study [51]. Our host course was shorter (8 weeks instead of a semester), so perhaps students needed more time to develop their skills before solving Rainfall. Our participants had only just started working with higher-order functions, which many students used in Fisler's study. Perhaps curricular differences addressed our observations for Fisler's study courses. It would be interesting to run similar think-alouds with students at the schools from Fisler's study to tease out these differences. Guzdial claimed that Fisler "beat the Rainfall problem" [59, 60], yet our study shows that even with mostly similar course curricula, the students in our study did not replicate the same success that students in her study did (for the potential reasons we mentioned prior). The research community can't claim to have "beaten the Rainfall problem" until we have findings that we can explain and reproduce across courses. This needs studies that report on finer-grained curricular details and how students draw on them when selecting designs; this study is an early effort towards this goal.

5.5 Status of Dissertation Research Questions

Our findings from this study highlight the interactions we've observed between how students use and think about the schemas they're taught, and how they think about the task-components of a multi-task problem (Rainfall):

RQ2. 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?

Interaction: Meaningful pattern use and task-level planning

The students in this study started by retrieving the list template, as prompted by the input (a list of numbers) in the Rainfall problem statement, but then shifted to an accumulator pattern later on. There's a clear difference, however, between (a) students who retrieved patterns mechanically without task-level insight, and (b) students who, in the process of retrieving the patterns, thought about how the problem-tasks would impact the use of their retrieved pattern.

The student who retrieved patterns mechanically did so, prompted by tasks (*e.g.* summing) that she has seen the patterns used with in prior examples (from their course). She then proceeded to populating the patterns with other task-related code without any clear plan/direction and without concretely thinking about how the tasks would impact her use of the patterns. She seems to only copy the syntactic structure of a pattern, without articulating its underlying concept. She generally got stuck in her process, with code that conflated different task-related code in ineffective ways within the patterns she retrieved.

Students who retrieved patterns while also relating the problem-tasks to specific parts of the patterns they retrieved generally ended up with mostly correct solutions. Two students concretely articulated a

task-level plan in advance: they used the insight from their plan to guide their use, or restructuring, of the patterns they retrieved (*e.g.* decomposing their code into multiple instances of templates that each dealt with a different task, concretely associating accumulator parameters with tasks and adding parameters for new tasks). One student engaged in on-the-fly decomposition of the problem, but still managed to get to a fairly-working solution (though not as close as the solution produced by students who concretely planned in advance). He also concretely associated parts of patterns with tasks as he wrote his solution, rather than straight-up copying a pattern without any task-level insight. Both types of students (students who planned in advance and those who decomposed the problem more on-the-fly) exhibited an awareness of the limitations of the patterns they're using (though the student who did on-the-fly decomposition did so later in his process), while also guided by the task-level plans they articulated.

RQ4. How do HTDP-trained students approach multi-task programming problems with novel components?

> Our previous studies suggest that HTDP-trained students approach new multi-task problems by:
>
> 1. Decomposing the problem on-the-fly around code they have already written.
> 2. Retrieving code-level plans without adjusting them to the need of the problem.
> 3. Using HTDP-prescribed design practices mechanically.

In this study, we observed some students approach Rainfall by:

- **Articulating a task-level plan in advance.**

Students who approached the problem this way identified concrete problem-tasks and articulated a task-level plan for solving the problem early in their process. They used the insight from their plan to guide their use of the patterns they retrieved: for example, they identified which tasks they allocated to separate template instances or which tasks they tracked in parameters (for accumulator-style solutions).

Chapter 6

Study: Movements Between Task-level and Code-level Thinking

Background and Context: Existing theories of how novices design solutions are primarily plan-based, but do not describe how novices move between task-level and code-level plans. Our own prior studies have shown that students think in terms of a problem's core tasks, but struggle with decomposition and compositions issues around their solutions. (Section 6.1)
Objective: We explored how HTDP-trained students from two different universities transition between thinking about high-level tasks and low-level code when solving multi-task programming problems.
Method: We conducted think-alouds with CS1 students at two universities (both used the HTDP curriculum) as they solved multi-task programming problems of varying degrees of novelty. We analyzed how students used the HTDP design techniques they've been taught and describe how patterns of high- and low-level thinking relate to their success on the problems we gave. (Section 6.2)
Findings: We developed a conceptual framework that captures students' task- and code-level thinking and identified three approach patterns students took towards solving multi-task programming problems. We also found interactions between how students applied their program design skills, and the variety of problems they have previously engaged with in class. (Sections 6.3 and 6.4)

A version of this chapter is published in the following venue:
[22] Francisco Enrique Vicente Castro and Kathi Fisler. 2020. Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), Association for Computing Machinery, Portland, OR, USA, 487–493. DOI: https://doi.org/10.1145/3328778.3366847

6.1 Exploring the Interplay of Task- and Code-level Thinking

Most theories of how novice programmers design solutions for programming problems draw from plan-based models of programming (Chapter 2). None of these theories, however, describe how novices move between a problem's high-level tasks and the tasks' low-level code implementations, especially when faced with programming problems that have both familiar and unfamiliar components. In these contexts, novices would need to navigate their use of prior plans and new tasks, both of which they need to build code for. In our prior studies, we have developed a concrete framework of the various program design skills that students are using (Chapter 4) and how students approach multi-task problems (Chapters 3 and 5); we have also began to describe specific interactions between certain design skills as students approached problems (Chapter 5). This study extends these prior works by exploring the interplay between task- and code-level thinking among novice programmers within the context of two CS1-level courses from two different universities that teach the HTDP design recipe. This context means that when students struggle with problems, they have design techniques to fall back on, which may, in turn, have an impact on how students think around the problem's tasks and their code. We thus focus on the following research questions for this study:

> **STUDY-RQ1.** What patterns of movements between high- and low-level thinking do we see among our students?
>
> **STUDY-RQ2.** How do students' patterns of task- and code-level thinking relate to their success on our programming problems?
>
> **STUDY-RQ3.** When students get stuck on a problem, what do they do and how do they use the HTDP design recipe as part of getting unstuck?

6.2 Study Design and Data Collection

We chose two programming problems with multiple subtasks and had students from HTDP-based CS1 courses at two universities work on each one. The first problem (Rainfall) consisted of subtasks that students had previously coded in other contexts, but not previously composed in this way. The second problem (Max-Temps) was harder, involving subtasks that students had not previously seen but that were solvable (though challenging) using the HTDP process as covered to date in each course. Problem details are provided in Section 6.2.4.

6.2.1 The HTDP Course Instances

The two HTDP courses in our study varied in topic orderings and concept emphasis. We collected data from CS1-level courses at WPI in Spring 2018 (WPI-CS1) and from Northeastern University (NEU) in Fall 2018 (NEU-CS1). At the time we ran our studies, each course had covered the basic design recipe, structures, lists, trees, and higher-order functions. WPI-CS1 spent the week prior the study covering

higher-order functions (`map`, `filter`) and was covering accumulator-style programming *during* the week of the study sessions. NEU-CS1, in contrast, had spent at least three weeks with higher-order functions (`map`, `filter`, `fold`) before the study and was scheduled to cover accumulator-style programming after the scheduled study sessions; study sessions for WPI-CS1 were done about a week before final exams, and sessions for NEU-CS1 about 2 weeks before final exams. WPI-CS1 ran at intense pace for about 8 weeks, while NEU-CS1 ran on a 14-week semester. Table 6.1 shows the topic sequences for each course.

6.2.2 Participants

Instructors of both courses publicized the study to their students. Interested students provided information on their intended major, prior programming experience, and an estimate of their current course grade on a volunteer survey (Appendices D.1 to D.4). A total of 13 WPI students and 84 NEU students signed up for the study. From each participant pool, we selected students based on their availability to participate in study sessions (sessions were 2 hours total per student) and their self-reported course performance (A, B, C, D), resulting in 12 WPI-CS1 and 10 NEU-CS1 participants. The following table summarizes the number of students self-reporting each grade, for each course; (M/F) indicates self-reported gender.

Course	**Self-estimated course grade**			
	A	**B**	**C**	**D**
WPI-CS1	2(M), 3(F)	1(M), 4(F)	-	2(F)
NEU-CS1	3(M), 1(F)	4(M)	2(F)	-

6.2.3 Logistics

Each participant did two 1-hour sessions, the first on Rainfall and the second on Max-Temps. Within a session, students had 30 minutes to work on the problem in think-aloud fashion. A retrospective interview followed, during which students described their process and responded to interviewer observations from the think-aloud; this study protocol follows the same protocol in Study 2 (Chapter 4: Section 4.2.1) that was adapted from Whalley and Kasto [140]. Students worked on their own computer (they could open notes they deemed relevant to the problem) and used the course's standard programming environment (DrRacket [44]). We audio recorded all sessions and collected students' solutions and scratch work. I conducted all of the sessions; I was not on the staff for either course, or even affiliated with NEU. Each participant received USD 20 per session.

Table 6.1: Topic sequences for the host courses in this study

(a) WPI-CS1 topic sequence (8 weeks)

Week	Topic	Assignments/Exercises
1	Arithmetic expressions and functions	Functions for arithmetic calculations, test cases
2	Helper functions, structures, conditionals, the design recipe	Functions over structs (online shopping data)
3	Lists of atomic data	Functions over lists of strings, functions over structs (course enrollment data)
4	Lists of structures	Lists of structures (lists of donors, lists of nonprofits)
5	Trees	Binary search trees (taxpayer database)
6	Locals and higher-order functions	N-ary trees (system of rivers and tributaries)
7	Accumulators, variables, mutation	Map, filter, and accumulators (revisit lists of nonprofits), variables, mutation (email system)
8	Mutation	None (end of course)

(b) NEU-CS1 topic sequence (14 weeks)

Week	Topic	Assignments/Exercises
1	Arithmetic expressions and functions	None
2	Booleans, conditionals, the design recipe	Functions for arithmetic calculations, test cases, composing images
3	Structures	Composing images, functions for arithmetic calculations (currency and measurement conversions), functions over structs
4	Unions	Functions over structs (midpoint game, manhattan distance, planet data, time data)
5	Lists, lists of structures	Functions over self-referential data (pet rock stored in layers of bags, natural numbers, monkey chain)
6	Abstractions	Functions over lists of numbers, functions over lists of structs
7	Abstractions, local	Functions over lists of structs (video playlist)
8	Abstractions, I/O	Local, map, filter, fold
9	Trees	Local, map, filter, build-list, fold, file I/O
10	Graphs	N-ary trees (file system, family tree, choose-your-own-adventure book)
11	Generative recursion	Game of life simulation
12	Generative recursion	Functions over graphs (friend network)
13	Accumulators, lambda	Generative recursion (k-means, string manipulation)
14	Lambda	Lambda, accumulators

6.2.4 The Study Problems

We describe here the programming problems we used, as well as our rationale for including each in our study.

Rainfall

The first problem we gave students was Rainfall (see Section 4.2.2 for the full problem description and viable solution structures). Rainfall provides an interesting context for our study: students have done most of the Rainfall tasks separately — summing, counting, and filtering lists (based on some criterion); they have not done list-problems that terminated at a specific value (rather than the end of the list) that may or may not appear. A main challenge in Rainfall is composition, as the problem is not itself structurally recursive (*i.e.* each of sum and count are straightforward applications of the HTDP template, but the rainfall function needs to decompose these computations into their own subtasks) and students have not composed the task-components together in a solution.

Max-Temps

The exact wording for Max-Temps follows, along with its most common solution approaches.

> **Max-Temps**: *Imagine that we have lists containing a combination of numbers and the string "new-day". The numbers represent temperature readings as taken by a sensor or weather monitoring device. The "new-day" string is sent at the start of each new calendar day.*
>
> *Design a program* `max-temps` *that takes one of these lists and returns a list of numbers representing the max temperature for each day. If the input list is empty, return empty.*
>
> Example: `(max-temps (list 40 42 "new-day" 50 "new-day" 52 56))` would return `(list 42 50 56)`.

Common solution structures:

1. **Reshape-first**
   Reshape the input into a list-of-lists that omit the delimiters, then recur over the outer list to compute the max of each inner-list.

2. **Collect-first**
   Collect sublist elements until the delimiter, then find the max of the collected elements before moving to the next sublist.

3. **Process-until**
   Find the max between consecutive list elements as the list is traversed. When a delimiter is found, concatenate the max onto the result of processing the rest of the input.

Max-Temps is more complex than Rainfall. Its viable solutions (which are similar to the *Adding Machine* problem–Section 3.2) require tasks students have not coded—or even seen—in class. *Reshape-first* requires restructuring the input into a list-of-lists. While nested lists are new to students, the design recipe templates handle them in similar ways to other nested data structures that students had used. *Process-until* requires keeping track of the current sublist's max either in an additional parameter (which WPI-CS1 students *may* have seen, given the study timing) or by modifying the head of the input mid-traversal. Finally, while functions over lists typically recur on the tail of the list, Max-Temps solutions may require recurring on a modified suffix (*e.g.* the one after the first sublist).

Overall, the plans for these tasks are largely unfamiliar to students, though the problems are on a familiar datatype (lists), so they at least have a viable schema to retrieve (the list template) that would extend to nested lists using techniques they had seen. These problems allow us to explore how students' task–code transitions, or use of design techniques (*i.e.* the design recipe), look when faced with some degree of novelty.

6.3 Analysis and Discussion

Our research questions (Section 6.1) revolve around understanding students' transitions between task- and code-level thinking as they solve problems, as well as how they navigate with their design techniques to get unstuck when struggling with the problems.

6.3.1 Framework: Task- and Code-level Thinking

Our work aims to capture *how* students think around problem-tasks and code-implementations, and their use of the design recipe within this dynamic. To do this, we randomly selected half the students from each cohort, from each self-estimated course grade: 6 WPI-CS1 (2 As, 2 Bs, 2 Ds) and 5 NEU-CS1 (2 As, 1 B, 2 Cs) students. We then open-coded [28, 69] their transcripts, field notes, and code by constructing qualitative narratives, similar to Whalley and Kasto's descriptive accounts of students' programming [140] (Figure 6.1 shows a snippet of the narrative coding done for student NEU1-STUD1), tagging student comments based on the following guide questions:

1. How did they interleave their thinking of tasks and patterns?
    - What patterns did they retrieve first?
    - What tasks did they identify first?
    - How did they think through the tasks before starting to retrieve code patterns?
2. What tasks did they implement first and what pattern did they use to implement it?
3. How did they use the design recipe in their process?

- What was the role of the different design recipe components in their work? (*e.g.* whether they used them merely for documentation or to help them get an initial understanding of the problem)

4. What dynamic do we observe in the way they think through the tasks and their implementations? (*e.g.* whether they returned to thinking about the tasks after getting stuck in code, or whether they stayed thinking entirely in code once they started implementing a task)
5. What tasks did they get stuck in?
6. What tasks were already implemented correctly when they got stuck?
7. What were they trying to do when they got stuck? (*e.g.* adding division operation to an add (+) expression)
8. What patterns did they struggle using?
9. How did they try to get unstuck?

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

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

- **Identifying and describing tasks**
  Describing tasks in terms of their role in the overall problem; novices may elicit tasks from the problem-statement or relevant plans they retrieve
- **Describing relationships between tasks**
  Describing how tasks relate to each other, such as how tasks' outputs relate to other tasks' inputs, or the ordering of the tasks (perhaps informed by their input-output relationships)
- **Plan-retrieval for familiar tasks**
  Retrieving plans for familiar problem-tasks that novices have in memory, such as a general formula for a computation or a strategy towards a task (*e.g.* to clean data, remove negatives until a sentinel)

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

- **Implementing code**
  Writing the code that novices retrieve, or modifying code to fit it within the problem context
- **Composing code**
  Putting together relevant code in a way that solves or actualizes components of a solution

Rainfall tasks

- HALT – halt the computation at the -999 sentinel
- SKIP – skip negative values (usually with another traversal computation)
- TRUNC – produce an intermediate list of values before the -999 sentinel
- RNEGS – produce an intermediate list of values without negative values
- SUM – produce the sum of a list of values
- COUNT – produce the count of a list of values
- AVG – produce the avg of a list of values
- EMP – handle cases of empty input (empty list, -999 as first element, all negatives)

References

- [# *] – transcript numbers
- {# *} – line numbers in code submission
- (FN# *) – field note item number
- (S) – scratch work

1. He identifies several tasks from the problem statement early on —
   a. HALT: *"So when -999 is encountered it's the end of reading."* [#2]
   b. AVG: *"Okay produce the average of the non-negative values"* [#2]
   c. RNEGS: *"So I need to filter the list of numbers to have only the positive numbers"* [#2]

2. He immediately associates a list-abstraction pattern with the expected general behavior of the function: the expected behavior of producing a singular value from processing a list prompts him towards the use of the foldr list-abstraction, even without mention of a concrete task connected to this:
   a. *"So it's consumes a list (sic) and produces a number so foldr most likely is my option so let's check with [the course notes]"* [#3]

3.1. Because of (2), he initially jumps into implementing rainfall using foldr, but stops upon recognizing the "special case" of the empty input, which prompts him to go back to using a list-template instead. He writes down the list-template for the rainfall function, but does not let go of the idea of using list-abstractions, thinking that he could use list-abstractions to "simplify" his function later on.
   a. *"Okay so for list of numbers so I need to use foldr [...] well no that's not going to work. I have a special case if the list is empty I cannot compute the value. I need to use cond [...] let me rewrite it using the regular list template [...] and I will see if I can simplify it using a list abstractions (sic) later"* [#4]

3.2. He writes a signature and purpose statement for rainfall, making sure to capture specific details about the problem in the purpose statement – (1) the average is only computed for positive values and (2) -999 is the terminating indicator for the computation. He writes a couple of trivial test-cases, one from the problem example and another the empty input case, and then leaves this in the meantime.
   a. Specific purpose statement: *"It takes a list of rainfall readings and returns an average of positive readings [...] should specify that -999 is [...] the indicator of end of reading the data of interest"* [#3], {#2-3}
   b. Trivial test-cases: *"let me use the list from the example [...] [check] empty list first."* [#3], {#4, 7}

4. The HALT task prompts him towards the idea of creating a function to TRUNC. In touching on this idea, he thinks about list-abstractions he could potentially use to implement TRUNC. He decides to write a list-template function – he starts by writing a signature and purpose that describe his specific goals for the TRUNC function, then test-cases that illustrate his expected behavior of the function, and finally the body of the function. The TRUNC function builds a new list of values until either empty or -999 – he realizes the similar roles of the empty and -999 cases.
   a. Touches on TRUNC while thinking about HALT, describing the TRUNC-function goal in the purpose statement: *"So I need to somehow filter the list when -999 encountered. I need to stop. [...] so first thing I need to do is to make a sub list [...] a helper function basically it will check every value and if it is -999 it will stop the function and return whatever it is [...] takes a list of rainfall readings and returns the readings prior to -999"* [#5-6], {#42}
   b. Looks for potentially applicable list-abstraction functions to implement TRUNC, deciding to use a list template when he cannot find applicable list-abstractions: *"So can I use any list abstractions. It takes a list and produces a list – I have [sort and map]. Sort doesn't change the number of items in the list. Map on the other hand also doesn't change. Filter. I may use it to filter negative numbers out of the sub list okay, but there are no list abstractions [that can be used for TRUNC]. [...] since there are no list abstractions I need to use a regular list template"* [#5-7]
   c. Recognizes the similarity in roles of the empty and -999 cases (terminating a traversal): *"So if first LON [is equal to -999] what do I do? [...] It will just return an empty list right? So I have two cases that return an empty list, I can use probably or statement if it is empty or if it is equal. Yes, I'll combine [the empty and -999 cases]. So it's going to be or empty and then equal, if any of these cases happen, then they return an empty list"* [#8-9], {#50-51}

Figure 6.1: A snippet of the narrative coding for student NEU1-STUD1

- **Plan-retrieval of task-relevant code**

  Retrieving code patterns for familiar tasks that novices have in memory; may come as built-in operations and functions, or entire code structures such as design recipe templates or previously-implemented code

The following excerpt illustrates an NEU-CS1 student touching on both high-level tasks and the

low-level code-patterns he retrieved. He describes an overall plan for Rainfall: he concretely *describes* the tasks he identified from the problem, *relates* tasks to each other by describing the ordering of the tasks and the output of some tasks, and *retrieves* concrete code patterns for familiar tasks:

> NEU1-STUD4 : *I'm thinking [the] best way to approach [Rainfall], you take your list of numbers, you get all the numbers before minus 999, create a new list from that [then] take out all the non-negative numbers and then [do]* `foldr` *with the average.* `Foldr` *to find the sum and then divide that by the length*

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

6.3.2 RQ1: Movements Between Tasks and Code

Our first *RQ* asks about patterns of student movement between high-level (HL) tasks and low-level (LL) code as students solved our programming problems. We found three main patterns of task-code transitions in our data[1]:

Cyclic

This pattern is characterized by a back-and-forth movement between task- and code-level descriptions of the components of a solution. *Cyclic* students concretely describe problem-tasks (HL) and describe code they will use to implement those tasks (LL). The composition of their code (LL) is guided by the concrete relationships they establish between tasks (HL), for example, by describing how the output of one task is used as input for another. Their descriptions of tasks are often within the context of an overall plan for a solution (*e.g. truncate at the sentinel first, then remove negatives, then compute average*); the connections they make between tasks fill the gaps between tasks, making a plan more complete. They maintain these connections that they make—both between tasks and between tasks and code (or parts of their code)—throughout their programming process. From the perspective of our multi-faceted skills taxonomy (Chapter 4), their descriptions of tasks and task-relationships reflect the *relational* level of the *decomposing tasks and composing solutions* skill.

Code-focused

These students primarily jump into writing code for the tasks they identify. They identify tasks *on-the-fly* as they program, often without concretely describing how the tasks relate to each other, and

[1] Illustrating observed patterns of task-code movement requires excerpts across the entirety of transcripts; see Figure 6.1 for a snippet of our narrative coding from which we drew task-code movement patterns.

often without an overall plan for a solution. Instead, they focus on retrieving and implementing code for a task at-hand, then add code for whichever tasks they shift their focus to next. Their descriptions of plans are often *fragmented*; they have a list of tasks that they identified, but no concrete descriptions of the connections between those tasks. They often exhibited, at most, a *multistructural* level of the *decomposing tasks and composing solutions* skill.

One-way

Students who exhibit this pattern often identified a high-level plan for a solution early on in their process, then focused on implementing code without going back to their high-level plan. Their process often shows characteristics observed from *code-focused* students: they have fragmented descriptions of plans later on as they fail to maintain the high-level insight of connections between tasks that they described initially.

6.3.3 RQ2: Success on Programming Problems

RQ 2 asks how students' movements between tasks and code relate to their success on our programming problems. Tables 6.2 and 6.3 show the number of WPI and NEU students, respectively, who attempted each solution approach per problem, the task–code movement they exhibited, and whether their code was close to a working solution.

Observation 6.3.1. *Students who followed the cyclic pattern generally developed more correct solutions than those with other patterns.*

Transcripts of *cyclic* students in both problems showed that they concretely described the tasks they implemented, capturing both the *role* of each task and how the tasks *connected* to each other. The descriptions of these connections were critical in informing the composition of their code. In Rainfall, at least half the students in each cohort exhibited a *cyclic* movement between tasks and code and were generally close to a correct solution. None of the *code-focused* or *one-way* students developed correct solutions. Their transcripts reveal that while they identified problem-tasks, they struggled to concretely relate these tasks, which seemed to influence their ability to implement these tasks in code. Others revealed a dissonance between their high-level plan and the code-pattern they retrieved, failing to recognize the limitations of their retrieved patterns in the context of the tasks they identified (exhibiting, at most, a *multistructural* level of the *meaningful use of patterns* skill). Across both courses, the number of *cyclic* students decreased in Max-Temps: of six students who were *cyclic* in Rainfall (4 from WPI-CS1, 2 from NEU-CS1), three moved to *code-focused* and three to *one-way* in Max-Temps. Their data also show that they struggled to concretely describe connections between tasks they identified; we discuss this further in the next observation.

Observation 6.3.2. *Some students struggle to capture identified task-level relationships at the code level.*

Table 6.2: WPI students implementing solution approaches for Rainfall and Max-Temps, grouped by task–code movement patterns. [C/F] indicates whether students' final code were [C]lose (minor errors on some tasks) or [F]ar from a correct solution (missing tasks, major implementation errors).

(a) WPI students' implementation of Rainfall solution approaches

Rainfall solution structure	**Cyclic**		**One-way**		**Code-focused**	
Clean-first	WPI3-STUD1 WPI3-STUD8 WPI3-STUD9 WPI3-STUD12	[C] [C] [C] [C]	-		WPI3-STUD6 WPI3-STUD7	[F] [F]
Process-multiple	WPI3-STUD3 WPI3-STUD4	[C] [C]	-		-	
Single-traversal	WPI3-STUD5	[C]	WPI3-STUD2 WPI3-STUD11	[F] [F]	WPI3-STUD10	[F]

(b) WPI students' implementation of Max-Temps solution approaches

Max-Temps solution structure	**Cyclic**		**One-way**		**Code-focused**	
Reshape-first	-		-		-	
Collect-first	WPI3-STUD5 WPI3-STUD8 WPI3-STUD12	[C] [C] [C]	WPI3-STUD9 WPI3-STUD11	[F] [F]	WPI3-STUD1 WPI3-STUD4	[F] [F]
Process-until	-		-		WPI3-STUD2 WPI3-STUD3 WPI3-STUD6 WPI3-STUD7 WPI3-STUD10	[F] [F] [F] [F] [F]

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

Observation 6.3.3. *Some students who got stuck failed to adapt patterns to new contexts.*

In Rainfall, some students got stuck on handling the -999 sentinel. Students mentioned in the interviews that they have not worked on problems that terminated list-computations prematurely at a

Table 6.3: NEU students implementing solution approaches for Rainfall and Max-Temps, grouped by task–code movement patterns. [C/F] indicates whether students' final code were [C]lose (minor errors on some tasks) or [F]ar from a correct solution (missing tasks, major implementation errors).

(a) NEU students' implementation of Rainfall solution approaches

Rainfall solution structure	**Cyclic**		**One-way**	**Code-focused**	
Clean-first	NEU1-STUD1 NEU1-STUD2 NEU1-STUD4 NEU1-STUD6 NEU1-STUD7	[C] [C] [C] [C] [C]	-	-	
Process-multiple	NEU1-STUD3	[C]	-	-	
Single-traversal	-		-	NEU1-STUD9 NEU1-STUD10 NEU1-STUD5	[F] [F] [F]
No clear plan	-		-	NEU1-STUD8	[F]

(b) NEU students' implementation of Max-Temps solution approaches

Max-Temps solution structure	**Cyclic**		**One-way**		**Code-focused**	
Reshape-first	NEU1-STUD4 NEU1-STUD6	[C] [F]	NEU1-STUD1 NEU1-STUD3	[F] [F]	NEU1-STUD10 NEU1-STUD5	[F] [F]
Collect-first	NEU1-STUD2 NEU1-STUD7	[C] [C]	-		-	
Process-until	-		-		NEU1-STUD9	[F]
No clear plan	-		-		NEU1-STUD8	[F]

specific element rather than the end of the list (the empty-list). They get stuck because they fail to see the similarity of roles between -999 and the empty-list as base-cases. The following excerpts illustrate this struggle with the list sentinel element.

WPI3-STUD11 : *The first thing that really stood out to me was may contain the number -999 indicating the end of the data of interest. We've never done anything like that in class, so for me at least, that was something new. [...] it was pretty unique in the whole -999 terminates the function. I've never done anything like that before [...] I've never done anything like that so I didn't know how to go about doing it.*

NEU1-STUD5 : *maybe that's kinda why I was having a hard time – I think that's why this function was – I was having so much trouble with it, was that we never did that in class before where we would take some value in the list and cut off the rest of the list. I know we've filtered the list by looking at the whole thing. But, again, we've never [terminated at a list value] [...] I think there's definitely a way to do it, but I couldn't come up with it on the spot.*

Observation 6.3.4. *Some students who got stuck failed to identify the limitations of the pattern they retrieved.*

A prevalent mistake among students who got stuck in Rainfall is that they started mechanically from the list-template and wrote code for the average formula within the recursive-case of the template, as in student WPI3-STUD2 's code in Figure 6.2.

```
;; average: ListOfNumber -> Natural
;; consumes a list of numbers and produces the average of those numbers
(define (average alon)
  (cond [(empty? alon) empty]
        [(cons? alon) (/ (+ (first alon) (average (rest alon)))
                         (number (alon)))]))
```

Figure 6.2: WPI3-STUD2 's Rainfall solution

From the perspective of the problem, this makes sense: the list-type input prompts the retrieval of the list-template, which students filled in with code for *average*. This, however, *overuses* the template. Students with similar code to WPI3-STUD2 did not decompose the *average* code around the *sum* and *count* subtasks into their own recursive templates, missing that the average task itself is not a recursive computation, and thus requires a slight modification to the template (*i.e.* no recursive call). They did not concretely think about how the retrieved average formula's task-components impact the use of the template code. Courses explain that a single template function can only perform one traversal-based operation; our data suggests that this task-level decomposition needs more emphasis, and that students may need to be taught how to recognize apriori when a problem requires them to modify the schemas they know.

6.3.4 RQ 3: How Did Students Get Unstuck?

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

Observation 6.3.5. *Even when students started with the design recipe steps, they did not come back to them when they got stuck.*

Missed opportunities in Rainfall

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

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

; Examples of input-output pairs/test-cases
(check-expect (rainfall data1) 2)
(check-expect (rainfall data2) 2)
(check-expect (rainfall data3) 2)
```

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

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

Students could also expand examples/test-cases to identify tasks or task-decompositions. For example, they could expand test-cases to say that this call to rainfall:

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

is similar to this:

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

which more explicitly shows a decomposition of the code relative to the tasks.

Missed opportunities in Max-Temps

Some students wrote data definitions, motivated by the novelty of a list with both numeric and string elements. Their data definitions, however, were either incomplete or incorrect. For example, student NEU1-STUD3 wrote a data definition for the elements of the input list, but missed writing a data definition for the actual list; this led her to use a template that was only good for a list element, but not the input list itself:

```
; A Newday is one of
;- "new-day"
;- Number
; The string "newday" represents the start of each new calendar day
; The number represents temperature readings as taken by a weather monitoring device
;(define NEWDAY-1 (list 40 42 "new-day" 50 "new-day" 52 56))
;(define NEWDAY-2 (list 40 42 "new-day" "new-day" 50 "new-day" 52 56))
#;
(define (nd-temp nd)
  (cond [(string=? nd "new-day")...]
        [(number? nd) ...]))
```

She then incorrectly uses the above template to write a function for processing a list:

```
(define (list-temp nd)
  (cond [(string? nd) (list (list-temps (rest nd)))]
        [(number? nd) (cons (first nd) (list-temps (rest nd)))]))
```

She eventually realizes that she used a template that was inappropriate for the input, but proceeds to use a basic list template for her functions regardless, without appropriate modifications for the data in the problem.

An appropriate input data definition and template for the list data in the problem would have been:

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

; (define (func input)
;   (cond [(empty? input) ... ]
;         [(string? (first input)) ... (first input)
                                   (func (rest input)) ... ]
;         [(number? (first input)) ... (first input)
                                   (func (rest input)) ... ]))
```

Even when students got stuck writing their code using a template that was not appropriate to the type of input they were working with, many never went back to reexamine and correct their data definitions (or template). The student (NEU1-STUD3) who wrote the template that was inappropriate for the input later on realized that the data definition and template did not fit the input she had to work with, so she attempted to go back to her data definition to identify how to structure the template correctly, but this was done late into her process so she ran out of time before finishing her solution. Students whose Max-Temps code was close to correct wrote their functions using list-traversal templates with appropriate modifications, *i.e.* they modified their templates based on the data in the problem. The reshaping function by NEU1-STUD4 below uses a list-traversal template that is similar to the template given above:

```
(define (readings->days loi)
  (cond
    [(empty? loi) (list '())]
    [(cons? loi) (cond
                   [(string? (first loi)) (cons '() (readings->days (rest loi)))]
                   [(number? (first loi)) (cons
                                           (cons (first loi) (first (readings->days (rest loi))))
                                           (rest (readings->days (rest loi))))])]))
```

6.3.5 Analyzing the Course Contexts

We conducted our studies towards the latter part of the courses; we thus expected students to have had a significant amount of practice with using the design recipe to solve programming problems. As we've discussed in the prior sections, some students did *not* use the techniques put forth by the recipe when working on our study problems; those who did, did not use them mid-process to get unstuck, or simply followed the recipe blindly. Many students used the design recipe in a *mechanical* manner, seemingly without insight into how the different techniques (and artifacts produced from the use of each technique) informed their solutions.

To understand our observations of students' use (or non-use) of the design recipe, we conducted a document analysis of the instructional materials used in our host courses. *Document analysis* is a qualitative research method of systematically reviewing and evaluating documents in order to elicit meaning to develop an understanding and knowledge of a phenomenon or subject of study [15]. Similar to other qualitative research methods, document analysis involves the organization of excerpts, quotations, or passages into major themes, categories, or case examples through content analysis and provides a context within which actors in a research project (*e.g.* participants) operate. In our specific case, we analyzed the course materials produced by the instructors, which include the web pages and electronic documents (*e.g.* PDFs, text files) used to disseminate homework problems, laboratory exercises, course lectures, and course notes. We also informally interviewed the instructors to ask about their practices in teaching the use of the design recipe to their students.

Ideally, a deeper study of the course contexts would have involved an ethnographic observation of the classes to identify and understand the cultural and social interactions between the instructors and their students; this would have helped us more deeply understand how classroom practices may have shaped how students used or understood course topics (*e.g.* the design recipe). We were not able to conduct this kind of study for a couple of reasons: our IRB proposals for both universities were limited to only observing students during our designed study sessions and did not involve classroom observations; in addition, it was our analyses of the student data we collected from the sessions that prompted us to the need to analyze course contexts, by which time the courses had already finished lectures and laboratory sessions.

Our main focus in this document analysis was to understand students' use of the design recipe from the perspective of the course context in which they learned to use them: in what ways were students asked to use the design recipe and in what ways were they shown to use it? In other words: how did they practice the use of the design techniques put forth by the recipe, given the activities that students were asked to do in their courses? We also looked at what problems students had been asked to work on (*e.g.* had they written accumulator-style functions with more than one accumulator?).

These questions guided our collection and analysis of the course materials.

Data Collection: Course Materials

We collected the following course materials from each course website:

- Overall course syllabus and topic outlines
- Homework problems
- Laboratory problems/activities
- Course handouts and lecture notes
- Sample exams (from previous course offerings)

A summary of the course syllabi and the general topics covered by homework and labs in each course is in Table 6.1 (page 92). The course handouts and lecture notes we retrieved contained information such as: the design recipe, design recipe templates, rules on course submissions (*e.g.* homework and lab exercises), vocabulary used in the course (*e.g.* definition of *value*/*expression*, how to evaluate functions, *etc.*), and course policies. The homework and laboratory exercise documents contained programming problem descriptions, instructions on submissions and logistics (*e.g.* setup of programming environments, naming conventions for functions and data, *etc.*), and the goals or expectations for the homework or lab.

Analysis of Course Materials

We focused our analysis on identifying the kinds of activities that students engaged in wherein they used techniques from the design recipe. Our goal is to find and describe the connections between our observations of the activities that students engaged in, and our earlier findings of students' use of the design recipe. I iteratively read through the course materials we extracted from the course web pages to find common themes of activities students engaged in. In my analysis, I found that the course handouts primarily served as quick reference notes, such as providing the list of the design recipe steps or the recipe templates that were covered in the lectures. I thus focused my analysis on the homework and lab exercise materials, as these provided more information on the actual activities students were asked to do. I looked at what the homework and lab instructions asked students to do, the kinds of problems students were asked to solve, and how students were asked to use the design recipe techniques.

Findings from the Document Analysis

A common theme across the homework problem sets were that students were generally instructed to "*Develop/Write a function to do* <*task*>", where <*task*> is a computation that has an input and an output, and may or may not have task-components (for example, homework given in the earlier parts of the course only had students write functions with one task, whereas homework problems at the latter parts of the course had multiple tasks). The following problem descriptions (Figure 6.3), both from the first homework of each course, illustrate this.

1. A local performing arts center books various events. Every event costs the center $1500 (for utility costs, custodial help, etc.) plus $2 per attendee (for program printing, etc.). You are to develop a set of functions that will calculate the profit for an event, based on the number of attendees and the cost of a ticket.
   - (a) develop a function that consumes the number of attendees at an event and produces the cost to the performing arts center to hold the event. Name your function `event-cost`.
   - (b) develop a function that consumes the number of attendees and the price per ticket and produces the income generated by ticket sales. Name your function `ticket-revenue`.
   - (c) develop a function that consumes the number of attendees and the price per ticket and produces the profit for the event. Use the functions you developed in parts (a) and (b). Name your function `event-profit`.

   Use defined constants where appropriate. Make sure each function you define is documented with a signature and a purpose. Each function should be tested using `check-expect`.

(a) WPI Homework 1 problem

Exercise 1 Write a function, `octuple`, that multiplies a number, supplied as an argument, by 8. Write three check-expects: one for a negative value, another for zero, and a third for a positive value.

(b) NEU Homework 1 problem

Figure 6.3: Problem descriptions from the first homework of the host courses

The examples in Figure 6.3 generally ask students to *follow* the design recipe to solve the programming problems given. Furthermore, note that in addition to the problem descriptions, there are supplementary instructions that remind students about writing artifacts related to certain design recipe steps. For example, Figure 6.3a reminds students to "document" their functions with a signature and purpose statement (recipe step 3) and to write test cases (recipe step 4); Figure 6.3b explicitly asks students to write three test cases, as well as the input scenarios that two of these test cases should cover. Other problems in the source homework of the sampled problems in Figure 6.3 also ask or remind students to explicitly apply specific design recipe steps when solving a problem (Figure 6.4); nonetheless, the problems in these homework 1 problem sets are all of the "*Develop/Write a function to do <task>*" form.

When developing functions, follow the design recipe: in particular, the check-expect's should be written before you write the function body.

(a) From WPI homework 1

Function signatures/purpose statements are not required, but attempting them would be good practice and you may receive useful feedback for future assignments.

(b) From NEU Homework 1

Figure 6.4: Homework 1 text of the host courses that remind students to apply the design recipe

Given that the source homework is the first homework of the course, it's reasonable that the students are further scaffolded (in addition to the design recipe itself being a scaffolded approach to program design) by explicitly asking or reminding them to apply specific recipe techniques, as doing so may help them get into the habit of using the design recipe. The rest of the homework for both courses still retain the additional scaffolding of asking students to write specific recipe artifacts for problems before writing code (*e.g.* writing data definitions and templates), but to a lesser extent. Most of the

problems in the homework at the latter parts of the courses have more "*Develop/Write a function to do <task>*" problems, with brief reminders to follow the design recipe. This is illustrated in Figure 6.5, which shows excerpts from an NEU homework on writing functions for lists. In the case of the WPI homework on the same topic of writing functions for lists (Figure 6.6), the homework provides the data definition for a list-of-string and asks students to write a template based on the list-of-string data definition, and functions over lists of strings.

All steps of the design recipe are required unless otherwise specified.

(a) Reminder at the beginning of the homework problem set to follow the design recipe

Exercise 9 Design a data definition for a list of `Booleans`. Be sure to use `cons` and `'()`. No interpretation is needed as this list of booleans has no specific context, but be sure to include the rest of the steps of the design recipe for data.

Exercise 10 Design a function which determines if *any* element in a list of booleans is `#true`. Do not use `member` or `member?`.

Exercise 11 Design a function which negates every element in a list of booleans. In other words, it converts every `#false` to `#true` and vice versa.

(b) Problem descriptions involving writing functions for a list of booleans

Figure 6.5: NEU Homework 7 text on writing functions for lists

Use this data definition for any problem that uses a ListOfString:

```
;; a ListOfString is one of
;;  empty
;;  (cons String ListOfString)
;; interp:  represents a list of strings
```

1. Write the template for ListOfString.
2. Write the data definition for a list of natural numbers. Then develop a function `list-of-lengths` that consumes a list of strings, and produces a list of the lengths of each string in the original list.
3. Develop a function `strings-of-length-n` that consumes a ListOfString and a number of characters and produces a ListOfString. The list that's produced contains only those strings from the original list that have the given number of characters.
4. Develop a function `got-word?` that consumes a ListOfString and a String and produces a Boolean indicating whether or not the given string occurs in the list.
5. Develop a function `found-duplicate?` that consumes a ListOfString and a String and produces a Boolean indicating whether or not the given string occurs at least twice in the list. (Hint: use your function from the previous problem as a helper.)

Figure 6.6: WPI Homework 2 text on writing functions for lists of strings

The NEU lab exercises are similar in nature to the homework given in the course: the labs are also problem sets on specific topics (*e.g.* writing functions over lists) that are meant to, according to the "General Information" page of the course, "explain some of the principles from lecture with hands-on examples". On the other hand, the WPI labs seem to serve as an extension of the homework: the activities in the lab ask students to write down the earlier steps of the design recipe for the problems in the corresponding homework. For example, the lab activities shown in Figure 6.7 ask students to work on the signatures, purpose statements, and examples/test cases for the problems in the corresponding homework 2 in Figure 6.6.

1. Do problem 1 on the homework.

2. Write a data definition for a list of natural numbers. Write the signature/purpose for `list-of-lengths`. Then, using check-expect, write an adequate set of test cases for the function. (Write the test cases only, don't write the function definition; you'll do that for homework.)

   Ask a lab assistant to check over your first two problems.

3. Write a signature/purpose for `strings-of-length-n`. Using check-expect, write an adequate set of test cases for the function.

4. Write a signature/purpose for `got-word?`. Using check-expect, write an adequate set of test cases for the function.

5. Write a signature/purpose for `found-duplicate?`. Using check-expect, write an adequate set of test cases for the function.

Figure 6.7: WPI Lab 2 text on writing functions for lists

We also looked at which patterns students had been asked to use, or had been exposed to, in their course material. From our analysis of students' solutions to our study problems (Section 6.3.3), two WPI students attempted to use the accumulator pattern to solve Rainfall by accumulating the sum in a parameter, but their transcripts show that they struggled to figure out how to generate the count: a viable solution would have been to use two additional parameters to track each of sum and count. WPI3-STUD5 noted in his interview that he had not used multiple accumulator parameters before, hence he did not think of the idea while developing his solution for Rainfall. WPI3-STUD11 also didn't figure out to use separate accumulators for sum and count. In our analysis of the WPI lecture notes, we found that students had only been shown examples of accumulator-style functions with only one accumulator parameter, and in most cases, the accumulator was returned in the base case. Students were shown accumulator-style examples of summing numbers, producing a list of words that meet specific criteria, and producing the largest number in a list. Only one example showed an accumulator-style function that did not return the accumulator in the base case; this example showed a context-preserving accumulator-style function. These examples (Figures 6.9 and 6.10) followed the steps that students were taught for defining accumulator-style functions (Figure 6.8).

```
;Follow these steps when using accumulator-style to make a function tail-recursive
;(i.e. to improve efficiency)
;
;- write two function
;    - one function calls the accumulator-style helper, initializing the accumulator
;    - the other function is the accumulator-style helper
;
;- the accumulator function
;    - introduces a new parameter, the same type as the type of the function result
;    - in the empty? case, the function returns the accumulator value
;    - in the cons? case, the function makes a recursive call on the rest of the
;      list, updating the accumulator parameter as needed
```

Figure 6.8: WPI lecture notes on defining accumulator-style functions

The homework also had students write accumulator functions that only needed single accumulators. The NEU students had not encountered accumulator functions at the time that we ran our study sessions, so a similar analysis of NEU material on accumulator programming is not meaningful. None of the NEU students had used accumulators in their solutions.

Other ways that students struggled with the problems is with adapting parts of the basic list template to new contexts. In particular, students struggled to figure out how to recur over a modified suffix

```
;; sum:  ListOfNumber -> Number
;; consumes a list of numbers and produces the sum of the
;; numbers in the list
(define (sum alon)
  (sum-accum alon 0))

;; sum-accum:  ListOfNumber Number -> Number
;; consumes a list of numbers and an accumulator, and produces
;; the sum of the numbers in the list, keeping track of the
;; sum so far in the accumulator
(define (sum-accum alon acc)
  (cond [(empty? alon) acc]
        [(cons? alon) (sum-accum (rest alon) (+ (first alon) acc))]))
```

(a) Accumulator-style summing function

```
;; long-words:  ListOfString -> ListOfString
;; consumes a list of string and produces a list of only those strings
;; with more than 5 characters
(define (long-words alos)
  (long-words-accum alos empty))

;; long-words-accum:  ListOfString ListOfString -> ListOfString
;; consumes a list of string and an accumulator and produces a list of
;; only those strings with more than 5 characters, building the list in
;; the accumulator as we go
(define (long-words-accum alos long-words)
  (cond [(empty? alos) long-words]
        [(cons? alos) (if (> (string-length (first alos)) 5)
                          (long-words-accum (rest alos) (cons (first alos) long-words))
                          (long-words-accum (rest alos) long-words))]))
```

(b) Accumulator function that produces a list of words that meet specific criteria

```
;; largest:  ListOfNumber -> Number
;; consumes a list of numbers and produces the largest number in the list
(define (largest alon)
  (largest-accum alon (first alon)))

;; largest-accum:  ListOfNumber Number -> Number
;; consumes a list of numbers and an accumulator, and produces the
;; largest number in the list, remembering the largest number seen so
;; far in the accumulator
(define (largest-accum alon acc)
  (cond [(empty? alon) acc]
        [(cons? alon) (if (> (first alon) acc)
                          (largest-accum (rest alon) (first alon))
                          (largest-accum (rest alon) acc))]))
```

(c) Accumulator function that produces the maximum value in a list

Figure 6.9: Accumulator function examples with one accumulator parameter returned in the base case that WPI students were shown in lecture

```
;; skipn:  ListOfString Natural -> LitOfString
;; consumes a list of strings and a number n of positions to skip, and produces
;; a list containing the first element, then skipping the next n elements, then
;; including an element, etc.
(define (skipn alos n)
  (skipn-accum alos n 0))

;; skipn-accum:  ListOfString Natural Natural -> ListOfString
;; consumes a list of strings, a number n of positions to skip, and an accumulator,
;; and produces
;; a list containing the first element, then skipping the next n elements, then
;; including an element, etc., keeping track of how many more to skip in acc
(define (skipn-accum alos n acc)
  (cond [(empty? alos) empty]
        [(cons? alos) (if (= acc 0)
                          (cons (first alos) (skipn-accum (rest alos) n n))
                          (skipn-accum (rest alos) n (- acc 1)))]))
```

Figure 6.10: A context-preserving accumulator function that does not return the accumulator in the base case that WPI students were shown in lecture

of the list (other than `rest`) for Max-Temps, and how to handle the -999 sentinel for Rainfall (we discussed these in Section 6.3.3). We found from our document analysis that students were only shown (list-based) examples and problems that required terminating computations at `empty` and recurring on the tail (`rest`) of the list.

Takeaways from the Document Analysis

One takeaway from this document analysis is that the design recipe is consistently reinforced and deeply woven into all the course materials: in lecture handouts, notes, homework, and exercises. Students thus had significant exposure to, and are consistently reminded about, the design recipe process. In our description of the curriculum (Section 1.1), we explained that HTDP courses show how to apply the design recipe to increasingly rich data structures, starting from designing programs over atomic data, to compound data (structures), to lists of atomic data and structures, to trees. Our document analysis further illustrates this repeated reiteration of the design recipe and shows the effort towards training the students in the design recipe process.

Another takeaway is that students primarily engaged in activities that asked them to *follow* the design recipe when solving problems. Many of the students in our study did, in fact, follow the design recipe when working on our problems, but did so *mechanically*. As we've discussed in Section 6.3.4, even when these students followed the recipe, they failed to solve our problems and often got stuck, missing opportunities to use their design techniques to get unstuck. Our recommendations about using the design techniques to get unstuck requires going *beyond* a mechanical use of the recipe: these require a *meaningful and insightful reflection* of how the recipe techniques informs and shapes a solution (and other design recipe artifacts) in order to use them meaningfully. In contrast with the students who struggled with the problems we gave, the students who were more successful described a more meaningful use of the recipe techniques, *in addition to* exhibiting a cyclic approach toward the problems. They explained how the input impacts the template to be used (instead of just mechanically using the basic list template, as prompted by the list-type input), or how the space of examples touched on certain tasks of the problem or connected to specific parts of code (rather than just using the example provided in the problem statement).

None of the homework or lab exercises from our host courses had activities that focused on getting students to *reason* about their use of the recipe, or explain how different recipe steps connect to each other. The design of the homework and exercises mainly prompted students to *use* or *follow* the design recipe, or even prompted them to write design recipe artifacts, which still remains in the activity space of "following" the design recipe. This current design of the course activities may have driven students to develop a habit of going through the motions of the design recipe steps; taking into consideration the difference between students who demonstrated an insightful use of the recipe versus those who didn't, our findings suggest that a habitual, mechanical use of the recipe may not be enough to solve our multi-task problems. In other words, developing a habit around using the design recipe may not necessarily imply an insightful use of it. As we did not (and could not) do a class observation in either of the host courses, we cannot make claims about whether or not the instructors only tried to teach the students to use the recipe mechanically, or whether or not (or to what extent) they aimed to prepare students for reasoning with the recipe. Our findings, however, suggest that students may potentially benefit from activities that not only directs them to follow the recipe process, but also to practice reasoning about their use of the recipe techniques and how the techniques connect to or inform each other; our recommended ways of using the design techniques fall into this space of activities.

Our observations about the kinds of problems that students have seen also point to the need for students to do a wider variety of data design activities. Students seem to develop a form of *functional fixedness* over the patterns they use, having only been exposed to examples and problems that don't require designing beyond the basic list template, or, in the case of accumulators (for the WPI students), accumulator functions that only used single accumulators. Our findings suggest that students seem to have just absorbed the syntactic structure of the patterns they've seen, rather than the underlying concepts around the patterns. Having students practice designing for problems with a wider variety of data contexts may help drive home the concepts underlying the patterns they retrieve and move them beyond a mechanical use of their patterns.

6.4 Insights and Takeaways

We developed a conceptual framework that captures the task- and code-level thinking students engaged in as they solved our programming problems. Our findings suggest that students who exhibit a *cyclic* pattern of task- and code-level thinking had the most success with the problems we gave. These students concretely described relationships between tasks, made concrete connections between tasks and code, as well as maintained these connections throughout their programming process. Students who struggled failed to capture how the tasks in the problems interconnected, could not transfer patterns to new contexts, or overused the patterns they retrieved.

6.4.1 Insights on Teaching the Design Recipe

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

6.4.2 Insights on Task- and Code-level Thinking

Our observations confirm findings from our previous study (Chapter 3) that students think in terms of a problem's core tasks. Where they struggle is in concretizing relationships between tasks, which affects their ability to compose tasks' code implementations, especially for tasks they have not seen before. Students could be taught to use techniques from the design recipe to help uncover some of these

relationships, for example, by teaching them how to expand examples to identify task decompositions of the code. Some task-relationships required new patterns students have not seen, for example, doing a recursion on a modified list suffix or prefix, or keeping track of lists using another list. This suggests that just because students have seen an instance of a pattern (*e.g.* a list of numbers; recursion on the tail of the list), does not mean that students can generalize those patterns to other contexts (*e.g.* a list of lists; recursion on a modified suffix of the list). As instructors, we should be careful not to assume that students have understood the underlying *idea* behind a pattern, simply because we've shown them an instance of it. Focusing on enforcing students' understanding of patterns they're taught may help them navigate between tasks and code better in new situations. This highlights the need for students to engage in learning activities that lets them practice the use of their design techniques in wider and more varied contexts, such as doing data design activities with a wider variety of data. Our analyses also highlight the importance of teaching task decompositions explicitly, as we have found that the more successful students used their task-level plans to guide the compositions of their code. One way to go about this is by having students do more activities around identifying and planning around problem-tasks, but without expecting them to write code; for example, students might be given several multi-task problems where they identify tasks and concretely describe how the tasks relate to each other, perhaps by using type signatures to relate how the tasks' outputs connects to other tasks' inputs.

Overall, our findings indicate that students need to be explicitly taught techniques for navigating back to high-level plans when they get stuck, how to identify when current code has diverged from a plan and needs to be rethought, and how to leverage their design techniques when rethinking their solutions.

6.5 Status of Dissertation Research Questions

Our findings from this study points to particular interactions between how students applied their program design skills, and the variety of problems they have previously engaged with in class. We also identified three approaches students took towards solving multi-task programming problems. Based on our findings, we also propose some activities that can be done in class that leverage some design recipe steps towards teaching task-level planning explicitly.

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?

> In our previous study (Chapter 5), we identified the following interaction between two program design skills:
> **Interaction:** *Meaningful pattern use* and *task-level thinking*
> Students who applied the above skills at a *relational* level were more successful towards writing a correct (or close to correct) solution for a multi-task programming problem. They used the insight from their task-level plan to guide their use, or restructuring, of the patterns they retrieved. Those

> who applied the skills at lower levels (at most *multistructural*) struggled to write a working solution, often retrieving and using patterns mechanically without any clear plan towards a solution.

Our analyses in this study allowed us to tease out certain interactions between (a) the variety of problems with which students have practiced their design techniques and (b) their application of their design skills on multi-task problems. While these are not skill-to-skill interactions (which is what **DRQ2** is about), these interactions seemed to have a critical influence in how students applied their design skills towards solving our multi-task programming problems.

Interaction: Problem variety and leveraging multiple representations of functions

Some students who got stuck in Max-Temps struggled because they used data definitions and/or templates that were inappropriate for the type of input they were writing functions for. Some struggled to figure out how to write a data definition (or template) for the Max-Temps data.

We inspected our host courses' syllabi, homeworks, and exercises to understand why students struggled to design data definitions and templates for Max-Temps. Our analyses revealed that the students had only practiced using their design techniques on a limited repertoire of data. Specifically with lists, students have only experienced designing for data that used the basic list data definition (and template), so they did not have experience using their design techniques for data such as the one in Max-Temps; while the problem's input data was a list (for which they at least had the list template schema to start with), manipulating their known schema to design a template for a list that had elements that played specific roles (*i.e.* delimiters that indicated sub-parts of the list that need to be processed separately) seemed to be a significant design challenge for students. This suggests that students may need to do a variety of data design activities that would expose them to various data situations they might encounter, as well as drive home the idea of how to design data definitions (and templates) for different data beyond the basic lists they've seen in class.

Interaction: Problem variety and meaningful use of patterns

Some students also struggled with adapting the patterns they've learned to new contexts. In Rainfall, for example, students got stuck figuring out how to handle the -999 sentinel; in Max-Temps, they got stuck figuring out how to recur over a modified suffix of the list, and not just the tail (`rest`) of the list. None of the students had worked on list problems that terminated prematurely at an element, rather than at the end of the list (`empty`); they also have not worked on list problems that recurred on anything other than the `rest` of the list. This suggests that these students may have attributed some form of functional fixedness [88, 96] over parts of the template (*i.e.* the base-case and recursive-call parts), having not seen or practiced on problems that required a use of these template parts beyond the basic `empty` base-case or recursion on the `rest` of the list. This suggests that students may need to be exposed to (and practice on) problems that require manipulating the base-case and recursive-call parts beyond their typical use. Such problems may help illustrate the "malleability" of parts of the template

beyond just the template "holes" that are typically considered the "malleable" parts, *i.e.* filled in with problem-specific computations.

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

> In our prior studies, we found that students approached multi-task problems by:
> 1. Decomposing a problem on-the-fly around code they have already written.
> 2. Retrieving code-level plans without adjusting them to the need of the problem.
> 3. Using HTDP design practices mechanically.
> 4. Articulating a task-level plan in advance.

In this study, we further synthesized the above practices we found into the following programming process patterns:

1. **Cyclic**
   Cyclic students move back-and-forth between task- and code-level thinking throughout their process. They concretely describe relationships between tasks in the context of an overall plan for a solution and use their insight from task-relationships to guide the composition of their code. They apply the *decomposing tasks and composing solutions* skill at a *relational* level consistently throughout their process.

2. **Code-focused**
   Code-focused students primarily work at the code-level throughout their process. They focus on retrieving and implementing code patterns and constructs relative to the tasks (which they often identify *on-the-fly*), but do not concretize how the tasks integrate with each other. Their descriptions of plans are thus fragmented and lack an overall task-level plan towards a solution. They often use the patterns they retrieve mechanically and without thinking about how a problem's task-components impact the code patterns they use.

3. **One-way**
   One-way students exhibit a *relational* application of the *decomposing tasks and composing solutions* skill by identifying a task-level plan for a solution early on in their process, but do not remain consistently at this level throughout their process. Once they move on to implement tasks in code, they regress to a *code-focused* process and fail to maintain the connections between tasks, and between tasks and code.

These programming process patterns that we found also suggests that it is not enough for students to exhibit their skills at the *relational* level, but that they must consistently remain at this level throughout their programming process to be successful in solving multi-task problems. This might be one explanation for the non-monotonic skill progressions we found in our second study (Chapter 4):

the lack of consistency in applying skills at certain levels may indicate the fragility of a student's skills. This could be a good indicator of when students might need help or interventions towards helping them practice applying their design skills; some of our proposed activities within this section could potentially be used towards this.

Other lessons learned

One of our takeaways from this study was that it was not enough to teach students to use the HTDP design techniques to plan solutions in advance. Students also needed to be explicitly taught how to leverage the design techniques they know *mid-process* or when they get stuck, as well as engage in activities that help them practice their design techniques on a variety of problems.

1. **Using examples**
   We observed that some students just copied the example provided in the problem-statement, suggesting that they are using the technique mechanically. Giving students code with errors and having them reason about the causes of the errors using specific examples may help students practice a more insightful use of examples.
   We had also hoped that students would expand examples to work out the task decompositions of their code; this was a similar hope we had way back in the first study (Section 3.6.3). None of the students across all our studies did this and none of our host courses taught students how to use examples in this way, yet this seems a potentially valuable practice given that most students struggle with (or do not even practice) task-level planning, and end up getting stuck while working purely in code.

2. **Teaching task-level planning explicitly**
   Building on the above point and as highlighted by our findings in this study, students need to be explicitly taught how to plan at the task-level, as we have found that the more successful students used their task-level plans to guide the design and composition of their code. One way this could be done is by having students do more activities that involve identifying and planning around problem-tasks, but without expecting them to write code. For example, students might be given several multi-task problems where they identify tasks and concretely describe how the tasks relate to each other, perhaps by leveraging design recipe steps such as the type signatures to relate how tasks' outputs connect to other tasks' inputs. One student in this study, NEU1-STUD9 , did something similar by using purpose statements to describe connections between helper functions. This at least suggests that the practice may be promising.

Chapter 7

Study: Impact of a Lecture on Program Plans in First-Year CS

Background and Context: One critical finding from our prior studies is that task-level planning needs to be made an explicit part of instruction. Some researchers have made notable efforts at teaching students high-level strategies or plans from the onset, but this requires a concerted effort to make planning a central part of the introductory curriculum. (Section 7.1)

Objective: We were interested in whether a lightweight approach to teaching planning could have any effect, or whether only a comprehensive overhaul of the courses—which may be impossible given a department's other needs—would suffice.

Method: We ran studies with first-year students at two universities. In a pre-assessment, we assigned students multi-task programming problems and leveraged those to give a single 50-minute lecture on plans and tradeoffs. In the post-assessment, we gave students a new set of programming problems and asked them to produce two solutions to each problem, each embodying a different plan. We also asked students to preference-rank their solutions to see what criteria they were learning to apply to plans. (Section 7.2)

Findings: Results suggest that lightweight instruction in planning can have significant impact, assuming students can produce correct solutions for the pre-assessment questions. Many students produced two different plans in the post-assessment, often choosing a general plan that was first introduced in the planning lecture. When preference-ranking their solutions, many students chose solutions with a different general structure than what they wrote on the pre-assessment. (Section 7.3)

A version of this chapter is published in the following venues:

[23] Francisco Enrique Vicente Castro, Shriram Krishnamurthi, and Kathi Fisler. 2017. The Impact of a Single Lecture on Program Plans in First-year CS. In Proceedings of the 17th Koli Calling Conference on Computing Education Research (Koli Calling '17), ACM, New York, NY, USA, 118–122. DOI: https://doi.org/10.1145/3141880.3141897

[17] Francisco Enrique Vicente Castro, Shriram Krishnamurthi, and Kathi Fisler. 2017. The Impact of a Single Lecture on Program Plans in First-Year CS. WPI Department of Computer Science Technical Report number WPI-CS-TR-17-02, released November 2017.

7.1 Study Context: Cognitive Foundations of Planning

As we discussed in Chapter 2, the landscape of first-year programming courses tends to focus instruction on low-level programming constructs, with the expectation that students implicitly build their knowledge base of problem solving and programming strategies from trial-and-error through extensive sets of exercises [36]. Some recent studies show students succeeding at plan composition in specific contexts [51, 122], but the pedagogic choices that help students with planning remain poorly understood. Some researchers have looked at improving planning skills through pedagogical frameworks and practices that explicitly teach systematic problem solving strategies in programming: Porter and Calder suggests a process for building a pattern vocabulary for guiding students through problem decomposition [108]; Muller, Haberman, and Ginat developed pattern-oriented instruction that involved attaching labels to algorithmic patterns [94]; and de Raadt, Watson, and Toleman incorporated programming strategies in an introductory programming curriculum and required students to apply specific strategies in their solutions [39].

Different strategies for teaching planning build on results of how people construct programs at a cognitive level. Given a programming problem, programmers (subconsciously) identify solutions to similar problems and adapt them to the constraints of the problem at hand [106, 107, 117, 130]. Repeated application of a pattern helps programmers form a mental schema for that problem (which could be recalled later for solving other problems); repeated use of a schema strengthens later recall of that schema [87, 99]. This basic architecture underlies curricular approaches to teaching patterns explicitly.

How much exposure students need to a solution schema before they can apply it to new problems remains an open question, and one that depends on a student's experience level. We would not expect a truly novice programmer to internalize a solution that used constructs (such as iteration) that the student had simply been shown in class: internalizing code patterns requires practice with actually using them. But what if a student had written several programs that traverse lists (for example), then saw a program that traverses a list to accomplish a slightly different goal than before? It seems plausible that the student could subsequently produce a program that handles the latter goal, even without separate practice (by virtue of having internalized both list traversal and any other constructs required for the latter goal). Given that there are differences between knowledge schemas and strategy schemas (as both de Raadt et al. and Caspersen cite [16, 39]), students may require less direct practice to internalize a new pattern that built on already-internalized schemas.

These results frame our experiment for this study: our intervention, a lightweight planning lecture, shows students new ways to cluster subtasks of planning problems (Section 7.2.3). All of our participants had been learning and practicing writing list traversals prior to the experiment. The planning lecture showed new (relative to the course contents) high-level ways to decompose a planning problem into (potentially multiple) list traversals. This explicit planning instruction had not been incorporated into any of our host courses in our studies in the previous chapters and we were interested to see whether students would apply these high-level strategies to new problems based just on the

single lecture (without us telling them which solution style to produce, as other pattern-based studies have done [39]). Building on the cognitive literature described (and more extensively covered in Chapter 2), as well as our findings on task-level planning from our own studies, our project explores the following research questions:

> **STUDY-RQ1.** Can planning be taught at all in the first year? Or is it a topic that can only be covered after students have had significant experience with programming, software engineering, and/or computer science?
>
> **STUDY-RQ2.** Assuming it can be taught, what are the differences between groups of students in their ability to construct multiple plans, rank different plans, and talk about programs with a plan-oriented vocabulary?
>
> **STUDY-RQ3.** Assuming students can engage in these plan-oriented activities, how much of an intervention is necessary before they can do so? Does the class need to be restructured to make planning a focus, or can it be done with a lightweight intervention?

7.2 Study Design

At a high level our study had three components, which we applied slightly differently in each of two courses (Section 7.2.1). Sections 7.2.2 and 7.2.4 discuss the specific problems used on the assessments. The three components were:

1. A pre-assessment (Section 7.2.2) in which students were asked to produce solutions to 2–3 programming problems. In one course, which had more time for this, students were also given 2–3 solutions to different problems and asked to rank them (with justification) in order of their preference between these solutions.

2. A single 50-minute lecture (Section 7.2.3) on planning and design tradeoffs. The lecture used the pre-assessment problems to frame the discussion, showing different high-level ways to decompose a planning problem into a collection of list traversals. The same instructor gave the same lecture in both courses.

3. A post-assessment (Section 7.2.4) in which students were asked to (a) produce two solutions with different plans to each of 4 programming problems and (b) to preference rank between their solutions (with justification).

Ideally, we would have asked students to write multiple solutions with different structures in both the pre- and post-assessments. This would have let us gauge whether students could construct different plans even before the lecture, and also given us insight into which planning strategies students already knew. Unfortunately, we were unable to find a way to phrase this task that was not either extremely frustrating to students (because they could not understand the required task) or that did not essentially give away the answer. We still gain some insight into their background from their rankings

(Section 7.3.3), but determining how to establish a baseline more authoritatively remains an open question.

The questions in the pre- and post-assessment were carefully chosen to introduce some broadly-applicable strategies in multitask programming problems. In particular, we exposed students to problems with the following features:

- Noisy data that could be cleaned prior to performing the main computation.
- Flattened data that could be reshaped to a structure that was better suited to the main computation.
- Overly-long data that could be truncated to a prefix of interest for the main computation.

In addition, the posttest included computations that targeted a *projection* of the data (say to a specific field within an object). We did not emphasize projection in the pre-assessment as students had experience with this idea from other assignments in both courses. We used the lecture to discuss cleaning, reshaping, and truncating in the context of the pre-assessment problems. We also discussed various design tradeoffs that these offered, including impact on run-time efficiency, ability to adapt the solution to a different dataset, and readability and maintainability of the resulting code.

7.2.1 The Host Courses

We conducted the study in two first-year CS courses at different universities. Each course was taught by a different instructor. Students in both courses had some prior programming experience, but the nature of that experience differed both across and within the populations. We describe each course in turn.

- CRS-BROWNU is an accelerated CS1 course that compresses much of the first year into a semester. Students test into the course after one month in the department's regular CS1 course. Though it is open to all, most students in the course have some prior experience, usually with imperative or object-oriented programming in Java or Python. The course is taught in Pyret, a functional language with syntax reminiscent of Python.
- CRS-WPI is a CS2 course on object-oriented programming and data structures, taught in Java. Students feed into the course from one of two introductory courses taught in functional programming: one feeder (CRS-WPI-NVC) course is for novice programmers, while the other (CRS-WPI-EXP) is for students with prior programming experience. Students from CRS-WPI-NVC have seen little to no imperative programming prior to CRS-WPI, while students from CRS-WPI-EXP have prior experience similar to that of CRS-BROWNU.

These descriptions indicate that we actually have three student populations within our two courses, with interesting overlaps among them. Table 7.1 summarizes these populations. "Prior Experience" estimates students' programming experience prior to the current course: courses are 7 weeks long at CRS-WPI, while most students in CRS-BROWNU and CRS-WPI-EXP may have had a year or more

of prior programming. "Prior Imperative" indicates whether students had previously programmed imperatively. "Used Iterators" says how much students had worked with higher-order functions (such as map and filter) before the pre-assessment. These constructs featured heavily in CRS-WPI-EXP and CRS-BROWNU, but were introduced more lightly in CRS-WPI-NVC.

Table 7.1: Student populations in the study

Course	Course Language	Prior Experience	Prior Imperative	Used Iterators
CRS-BROWNU	Pyret	maybe more	yes	yes
CRS-WPI-NVC	Java	7 weeks	no	a bit
CRS-WPI-EXP	Java	maybe more	yes	yes

Both the CS1 course that preceded CRS-WPI and the CS1 course from which students placed into CRS-BROWNU followed a similar program design curriculum, *How to Design Programs* [48], taught in functional programming with Racket. While the overall assignments and lectures in these two CS1 courses were not identical, they taught largely the same concepts and the same method for developing programs and writing good test suites. While we cannot control for differences in the pre-university programming background of students participating in this study, the common CS1 foundations provide some degree of a shared baseline.

Prior to the pre-assessment, CRS-WPI had covered both kinds of `for`-loops for iterating over Java linked lists. In-class examples of `for`-loops consisted of simple list traversals that accumulated answers (such as summing a given field across a list of objects) or filtering out a subset of elements. The pre-assessment was the first assignment in the course on programming with lists and `for`-loops.

Sampled Populations

All in all, there were 75 students in CRS-BROWNU and 290 in CRS-WPI. While all students completed the study, our (manual) analysis uses a sample based on final course grade. Acknowledging that overall course performance (as indicated by formal course grade) could be a relevant factor, we aimed for a sample of 10 students from each passing grade (A, B, and C) in each course. Since CRS-WPI had two different feeder populations, we sampled separately from both feeder populations. Some subpopulations had fewer than 10 students in a grade band who submitted both assessments working individually (CRS-WPI allowed pair work). We had seven C-range students in each course, eight B-range students in CRS-WPI-EXP, and a full 10 students in each other population. In total, our sample included 27 CRS-BROWNU students, 27 CRS-WPI-NVC students, and 18 CRS-WPI-EXP students. The different grade bands did not manifest meaningfully in our analysis, so we do not discuss them further.

7.2.2 Pre-Assessment

The pre-assessments for both courses contained 2 or 3 programming problems; in CRS-BROWNU, students were also asked to preference rank among solutions to 3 additional problems. For the pre-assessment, we used the problems from Fisler *et al.* 's recent study [53], as they had been designed for

multi-linguistic contexts such as ours. We reproduce here the statements of problems that feature in our analysis, but defer descriptions of the other study problems to their paper or to Appendix E, which shows the problem statements and the solutions given in the ranking questions.

In CRS-BROWNU, the pre-assessment consisted of the *Palindrome*, *Sum Over Table*, and *Adding Machine* problems. Our analysis looks at *Adding Machine*, which features flattened data that could be reshaped into a list of sublists and a prefix of data (prior to the consecutive zeros) that could be truncated.; the full problem description and typical solution structures are described in Section 3.2 (page 29). For the purpose of discussion, we briefly revisit the solution structures here (we use these solution structures to bin students' solutions in our analysis, which we discuss further in Section 7.3.1):

> **Single Traversal**: Traverses the input data once, accumulating (a) the sum of the current sublist and (b) the output, returning the output when the consecutive 0s are detected.
>
> **Nested Traversal**: Like Single Traversal, but an extra inner loop re-traverses the sublists to compute their sums.
>
> **Reshape**: One traversal converts the pre-00 input into a list of sublists; a second traversal produces the list of sums of each sublist.
>
> **Clean**: One traversal truncates the data prior to 00; subsequent traversals follow one of the other solution structures.

We did not give the same programming problems in CRS-WPI because the instructor felt they were beyond what the students were prepared to do. Students did not know the string- or array-operations needed for *Palindrome*. The consecutive-position delimiter in *Adding Machine* would also have been new to the students from CRS-WPI-NVC. For CRS-WPI, we instead used two different problems from the Fisler *et al.* study, specifically the classic *Rainfall* problem and *Length of Triples*. *Rainfall* involves noisy data (negative numbers that should not be averaged) and a delimiter for the relevant data. *Length of Triples* asks for the longest concatenation of three consecutive elements from a list of strings.

For the questions that asked students to rank solutions in CRS-BROWNU, students were given multiple solutions to three programming problems, asked to state their preferences among the solutions, and told to justify their decision (we did not suggest criteria for the comparison). For this component, we used the same ranking problems as in the Fisler *et al.* paper [53] (their paper includes a link to their detailed problem statements; we simply adapted their solutions to the programming languages used in our courses). The specific problems were the *Rainfall* and *Length of Triples* problems given to the CRS-WPI students as programming problems, as well as the *Shopping Cart* problem that asked students to compute the total cost of a shopping cart after applying discounts when certain volumes of items were being purchased.

We did not include a ranking problems portion in CRS-WPI due to time constraints within that course. The programming problems were given within a larger assignment in that course, and the instructor did not feel the students could handle the additional work of the ranking problems in the time available for the assignment.

7.2.3 The Lecture Intervention

Within two days after the pre-assessment was due, one of the course instructors (from CRS-WPI) lectured about the problems in each course (guest lecturing in the course for which she was not the regular instructor). The lectures were not identical since the pre-assessment questions were different, but they covered similar content.

In CRS-BROWNU, the lecture started with a discussion of the ranking problems and the tradeoffs students considered. The instructor moderated discussion among the students, making sure that each of efficiency, aesthetics, maintainability, and code structure were given due attention. The instructor showed possible solutions to *Adding Machine*, explicitly discussing *reshaping* the input and *truncating* the input at the sentinel pattern as applicable strategies.

In CRS-WPI, the instructor showed multiple solutions to each of *Rainfall* and *Length of Triples*. The former was used to point out cleaning and truncating as strategies; the latter was used to point out reshaping. These solutions were posted for later reference. Again the instructor moderated a class-wide discussion among the students of the tradeoffs across these solutions to both problems.

In both lectures, the instructor described planning as the general task of allocating subproblems to traversals of the data. While this is a somewhat more code-focused definition that we might otherwise like, it was designed to give students a way to assess whether their two solutions would be "different" from the perspective of the post-assessment, though at this point they did not know what they would be asked to do on that assignment. However, the emphasis of the lecture was not on this definition but on concrete strategies, to help them build a vocabulary of planning operations.

7.2.4 Post-Assessment

For the post-assessment, we sought problems that were amenable to the reshaping, cleaning, and truncating strategies discussed in the lecture. We wanted problems that resembled, but were not identical to, the pre-assessment problems. The post-assessment contained four problems; students were required to (a) submit two solutions (with different structures) for each problem, and (b) state a preference between their two solutions (with justification). Teaching assistants in both courses were instructed not to give students much help in figuring out what a second solution would look like, but rather to refer them to their notes from each lecture.

We focused our analyses on two of the problems, due to their particular similarities with the pre-assessment problems:

> **Data Smoothing**: Given a list of health records with a numeric `heartRate` field, design a program `dataSmooth` that produces a list of the `heartRate`s but with each (internal) element replaced with the average of that element and its predecessor and successor. For example, given a list of health-records with heart rates `[95, 102, 98, 88, 105]`, the resulting smoothed sequence should be `[95, 98.33, 96, 97, 105]`.

This problem, like the *Length of Triples* problem which both courses saw in the pre-assessment, looks at three-element windows within an input list. This problem is a good candidate for reshaping.

Other viable strategies include first extracting all the heart-rates from the health records (resulting in a list of numbers to smooth), or simply doing the entire computation in a single traversal of the input data.

> **Earthquake Monitor**: Write a program that takes a month and a list of readings from an earthquake sensor. In the input, 8-digit numbers are dates and numbers below 500 are readings for the preceding date. Produce a list of reports containing the highest reading for each date in the given month. For example, given the list `20151004 200 150 175 20151005 0.002 0.03 20151207` and 10 for the month, the program should yield `[report(4, 200), report(5, .03)]`.

This problem resembles *Adding Machine* in having sublists within the data, which makes it a candidate for reshaping. Like both *Adding Machine* and *Rainfall*, it has a sentinel (in the form of data from a later month). It could be approached with a single traversal that accumulates the max value per date, a cleaning phase that restricts the input data to the desired month, or reshaping prior to computing the reports.

7.3 Analysis and Findings

In discussing our findings, we find it useful to organize our results by course, first discussing what we observe from CRS-BROWNU, then contrasting those results to the data from CRS-WPI. A direct comparison of the results from the two courses is not meaningful due to differences in the students' backgrounds and the smaller set of questions used with CRS-WPI. We still find value, however, in seeing how two groups of students with some similarities in their backgrounds fared in this experiment. The analysis will show interesting differences across the two courses; this gives a much more realistic picture of the proposed intervention than if we had reported on one course alone, despite the differences in the details of problem selection.

7.3.1 Coding Analysis of the Data

Coding the Solution Structures

We coded individual solution structures by (a) enumerating the subtasks for each programming problem, and (b) writing a regular expression to capture the clustering and sequencing of subtasks within the solution. We grouped the regular expressions into larger bins based on how they handled the main strategies (reshaping, cleaning, truncating) covered in this study. For *Data Smoothing*, for example, the subtasks were (E)xtracting the heart-rate, (S)moothing the data, and optionally (R)eshaping the data. A code of R;(E+S) captures a solution that reshapes the data then extracts the heart-rates and computes smoothed values in a subsequent traversal. This solution would be classified as "Reshape First". Section 7.2.2 outlined four larger bins that arise for *Adding Machine*; similar terms describe bins for the other problems.

Coding the Preference Criteria

Ranking criteria were processed through open-coding. Four main themes emerged:

1. **Efficiency**: Mentions runtime, performance (*e.g.* Big-O), memory use, or efficiency of operations used
2. **Structure**: Discusses use of specific operations, constructs, or clusterings of subtasks
3. **Aesthetics**: Readability, comprehensibility, or reflecting the problem statement in the code; often positive tone
4. **Maintainability**: Mentions ability to maintain, debug, or adapt the solution to new data

7.3.2 The view from CRS-BROWNU

The data from CRS-BROWNU suggest that our intervention lecture had a noticeable impact on students' planning behavior. Figures 7.1 and 7.2 show the structures that CRS-BROWNU students used in *Adding Machine* on the pre-assessment and *Earthquake Monitor* on the post-assessment, respectively. We contrast these two problems because they have similar attributes: flattened sequences of structured data, with a computation (sum or max) to be performed on a delimited subsequence of relevant data. Students took a variety of approaches in the pre-assessment, with some using reshaping. Usage of reshaping jumps significantly ($p < .006$ with a McNemar's test) in the post-assessment: about 70% of CRS-BROWNU students used reshaping in one of their two *Earthquake Monitor* solutions. Even in *Data Smoothing*, roughly the same number of students chose to reshape as did a single data traversal. Thus, there is evidence that CRS-BROWNU students learned and applied the reshaping strategy from the single lecture.

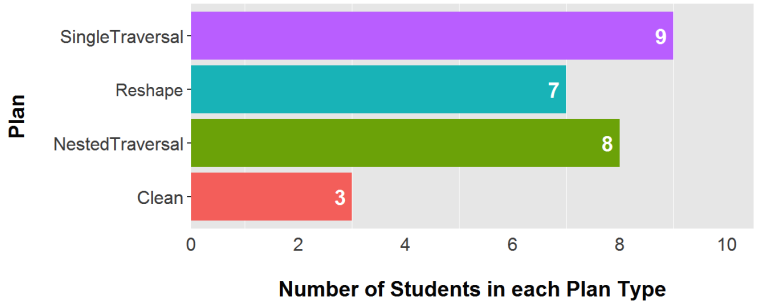


Figure 7.1: Structures of *Adding Machine* solutions in CRS-BROWNU from the pre-assessment

Significant contrasts also arise when we examine CRS-BROWNU students' ranking preferences between the two assessments. Figure 7.3 shows the criteria that students used when ranking the *Rainfall* and *Shopping Cart* problems from the pre-assessment, alongside those for *Earthquake Monitor* on the post-assessment (these are for the entire population of 75 students, not just the sampled subset). The three most common categories (efficiency, structure, and aesthetics) are the same in the two pre-assessment problems, though aesthetics is more prominent on *Shopping Cart*, which involves

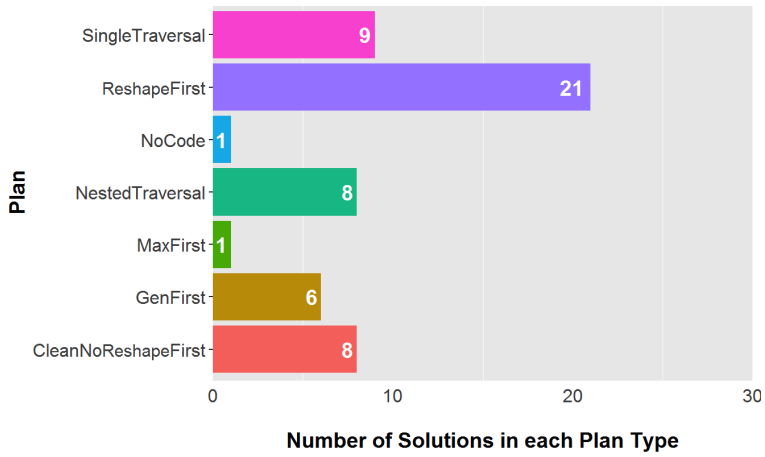


Figure 7.2: Structures of *Earthquake Monitor* solutions in CRS-BROWNU from the post-assessment

more subtasks than *Rainfall*. In the post-assessment, efficiency is cited significantly less often ($p <$ .0001) while maintainability grows significantly ($p = .02$). Maintainability is a potential issue for both *Rainfall* and *Shopping Cart*, but particularly the latter (as the store could begin to offer more or different discounts). Table 7.2 shows the evolution of these criteria on a per-student (rather than aggregate) level, summarizing numbers of CRS-BROWNU students who raised various criteria in each of the assessments. The table shows that many students both dropped and gained criteria over the course of the study.

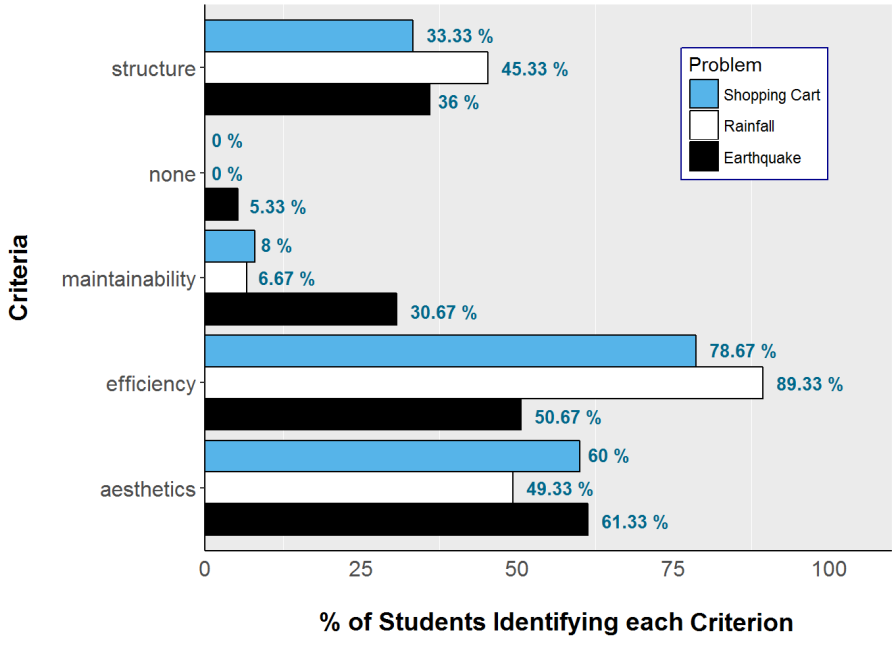


Figure 7.3: Criteria CRS-BROWNU students cited while ranking solutions to *Rainfall* (pre), *Shopping Cart* (pre), and *Earthquake Monitor* (post)

All in all, the lecture had the impact we hoped for in CRS-BROWNU: students showed their ability to produce solutions with multiple plans (only 4 students had the same high-level plan on *Earthquake Monitor* and only 5 had the same high-level plan on *Data Smoothing*; only one student overlaps these two groups), most students raised more issues when discussing tradeoffs among solutions, and many

Table 7.2: Criteria raised by CRS-BROWNU students across pre- and post-assessment rankings

Criterion	Pre, not post	Post, not pre	Pre and post
Efficiency	31	-	36
Structure	24	9	15
Aesthetics	14	1	35
Maintainability	7	20	3

students changed the solution structures that they preferred in the post-assessment (which is merely a sign that the lecture impacted their thinking, not that their analyses necessarily grew more accurate).

7.3.3 The view from CRS-WPI

Contrasting the CRS-BROWNU data with those from CRS-WPI paints a more nuanced picture of the impact of the single lecture. In particular, taken as a whole, the lecture's impact is less significant; we also see interesting differences between the CRS-WPI-NVC and CRS-WPI-EXP subgroups of CRS-WPI. We also see differences between CRS-BROWNU and CRS-WPI-EXP, who had been working in different programming languages despite a fairly common curriculum (and common programming language) just a month or two prior to the study.

Figure 7.4 contrasts the *Earthquake Monitor* solutions across all three populations in the post-assessment. Two observations jump out. First, a significant percentage of students in CRS-WPI were unable to solve the problem at all (the "No Code" group): of the 45 students sampled, 11 turned in no solution for *Earthquake Monitor* (9 from CRS-WPI-NVC, 2 from CRS-WPI-EXP), while another 6 students turned in only one solution. Of those who turned in only one solution, half used reshaping while the others did a straightforward loop-based traversal or nested traversal. In contrast, there was only one case of a "No Code" solution in CRS-BROWNU. We suspect that the "No Code"s came partly from the lack of programming experience in CRS-WPI-NVC and partly from students running out of time (*Earthquake Monitor* was the last problem on the assignment, which was due just before students left campus for Thanksgiving, a mid-course holiday.)

Setting aside the "No Code" students, the dominant solution structures differ across the three populations: "Reshape First" dominates in CRS-BROWNU, "Single Traversal" dominates in CRS-WPI-NVC, while these two are fairly even in CRS-WPI-EXP. This suggests that reshaping strategies may be harder for students to adopt with only novice programming experience.

On *Data Smoothing* (Figure 7.5), the two CRS-WPI populations are more similar to each other, with much heavier use of "Single Traversal" solutions, especially compared to the dominance of "Extract First" solutions in CRS-BROWNU. Here, we strongly suspect that programming language constructs were a factor. Most of the CRS-BROWNU students used a built-in `map` function to extract the heart rates from the health records. While Java 8 provides `map`, it is somewhat clumsy to use and only a handful of CRS-WPI students had been exposed to it by the class (in an optional lab for students who wanted a challenge assignment). A basic Java `for` loop is straightforward for *Data Smoothing*, so we should hardly be surprised that students used it.

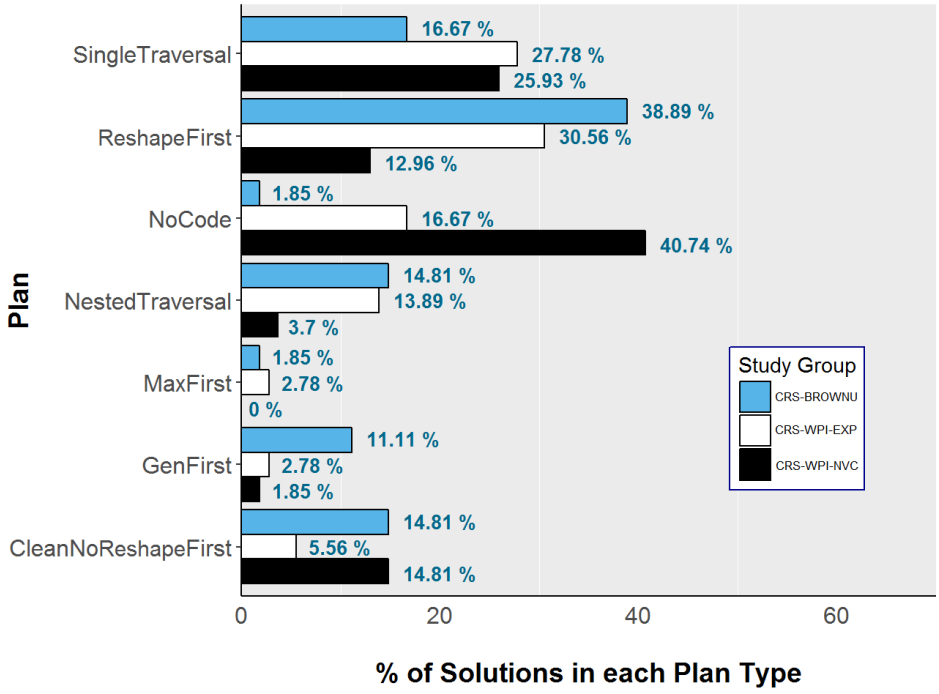


Figure 7.4: Structures of *Earthquake Monitor* solutions from the post-assessment

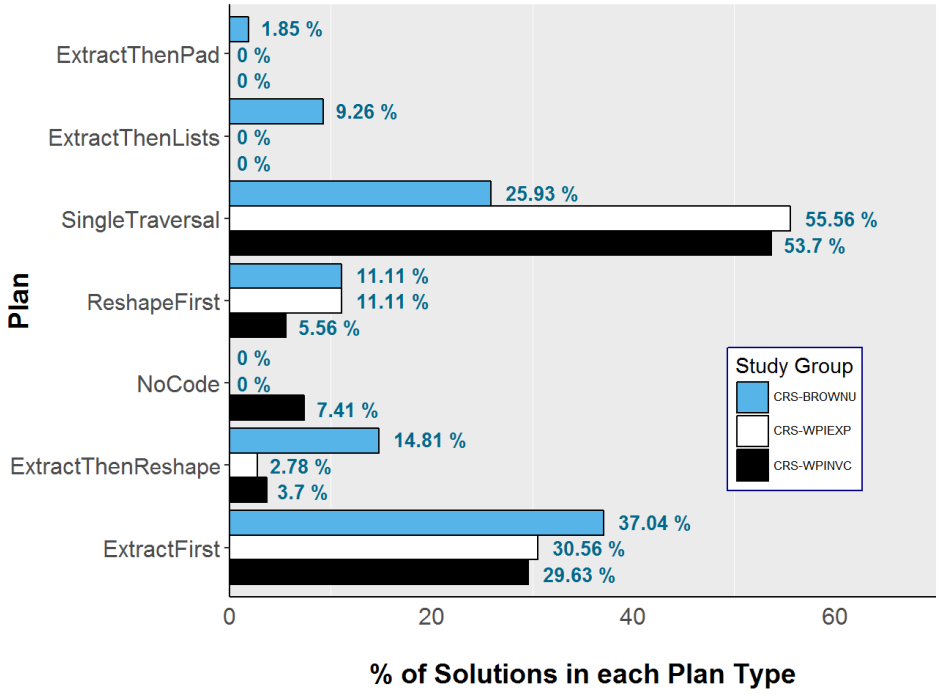


Figure 7.5: Structures of *Data Smoothing* solutions from the post-assessment

Of the 45 CRS-WPI students, 16 produced two *Data Smoothing* solutions with the same high-level structure. This suggests that many CRS-WPI students didn't really understand the idea of multiple program plans just from the single lecture, or perhaps that the alternate plans for *Data Smoothing* were too subtle for many students. Of the strategies provided by the lecture (cleaning, reshaping, and truncating), only reshaping applies to *Data Smoothing*; if reshaping was indeed too hard for students, they would have been left without named strategies to apply to the problem. Instead, they would have to have understood the more general point about different structures allocating tasks

differently to traversals of data. The pre-assessment data did not shed light, as all but one student used a "Single Traversal" structure to program *Rainfall*. Prior experience is not the explanation either: these 16 students were roughly evenly split between CRS-WPI-NVC and CRS-WPI-EXP. Whether CRS-WPI students would have understood this idea better had they also done ranking problems on the pre-assessment is a question for future studies.

7.3.4 Preference Ranking of Own Post Solutions

Recall from Section 7.2.4 that students were asked not only to generate two different plans for the solutions, but also to state a preference between the two. Here, we study the outcome of this activity from both courses (CRS-BROWNU and CRS-WPI).

All the CRS-BROWNU students submitted preferences and mentioned some criteria. Figure 7.6 shows their preferences for *Earthquake Monitor* solutions. For now, we ignore the colors and look at each bar as a whole. In so doing, we notice that students have a significant preference for reshaping first, which suggests at least the ability to recognize the lecture's views on structures that better decomposed plans. The *Earthquake Monitor* bars of Figure 7.3 also show the variety of criteria that the students mentioned.

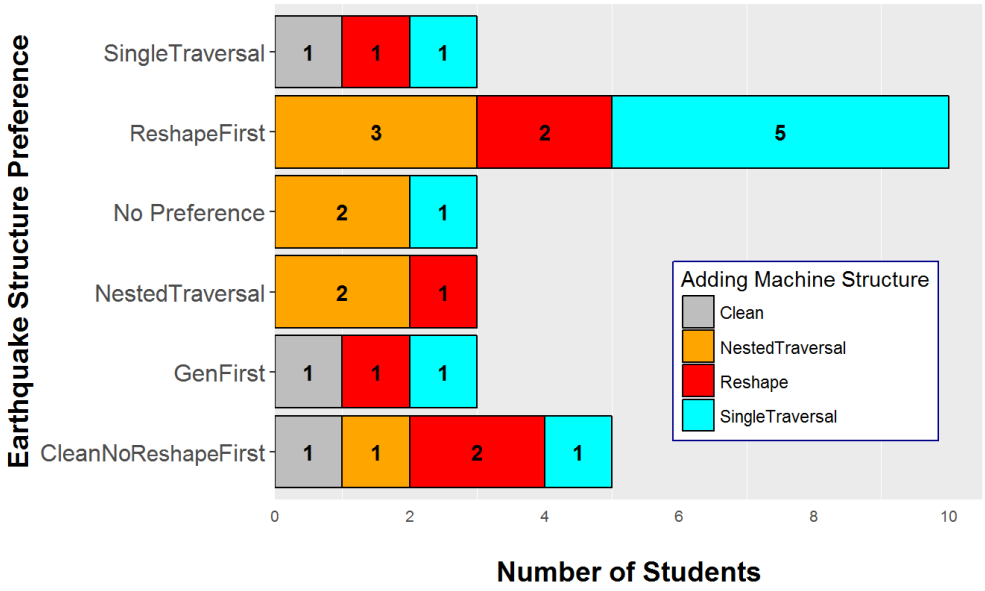


Figure 7.6: Comparing individual CRS-BROWNU students' *Adding Machine* structures (pre) to that of their preferred *Earthquake Monitor* solution (post)

We now contrast this to CRS-WPI. Figure 7.7 shows which of their solutions students preferred. Of the 34 students who submitted a solution for *Earthquake Monitor*, all but 5 had at least one "Single Traversal" or "Nested Traversal" solution, yet half preferred a reshaping- or cleaning-based solution. When we now focus on their criteria, however, we see something more disturbing. Of the 45 students sampled: only 15 mentioned any criteria at all; 9 didn't submit a ranking; the other 21 just described the implementations (6 of these were from CRS-WPI-EXP, the rest from CRS-WPI-NVC). Among the 15 CRS-WPI students who did describe criteria, code structure and aesthetics came up most often (9 and 10 instances, respectively), while efficiency got only 3 mentions.

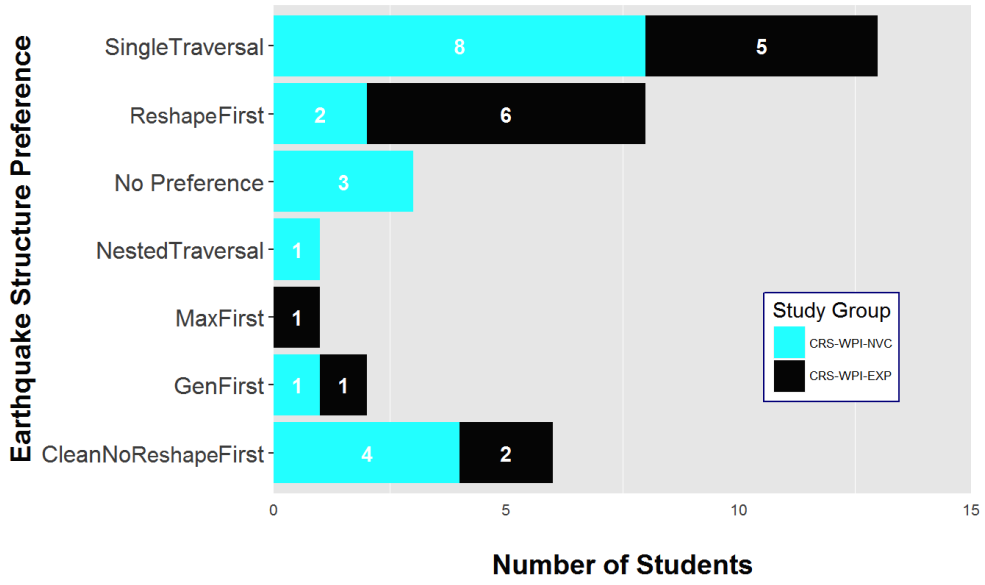


Figure 7.7: CRS-WPI students' preferred *Earthquake Monitor* structures (post)

The contrast between the courses in students' ability to discuss solutions by attributes is striking, and not readily explained. Neither course had practiced this skill, either explicitly or implicitly that the instructors can recall. Both courses had covered rudimentary big-O prior to the pre-assessment, so students at least had "efficiency" in their vocabulary. Therefore,we do not yet have a proper explanation for these differences.

7.3.5 Changes in Solution Structures

Changes in the solution structures that students wrote from the pre- to the post-assessment might indicate that the planning lecture had impact. Because students were asked to write two solutions in the post-assessment, it might not be clear which one to compare. However, given that students were asked to rank their two solutions, we believe it is reasonable to compare the structure of the pre-solution with that of the preferred post- one.

We present this comparsion for CRS-BROWNU, using *Adding Machine* from the pre-assessment and the preferred *Earthquake Monitor* solution from the post-assessment. This is a meaningful comparison due to the similarities in tasks between these two problems: both involve flattened data, a numeric calculation on subsegments of the data, and consideration of only a subset of the data.

Figure 7.6 shows this information. Now we can consider the colors. Each row shows a structure for the preferred post- solution binned by their pre- solution structure. The comparisons are per student. Perhaps the most interesting feature of this graph is its lack of a clear internal pattern: students from each *Adding Machine* structure are dispersed across the *Earthquake Monitor* bins, and each *Earthquake Monitor* bin is populated with students from multiple *Adding Machine* structures. Our key takeaway from this graph is that the lecture got CRS-BROWNU students to think about plans and tradeoffs, with many reconsidering choices that they might have made reflexively during the pre-assessment.

A similar graph for CRS-WPI is not meaningful, since all but one student produced the same

structure for *Rainfall* on the pre-assessment. However, we do see diversification in students' preferred solutions in the post-assessment. While "Single Traversal" solutions remain the most popular among CRS-WPI students in both the *Earthquake Monitor* and *Data Smoothing* problems on the post-assessment, there is considerable diversity in the *Earthquake Monitor* preferences (which admits more interesting plans). Whether this diversification was caused by the lecture, or by the relatively greater difficulty of *Earthquake Monitor* compared to *Rainfall*, is open to question.

7.4 Threats to Validity

Students' prior programming experience, including the kinds of problems they have been exposed to and in which programming languages, is a significant factor in studies of planning (Section 7.1). We have a rough characterization of our participants' prior experience: students in CRS-WPI-EXP and CRS-BROWNU all took courses designed for students with non-trivial programming experience prior to starting at university (for CRS-WPI-EXP, that course was the one preceding CRS-WPI itself; novice and experienced students feed into a common CRS-WPI as a second course). The nature of that experience could vary significantly across students (in practice, most students had experience in at least Java). More careful accounting of the details of prior background would help refine conclusions in planning studies.

Student motivation to master programmming could affect how seriously they engage in the task of writing two solutions. All students in our study courses were likely interested in majoring in computer science, or at least studying it in some depth. This might give them greater motivation for understanding the planning concepts we discussed, and they might also have more aptitude for computational problem composition and decomposition. It would be interesting to perform similar studies on non-major populations, in particular exploring how much training they require before they appreciate planning. Data on this question will likely evolve in the next few years, given the growing adoption of computing in middle- and high-schools in many countries, which might prepare students for this material.

As noted in Section 7.2.2, we did not use the same questions in both courses. The questions we gave to CRS-BROWNU reflected the richness and parallel structures across pre and post that we would ideally like to explore, but some of these questions were beyond what the CRS-WPI-NVC students would be able to handle based on the problems presented in class (for example, they had not yet done any string manipulation, so *Palindrome* would not have been approachable). The extent to which this is a problem depends a bit on the nature of the conclusions one wishes to draw. Had the study results been positive with all three groups of students and we drawn conclusions about relative performance, the variations in questions would confound the results. As it were, however, the CRS-WPI students did not fare as well, despite having arguably easier questions. In this context, the difference in questions is not as significant.

In a prior review of this work (for a submission for publication), one reviewer posited that our questions were biased in favor of functional programming. We drew our questions from a collection

of planning problems [53] that had been curated by multiple instructors, some of whom vastly prefer functional programming and some who vastly prefer imperative programming. Given this curation, we were confident that our problems were reasonable ones to pose of students working in various programming styles.

7.5 Discussion

We now discuss in some depth the many questions we have raised throughout our discussions of this study.

1. The studies we conducted were roughly near the end of the first semester (in US terminology). If we could wait another semester and get students at the end of the first year, we might find that even those without prior computing are much more sophisticated programmers. It would be especially interesting to see whether, at that point, the CRS-WPI-NVC group performs like the CRS-BROWNU and CRS-WPI-EXP populations did in our first-semester study.

2. What if we were to do the same study in an imperative setting? Would students who have primarily been exposed to `for` loops, and not seen higher-level constructs like mapping and filtering, readily grasp the idea of planning? Clearly, exposure to such constructs helps students understand planning concepts; but while sufficient, is it also necessary? This is a topic that needs further study.

   In addition, programming classes may also start changing in flavor to keep pace with languages. Increasingly, imperative and object-oriented languages have adopted some of the basic features of functional programming, such as higher-order functions and higher-order operators (whether in the form of functions like `map` or `filter` or as syntactic constructs such as comprehensions). The pedagogy is also catching up: new (editions of some) Java books provide a thorough discussion of using higher-order functions and functional style [123]. Therefore, it is no longer necessary to switch to a functional language to teach a more functional style of programming. This removes a significant source of friction that introductory course instructors sometimes feel. Nevertheless, this does require adopting a new style of programming, which may be considered a significant intervention in some departments, far removed from our view of it as a "lightweight" one.

3. What do students already understand about planning before our intervention? Can we phrase our "construct two different plans" task (Section 7.2.4) in a way that meaningfully measures student knowledge and ability?

4. The fact that students changed their ranking criteria in the post-lecture assessment does not mean they genuinely changed their preference: they may be reflecting what they believe the course staff want to hear. We can probe their true beliefs by giving them significant problems that they can decompose in different ways and checking whether their decompositions match their stated preferences. In turn, creating multi-part exercises where they have to create an initial solution and

later modify it—so that issues like maintainability come to the fore—can help reinforce the value of some plans over others.

5. When are students ready to understand and adopt a planning strategy like reshaping? There are many reasons why several CRS-WPI-NVC students were unable to use it: maybe they didn't understand it in the first place; maybe they understood it but didn't see its value (especially if they have had no experience building larger systems, it can be hard to see the benefit of an abstraction); or, having passed both hurdles, they may have been unable to implement it.

6. We find puzzling the contrast between CRS-BROWNU and CRS-WPI students when it comes to ranking their own solutions. Do the CRS-WPI students not appreciate tradeoffs at all? Do they appreciate them but lack the vocabulary to articulate them? Perhaps they simply did not understand what the problem was asking for?

   There is one important factor that may have played a part. The CRS-BROWNU students had to preference-rank in their pre-assessment, and discussed this during the intervention lecture. Therefore, it is possible they had already had "training" to think about this issue. However, given that they already mentioned several criteria—instead of just describing implementations—in their rankings, this cannot be the whole explanation. Having the CRS-WPI students preference-rank in the pre-assessment would clearly help shed light on this phenomenon.

Going beyond these questions, future studies could include control groups or find ways to determine what kinds of solutions students can imagine as part of the pre-test. Given the relationship between planning and prior exposure to similar problems, control groups would likely need to come through the same sequence of courses as study participants. Naturally, this requires a different intervention design than ours, which covered planning as one of the regular lectures within the participating courses.

Chapter 8

Discussion and Conclusions

We started this project with the following broad question:

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

Our overall goal with this dissertation is to develop a *conceptual framework* that describes how HTDP-trained novice programmers design programs to solve multi-task programming problems. We concretized this goal along four research questions, asking (1) what specific program design skills we observed students apply in their programming process, (2) what interactions exist between these design skills and what the impact of these interactions are on how students solved programming problems, (3) how do students' design skills evolve over a CS1 course, and (4) how do students approach novel multi-task programming problems. Overall, the project aims to create a theoretical grounding for how students use the design recipe, which can be used towards refining the practice of teaching the curriculum, or informing the design of learning activities and teaching artifacts (*e.g.* tools and assessments) that are based on, informed by, or towards the teaching of, the curriculum.

In the following sections, we synthesize our findings around our research questions and describe how our findings inform the teaching of HTDP, and our broader reflections toward practices in teaching introductory CS. Section 8.1, revisits each of our dissertation research questions: we summarize and synthesize our findings from our studies as they relate to each question. Our discussions within each research question synthesize the discussions we made in the "*status update*" subsections at the conclusion of each of our study chapters (Chapters 3 to 6). In Sections 8.2 and 8.3, we discuss how our findings inform the teaching of HTDP-based classes/curricula and introductory CS in general, as well as discuss concrete recommendations for teaching program design. Sections 8.5 and 8.6 discusses the threats to validity of our work and several paths toward continuing our research on planning and program design.

8.1 Answering the Dissertation Research Questions

8.1.1 Students' program design skills

> **DRQ1**. *What program design skills do* HTDP*-trained students apply/practice when developing solutions for multi-task programming problems?*

We distilled three main design skills that students practice, from our observations of students solving multi-task programming problems. We also found that students varied in the way they demonstrated these skills: we captured these variations using the SOLO levels of conceptual complexity (Chapter 4). The following are the broader-level skills that we found from our data, and the more specific subskills within each that we found students apply:

1. Working through core problem tasks
    a) Decomposing tasks and composing solutions
2. Using code-level plans
    a) Composing expressions to build function bodies
    b) Meaningful use of patterns
3. Applying HTDP-prescribed design practices
    a) Methodical choice of tests and examples
    b) Leveraging multiple representations of functions

Skill: Working through core problem tasks

Students *decomposed problems into tasks and composed task-solutions* at different levels. Supporting Rist's findings, students generally started their programming process by eliciting tasks from problem statements. Depending on the familiarity of the problem, students may identify tasks that are familiar from knowledge they've gained outside of class (*e.g.* averaging), they may identify tasks they've seen from their class exercises or homework (*e.g.* summing elements), or they may identify tasks, driven by particular characteristics of a given problem (*e.g.* reshaping the input based on windows of data in the input).

Some students apply this skill at a *relational* level: they concretely describe the relationships between the identified tasks and use the insight from the described relationships to articulate an overall plan around the tasks. These students often write their code in the context of the task-level plan they articulate: they delegate tasks into appropriate functions or expressions, and the composition of their code is guided by the task-relationships they described. When describing their program, they describe the tasks embodied by their code and the inter-operation of the tasks.

Some students focus solely on the implementation of the identified tasks in code without articulating an overall plan around the tasks. They often do not concretely identify how the tasks relate to each

other as they instead focus on the code-specific mechanisms of their program, even when describing their overall program. Some of these students do not delegate tasks into appropriate functions or expressions (*unistructural* level); others attempt to delegate tasks to separate traversals (*multistrucrural* level). These students, however, still fail to concretize task-relationships, which often leads them to struggle (and fail) in composing their code.

Skill: Using code-level plans

Two subskills are involved in students' application of code-level plans. Firstly, students *compose expressions to build function bodies*. Most of the students in our studies applied this at a *relational* level: when describing the code-specific mechanisms of their programs, they are able to correctly describe the low-level syntax structure and evaluation mechanisms of the expressions and function calls in their code. They are also able to correctly write syntactically-correct functions and expressions. This is not surprising given the timing of our studies: students had been writing functions and expressions for at least three weeks and have taken at least the first exam by the time we collected data. We expect that had we assessed students from the start of their course, students would be applying this skill at lower SOLO levels as they begin to learn the syntax and semantics of the course's programming language. In general, this skill is the most mechanical of the design skills we identified.

To solve our multi-task programming problems, students also need to *use code patterns meaningfully*. This is where students often struggle in their work: students blindly retrieve the list template (by virtue of the list-type input) and populate the template with code for problem-tasks without thinking about the limitations of the template pattern in the context of the traversal tasks they are implementing. They thus inappropriately conflate multiple traversal tasks in a single template (*multistructural*). The more successful students (*relational*) think about how the tasks impact the use of the patterns they retrieve and separate the traversal tasks in a meaningful way, for example, by appropriately delegating tasks to patterns (*e.g.* multiple templates) or parts of patterns (*e.g.* multiple accumulators).

Skill: Applying HTDP-prescribed design practices

We found two subskills that students applied in their use of the HTDP design recipe. Students varied in how they *methodically selected and wrote tests and examples*. Many students simply copied and worked off of the example provided in the problem statements, or wrote multiple examples and test-cases, but which illustrated essentially the same scenarios, without thinking about how the consideration of different input and output scenarious might impact the design of their code (*unistructural*). Some students explored different scenarios for the input examples, but only mechanically and did not use these to guide the design of their code (*multistructural*). A few explored a broad range of inputs and attempted to write test cases that captured different input-to-output scenarios. They explicitly describe how their functions' execution relate to the different scenarios illustrated by the examples and test cases they wrote (*relational*).

Students also varied in how they described or reasoned about how the different steps of the design

recipe related to each other. Many students jumped into writing their code and did not use the design recipe at all. Some blindly followed the design recipe without any insight about how the recipe steps informed their work (*unistructural*), describing the recipe simply as a static process to follow. A few talked more concretely about how different recipe steps captured different details about the problem, but still lacked an insight about how the steps relate to each other (*multistructural*). Few students concretely articulated relationships between the programming artifacts produced from each step (*e.g.* how the data definition drives the structure of the template; how the execution of their program connects to a test space); this usually happened when students attempted to solve a multi-task problem that was mostly new to them (*i.e.* Max-Temps), which shows some evidence of students falling back on certain design recipe steps when rethinking their solution.

Students reflected less about how to use the design recipe as an inter-operating set of steps towards developing a solution, than they did writing examples or test cases independent of the overall process. Given that our host courses graded students heavily on the thoroughness of their test cases and don't engage students in class activities that directly practice and focus on explaining how the different design recipe steps inter-operate (which we ask them to do in our semi-structured interviews), this isn't surprising. Overall, students seem to develop a *mechanical habit* around following the design recipe as a series of steps, but not necessarily an *insight* around how each recipe step is a technique towards building an understanding of the problem-space and which they can use as a guide towards the design of their solutions; even the HTDP instructors who participated in our validation study (Section 4.7) confirm this mechanical and uninsightful use of the design recipe. These issues raise potential deficiencies in how the recipe is taught in our host courses; we discuss these issues at various points in the succeeding sections.

8.1.2 Interactions between program design skills

> **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?*

We found a critical interaction between students' *use of patterns* and their *task-level planning*. Interestingly, other important interactions we found instead involved interactions between students' use of their design skills and the variety of problems students have seen in class. Our findings suggest an important influence of these interactions on how students applied their design skills towards solving multi-task programming problems.

Interaction: Meaningful pattern-use and task-level planning

All of the problems we gave students were list-based problems. Students thus retrieved the list template pattern, the accumulator pattern, or list-abstraction patterns, as prompted by the input type or by tasks they identify that they've seen previously (*e.g.* summing a list via a template-based summing function,

an accumulator-style summing function, or by using `fold`). Our findings suggest an interaction between students' *understanding and use of patterns* and their *task-level planning*. We found that students who are applying both skills at the *relational* level generally developed correct (or close to correct) solutions. Even when students realize that specific tasks need their own traversals, they still need to identify the limits of the patterns they're using relative to the tasks. In other words, they need to identify how the problem-tasks would impact their use of their retrieved patterns (such as the need for separate traversal functions). We illustrate these findings below.

Students who retrieved the list template pattern mechanically only copied the syntactic structure of the pattern, without articulating its underlying concept. They would blindly populate the template with task-related code without concretely identifying the limits of the pattern relative to the tasks they are implementing. This often leads students to get stuck with code that ineffectively conflates different task-related code within the template. A recurring example of this is in Rainfall, where students ineffectively combine the code for *divide*, *sum*, and *count*:

```
(define (rainfall input)
  (cond [(empty? input) ... ]
        [(cons? input) (/ (+ (first input) (rainfall (rest input)))
                          (+ 1 (rainfall (rest input))))]))
```

Some students who implement code similar to the above recognize that the *sum* and *count* tasks each need their own traversals. Because they don't identify the limit of the list template relative to the tasks they're implementing, they ineffectively combine them within the same template. Students who concretely identify the template's limit recognize that a template only handles one task and address this by pulling each of *sum* and *count* into their own separate template functions. They then use their task-level plan to inform the composition of the functions into the overall average function.

Some students use the insight from their task-level plan, along with an understanding of a pattern's underlying concept, to guide the modification of the patterns they retrieved. For example, some students who use the accumulator pattern to implement *average* recognize that they need to track the results of two tasks (*i.e. sum* and *count*). The students in our courses have typically only seen accumulator functions that used only one accumulator. Students who articulate the accumulator's underlying concept, however, recognize that they can use an additional accumulator to keep track of multiple tasks. In this case, students concretely associate tasks with separate accumulator parameters, and their understanding of the accumulator pattern's underlying concept enables them to go beyond the typical structure of accumulator-style functions that they have seen.

Interaction: Problem variety and leveraging the design recipe

In Max-Temps, some students got stuck because they used data definitions and templates that were inappropriate for the type of input they were writing functions for. In general, students struggled to figure out how to write a data definition (or template) for the input data in the Max-Temps problem. Our findings reveal that this difficulty is related to the limited repertoire of data that students had practiced

their design techniques on. Students had only experienced designing functions for which the basic list data definition and template sufficed; they had not experienced using their design techniques for data such as the one in Max-Temps. Students also mentioned that they had not encountered problems that had the kind of data that Max-Temps had.

Given Max-Temps' list-type input, the students had at least the list template schema to start with, but manipulating the schema they know to design a template for a list with elements that had specific roles (*i.e.* delimiters indicated windows of elements to be processed separately) was a significant design challenge for the students. This suggests two things: firstly, that the limited set of list problems that students practiced on in class (*i.e.* examples, labs, or homework) may have constrained students' knowledge or understanding of how to write data definitions and templates for lists. They may have instead developed a *mechanical habit* of writing the basic list data definition and template from which they struggled to deviate from, or that they potentially did not develop the idea that the basic list data definition and template can be manipulated based on the context of the input (beyond changing the type of the individual list elements). Second, this suggests the need for students to do various data design activities to practice designing functions for various data situations they might encounter beyond those requiring the basic list data definition and template. This seems critical to drive home the idea that the practice of designing data definitions and templates does not just lead to one form of data definition or template, but that these are *malleable* constructs that can be manipulated based on the context of the input (*e.g.* lists with *significant* elements, *i.e.* elements with distinct roles such as sentinels or delimiters).

Interaction: Problem variety and meaningful use of patterns

Some students also struggled with adapting the patterns they've learned to new contexts. In Rainfall, for example, students got stuck figuring out how to handle the -999 sentinel; in Max-Temps and Adding Machine, they got stuck figuring out how to recur over a modified suffix of the list and not just the tail (`rest`) of the list. None of the students had worked on list problems that terminated prematurely at a specific element (-999 for Rainfall) or element pattern (0 0 for Adding Machine) rather than at the end of the list (*empty*), or list problems that recurred on anything other than the `rest` of the list (as indicated by specific delimiter-role elements). This suggests that students may have attributed some form of functional fixedness over parts of the template (*i.e.* the base-case and recursive-call parts), having not seen or practiced on problems that required using or modifying these template parts beyond the basic `empty` base-case or recursion on the `rest` of the list; this is in a similar vein as the habitual, mechanical writing of the design recipe steps in the previous interaction. This likewise suggests the need for students to be exposed to, and practice on, problems that require manipulating the base-case and recursive-call parts beyond their typical use. Such problems may help put forth or drive home the idea of the *malleability* of *all* parts of the template, beyond the typical template "holes" that students fill in with problem-specific computations.

8.1.3 Evolution of students' program design skills

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

Students applied program design skills at different levels of conceptual complexity.

An important benefit to collecting students' programming process data *in situ* through think-alouds and semi-structured interviews is that our data enabled us to capture the gradations at which students applied the program design skills that we observed them demonstrate. Through a grounded theory-based synthesis of both student and instructor data, we found five core design skills that students practice when solving multi-task programming problems, and that students applied each skill at different levels of conceptual complexity. We used the SOLO taxonomy as a framework to capture the skill variations in conceptual complexity: the multi-strand SOLO taxonomy of design skills is a key contribution of this work. We have also begun to qualitatively validate this taxonomy in two ways: firstly, we used the taxonomy to categorize data from students beyond the pool we used to design the taxonomy; second, we checked with other instructors to see whether the taxonomy captures the same things they look for when assessing how students apply their design skills. We have also used the taxonomy as a descriptive framework to describe our findings in our other studies of students' program design work.

Students evolve in different skills at different paces.

We assessed how students used their design skills at multiple points throughout their CS1 course. We found that they can be at different levels for different skills at a given time; students do not necessarily progress through the different skills simultaneously. We've described some of the effects of students applying the skills at different levels in ***DRQ1*** and ***DRQ2*** (Sections 8.1.1 and 8.1.2); one example is that even when students are *relational* for *building function bodies*, they struggle to write correct solutions for multi-task problems if they apply *task-decomposition* at the *multistructural* level or lower because they fail to make concrete connections between the tasks they're implementing, which is critical to informing the composition of their code.

Students may show non-monotonic progression of skills.

In addition to a non-simultaneous evolution of skills, we also found some students demonstrate some skills at a lower level than they have previously demonstrated. We have some hypotheses that may explain this:

- *Students may not have internalized the skills they displayed at earlier sessions.*
  Some students may not have explicitly internalized their use of certain skills as good practices, so they may not be consistent in applying them intentionally as they solve programming problems (*e.g.* writing examples and test cases to develop an understanding of a problem space). This

is related to the *value judgments* that they may attribute to the use of certain design skills or practices.

- *The problems used in our study may have pushed students towards particular skill levels.*
  The problems we selected for our study may have affected how students performed the skills; this could thus be an instrument or study design problem. For example, in one of the study sessions we conducted, students described their design process on a homework problem for which an earlier problem on the same homework provided a usable helper function. This may have prompted some students to make comments at a higher task-decomposition skill level than they would have if they solved the problem unscaffolded.

- *The drops in skill level may reflect the level of problem complexity at which students can apply their skills.*
  This is a useful construct for assessing when a curriculum prepares students to work at various levels; in other words, this idea can be used for contrasting when students *can* versus *do* achieve certain skill levels relative to the curriculum. For example, problems in earlier parts of the course don't require students to perform at higher levels for *task-decomposition* and may instead focus on getting students to first understand the course's programming language, or the concept and purpose of writing examples and test cases. Latter parts of the course invite students to apply their skills towards more complicated multi-task problems. Under this model, assessing students' skills with our taxonomy may be directed towards examining whether students can scale their skills or break down at a certain problem complexity.

8.1.4 Approaches to solving multi-task problems

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

We found three programming process patterns by which students approached solving the multi-task problems we gave:

1. **Cyclic**
   Cyclic students move back-and-forth between *task-* and *code*-level thinking throughout their programming process. They often apply *task-decomposition and composition* at the *relational* level *consistently* throughout their process: they concretely describe relationships between tasks in the context of an overall plan for a solution and use their insight from task-relationships to guide the composition of their code. At the same time, they use the insight from their task-level plan to guide their use, or restructuring, of the patterns they retrieved. Their meaningful use of the patterns they retrieve enables them to concretely describe how to appropriately delegate and connect the tasks they identify to specific code or parts of code.

2. **Code-focused**
   Code-focused students primarily work at the code-level throughout their programming process. They focus mainly on retrieving and implementing code patterns and constructs relative to the problem-tasks (which they often identify *on-the-fly*), but do not concretize how the tasks integrate and inter-operate with each other. Their descriptions of plans are *fragmented* and lack an overall task-level plan towards a solution; their code implementations are thus often haphazardly done and lacks any concrete direction, leading to difficulties in appropriate code decomposition and composition. They often use the patterns they retrieve mechanically and without thinking about how a problem's task-components impact the code patterns they use.

3. **One-way**
   One-way students apply *task-decomposition and composition* at the *relational* level and identify a task-level plan for a solution, but only at the beginning of their programming process. They do not remain consistently at this level throughout their process. Once they move on to implement tasks in code, they regress to a *code-focused* process and fail to maintain the connections between tasks, and between tasks and code.

These programming process patterns suggest that it is not enough for students to exhibit their skills at the *relational* level; they must also be *consistent* in applying their skills at this level throughout their programming process. Students who do so (*i.e. cyclic*) tend to be more successful in solving the multi-task problems we gave. The process that students engage in may be another explanation for the non-monotonic skill progressions we found: the lack of consistency in applying skills at certain levels may indicate the fragility of a student's skills and could be an indicator of when students might need help or interventions towards helping them practice their use of their design skills to consistently apply their skills at *relational* levels.

8.2 What did we learn about teaching program design with HTDP?

This dissertation is an exploratory project towards understanding how students who are taught the HTDP design recipe use the process when solving multi-task programming problems. Our discussions in the following sections describe what we learned about how our students used the design recipe for solving multi-task problems and what we learned about, as well as concrete recommendations for, teaching HTDP-based classes/curricula.

8.2.1 What did we learn about how students use HTDP?

We wanted to understand how students use the design recipe for multi-task programming problems. For list-based problems, we saw that the design recipe works particularly well for problem-tasks that are list-template-instantiable. For example, our findings suggest that many of our students did not struggle with following the recipe to write functions for single traversal tasks like *sum*, *count*, or

finding the *max* (even when they had to recreate them from scratch sometimes). Giving the students multi-task programming problems is what makes this an interesting project: one main challenge for students, given our problem contexts, is figuring out how to decompose multi-task problems into component subtasks that can be directly implemented using the template, and then recomposing their sub-solutions into an overall solution.

Many of the students in our studies struggled with our multi-task problems. Even when some started (successfully) with single-task, template-instantiable functions, many started to thrash when they had to figure out how the different task-related code needed to fit together, or when they attempted to populate their single-task functions with code for other tasks. There's an expectation (or a hope) that the students will use the design recipe to understand the problems better toward the goal of writing effective solutions (problem comprehension is one of the underlying motivations for the recipe steps, after all [48, 49, 146]). Instructors hope that in working with the design recipe, students won't just follow the recipe blindly or mechanically (although this may be how many students initially start), but learn to use the techniques put forth by the recipe meaningfully. For example, students might describe the concrete relationships between input data, the data definition that describes the structure of the input, and the template that concretizes this structure in code; students might describe how the examples *drive* function design or even the design of the other artifacts within the recipe (for example, in Rainfall, a list with a guaranteed sentinel would have a different data type, which in turn leads to a different template). These relationships between recipe steps, in fact, are what the curriculum teaches students to internalize as they use the recipe: the curriculum teaches to reason about programs with the steps, as each step is a *comprehension* technique (with concrete artifacts that concretize these steps) towards understanding a problem. Even the examples in the HTDP book [48] walk a reader carefully through the problems by explaining how each step informs other or succeeding steps and artifacts until a solution program is achieved.

Going beyond a mechanical use of the design recipe is critical to solving multi-task problems.

Our findings, however, reveal and highlight a *disconnect* between the expectations about using the design recipe, and how students used the design recipe in practice. There's a clear difference between students who followed the recipe *mechanically* and those who *reasoned* about their work using the recipe steps. Many students simply followed the recipe without reasoning about how the steps connect to each other and these students often failed to solve the problems we gave, even struggling to fix smaller issues that could have been addressed with techniques from the recipe; for example, output inconsistencies within functions could have been addressed by using the signatures and examples to analyze or reason about the behavior of an erroneous function. From the perspective of our skills taxonomy (Table 4.3), many of these students are rated at most *multistructural* or below for the *leveraging multiple function representations* skill. In other words, the way they internalized the recipe is *syntactic* at best. Some skipped using the recipe entirely and jumped to writing code (if anything, they at least used the template), showing a lack of appreciation and understanding of the use the recipe techniques; most of these students often got stuck and failed to solve our problems.

Some students who were more successful on our problems exhibited a more meaningful use of the design recipe techniques (among other factors; we discuss these in succeeding subsections). For example, some concretely discussed how different examples they have written illustrated the tasks they needed to be addressed, which in turn drove how they designed the structure of their solutions. Some concretely described how particular characteristics of the input data had to be reflected into the design of the template. There were a few cases of students who were not successful in solving Max-Temps, but exhibited a meaningful use of the recipe: for example, some realized during their programming process that the function template that they were using did not fit the input they had to work with, so they went back to their data definition to identify how to structure their function (these often happened later in their design processes, however, so they ran out of time before finishing their solution). These illustrate a more intentional and thoughtful use of the recipe steps, and provide some evidence that achieving a *relational*-level use of the process may lead to more success in solving multi-task problems. One instructor who participated in our validation study explained "inducing" their students to follow the recipe even when students didn't completely understand why. While this may be fine at the onset of an HTDP-based course (perhaps, to build familiarity while practicing the process), our findings suggest that this practice may not be effective (or loses its efficacy) for when students need to solve more complex problems such as the multi-task ones that we used in our studies.

8.2.2 Make problem decomposition an explicit *early* step of the design recipe

***Advanced task-level planning* (+ HTDP) worked for successful students.**

Many students did not explicitly describe task-level plans in the process of developing their solutions. Some of them *did* follow the design recipe as part of their programming process, but they did not use the design recipe steps to figure out task-decompositions. Students who did (or attempted to) decompose the problem did so usually at the *beginning* of their process. They then applied the design recipe steps on the individual tasks they have identified, and used their articulated task-level plan to *guide* the composition of their task-implementations. This at least shows some evidence that task-level planning *in advance* helps HTDP-trained students in solving multi-task problems.

"*One function per task*" does not seem to be enough to encourage consistent planning.

When we examined the curriculum and learning activities (*i.e.* homework, exercises, class examples) of our host courses to understand why students did *not* plan in advance, we found that task-level planning was not a fundamental part of instruction, or was at least given only a light treatment. Both the WPI and NEU CS1 host courses first teach the *idea* of task-level decomposition in the second week[1] of their respective school terms by teaching students a recommended rule from section 2.3[2] of the HTDP textbook [48]: "*Define one function per task*". This section of the textbook presents the rule as a "slogan", motivated by the ideas of writing "reasonably small" and "easy to comprehend" functions,

[1] Course syllabi are in Table 6.1; the topic sequence for all WPI courses are the same.
[2] Second Edition of How to Design Programs [48]

and followed by a sample problem on implementing a formula with sub-computations (subtasks). Other WPI course materials revealed no additional treatment of task-decomposition beyond the given textbook rule. In the NEU course web page on the design recipe, the instructor had added one guide question in the *function definition* step (step 6) of the design recipe: *Can what I'm trying to achieve be broken down into multiple steps (i.e. should I use helpers)?* (Appendix F.1). The lab exercises and homework for both courses are all directed towards using the design recipe to solve problems, with brief reminders to "use helper functions".

Given how the students in our studies performed on our problems, the "*one function per task*" rule doesn't seem to be enough to habituate task-level planning in advance among students. Some students explained that they struggled with the multi-task problems because their homework problem sets already provide the decomposition for them, *e.g.* earlier problems in the problem set served as helper functions for latter problems in the set (*e.g.* Section 5.4.3 and appendix C.2).

Have students do more activities on task-level planning.

Our work highlights the importance of teaching task-level planning explicitly. One way to do this is by having students do more activities that involve identifying and planning around problem-tasks, but without expecting them to write code. For example, students might be given several multi-task problems where they identify tasks and concretely describe how the tasks relate to each other. Students could also be taught to leverage specific design recipe steps to facilitate this. We had already observed some students practice a form of this with purpose statements, where they described where a helper function is being called within an overall program (*e.g.* Chapter 6). We further illustrate an example of this in Section 8.2.3.

In Sections 3.6.3 and 6.3.4, we described a way of expanding concrete examples to work out task-decompositions. For example:

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

No student across all of our study cohorts did this and none of the courses taught students how to use examples in this way. A systematic way of using examples to identify task decompositions, such as this, might suggest specific functions that a student could write as it can be used to illustrate a decomposition of the code relative to the tasks as well as their composition. Understanding whether or not these planning exercises help students practice task-level planning merits future exploration.

8.2.3 Focus on teaching *how* to use the design recipe steps

Many students developed a blind habit of following the design recipe.

One of our main findings is that many students primarily *followed* the HTDP design recipe *mechanically* and without an insight towards how the recipe steps connect to each other, or how the steps connect to an understanding of a problem (Section 8.2.1). Students developed a *blind habit* of following the design recipe, but did not necessarily gain any *insight* around how each recipe step is a technique towards building a concrete understanding of a problem at hand. For example, when explaining their use of the techniques, many students describe "following the recipe" but don't make concrete connections about how the different steps/techniques guide the design of their code. Students also sometimes skipped over some of the recipe steps (these students just jumped into writing code). When they struggled with the problems, they did not seem to think about using their design techniques to figure out where or why they may be stuck, further suggesting a mechanical and shallow understanding of their design techniques. Students also struggled to figure out *how* to use their design techniques (*e.g.* writing data definitions and templates for Max-Temps) towards the novel aspects of the problems we gave.

These observed behaviors illustrate several problems: (a) there's a *disconnect* between expectations about using the design recipe and how students use it in practice, (b) students have only used their design techniques towards template-instantiable problem-tasks, and (c) students have not understood that their design techniques can be used for debugging mid-process, and not only at the earlier stages of their programming (*i.e.* when they start writing their functions after following the early steps of the recipe). I argue that practicing the techniques shouldn't just be focused towards activities that ask students to use them to write programming solutions for problems; students should also engage in activities that help them learn and practice to *reason about* and *explain* plans and programs using the techniques. The latter does not seem to be implemented as activities in our host courses given that the lab exercises and homework given are primarily geared towards using the recipe to write solutions for programming problems. We discuss in the next sections, recommendations towards concrete class activities for addressing these problems, which also provide opportunities towards further research.

Students may need targeted exercises towards an insightful use of the design techniques.

While the recipe suggests writing examples of data and input–output pairs (*i.e.* test-cases) early, many skip this and write them *after* they have written their functions. This is unfortunate, as the goal of writing examples is to have students concretely think through and write down the space of program inputs (and the expected output for each) before writing any code [51]. Doing so can help illustrate whether or not a student has understood the problem, at least within the context of the problem's input; for example, if a student can't explain what output is produced from a specific input, or why a specific output is produced from a given input, the student likely hasn't understood the problem just yet. One activity towards teaching an insightful use of examples would be expanding concrete examples to illustrate task-decompositions, as we've previously described in Sections 3.6.3, 6.3.4 and 8.2.2.

Another activity leveraging examples would be to give students problem implementations with errors and to have them *reason* about and *explain* the errors using concrete examples. For example, given the Rainfall problem statement and the following implementation:

```
(define (rainfall list-input)
  (average (get-data-of-interest list-input)))

(define (get-data-of-interest list-input)
  (filter positive? list-input))

(define (average list-input)
  (/ (foldr + 0 list-input)
     (length list-input)))
```

students may be asked to both identify and explain the errors using examples. Students will have to think concretely about the tasks of Rainfall and illustrate, using concrete examples, input scenarios that will pass and fail the implementation (possibly with brief descriptions/explanations). The given code above fails because of the following:

- Does not terminate at the sentinel
- Does not include zeros in the average computation
- Does not account for an empty-list input
- Does not account for a list input with all-negative elements

From a tool perspective, Wrenn and Krishnamurthi developed *Examplar* [146], a tool that provides students with feedback on whether they have correctly and thoroughly explored a problem with their examples. Students can use Examplar to develop and assess their examples independent of how far along they have written code [146]. A tool such as this, when used as part of a class, may encourage students to be more reflective and intentional about the examples they write towards understanding the problems they're given. Indeed, in assessing students' use of Examplar in an accelerated CS1 course, the authors found that students used the tool even when it was not required for some course exercises and that the quality of students' test suites generally improved, although they remained inconclusive about whether the tool helped students improve their solutions. Nevertheless, it remains an open question whether this tool, in addition to the class activities towards writing examples that we recommend in this dissertation, may help students towards a more meaningful use of examples towards developing programming solutions.

A meaningful way to use *purpose statements* would be using them to relate tasks or functions to each other. Suppose that students had been asked to identify and describe the problem-tasks in advance (as we recommended in Section 8.2.2). As it stands, purpose statements are meant to be brief descriptions of functions [48] (*e.g.* Appendix F.1), but there are no guidelines around what constitutes a *good* (or useful) purpose statement. Our findings highlight that students who had *concrete*

explanations and descriptions about their task-level planning of the problems were more successful towards their solutions. What if we leveraged purpose statements further to be more concrete about task-level decompositions and compositions? For example, purpose statements could be slightly expanded to include: (a) a list of the problem-tasks that a function is addressing, and (b) how the function relates to other functions. The first item (a) could be used as a concrete descriptive tool towards task-decomposition; if the list of tasks that a function is addressing has too many task-items (instructors may define "too many" using rules, such as the "one task per function" rule), then that may signal to a student, opportunities for further task-decomposition. The second item may be used to illustrate task-compositions, by indicating, for example, what other functions the current function uses/calls, and which other functions use/call the current function. We have already observed some students (Section 8.2.2) do this with their purpose statements, which suggests that it may be reasonable to have students practice writing purpose statements in this way. We illustrate a possible version of this in the following code snippets (type signatures added for clarity):

```
; sum: list[numbers] -> number
; Sum a list of numbers up to -999, ignoring negatives
; Tasks: sum, sentinel, ignore-negatives
; Called by: average function
; Calls: none
```

Compare the above purpose comments with the following purpose comments:

```
; sum: list[numbers] -> number
; Sum a list of numbers
; Tasks: sum
; Called by: average function
; Calls: truncate and rem-negs functions to clean list data

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

; rem-negs: list[numbers] -> list[numbers]
; Produces a list of numbers with negatives removed
; Tasks: ignore-negatives
; Called by: sum function
; Calls: truncate function to get list data before -999
```

In practice, either version works for the problem, as students can certainly weave multiple tasks together in a single function, or pull them apart into separate functions. Students who might struggle with writing functions that integrated multiple subtasks, however, may benefit from the version that further separates the tasks into their own functions. The parts of the purpose comments that indicate where a function is used/called (`Called by`) and what other function it uses/calls (`Calls`) illustrate the

compositions of the functions and requires students to concretely articulate these relationships. This is another activity that may help practice task-level planning, as we recommended in Section 8.2.2. Since purpose statements are about *describing* functions, leveraging this description to further capture *relationships* across tasks/functions may be beneficial to students. It would be interesting to explore whether such a use of the purpose statements may help students with task-level planning, particularly for multi-task problems.

Students need to practice using their design techniques in more varied data contexts.

Having students practice on a wider variety of data design activities exposes them to various data situations they might encounter and provides opportunities to apply their design techniques towards different data situations. We discussed in Section 8.1.2 that there is an interaction between the variety of problems students have seen and their skill towards adapting the patterns they've learned. In particular, students struggled with adapting the patterns they've learned to the novel aspects of our programming problems, even when the problems used a data type (lists of numbers) that were largely familiar to them.

An interesting characteristic of our study problems (on top of having multiple tasks) is that they use *significant* list elements (*i.e.* list elements with distinct roles). For example, Rainfall has the -999 *sentinel*, which serves the role of terminating a computation, akin to the empty-list base-case; similarly, Adding Machine has a *sentinel pattern* (the double zeros). Both Adding Machine and Max-Temps have *delimiters* that are indicators to the windows of elements within the list to be processed separately. At a problem-level perspective, our study problems involving these significant elements are data-processing problems that require students to *cleanse* noisy data or address data with *underlying structure* [53]. The significant elements add additional complexity to traversing lists and were significant design challenges for some students. For example, students got stuck figuring out how to handle sentinel tasks, how to recur over a modified suffix of the list beyond the usual recursion on the tail (`rest`) of the list, or how to design data definitions (and templates) for lists with significant elements. These highlight a functional fixedness that students may have attributed to the parts of the template (*i.e.* the base-case and recursive-call parts) or even to the use of their design techniques.

Engaging students in design activities through problems of various data contexts or data-processing needs, as well as teaching them concrete techniques for such situations (*e.g.* identifying significant elements in a list and how they may impact the use of template parts like the base-case and recursive-call parts), may benefit students by helping them further understand the *malleability* of *all* parts of the template beyond the template "holes". Data design activities with varying data contexts/scenarios may also help students understand how to leverage their design techniques beyond basic list template scenarios. For example, another reason students struggled in Max-Temps is because they got stuck trying to figure out how to write a data definition or template for the problem's input data (Sections 6.3.4 and 8.1.2); this suggests that they only see data definitions in the context of the basic list definitions that they've seen. While the students recognized the roles of the delimiter elements (based on the problem statement), they did not know how to design or manipulate the data definitions they know for

basic lists in a way that accounts for these significant elements, thereby affecting how they wrote their function template later on. The critical point here is to expose students to a variety of design problems that can be used to illustrate (and practice) how their patterns and techniques are malleable constructs that can be manipulated based on the context of the input (*e.g.* significant elements) and that what they already know do not just lead to a single way of applying these constructs. Data-processing problems with *significant elements* seem to be a class of problems that are good candidates for such design activities. Thinking about *significant elements* and the tasks embodied by these elements fit into the idea of "data-thinking" that motivates the HTDP design recipe, as students would have to think about how pieces of the data (*i.e.* the specific roles of certain list elements), and the tasks these represent, impact how they might design functions for processing such data.

Teach the underlying concept of patterns through varied contexts of use.

Even when having associated some behavior with accumulators (*e.g.* "storing" values), some students retrieved and used the accumulator pattern mechanically when they attempted to use it for Rainfall. They struggled to adapt the pattern to the needs of Rainfall (*e.g.* adding parameters to track additional values); an accumulator-based Rainfall solution either needs two parameters (one for each of the running sum and count) or one to keep track of data (for a list of clean data). Some students explained not understanding why accumulator functions were structured the way they were, even when they describe using the pattern for previously-seen tasks in class, such as summing or finding the max, and replicating the code during their study sessions. These suggest that they had only understood the syntactic schema of accumulators, rather than its underlying concept. The WPI students in our courses had only seen examples with a single accumulator parameter, and in each of those programs, the parameter value was returned in the base case of the recursion (NEU had not yet reached accumulators when we ran our studies). This drives home the power and hold of previously-seen patterns on novice programmers and suggests that the examples instructors choose may inadvertently constrain how students understand the use of the patterns they're learning.

This further supports the need for students to practice on problems that require them to use their patterns in various contexts. Instructors may think they are teaching a general approach (*i.e.* using accumulators), but if students have only seen a limited set of examples that apply the approach in a single way, they may struggle to adapt the patterns to new situations. Our findings suggest that teaching the concept that underlies accumulators requires showing students varying scenarios for which the pattern applies. The students who used accumulators and produced more correct code used the pattern beyond the ways that they had seen in their course, articulated clear roles for their accumulators, connected accumulator parameters to specific tasks, and maintained those connections throughout their programming process. Giving students activities and exercises on multi-task problems that requires concretely articulating the connection between (traversal) tasks and accumulators seem critical not only to producing effective solutions using the pattern, but also for understanding the pattern's general concept.

8.3 On teaching program design in general

The limited range of problems and activities that students engaged in in their courses seem to have constrained students' understanding of how to apply their techniques and patterns.

Our findings highlight an interaction between how students learn to use the HTDP design recipe and the examples and learning activities that instructors teach the curriculum with. Many students generally followed the design recipe process top-down, and would even use (or reproduce) the code patterns they know, but they still ended up struggling to solve our programming problems. Our findings show that this is related to their *mechanical* use of their techniques and patterns. While they *retrieve* the knowledge of their techniques and write the artifacts relevant to the techniques (*e.g.* data definitions, examples, templates, *etc.*), they don't articulate how the techniques relate to, or inform, each other, or the patterns they retrieve (*e.g.* templates) as part of the recipe process. Their following and use of the recipe thus becomes just another set of artifacts to be written down with their code. In other words, they applied their techniques syntactically (*e.g.* "*Write a list data definition then the list template...*") rather than with a relational understanding (*e.g.* "*How do I capture characteristics of the input in the data definition and how does this translate into a template?*"); this highlights the disconnect between instructional goals with the curriculum and the students' actual use of the techniques within HTDP. Our analyses of the course contexts reveal that the students mostly applied the design recipe towards writing solutions for programming problems and that they have only seen their patterns applied to a limited set of problems (those that could be directly implemented with templates). The limited range of problems and activities that students have seen thus seems to have also limited their understanding of how to use their techniques and patterns. Students who struggled also primarily focused their thinking on retrieving and writing code constructs, without thinking about the task-level decomposition of the problem and how this task-decomposition impacts their code implementations and compositions. This suggests that the activities and problems used to teach the HTDP program design techniques matter as much as the program design techniques being taught.

Our recommendations of activities and problems in Section 8.2 are directed towards explicitly teaching, and having students practice, (1) task-level planning, (2) how to use their design techniques to reason about and explain their task-level and code-level plans, and (3) how to apply their design techniques and patterns towards more varied data contexts. These are informed by our own findings: the students who had more success on our programming problems concretely *described* task-level plans, *reasoned* about their work using their design techniques and plans, and adapted their use of their techniques and patterns to the needs of our problems. Being able to explain and reason about plans, techniques, and patterns, and going beyond a syntactic use of these seem to be the critical ingredient that differentiates our more successful students from the less successful ones when solving multi-task problems. Our recommendations also align with recommendations from prior work on teaching students to integrate and organize program design knowledge. Work by Linn and Clancy [80–82] found that concrete and explicit descriptions, explanations, and narratives of how to select plans and relationships among plans encouraged students to focus on the central aspects of examples and plans

rather than their superficial features, as well as use plans towards new contexts; getting students to make similar descriptions and explanations of plans (with their design techniques as tools towards this) is an explicit goal of our recommendations.

Our work shows that even a carefully-designed curriculum for teaching design techniques (*i.e.* HTDP) can be absorbed by students at a superficial or syntactic level (even when a primary goal for the design recipe, at least at the collegiate level[3], is to relate steps to each other during the design process); prior cognitive science work on how students retrieve programming plans and processes corroborate this [6, 57, 118]. While there are ways in which the design recipe techniques can be used syntactically, our findings show that this is not enough to get our students to solve the multi-task problems that we gave. We should also note that the curriculum was designed around a specific set of problems for which it is suited, that is, problems for which structural recursion is a natural approach [48, 49]. While none of our multi-task problems are directly template-instantiable, most of our problems' subtasks are, making problem decomposition (or task-level planning in general) a critical step alongside the use of HTDP (and pushing students just a little beyond their training with HTDP). This further supports the need for interventions and activities that drive students to practice using their skills and techniques beyond a syntactic and mechanical use.

In the broader sense, studying how other program design curricula help students solve programming problems, such as the Pattern-Oriented Instruction approach by Muller *et al.* [94] and de Raadt's work on explicit strategy instruction in CS1-level courses [35] must be considered alongside a closer analysis of their respective course contexts to truly understand how students are integrating their knowledge around the techniques, skills, and patterns put forth by their curricula. In his dissertation [35], de Raadt showed an example of an exam question that asked students to explain their strategy towards a problem in prose, rather than asking them to generate code, for example, by asking novices to *identify or describe strategies*, *relate strategies*, or *identify how a strategy is incorrectly applied*. Rather than use similar questions in exams (which are primarily used for summative assessment; at this point, it may be too late for students to practice or learn skills), such questions can be used towards class activities for students to practice explaining and reasoning about plans and programs. Additionally, instructors should be aware of their own biases and blind spots around designing curricula or learning activities. Instructors must closely examine the activities and assessments they use towards their teaching to understand whether or not these truly help them achieve their curricular goals. As Ramsey put it: "*Plenty of problems (in teaching program design) are still open, of which the most difficult is assessing whether students can do what we think they can do*" [112].

[3] In Bootstrap (an early-programming curriculum for middle- and high-school math and computing classes in the USA), the design recipe is used to aid in a syntactic exercise of creating functions by abstracting over examples (Personal communication with Kathi Fisler, Bootstrap Co-Director)

Developing a data-grounded theory for program design curricula helps create a critical understanding of their use by students in real-world contexts.

In her study of students' solutions for Rainfall, Fisler found that HTDP-trained students generally did well on the problem; they produced a diverse set of high-level structures and made fewer low-level errors (compared to students from prior Rainfall studies) [51]. While she identified the language constructs that students used (*e.g.* higher-order functions) to structure their solutions, she did not identify the underlying processes that describe students' movements toward a solution structure, or how they used the techniques they had at the time to solve the problem (her data did not allow these kinds of analyses). In other words, her work answered the question of *what* HTDP-trained students did, but misses out on critical details about *how* students approached the problem and *why* they did so. Even more recent Rainfall studies focus on the *what*: the number of students who get it right, the kinds of errors they commit, the details that students miss out, the solution structures they use, or the code constructs they use [51, 122, 126]. Answering these questions certainly gives clues towards what may or may not work, but being able to explain *why* things do or do not work or *how* they work are critical towards understanding how to teach program design better. As Guzdial puts it, "*we don't yet have enough theory to explain why [CS teaching contexts] works when we get it right* [59]."

We did not find the same level of success among our students as Fisler found with hers, but we were able to tease out how students were using their design techniques and how they were thinking through the problems we gave. In addition to Rainfall, we also used two other multi-task programming problems that required different techniques than what Rainfall required. We were thus able to see student work with the design recipe in new problem contexts. Beyond students' code submissions, we also collected think-aloud, interview, field observation, and scratch work (whenever students did this) data which allowed us to analyze students' work at a finer level of detail than what just a final code submission would allow. This enabled us to concretely identify and describe programming process patterns (*cyclic*, *code-focused*, *one-way*) that we found were indicative of whether or not a student would be successful in designing an effective solution. We were also able to identify and describe the varying levels of complexity at which students were applying their skills and techniques, specific interactions between skills, as well as interactions between skills and the kinds of examples and class activities that students have seen or engaged in in their courses. These findings and observations became components to our theoretical framework that describes how HTDP-trained novice programmers design programs toward multi-task programming problems.

Developing theoretical frameworks for the curricula we teach is essential to understanding them better. Theories can explain observed phenomena (*e.g.* how students use techniques to program) or predict occurrences (*e.g.* whether students succeed in writing programs) [100]. Data-grounded theories on curricula (such as ours) help explain what works or doesn't when used by learners in the real world and provides critical insight for instructors to understand how to teach skills, techniques, patterns, or other components of curricula better. For example, in her Rainfall study, Fisler found that the students who failed to write working Rainfall solutions used the template naively; our own theory explains that students who do so engaged in a *code-focused* programming process, did not

engage in task-level planning, and did not explicitly describe the limits of their retrieved patterns in the context of problem-tasks. From Pai *et al.*'s [102] perspective, the theoretical framework we've developed in this dissertation both provide '*How*' theories and '*Why*' theories. Computing education researchers have recently called on others in the field to develop theories for CS teaching and learning, citing the unique challenges faced by both CS educators (in teaching CS) and learners (in learning CS concepts and within CS contexts) [59]. Xie *et al.* noted that prior theories on programming skills did not translate to concrete instruction for supporting skill development; they proposed a theory for structuring and sequencing programming skills in a way that can be translated to instruction that scaffolds the development of these skills [147]. Our own theory highlighted gaps in how courses taught HTDP and we proposed instructional activities that can be used to address these gaps (that also leverage techniques from the design recipe). Theory development for CS education connects research and practice on how to teach CS and how people learn to program, and is becoming increasingly necessary to address the needs of the growing number, and diversity, of learners enrolling in CS courses or wanting to learn programming.

Using formative measures of programming skills may be valuable for students as they work towards developing their skills during their CS1 courses

One of the contributions of this dissertation research is a framework towards assessing students' program design skills. While we found many students fail to solve our programming problems, our analyses did not focus on rating the code they produced with some numeric grade. Rather, we focused on understanding *why* they failed to write effective code (or *why* the successful ones do) and *how* they had approached the problems. This focus on students' processes and the interactions of factors within these processes (*e.g.* planning, knowledge of design techniques) is important when assessing students' program design skills. Grading students' final code can only say so much about students' design process toward their solution (they could, for instance, do a trial-and-error and still get a correct answer without truly understanding how or why their solution works). A student who failed to write a working solution will also not know what they could have done better if the assessment of their work that they receive from their instructor is a single numeric rating. Numeric/summative ratings can also have arbitrary meanings associated to them; for example, different students who get a B for their work may have received them for completely different reasons. This lack of granularity does not have actionable components that reflect how to go about with progressing in one's development of their skills, which may affect learners negatively. Students use grades differently and they may feel demotivated or see the grades as a reflection of their programming ability, or even a reflection of their (in)ability to learn, shaping their self-efficacy and beliefs about their intelligence and identity [72, 98].

HTDP-based courses approach grading student work by considering each design recipe step, that is, each recipe step counts in grading. For example, functions that produce correct answers but deviates from the data definition (or the "shape" of the input), or lacks signatures, purpose statements, or test

cases only get a fraction of credit[4]. This seems preferable to simply assigning a grade only to the final code, for example: missing data examples and test cases may suggest that a student did not think through the space of input, or functions with output inconsistencies may suggest that a student did not consider the signatures/contracts they wrote in their design. Our findings, however, suggest that many students don't even engage in reasoning about their work with the recipe steps, which we've mentioned may be due to several reasons, such as the *value judgments* they attribute to the use of the techniques, that they didn't know how to use the techniques towards novel data scenarios beyond what they've seen in class (*e.g.* list problems with particular data-processing challenges, see Section 8.2.3), or that they've only understood the use of the techniques superficially.

Our studies teased out some of the nuances of how students approached the problems and the skills they used during their programming process. In particular, our SOLO-based skills taxonomy captured the skills that students used and the levels of complexity at which they applied their skills. It helped us identify the extent to which they reasoned about their examples and test cases and the inter-operation of the different techniques put forth by the design recipe. It also captured how students think about the problems at the task-level and how they understood the use and limits of the patterns they retrieved; these cannot easily be assessed with just the techniques from the recipe alone (although we have begun to recommend possible ways of using the techniques towards thinking about tasks, as we've described in Sections 8.2.2 and 8.2.3). We've used the taxonomy in a couple of ways: we used it to assess students' design work at multiple points during a CS1 course, and as a lens towards a descriptive analysis of how students from other HTDP-based courses approached multi-task programming problems.

Our taxonomy is in a similar vein as de Raadt's work of assessing students' use of their strategies; he explained that assessing students only by the code they generated may be a poor measure of their actual programming skills and that assessing students' comprehension of programming strategies through explanations may be more meaningful [35]. More broadly, it may benefit learners to have *formative measures* that help both learners and teachers understand what the learners do and do not know, or can and cannot do, and more concretely identify what problems need to be addressed or what concepts need to be strengthened, rather than a vague and less helpful, "your code is worth a B". Our SOLO-based skills taxonomy is our contribution towards the goal of developing a formative measure for program design skills; the skills we described in the taxonomy are certainly not limited to HTDP and future work can look at how the taxonomy can serve as a rubric in other curricula. When assessing student design work with the taxonomy, the level that a student falls within a skill may be used for more concrete feedback to the student on what they may need to practice on (*e.g.* they may need to practice thinking about how tasks relate to each other to guide their code composition).

So far, however, the use of our taxonomy is contingent on having/collecting data from which students' understanding of skills can be concretely drawn, such as explanations of how they perceive relationships between design techniques, plans, or program representations. This is not always

[4]This is based on Fisler's description of HTDP-based courses (she is a long-time HTDP instructor) [51], discussions with the instructors of our host courses, and my direct experience as a teaching assistant for HTDP courses.

practical especially in courses with hundreds of students; yet it seems that this kind of data is critical to understanding at a granular level *why* students struggle in their learning of programming and can be the basis of meaningful feedback for students. This highlights the importance of the activity recommendations we made in Sections 8.2.2 and 8.2.3, which leverage the design recipe techniques towards having students explain and reason about their plans, techniques, and programs. Future work could look at designing formative assessments around these activities that can be used within CS1 courses to give concrete and meaningful feedback to students about how they are progressing in their learning during a course. Such formative assessments could help educators (even those teaching with different curricula) better understand where students are in their learning and have a more concrete basis of where and how to focus their teaching better to help their students.

8.4 Bricolage, Planning, and the HTDP Design Recipe

Researchers describe *bricolage* as an exploratory trial-and-error approach for solving problems, a form of experimental tinkering that involves a "dialogue" and "negotiation" with artifacts to develop a solution for a problem at hand [9, 30, 43, 131, 136, 148]. Turkle and Papert suggest that bricolage is the opposite of planning, but that there is value to both ways of thinking [30, 136]. What do our findings suggest about the design recipe as a way of thinking about problems?

As we've described in Section 1.1, the systematic process of the design recipe leads a novice through different representations of a problem, which suggests that the recipe is primarily a planning approach. Our data, however, suggests a more nuanced use *in practice* of the design recipe as a way of thinking about (programming) problems. Our observations and findings show that the students' use of the recipe reflects elements of both planning and bricolage; we illustrate this in the following discussions.

8.4.1 Productive bricolage among HTDP-trained students

Some of the students who used the design recipe at the *relational* level made concrete connections across the different recipe steps and artifacts. In doing so, they used their understanding of the recipe techniques to reflect novel characteristics of data into the structure of their code (*e.g.* data definitions and templates), or identify problem tasks through the analysis of concrete examples in relation to the problem prompts (*i.e.* planning using the design recipe). Other students followed the recipe mechanically and would tinker around their code based on feedback from their test cases (*i.e.* through error messages), even when this response to feedback is more about figuring out how to edit the code structure "to make it run", rather than concretely making connections about what errors the test cases are highlighting and how this relates to the design of their code (*i.e.* bricolage with a focus on tinkering around code).

The design processes we observed weren't always clean, straightforward, well laid-out processes, and this can be attributed to the nature of our study problems. Our selected problems were designed to

push students slightly beyond what they know and what they have seen in their classroom instruction, and this manifested in our students' design approaches. In a few cases, some students approached problems systematically through task-level planning and the design recipe process (students who demonstrated a *cyclic* design process), but the unfamiliar components of the problems drove them to go back to the recipe artifacts that they wrote to analyze how they might refine artifacts to accommodate the novel aspects of the problems.

When components of the problems are unfamiliar to some students, they would go back to the techniques that they have learned and attempt to discover ways to use or adapt their techniques (or their associated artifacts) to accommodate the new contexts that they're discovering. For example, some students thought about how they might modify the template structure to capture significant elements that had a direct impact on how a list is traversed (*e.g.* the -999 or double-zero sentinels in Rainfall and Max-Temps, respectively) or used examples to ask about how their functions should account for computations that need to be addressed (*e.g.* choosing whether to integrate removing negatives with summing for Rainfall, or what input might look like for input with empty sublists in Max-Temps); we have illustrated these observations further in the previous chapters and sections detailing our studies (Chapters 3, 4 and 6 and section 8.2). These illustrate a form of bricolage *negotiation* that students are engaging in with their design teachniques and artifacts to capture the problem aspects they identified into their currently evolving solution. This behavior reflects what Perkins *et al.* [103] and Berland *et al.* [9] call a *cautious tinkering* approach and which, similar to our own findings, seemed beneficial to students when developing solutions for more novel problems. Unfortunately, our studies also highlighted the fact that the use of the design recipe as a way of productive tinkering wasn't a frequent occurrence among the students in our studies. Many students instead used the design recipe in a less productive, mechanical, and superficial manner; we discuss this use of the recipe from the perspective of bricolage next.

8.4.2 Haphazard bricolage among HTDP-trained students

Some students jumped into writing their code and proceeded to tinker with their code without any concrete plan (*code-focused* approach pattern). Their use of the design recipe techniques and artifacts (if they ever used them) was primarily ad hoc and without concrete connections across the steps or artifacts. In these instances, the students' negotiations with artifacts primarily revolved around the code that they wrote and the feedback they were getting about the status of their code (*e.g.* errors), and our findings showed that these students often ended up getting stuck. For example, some students jumped immediately into writing functions without making use of any of the design recipe techniques; other students simply copied the examples provided in the problem prompts and then completely ignored them throughout the rest of their design process. These students would then proceed with writing their functions, with a back-and-forth process of writing code and running the code until they hit on a solution. Without recipe artifacts that capture problem aspects and on which concrete descriptions or explanations can be anchored onto to explain what logic or tasks are missing from the solutions,

students are forced to reason about their code using only the code that they have written. At best, the error messages provide some insight about what's wrong with the implementation (*e.g.* syntax errors or output inconsistencies), but beyond that, the problem tasks or interesting aspects about the problem such as sentinels and delimiters are not articulated in the error messages; students have to concretely capture these through data definitions, examples, signatures, or purpose statements). Perkins *et al.* and Berland *et al.* described this bricolage approach as a *haphazard tinkering* behavior and their findings seem to align with our own: when students are tinkering haphazardly, that is, a purely experimental trial-and-error and not a purposeful tinkering, the tinkering has a negative impact on students' progress.

8.4.3 Supporting productive bricolage through HTDP

These findings suggest that the design recipe may be used to support *productive* bricolage [9] (as opposed to Yeshno and Ben-Ari's characterization of bricolage as a purely "*aimless*" process [148]). Our observations (as described in Sections 8.4.1 and 8.4.2) suggest that the different recipe artifacts can provide students with a *language* for negotiating between their understanding of a problem and how they manifest this understanding into code. These negotiations between a student's cognitive representations of a problem's elements, the recipe artifacts that concretely illustrate or capture these elements, and the code that implements these elements, are possible provided that there are meanings attributed to the recipe artifacts in relation to the code (or as Fisler describes it, the recipe artifacts provide an *explanation* of the code [51]). This also explains why students who used the recipe mechanically get stuck even when using the recipe: when students merely copy or follow the recipe without insight, the recipe process and artifacts lose their value as vehicles for negotiating students' understanding of the problem with the code they hope to produce. The recipe thus does not serve any purpose other than being another set of artifacts to write without any meaning-making [149] involved.

Overall, these provide further evidence for the benefits offered by tools, scaffolds, or systematic approaches to design (not only for *program* design) as providing a language for negotiating the discovery of insights and solutions and supporting learning through productive bricolage. What we have articulated here is a concrete characterization of the role that the design recipe plays from a bricolage programming perspective; this is akin to the description of McLean and Wiggins [91] of bricolage programming as a creative feedback loop between concepts, algorithms, outputs, and a programmer's perceptions and reactions to output or behavior. Interesting future work along this line could focus on the negotiating affordances that each design recipe artifact provides and the specific negotiating and meaning-making mechanisms that novices practice in relation to the techniques taught by HTDP. This could further explore the ways in which the design recipe may facilitate further bricolage, or whether it just gets in the way for students who otherwise are going to just tinker with code.

8.5 Threats to validity

8.5.1 Improving the contextual grounding for our taxonomy design

Our taxonomy was initially designed based on the data from one homework that a subset of our students in our longitudinal study (Chapter 4) worked on. As a team, we then used the taxonomy to collaboratively code the program design work by the sampled students in succeeding study sessions, and the program design work in all study sessions by students outside the initial sample, refining our taxonomy based on new data. While this iterative refinement of the taxonomy was meant to deepen and strengthen the descriptive power of the taxonomy, the genesis of the taxonomy is still deeply rooted in the contexts of the CS1 course that we studied, the programming problems that the students designed solutions for (thus exposing their design process), as well as the individual contexts of the students that participated in the study.

This contextual limitation is partly mitigated by our use of the taxonomy as an explanatory framework for describing the design work of students in our later studies. As we used the taxonomy as an analytical lens in our subsequent studies, we found that it helped us concretely explain how students approached our multi-task problems, even enabling us to identify concrete relationships between our articulated program design skills (Sections 8.1.1 and 8.1.2). Unlike the original study session which produced the data on which the taxonomy was designed from, the succeeding studies used different student cohorts, from different HTDP course instances, from two different universities, and focused on more than one multi-task problem that the students had not seen in their course. This at least illustrates that the taxonomy is not strictly contextually-bound to the original study.

The contextual grounding of our taxonomy could, however, still be improved in several ways, some of which include: (a) replicating the study and the analyses conducted on a different HTDP-based CS1 course, student cohort, and set of programming problems (we also discuss this in Section 8.6.1), or (b) varying the study in small ways (such as using the taxonomy as an interpretive lens to understand focused observations during programming labs, for example) on new student cohorts to determine how well the taxonomy makes sense outside of our own study contexts (a form of ecological or external validity[5]). These could help refine our taxonomy descriptions through the identification of the persistent aspects and the contextual limits of its theoretical constructs.

We also attempted to broaden the contextual grounding of our taxonomy through expert-checking [77, 78, 142] with other HTDP instructors to understand how well the taxonomy reflects how other instructors assess student design work (Section 4.7). This allowed us to tease out and capture another program design skill that we did not capture in our previous iterations of taxonomy design. A limitation in this regard was that some of the instructors only gave very vague descriptions or explanations of how they rated students' design skills, or that some instructors completely skipped writing explanations (Section 4.7.4). This limits the extent to which our taxonomy actually captures

[5]Note, however, that qualitative research does not primarily or necessarily aim for broad generalizability of findings, but rather, deep and close understanding of phenomena and the contexts on which they are grounded. Caution should be exercised when interpreting constructs and observations from similar studies towards generalizability.

the authentic constructs and practices that HTDP instructors look for when assessing students' program design work or processes. We suspect that had we conducted this instructor study with a semi-structured interview or even a think-aloud protocol (rather than with a free-form survey/worksheet), we could have addressed the lack of detail in their descriptions with additional prompts to delve deeper into the instructors' reasoning or thought-processes as they assessed student work. Future work on expert-checking should take into consideration designing similar studies that prompt for richer responses from experts. This would allow for a more grounded refinement of the taxonomy based on experts' narratives, so long as the methodology and analytic process is concretely explained in detail to account for audit trails [77, 78, 142].

8.5.2 Problem context limitations

One of our motivations with our choice of study problems was that we wanted to understand how students used the design recipe techniques "in the wild", on problems that are slightly more challenging (while still having components or aspects that can be addressed with the design recipe), especially since the design recipe was meant to be a scaffold to get students beyond a blank page when solving (new) programming problems. Using the problems in our studies helped us understand the nuances of how students used the design recipe; our findings gave us insight into how we might improve on the teaching or design of the curriculum. These insights are embodied in the conceptual framework we developed for HTDP (Section 8.1), as well as our recommendations for improving the teaching of HTDP-based courses and introductory-CS in general (Sections 8.2 and 8.3).

Our findings, however, are based on a limited set of problems, all with list-type input and sharing some similar characteristics across the problems: sentinels that denote a prefix of data to process, delimiters and noise in the data, and underlying structure in the data. Ideally, we would have liked to test students on a variety of multi-task problems, varying input type and viable plans, for example. Doing so would have helped us tease out and understand constructs such as: what planning issues persist across different multi-task problems, how the students may have applied the program design skills we identified in our taxonomy, as well as identify potential hierarchies of complexity among plans (or problems that embody these plans), which may be useful in designing course activities. As it stands, the limited range of our problems suggests a potential limitation in the contexts in which our findings may apply, as well as our own descriptions of students' design skills and processes.

Part of why we were not able to run more studies with a broader set of multi-task problems was due to the time demand (and monetary cost) of study sessions: a think-aloud session on a single multi-task problem, followed by a retrospective interview already consumed a full hour, and our study design already had students work on two problems each (two hours total per student). We had begun to run studies wherein we gave students two versions of either Adding Machine or Max-Temps, with one version in a reshaped form (*i.e.* list-of-lists), and another in the original form, to explore whether students could solve the problems in their already-reshaped forms, and whether having seen a reshaped version first helps them solve the problems in their original form. We sought to understand whether

giving the students these problems in a "leveled-down" form might suggest to them certain tasks that they can apply to the original forms of the problems, and whether they can recognize to apply the design recipe to the "leveled-down" versions of the problems (as these versions would be directly template-instantiable). Articulating findings from this study will be part of the future work on this project.

8.5.3 Computing intercoder reliability for our skills taxonomy

My advisor and I iteratively developed the taxonomy, revisiting our taxonomy definitions several times after its initial development to refine them as we collected new data. Whenever we used the taxonomy to score or assess student work (with transcripts, code, scratch work, and field observations as data sources), we discussed scorings as a team and thus did *not* compute intercoder reliability. Teams of instructors or researchers could replicate our studies (or use our existing data) and use the taxonomy to code the design work of students to determine the taxonomy's usability as an assessment scheme based on computed reliability scores. Such a study could be used as another basis for refining the taxonomy or understanding better how instructors use the taxonomy.

8.5.4 Mitigating short-term learning gains

One major factor that we had to work with when running studies with our students cohorts was their schedule and availability. Some students were only available to do successive study sessions (*e.g.* two 1-hour sessions one after the other), while some were able to do separate days. It is likely that students who did successive sessions could have benefited from solving one multi-task problem after another; in the case of solving Max-Temps after solving Rainfall, for example, working on the sentinel-task in the first session could have given them practice working with the task and influenced how they worked with similar tasks in the second problem (such as handling delimiters). Distractor tasks, similar to what Xie *et al.* [147] have done, could be used to occupy students' working memory with unrelated content to mitigate short-term temporary learning gains. This may help us better tease out how students performed on the problems independently.

8.5.5 Teasing out workload factors of our study sessions

Think-aloud protocols are widely used in many fields as a way of accessing study subjects' thought-processes in short-term working memory [29, 31, 47]. They are used for studies such as those that concern problem-solving, usability, process-analysis, and is particularly popular in computing education research for understanding how people learn programming, among many other uses [41, 86, 97, 109, 140]. One concern about this methodology is the potential overload it may impose on a subject's working memory, which is usually mitigated by having participants practice thinking-aloud first on a (usually unrelated) practice activity before engaging in an actual study protocol; this is the same approach we've taken in our own studies. Nevertheless, our own study protocols require students

to: solve our multi-task problems (which are already designed to be challenging for them), think aloud to verbalize their thought-processes during their design process, recall relevant knowledge from their CS1 course or other prior knowledge, and engage in other activities they deem necessary for solving our problems (*e.g.* sketching their solutions on paper, looking up notes, *etc.*). This combination of activities may have imposed a workload on our students that could have affected how they performed on our problems. A potential way of teasing out how the workload impacted students' performance is with the use of the NASA Task Load Index (NASA-TLX), a validated workload assessment tool developed by the Human Performance Group at NASA Ames Research Center, to assess students' workload during the study sessions [4, 75]. The NASA-TLX has been used in several computing studies (particularly those that require usability studies) and by a few in computing education, such as in DesPortes' physical computing work [40]. Using the NASA-TLX as part of our assessment of students' programming processes could help us further identify the factors that contribute to students' planning and design work.

8.6 Open Questions and Future Work

Research towards theory-development for HTDP can continue along several paths. This dissertation has focused on developing a descriptive conceptual framework of HTDP-trained students' program design skills and skill levels, the interactions between design skills and course contexts, and observed patterns of program design processes. The conceptual framework was developed from a grounded-theory-based analysis of about 180 hours of think-alouds, interviews, and observations of students from several HTDP-based CS1 courses from two universities, as well as data from some HTDP instructors.

8.6.1 Further validation of the SOLO-based program design skills taxonomy

We have begun to validate our SOLO-based skills taxonomy in a couple of ways: first, by using it to categorize data from students beyond those from whose data we derived the taxonomy, and second, by checking the taxonomy with other HTDP instructors to see whether their descriptions of what they look for when assessing student design work were captured by the taxonomy descriptions we defined (the latter refined the taxonomy through the addition of a new skill strand on pattern-use that we were not able to previously capture). Validating the taxonomy further will not only refine our conceptual framework for program design, but also provide evidence and refinement to the taxonomy as a potential tool for assessing student design work. We envision several ways of validating this taxonomy:

- Run studies with CS1 courses that will fix the same problems that students will attempt at multiple points in a course, apply the taxonomy to gauge students' skill levels at each point, and then check whether there is a linear progression in student skills over time. Because students may gain familiarity with the problems and potentially "learn" the solution, the problem cover story could be changed to mitigate short-term temporary learning gains (this is similar to Morrison's methodology on varying example cover stories in her experiments on measuring

cognitive load [93]). We've done a preliminary study using this methodology (Chapter 7 [23]), although our goals for that study was to understand whether students learn high-level plans with a lightweight lecture intervention. Applying the taxonomy to such studies could better tease out how students were thinking around the plans they have seen.

A variation of the above would be to give students a sequence of problems of varying familiarity (reflecting the growing complexity of topics covered in a course), apply the taxonomy to gauge students' levels at each point, then examine whether their skills break down at certain levels of problem complexity; this is what we've done in our own studies (Chapter 4).

- Our taxonomy descriptions are not purely specific to HTDP-based courses, so running the studies described in the previous bullet with multiple courses that use different curricula can help further identify the extent to which the taxonomy can capture how students from different curricular contexts apply their skills, or the extent to which the taxonomy is curriculum-specific.

One other challenge that we encountered in using our taxonomy has to do with the skill on *composing expressions to build function bodies*. We often found that this skill strand often did not lend a strong discriminating power in our analyses of our students' work. We explained that this skill strand emerged from our students' discussions of their design work at the code level, but that oftentimes, we found the students always at the *relational* level given the timing of our studies. We argued that this skill strand was the most mechanical of the design skills in our taxonomy, and that it did not seem to require more from the student than having learned richer code patterns to write. It would be interesting to see whether potential findings from the studies described above would further enhance or deepen our definitions for this skill strand.

8.6.2 Using our skills taxonomy as a framework for designing assessments

Our SOLO taxonomy provides a framework for constructing assessments that witness to various skill levels. Earlier topics within a CS1 course aren't likely to witness to the higher skill levels in the taxonomy, but as the course progresses, students need to be assessed on whether their skills are improving as concepts increase in complexity. The taxonomy can be used as a framework towards designing assessments that require students to demonstrate skills at specific skill levels. How to design such assessments and whether the use of these assessments are, in fact, accurate in determining whether a students is performing a specific skill at a specific level, remains an open question.

8.6.3 Student performance with new instructional activities

We have several recommendations (Section 8.2) for activities aimed at getting students to practice using their design techniques to reason and explain their (task-level and code-level) plans, as well as to use them towards novel problems of varying data scenarios. How would the implementation of such activities in HTDP courses impact how students perform on the multi-task programming problems

used in our studies? For example, do they develop task-level plans earlier in their programming process; in other words (using the programming process patterns we found), will more students exhibit a cyclic vs. a code-focused process? Do students use the design techniques put forth by the recipe more meaningfully rather than in a mechanical way (which is what we observe among many of our students)? Do activities that require students to solve programming problems by developing multiple solutions of varying high-level structures, or having them compare or explain preferences between multiple solutions with different structures, drive or encourage them to engage in planning?

8.6.4 Impact of programming language on students' planning

Most existing work on the challenges of planning (particularly with the Rainfall problem) was conducted in the context of imperative programming [122, 126, 127, 137]. Fisler studied how students who used functional programming (also through HTDP) performed on Rainfall [51]. Given that different programming languages have different affordances and linguistic idioms, a better understanding of how students solve or struggle with multi-task programming problems in different pedagogic contexts and programming languages will enhance our understanding of students' planning towards multi-task programming problems.

All of our host courses taught HTDP through functional programming in Racket. The high-level structures produced by our students are similar to the ones produced by students in Fisler's study, although there were some differences in the language constructs that students used: for example, some of the students from WPI used accumulator-style functions and very few attempted to use higher-order functions; many NEU students used higher-order functions and none used accumulator-style functions. These differences were related to the amount of exposure students from each school had to the constructs. At the time we ran our studies, WPI students had only about a week's exposure to higher-order functions (and mostly focused on `filter` and `map`) and had used accumulator-style programming in their course more; NEU students, on the other hand, had been working on higher-order functions for at least three weeks by the time we ran our studies with the cohort and were scheduled to discuss accumulator-style programming after the study.

Planning studies (for Rainfall) done in imperative contexts have often assumed a default high-level structure, that is, one that took a single pass over the input sequence (usually with `for` or `while` loops) and keeping track of the *sum* and *count* in variables [122, 126, 127, 137] (we call this *single-traversal* in our codings for our studies). We've found students from our own study cohorts attempt this: some used the list template naively and failed to decompose the problem over the *sum* and *count*, thus incorrectly composing the code for the tasks within a single list template. We observed an interesting difficulty/confusion with some of the students who attempted this solution structure with accumulator-style functions: they did not realize to adapt the accumulator pattern to have multiple parameters to track multiple tasks/values. Interpreting this in an imperative context, it would be as if students had only ever seen programs with a single numeric variable; this is not a confusion that we have seen reported in prior planning studies, particularly with the ones done in imperative settings. We also

found that this was related to the examples and course exercises that students have seen (examples and exercises only showed/required the use of a single accumulator parameter).

Many of the prior planning studies also focused on students' errors, rather than their programming processes. It would be interesting to replicate our studies in CS1 courses with imperative contexts to conduct a more granular analysis of those students' design processes to understand better the interplay between design skills and linguistic factors. Do students in those courses struggle with our multi-task problems in similar ways to our students? What specific aspects of the languages have an impact on how students from these different contexts design programs? In discussions with Michelle Craig and Jennifer Campbell from the University of Toronto[6], we discussed that they had adapted the HTDP design recipe in their CS1 course that used Python. They haven't formally evaluated their approach, but have anecdotally described the approach to be "very helpful". Their HTDP-based CS1 course would be a good candidate on which to replicate our studies.

[6]Email communication with Michelle Craig (October 2018) and Jennifer Campbell (June 2019)

Bibliography

[1] Beth Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9(4):422–433, July 1981.

[2] Beth Adelson. When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 10(3):483–495, 1984.

[3] Beth Adelson and Elliot Soloway. The Role of Domain Experience in Software Design. *IEEE Trans. Softw. Eng.*, 11(11):1351–1360, November 1985.

[4] National Aeronautics and Space Administration. TLX @ NASA Ames - NASA TLX Paper/Pencil Version.

[5] John R. Anderson. Acquisition of cognitive skill. *Psychological Review*, 89(4):369–406, 1982.

[6] John R. Anderson, Robert Farrell, and Ron Sauers. Learning to Program in LISP1. *Cognitive Science*, 8(2):87–129, April 1984.

[7] Frances K. Bailie. Improving the Modularization Ability of Novice Programmers. In *Proceedings of the Twenty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '91, pages 277–282, New York, NY, USA, 1991. ACM.

[8] Brett A. Becker. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, New York, NY, USA, 2016. ACM. event-place: Memphis, Tennessee, USA.

[9] Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, 22(4):564–599, October 2013.

[10] Himika Bhattacharya. Interpretive Research. In *The SAGE Encyclopedia of Qualitative Research Methods*. SAGE Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States, 2008.

[11] Irving Biederman and Margaret M. Shiffrar. Sexing day-old chicks: A case study and expert systems analysis of a difficult perceptual-learning task. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 13(4):640–645, 1987.

[12] Annette Bieniusa, Markus Degen, Phillip Heidegger, Peter Thiemann, Stefan Wehr, Martin Gasbichler, Michael Sperber, Marcus Crestani, Herbert Klaeren, and Eric Knauel. 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, pages 1–12, New York, NY, USA, 2008. ACM.

[13] John Burville Biggs and K. F. Collis. *Evaluating the quality of learning : the SOLO taxonomy (structure of the observed learning outcome)*. New York : Academic Press, 1982.

[14] Jeffrey Bonar and Elliot Soloway. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Hum.-Comput. Interact.*, 1(2):133–161, June 1985.

[15] Glenn A. Bowen. Document Analysis as a Qualitative Research Method. *Qualitative Research Journal*, 9(2):27–40, August 2009.

[16] Michael E. Caspersen. *Educating Novices in the Skills of Programming*. PhD thesis, University of Aarhus, Denmark, 2007.

[17] Francisco Enrique Vicente Castro. The Impact of a Single Lecture on Program Plans in First-Year CS. Technical Report, WPI Department of Computer Science, Worcester, Massachusetts, USA, November 2017.

[18] Francisco Enrique Vicente Castro, Seth Adjei, Tyler Colombo, and Neil Heffernan. *Building Models to Predict Hint-or-Attempt Actions of Students*. International Educational Data Mining Society, June 2015.

[19] Francisco Enrique Vicente Castro and Kathi Fisler. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 205–210, New York, NY, USA, 2016. ACM.

[20] Francisco Enrique Vicente Castro and Kathi Fisler. Designing a Multi-faceted SOLO Taxonomy to Track Program Design Skills Through an Entire Course. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*, Koli Calling '17, pages 10–19, New York, NY, USA, 2017. ACM.

[21] Francisco Enrique Vicente Castro and Kathi Fisler. Balancing Act: A Theory on the Interactions Between High-Level Task-thinking and Low-Level Implementation-thinking of Novice Programmers. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pages 295–295, New York, NY, USA, 2019. ACM. event-place: Toronto ON, Canada.

[22] Francisco Enrique Vicente Castro and Kathi Fisler. Qualitative Analyses of Movements Between Task-level and Code-level Thinking of Novice Programmers. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, pages 487–493, Portland, OR, USA, February 2020. Association for Computing Machinery.

[23] Francisco Enrique Vicente Castro, Shriram Krishnamurthi, and Kathi Fisler. The Impact of a Single Lecture on Program Plans in First-year CS. In *Proceedings of the 17th Koli Calling Conference on Computing Education Research*, Koli Calling '17, pages 118–122, New York, NY, USA, 2017. ACM.

[24] Francisco Enrique Vicente G. Castro. Investigating Novice Programmers' Plan Composition Strategies. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 249–250, New York, NY, USA, 2015. ACM.

[25] Francisco Enrique Vicente G. Castro. Pedagogy and Measurement of Program Planning Skills. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 273–274, New York, NY, USA, 2016. ACM.

[26] Francisco Enrique Vicente G. Castro. Towards a Theory of HtDP-based Program-Design Learning. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, ICER '18, pages 260–261, New York, NY, USA, 2018. ACM.

[27] Richard Catrambone. The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General*, 127(4):355–376, 1998.

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

[29] Michelene T. H. Chi. Quantifying Qualitative Analyses of Verbal Data: A Practical Guide. *Journal of the Learning Sciences*, 6(3):271–315, July 1997.

[30] Tamara L. Clegg and Janet L. Kolodner. Bricoleurs and planners engaging in scientific reasoning: a tale of two groups in one learning community. *Research and Practice in Technology Enhanced Learning*, 02(03):239–265, November 2007. Publisher: World Scientific Publishing Co.

[31] John Cowan. The potential of cognitive think-aloud protocols for educational action-research. *Active Learning in Higher Education*, 20(3):219–232, November 2019. Publisher: SAGE Publications.

[32] Marcus Crestani and Michael Sperber. Experience Report: Growing Programming Languages for Beginning Students. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 229–234, New York, NY, USA, 2010. ACM.

[33] Holger Danielsiek, Laura Toma, and Jan Vahrenhold. An Instrument to Assess Self-Efficacy in Introductory Algorithms Courses. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 217–225, Tacoma, Washington, USA, August 2017. Association for Computing Machinery.

[34] Simon P. Davies. Models and theories of programming strategy. *International Journal of Man-Machine Studies*, 39(2):237–267, August 1993.

[35] Michael de Raadt. *Teaching programming strategies explicitly to novice programmers*. Thesis (PhD/Research), University of Southern Queensland, 2008.

[36] Michael de Raadt, Mark Toleman, and Richard Watson. Training Strategic Problem Solvers. *SIGCSE Bull.*, 36(2):48–51, June 2004.

[37] Michael de Raadt, Mark Toleman, and Richard Watson. Incorporating Programming Strategies Explicitly into Curricula. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research - Volume 88*, Koli Calling '07, pages 41–52, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[38] Michael de Raadt, Richard Watson, and Mark Toleman. Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 55–62, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[39] Michael de Raadt, Richard Watson, and Mark Toleman. Teaching and Assessing Programming Strategies Explicitly. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 45–54, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[40] Kayla DesPortes. Learning and Collaboration in Physical Computing. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 283–284, New York, NY, USA, 2016. ACM.

[41] Kayla DesPortes and Betsy DiSalvo. Trials and Tribulations of Novices Working with the Arduino. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pages 219–227, New York, NY, USA, 2019. ACM. event-place: Toronto ON, Canada.

[42] Zoltán Dienes and Dianne Berry. Implicit learning: Below the subjective threshold. *Psychonomic Bulletin & Review*, 4(1):3–23, March 1997.

[43] Brian Dorn and Mark Guzdial. Learning on the job: characterizing the programming knowledge and learning strategies of web designers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 703–712, Atlanta, Georgia, USA, April 2010. Association for Computing Machinery.

[44] DrRacket: The Racket Programming Environment. https://docs.racket-lang.org/drracket/index.html.

[45] Sebastian Dziallas and Sally Fincher. Aspects of Graduateness in Computing Students' Narratives. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 181–190, New York, NY, USA, 2016. ACM.

[46] Dennis E. Egan and Barry J. Schwartz. Chunking in recall of symbolic drawings. *Memory & Cognition*, 7(2):149–158, March 1979.

[47] K. Anders Ericsson and Herbert A. Simon. *Protocol Analysis : Verbal Reports As Data*, volume Rev. ed. MIT Press, Cambridge, Mass, 1993.

[48] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

[49] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The Structure and Interpretation of the Computer Science Curriculum. *J. Funct. Program.*, 14(4):365–378, July 2004.

[50] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! Project: Computing and Programming for Every Student. *Computer Science Education*, 14(1):55–77, January 2004.

[51] Kathi Fisler. The Recurring Rainfall Problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 35–42, New York, NY, USA, 2014. ACM.

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

[53] Kathi Fisler, Shriram Krishnamurthi, and Janet Siegmund. Modernizing Plan-Composition Studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 211–216, New York, NY, USA, 2016. ACM.

[54] Kathy Garvin-Doxas and Lecia J. Barker. Communication in computer science classrooms: understanding defensive climates as a means of creating supportive behaviors, March 2004.

[55] David Ginat and Eti Menashe. SOLO Taxonomy for Assessing Novices' Algorithmic Design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 452–457, New York, NY, USA, 2015. ACM.

[56] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Transaction Publishers, August 2009. Google-Books-ID: tSi7KiOHkpYC.

[57] Wayne D. Gray and John R. Anderson. Change-episodes in coding: when and how do programmers change their code? In *Proceedings of the Second Conference on the Experimental Study of Programming*, pages 185–197, 1987.

[58] Raymonde Guindon, Herb Krasner, and Bill Curtis. Breakdowns and processes during the early activities of software design by professionals. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 65–82. Ablex Publishing Corp., Norwood, NJ, USA, 1987.

[59] Mark Guzdial. Learning Computer Science is Different than Learning Other STEM Disciplines.

[60] Mark Guzdial. A challenge to computing education research: Make measurable progress. https://computinged.wordpress.com/2010/08/16/a-challenge-to-computing-education-research-make-measurable-progress/, August 2010. Accessed April 14, 2017.

[61] Arto Hellas. *Retention in Introductory Programming*. PhD thesis, University of Helsinki, October 2017. Accepted: 2017-10-17T10:48:26Z ISBN: 9789515137982 Publisher: Helsingin yliopisto.

[62] Felienne Hermans and Marileen Smit. Explicit Direct Instruction in Programming Education. In *Psychology of Programming Interest Group 2018 29th Annual Conference*, 2018.

[63] Jacqueline Hundley. A Review of Using Design Patterns in CS1. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, pages 30–33, New York, NY, USA, 2008. ACM.

[64] WPI ITS. WPI ITS | Article | Use Qualtrics. https://its.wpi.edu/article/283/use-qualtrics.

[65] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and Renske Weeda. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '19, pages 27–52, Aberdeen, Scotland Uk, December 2019. Association for Computing Machinery.

[66] Cruz Izu, Amali Weerasinghe, and Cheryl Pope. 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, pages 251–259, New York, NY, USA, 2016. ACM.

[67] Robin Jeffries, Althea Turner, Peter Polson, and Michael Atwood. The processes involved in designing software. In *Cognitive skills and their acquisition*, pages 255–283. Erlbaum, Hillsdale, New Jersey, 1981.

[68] Aaron Keen and Kurt Mammen. Program Decomposition and Complexity in CS1. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 48–53, New York, NY, USA, 2015. ACM.

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

[70] Päivi Kinnunen and Lauri Malmi. Why Students Drop out CS1 Course? In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 97–108, New York, NY, USA, 2006. ACM.

[71] Päivi Kinnunen and Beth Simon. CS Majors' Self-efficacy Perceptions in CS1: Results in Light of Social Cognitive Theory. In *Proceedings of the Seventh International Workshop on Computing Education Research*, ICER '11, pages 19–26, New York, NY, USA, 2011. ACM.

[72] Amy J. Ko. Grading is ineffective, harmful, and unjust — let's stop doing it, March 2019. Library Catalog: medium.com.

[73] Antti-Jussi Lakanen, Vesa Lappalainen, and Ville Isomöttönen. Revisiting rainfall to explore exam questions and performance on CS1. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 40–49, Koli, Finland, November 2015. Association for Computing Machinery.

[74] Celine Latulipe, Audrey Rorrer, and Bruce Long. Longitudinal Data on Flipped Class Effects on Performance in CS1 and Retention after CS1. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 411–416, Baltimore, Maryland, USA, February 2018. Association for Computing Machinery.

[75] Cynthia Laurie-Rose, Meredith Frey, Aristi Ennis, and Amanda Zamary. Measuring Perceived Mental Workload in Children. *The American Journal of Psychology*, 127(1):107–125, 2014. Publisher: University of Illinois Press.

[76] Michael J. Lee and Andrew J. Ko. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*, ICER '11, pages 109–116, Providence, Rhode Island, USA, August 2011. Association for Computing Machinery.

[77] Lawrence Leung. Validity, reliability, and generalizability in qualitative research. *Journal of Family Medicine and Primary Care*, 4(3):324, July 2015. Company: Medknow Publications and Media Pvt. Ltd. Distributor: Medknow Publications and Media Pvt. Ltd. Institution: Medknow Publications and Media Pvt. Ltd. Label: Medknow Publications and Media Pvt. Ltd. Publisher: Medknow Publications.

[78] John Lewis. Redefining Qualitative Methods: Believability in the Fifth Moment. *International Journal of Qualitative Methods*, 8(2):1–14, June 2009. Publisher: SAGE Publications Inc.

[79] Hongjing Liao and John Hitchcock. Reported credibility techniques in higher education evaluation studies that use qualitative methods: A research synthesis. *Evaluation and Program Planning*, 68:157–165, June 2018.

[80] Marcia C. Linn. How can hypermedia tools help teach programming? *Learning and Instruction*, 2(2):119–139, January 1992.

[81] Marcia C. Linn and Michael J. Clancy. Can experts' explanations help students develop program design skills? *International Journal of Man-Machine Studies*, 36(4):511–551, April 1992.

[82] Marcia C. Linn and Michael J. Clancy. The case for case studies of programming problems, March 1992.

[83] Alex Lishinski, Aman Yadav, Jon Good, and Richard Enbody. Learning to Program: Gender Differences and Interactive Effects of Students' Motivation, Goals, and Self-Efficacy on Performance. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 211–220, New York, NY, USA, 2016. ACM.

[84] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '04, pages 119–150, New York, NY, USA, 2004. ACM.

[85] 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. Naturally Occurring Data As Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *SIGCSE Bull.*, 41(4):156–173, January 2010.

[86] Raymond Lister, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad. Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 118–122, New York, NY, USA, 2006. ACM.

[87] S.P. Marshall. *Schemas in Problem Solving*. Cambridge University Press, 1995.

[88] R. E. Mayer. Problem Solving and Reasoning. In Penelope Peterson, Eva Baker, and Barry McGaw, editors, *International Encyclopedia of Education (Third Edition)*, pages 273–278. Elsevier, Oxford, January 2010.

[89] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

[90] Andrew McGettrick, Roger Boyle, Roland Ibbett, John Lloyd, Gillian Lovegrove, and Keith Mander. Grand Challenges in Computing: Education—A Summary. *The Computer Journal*, 48(1):42–48, January 2005.

[91] Alex McLean and Geraint Wiggins. Bricolage Programming in the Creative Arts. In *Psychology of Programming Interest Group 2010 22nd Annual Workshop*, 2010.

[92] George Armitage Miller. The Magical Number Seven, Plus Or Minus Two: Some Limits on Our Capacity for Processing Information. In *Psychological Review*, volume 63, pages 81–97. American Psychological Association, 1956. Google-Books-ID: AFn0ngEACAAJ.

[93] Briana B. Morrison. Computer Science Is Different!: Educational Psychology Experiments Do Not Reliably Replicate in Programming Domain. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, ICER '15, pages 267–268, New York, NY, USA, 2015. ACM.

[94] Orna Muller, David Ginat, and Bruria Haberman. Pattern-oriented Instruction and Its Influence on Problem Decomposition and Solution Construction. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 151–155, New York, NY, USA, 2007. ACM.

[95] Orna Muller, Bruria Haberman, and Haim Averbuch. (An Almost) Pedagogical Pattern for Pattern-based Problem-solving Instruction. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '04, pages 102–106, New York, NY, USA, 2004. ACM.

[96] Felipe Munoz-Rubke, Devon Olson, Russell Will, and Karin H. James. Functional fixedness in tool use: Learning modality, limitations and individual differences. *Acta Psychologica*, 190:11–26, October 2018.

[97] Brad A. Myers, Andrew J. Ko, Thomas D. LaToza, and YoungSeok Yoon. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer*, 49(7):44–52, July 2016.

[98] Greg L. Nelson, Andrew Hu, Benjamin Xie, and Amy J. Ko. Towards validity for a formative assessment for language-specific program tracing skills. In *Koli Calling '19: Proceedings of the 19th Koli Calling International Conference on Computing Education Research*, Koli Calling '19, Koli, Finland, November 2019. Association for Computing Machinery, New York, NY, United States.

[99] A. Newell, P.S. Rosenbloom, and J.E. Laird. Symbolic architectures for cognition. In *Foundations of Cognitive Science*, pages 93–131. MIT Press, 1989.

[100] Mogens Allan Niss. The concept and role of theory in mathematics education. In *Proceedings of Norma 05*, Trondheim, 2006.

[101] Jun Rangie C. Obispo and Francisco Enrique Vicente G. Castro. Incidence of Einstellung Effect among Programming Students and its Relationship with Achievement. In *Proceedings of the Information and Computing Education Conference 2018*, 2018.

[102] Alexandre Manuel Pais, Diana Stentoft, and Paola Valero. From questions of how to questions of why in mathematics education research. In *Proceedings of the Third International Mathematics Education Society Conference*, volume 2, pages 369–378, Berlin, 2010.

[103] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. Conditions of Learning in Novice Programmers:. *Journal of Educational Computing Research*, January 1995. Publisher: SAGE PublicationsSage CA: Los Angeles, CA.

[104] Andrew Petersen, Michelle Craig, Jennifer Campbell, and Anya Tafliovich. Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, Koli Calling '16, pages 71–80, New York, NY, USA, 2016. ACM.

[105] Peter Pirolli. A Cognitive Model and Computer Tutor for Programming Recursion. *Human–Computer Interaction*, 2(4):319–355, December 1986.

[106] Peter L. Pirolli and John R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 39(2):240–272, 1985.

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

[108] Ron Porter and Paul Calder. A Pattern-based Problem-solving Process for Novice Programmers. In *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*, ACE '03, pages 231–238, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[109] James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, pages 74–82, Tacoma, Washington, USA, August 2017. Association for Computing Machinery.

[110] Racket. https://racket-lang.org/.

[111] Vennila Ramalingam and Susan Wiedenbeck. Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Group Analyses of Novice Programmer Self-Efficacy:. *Journal of Educational Computing Research*, February 1999. Publisher: SAGE PublicationsSage CA: Los Angeles, CA.

[112] Norman Ramsey. On Teaching *How to Design Programs*: Observations from a Newcomer. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 153–166, New York, NY, USA, 2014. ACM.

[113] T. M. Rao and Sandeep Mitra. An Early Software Engineering Approach to Teaching Cs1, Cs2 and Ai. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 143–147, New York, NY, USA, 2008. ACM.

[114] Arthur S. Reber. Implicit Learning and Tacit Knowledge: An Essay on the Cognitive Unconscious. *Oxford University Press*, 1993.

[115] Yanyan Ren, Shriram Krishnamurthi, and Kathi Fisler. What Help Do Students Seek in TA Office Hours? In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pages 41–49, Toronto ON, Canada, July 2019. Association for Computing Machinery.

[116] Carol Righi, Janice James, Michael Beasley, Donald L. Day, Jean E. Fox, Jennifer Gieber, Chris Howe, and Laconya Ruby. Card Sort Analysis Best Practices. *J. Usability Studies*, 8(3):69–89, May 2013.

[117] Robert S. Rist. Schema creation in programming. *Cognitive Science*, pages 389–414, 1989.

[118] Robert S. Rist. Variability in program design: the interaction of process with knowledge. *International Journal of Man-Machine Studies*, 33(3):305–322, September 1990.

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

[120] Robert S. Rist. Search Through Multiple Representations. In *User-Centred Requirements for Software Engineering Environments*, NATO ASI Series, pages 165–176. Springer, Berlin, Heidelberg, 1994.

[121] Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, June 2003.

[122] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. Do We Know How Difficult the Rainfall Problem is? In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 87–96, New York, NY, USA, 2015. ACM.

[123] Peter Sestoft. *Java Precisely*. MIT Press, third edition, 2016.

[124] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, June 1979.

[125] Shuhaida Shuhidan, Margaret Hamilton, and Daryl D'Souza. A Taxonomic Study of Novice Programming Summative Assessment. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 147–156, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[126] Simon. Soloway's Rainfall Problem Has Become Harder. In *2013 Learning and Teaching in Computing and Engineering*, pages 130–135, March 2013.

[127] E. Soloway. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM*, 29(9):850–858, September 1986.

[128] Michael Sperber and Marcus Crestani. Form over Function: Teaching Beginners How to Construct Programs. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*, Scheme '12, pages 81–89, New York, NY, USA, 2012. ACM.

[129] James C. Spohrer and Elliot Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Commun. ACM*, 29(7):624–632, July 1986.

[130] James C. Spohrer and Elliot Soloway. Simulating Student Programmers. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, pages 543–549, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[131] Evelyn Stiller. Teaching programming using bricolage. *Journal of Computing Sciences in Colleges*, 24(6):35–42, June 2009.

[132] Donna Teague and Raymond Lister. Longitudinal Think Aloud Study of a Novice Programmer. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 41–50, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[133] Donna Teague and Raymond Lister. Manifestations of Preoperational Reasoning on Similar Programming Tasks. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148*, ACE '14, pages 65–74, Darlinghurst, Australia, Australia, 2014. Australian Computer Society, Inc.

[134] TechSmith Snagit | Screen capture and screen recorder. https://www.techsmith.com/screen-capture.html.

[135] Errol Thompson. Holistic Assessment Criteria: Applying SOLO to Programming Projects. In *Proceedings of the Ninth Australasian Conference on Computing Education - Volume 66*, ACE '07, pages 155–162, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[136] Sherry Turkle and Seymour Papert. Epistemological Pluralism: Styles and Voices within the Computer Culture. *Signs*, 16(1):128–157, October 1990.

[137] Anne Venables, Grace Tan, and Raymond Lister. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, ICER '09, pages 117–128, New York, NY, USA, 2009. ACM.

[138] David Weintrop and Nathan Holbert. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 633–638, New York, NY, USA, 2017. ACM.

[139] Jacqueline Whalley, Tony Clear, Phil Robbins, and Errol Thompson. Salient Elements in Novice Solutions to Code Writing Problems. In *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, pages 37–46, Darlinghurst, Australia, Australia, 2011. Australian Computer Society, Inc.

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

[141] Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[142] Robin Whittemore, Susan K. Chase, and Carol Lynn Mandle. Validity in Qualitative Research. *Qualitative Health Research*, 11(4):522–537, July 2001. Publisher: SAGE Publications Inc.

[143] Michael R. Wick. Teaching Design Patterns in CS1: A Closed Laboratory Sequence Based on the Game of Life. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 487–491, New York, NY, USA, 2005. ACM.

[144] Jed R. Wood and Larry E. Wood. Card Sorting: Current Practices and Beyond. *J. Usability Studies*, 4(1):1–6, November 2008.

[145] WPI ITS | Software | Snagit. https://its.wpi.edu/software/215/snagit-.

[146] John Wrenn and Shriram Krishnamurthi. Executable Examples for Programming Problem Comprehension. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*, ICER '19, pages 131–139, New York, NY, USA, 2019. ACM. event-place: Toronto ON, Canada.

[147] Benjamin Xie, Dastyni Loksa, Greg L. Nelson, Matthew J. Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Andrew J. Ko. A theory of instruction for introductory programming skills. *Computer Science Education*, 0(0):1–49, January 2019.

[148] Tzippora Yeshno and Mordechai Ben-Ari. Salvation for Bricoleurs, 2001. Place: PPIG 2001 - 13th Annual Workshop.

[149] Tania Zittoun and Svend Brinkmann. Learning as Meaning Making. In Norbert M. Seel, editor, *Encyclopedia of the Sciences of Learning*, pages 1809–1811. Springer US, Boston, MA, 2012.

Appendix A

Solutions to the Study Problems

A.1 Rainfall: Clean-first

```
;; rainfall: list[numbers] -> number
;; Produces the average of nonnegatives up to, excluding, -999
(define (rainfall input)
  (compute-average (remove-negatives (truncate input))))


;; truncate: list[numbers] -> list[numbers]
;; Produces the prefix of the list before -999
(define (truncate input)
  (cond [(or (empty? input) (= -999 (first input))) empty]
        [(cons? input) (cons (first input)
                             (truncate (rest input)))]))


;; remove-negatives: list[numbers] -> list[numbers]
;; Produces a list of only nonnegative numbers
(define (remove-negatives input)
  (cond [(empty? input) empty]
        [(negative? (first input)) (remove-negatives (rest input))]
        [else (cons (first input)
                    (remove-negatives (rest input)))]))


;; compute-average: list[numbers] -> number
;; Produces the average of a list of numbers
(define (compute-average input)
  (cond [(empty? input) -1]
        [(cons? input) (/ (sum input)
                          (length input))]))


;; sum: list[numbers] -> number
;; Produces the sum of a list of numbers
(define (sum input)
  (cond [(empty? input) 0]
        [(cons? input) (+ (first input)
                          (sum (rest input)))]))
```

Figure A.1: Rainfall solution using the *Clean-first* approach

A.2 Rainfall: Process-multiple

```
;; rainfall: list[numbers] -> number
;; Produces the average of nonnegatives up to, excluding, -999
(define (rainfall input)
  (cond [(or (empty? input) (= 0 (count input))) -1]
        [else (compute-average input)]))


;; compute-average: list[numbers] -> number
;; Produces the average of a list of numbers
(define (compute-average input)
  (cond [(empty? input) -1]
        [(cons? input) (/ (sum input)
                          (count input))]))


;; sum: list[numbers] -> number
;; Produces the sum of nonnegatives in a list up to, excluding, -999
(define (sum input)
  (cond [(or (empty? input) (= -999 (first input))) 0]
        [(negative? (first input)) (sum (rest input))]
        [else (+ (first input) (sum (rest input)))]))


;; count: list[numbers] -> number
;; Counts the nonnegatives in a list up to, excluding, -999
(define (count input)
  (cond [(or (empty? input) (= -999 (first input))) 0]
        [(negative? (first input)) (count (rest input))]
        [else (+ 1 (count (rest input)))]))
```

Figure A.2: Rainfall solution using the *Process-multiple* approach

A.3 Rainfall: Single-traversal

```
;; rainfall: list[numbers] -> number
;; Produces the average of nonnegatives up to, excluding, -999
(define (rainfall input)
  (cond [(empty? input) -1]
        [else (rainfall-acc input 0 0)]))


;; rainfall-acc: list[numbers], number, number -> number
;; Produces the average of nonnegatives up to, excluding, -999
(define (rainfall-acc input sum-acc count-acc)
  (cond [(or (empty? input) (= -999 (first input))) (cond [(= 0 count-acc) -1]
                                                          [else (/ sum-acc
                                                                   count-acc)])]
        [(negative? (first input)) (rainfall-acc (rest input)
                                                 sum-acc
                                                 count-acc)]
        [else (rainfall-acc (rest input)
                            (+ (first input) sum-acc)
                            (+ 1 count-acc))]))
```

Figure A.3: Rainfall solution using the *Single-traversal* approach

A.4 Max-Temps: Reshape-first

```
;; max-temps1: list[numbers/strings] -> list[numbers]
;; Produce a list of the maximum elements within each sublist
;; This version uses a higher-order function
(define (max-temps1 list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (map find-list-max (reshape list-input))]))


;; max-temps2: list[numbers/strings] -> list[numbers]
;; Produce a list of the maximum elements within each sublist
;; This version uses a regular list traversal (in a helper function)
(define (max-temps2 list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (max-per-list (reshape list-input))]))


;; reshape: list[numbers/strings] -> list[list[numbers]]
;; Reshape a flat list into a list of lists, excluding delimiters
(define (reshape list-input)
  (cond [(empty? list-input) empty]
        [(number? (first list-input)) (cons (get-front-sublist list-input)
                                            (reshape (skip-front-sublist list-input)))]
        [(string? (first list-input)) (reshape (skip-front-sublist list-input))]))


;; get-front-sublist: list[numbers/strings] -> list[numbers]
;; Get the sublist at the front of the list before the first delimiter
(define (get-front-sublist list-input)
  (cond [(or (empty? list-input)
             (string? (first list-input))) empty]
        [(number? (first list-input)) (cons (first list-input)
                                            (get-front-sublist (rest list-input)))]))


;; skip-front-sublist: list[numbers/strings] -> list[numbers/strings]
;; Get the tail of the list excluding the first sublist
(define (skip-front-sublist list-input)
  (cond [(empty? list-input) empty]
        [(string? (first list-input)) (rest list-input)]
        [(number? (first list-input)) (skip-front-sublist (rest list-input))]))


;; max-per-list: list[list[numbers]] -> list[numbers]
;; Produces a list of the max values from each inner-list
(define (max-per-list list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (cons (find-list-max (first list-input))
                                  (max-per-list (rest list-input)))]))


;; find-list-max: list[numbers] -> number
;; Produce the maximum value in a list
(define (find-list-max list-input)
  (cond [(= 1 (length list-input)) (first list-input)]
        [else (max (first list-input)
                   (find-list-max (rest list-input)))]))
```

Figure A.4: Max-Temps solution using the *Reshape-first* approach

A.5 Max-Temps: Collect-first (accumulator-style)

```
;; max-temps: list[numbers/strings] -> list[numbers]
;; Produce a list of the maximum elements within each sublist
;; This version uses an accumulator to collect
;; the elements in the current sublist
(define (max-temps input)
  (cond [(empty? input) empty]
        [(cons? input) (max-temps-collect input empty)]))


;; max-temps-collect: list[numbers/strings] list[numbers] -> list[numbers]
;; Produce a list of the maximum elements within each sublist
;; Accumulator curr-list keeps track of the current sublist being processed
(define (max-temps-collect input curr-list)
  (cond [(empty? input) (cond [(empty? curr-list) empty]
                              [(cons? curr-list) (cons (find-list-max curr-list) empty)])]
        [(string? (first input)) (cond [(empty? curr-list) (max-temps-collect (rest input) empty)]
                                       [(cons? curr-list) (cons (find-list-max curr-list)
                                                                (max-temps-collect (rest input) empty))])]
        [(number? (first input)) (max-temps-collect (rest input)
                                                    (cons (first input) curr-list))]))


;; find-list-max: list[numbers] -> number
;; Produce the maximum value in a list
(define (find-list-max list-input)
  (cond [(= 1 (length list-input)) (first list-input)]
        [else (max (first list-input)
                   (find-list-max (rest list-input)))]))
```

Figure A.5: Max-Temps solution using the *Collect-first* approach (accumulator-style)

A.6 Adding Machine: Reshape-first

```
;; adding-machine1: list[numbers] -> list[numbers]
;; Produce a list of the sums within each sublist
;; This version uses a higher-order function
(define (adding-machine1 list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (map sum (reshape (clean-extra-zeros (truncate list-input))))]))


;; adding-machine2: list[numbers] -> list[numbers]
;; Produce a list of the sums within each sublist
;; This version uses a regular list traversal (in a helper function)
(define (adding-machine2 list-input)
    (cond [(empty? list-input) empty]
          [(cons? list-input) (sum-per-list (reshape (clean-extra-zeros (truncate list-input))))]))


;; truncate: list[numbers] -> list[numbers]
;; Produce the list before the double-zero sentinel pattern
(define (truncate list-input)
  (cond [(empty? list-input) empty]
        [(= 1 (length list-input)) list-input]
        [(and (= 0 (first list-input)) (= 0 (second list-input))) empty]
        [(cons? list-input) (cons (first list-input) (truncate (rest list-input)))]))


;; clean-extra-zeros: list[numbers] -> list[numbers]
;; Remove the zero at the head of the list, if any
(define (clean-extra-zeros list-input)
  (cond [(empty? list-input) empty]
        [(= 0 (first list-input)) (rest list-input)]
        [else list-input]))


;; reshape: list[numbers] -> list[list[numbers]]
;; Reshape a flat list into a list of lists, excluding delimiters
(define (reshape list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (cons (get-front-sublist list-input)
                                  (reshape (skip-front-sublist list-input)))]))


;; get-front-sublist: list[numbers] -> list[numbers]
;; Get the sublist at the front of the list before the first delimiter
(define (get-front-sublist list-input)
  (cond [(or (empty? list-input) (= 0 (first list-input))) empty]
        [(cons? list-input) (cons (first list-input)
                                  (get-front-sublist (rest list-input)))]))


;; skip-front-sublist: list[numbers] -> list[numbers]
;; Get the tail of the list excluding the first sublist
(define (skip-front-sublist list-input)
  (cond [(empty? list-input) empty]
        [(= 0 (first list-input)) (rest list-input)]
        [else (skip-front-sublist (rest list-input))]))


;; sum-per-list: list[list[numbers]] -> list[numbers]
;; Produces a list of the sums from each inner-list
(define (sum-per-list list-input)
  (cond [(empty? list-input) empty]
        [(cons? list-input) (cons (sum (first list-input))
                                  (sum-per-list (rest list-input)))]))


;; sum: list[numbers] -> number
;; Produce the sum of the list of numbers
(define (sum list-input)
  (cond [(empty? list-input) 0]
        [(cons? list-input) (+ (first list-input)
                               (sum (rest list-input)))]))
```

Figure A.6: Adding Machine solution using the *Reshape-first* approach

A.7 Adding Machine: Accumulator-style

```
;; adding-machine: list[numbers] -> list[numbers]
;; Produce a list of the sums within each sublist
(define (adding-machine input)
  (cond [(empty? input) empty]
        [(cons? input) (adding-machine-accum input empty)]))


;; adding-machine-accum: list[numbers] list[numbers] -> list[numbers]
;; Produce a list of the sums within each sublist
;; Accumulator curr-list keeps track of the current sublist being processed
(define (adding-machine-accum input curr-list)
  (cond [(empty? input) (cond [(empty? curr-list) empty]
                              [(cons? curr-list) (cons (sum curr-list) empty)])]
        [(= 0 (first input)) (cond [(empty? (rest input)) (cond [(empty? curr-list) empty]
                                                                [(cons? curr-list) (cons (sum curr-list) empty)])]
                                   [(= 0 (second input)) (cond [(empty? curr-list) empty]
                                                               [(cons? curr-list) (cons (sum curr-list) empty)])]
                                   [else (cond [(empty? curr-list) (adding-machine-accum (rest input) empty)]
                                               [(cons? curr-list) (cons (sum curr-list)
                                                                        (adding-machine-accum (rest input) empty))])])]
        [(not (= 0 (first input))) (adding-machine-accum (rest input)
                                                         (cons (first input) curr-list))]))


;; sum: list[numbers] -> number
;; Produce the sum of the list of numbers
(define (sum list-input)
  (cond [(empty? list-input) 0]
        [(cons? list-input) (+ (first list-input)
                               (sum (rest list-input)))]))
```

Figure A.7: Adding Machine solution using an accumulator-style approach

Appendix B

HTDP and Focal Expansion Model Study: Instruments and Data

B.1 SnagIt setup web page

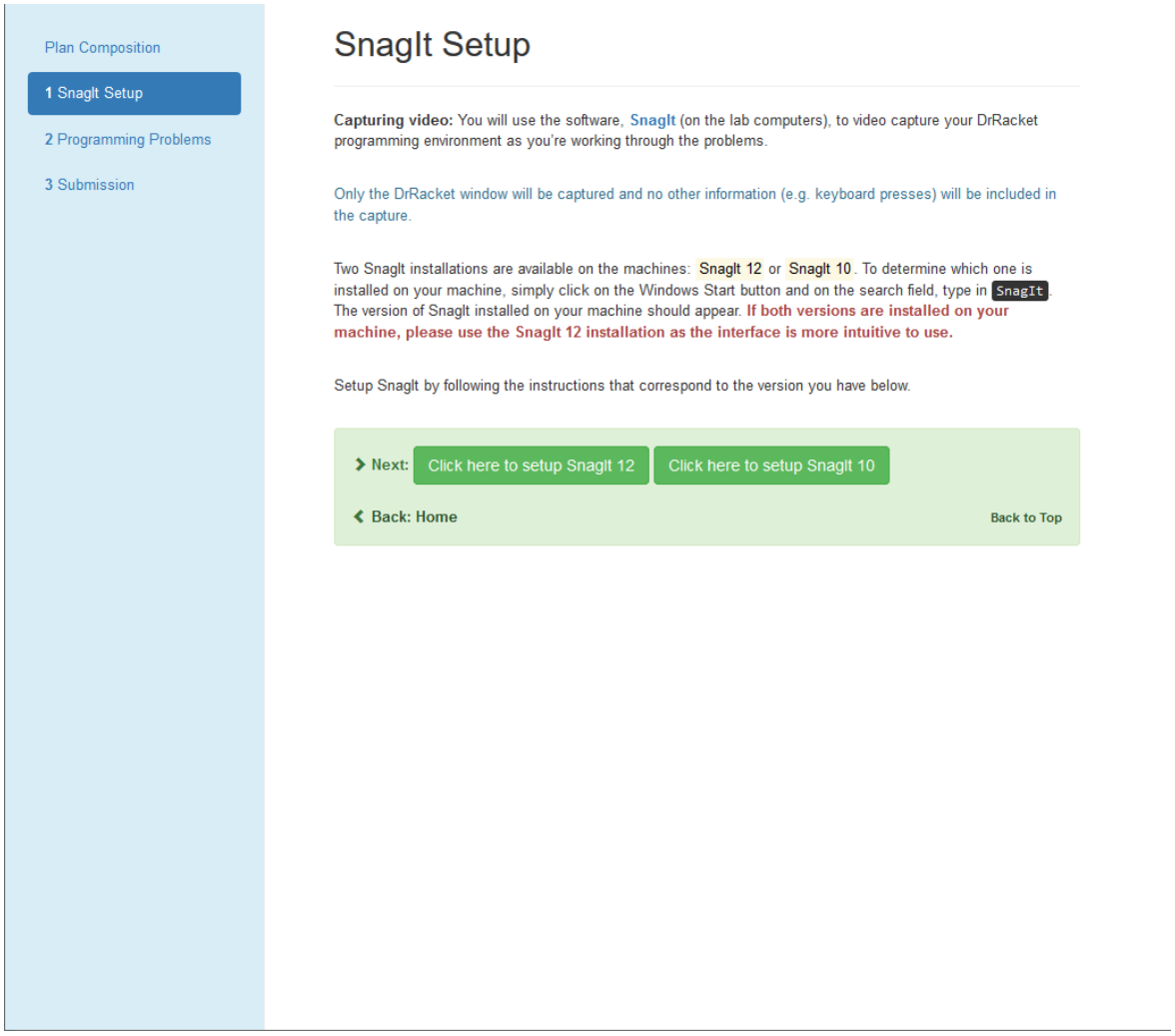


Figure B.1: Instructions for setting up SnagIt to capture programming sessions.

B.2 Adding Machine problem statement web page

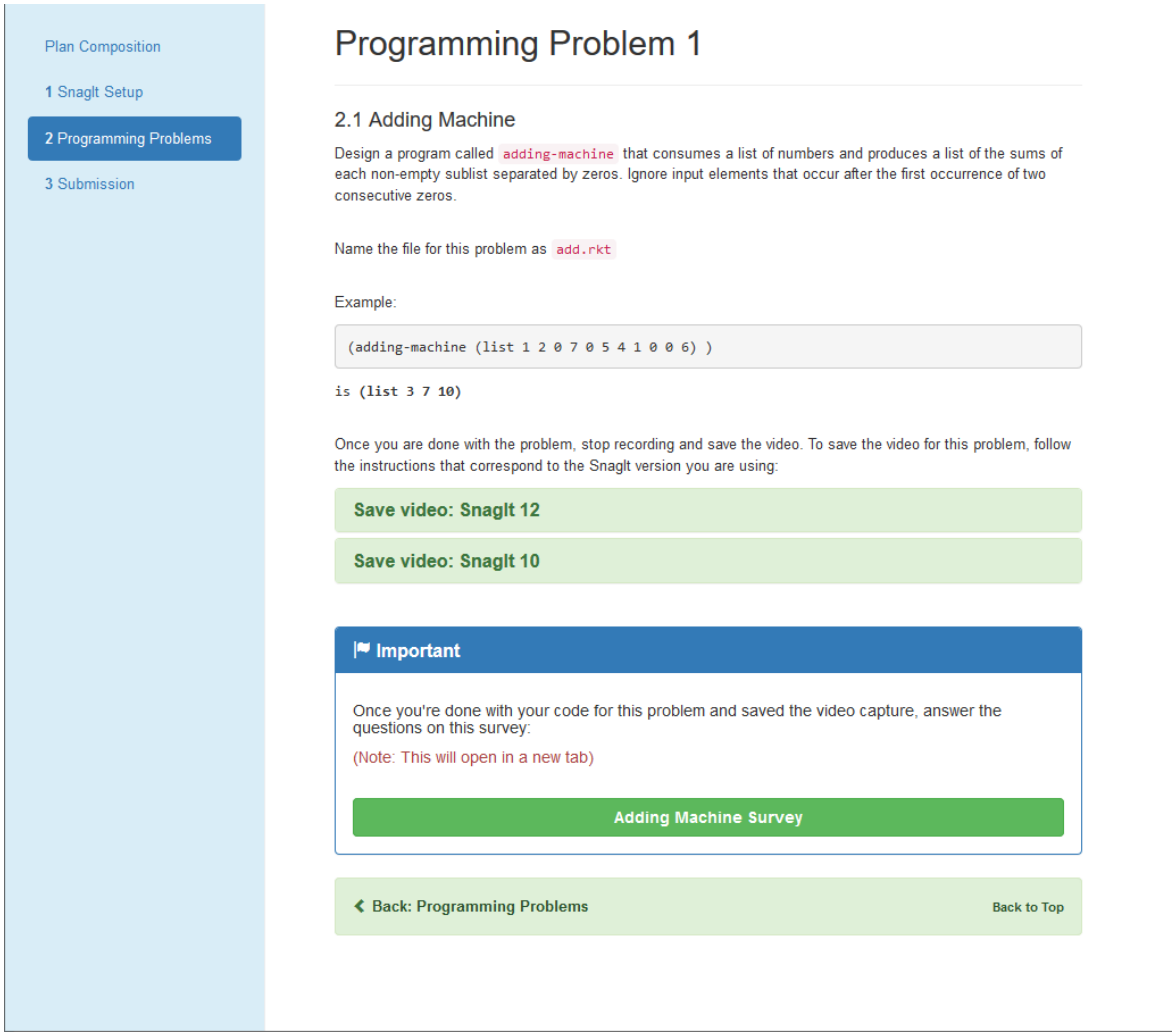


Figure B.2: Adding Machine problem statement used for the study.

B.3 SnagIt video save web page

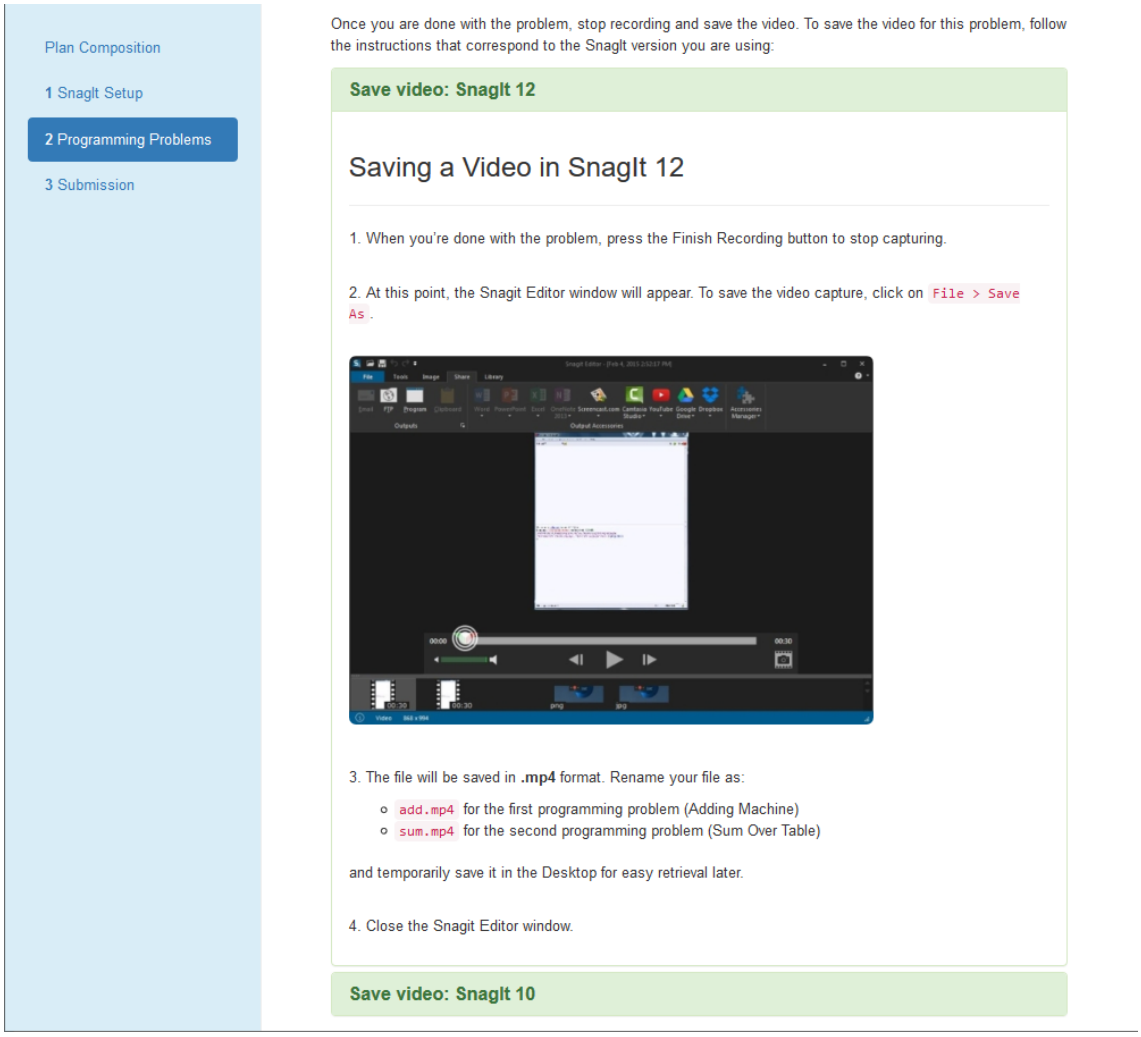


Figure B.3: Instructions for saving the SnagIt programming video capture (SnagIt12 version).

B.4 Code and programming video submission web page

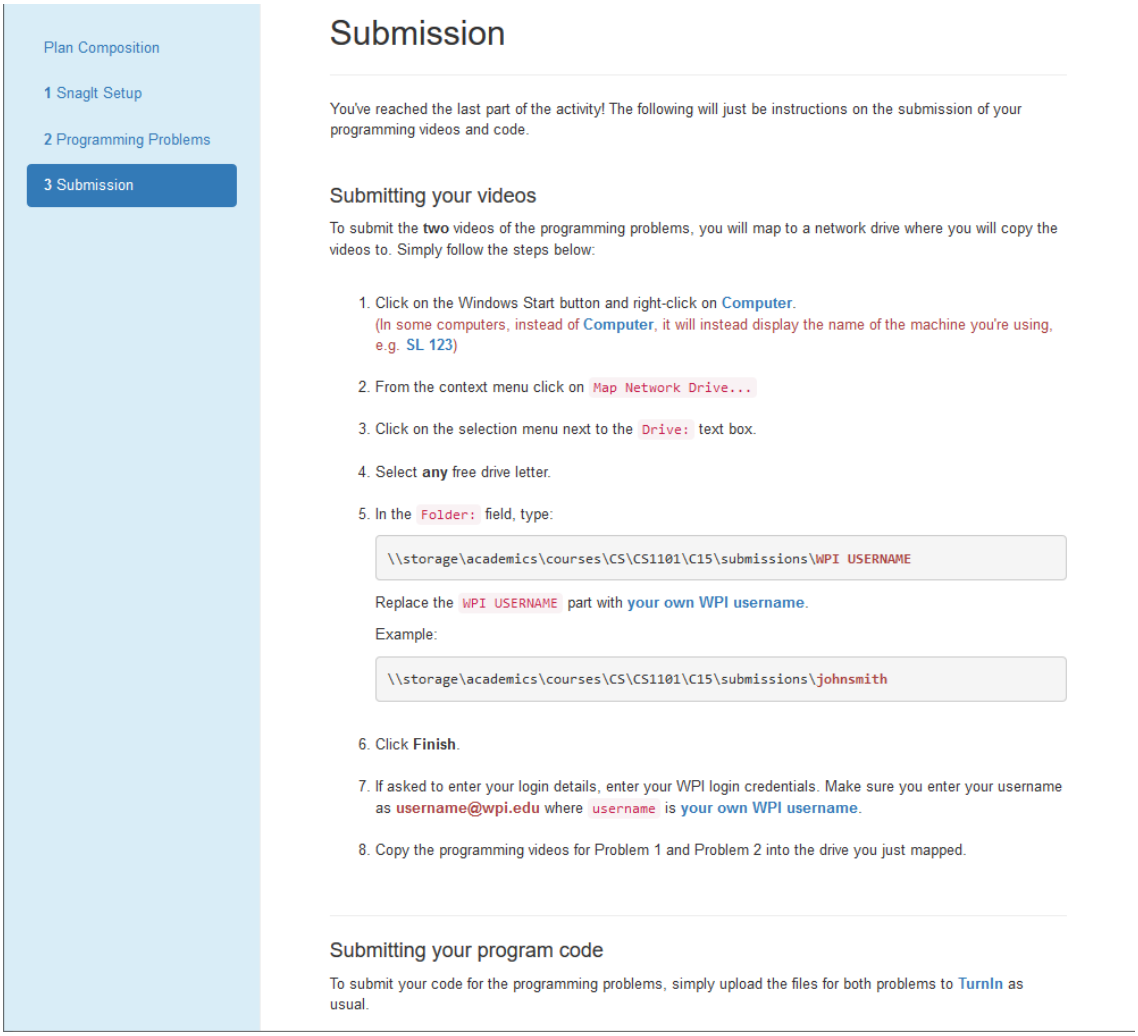
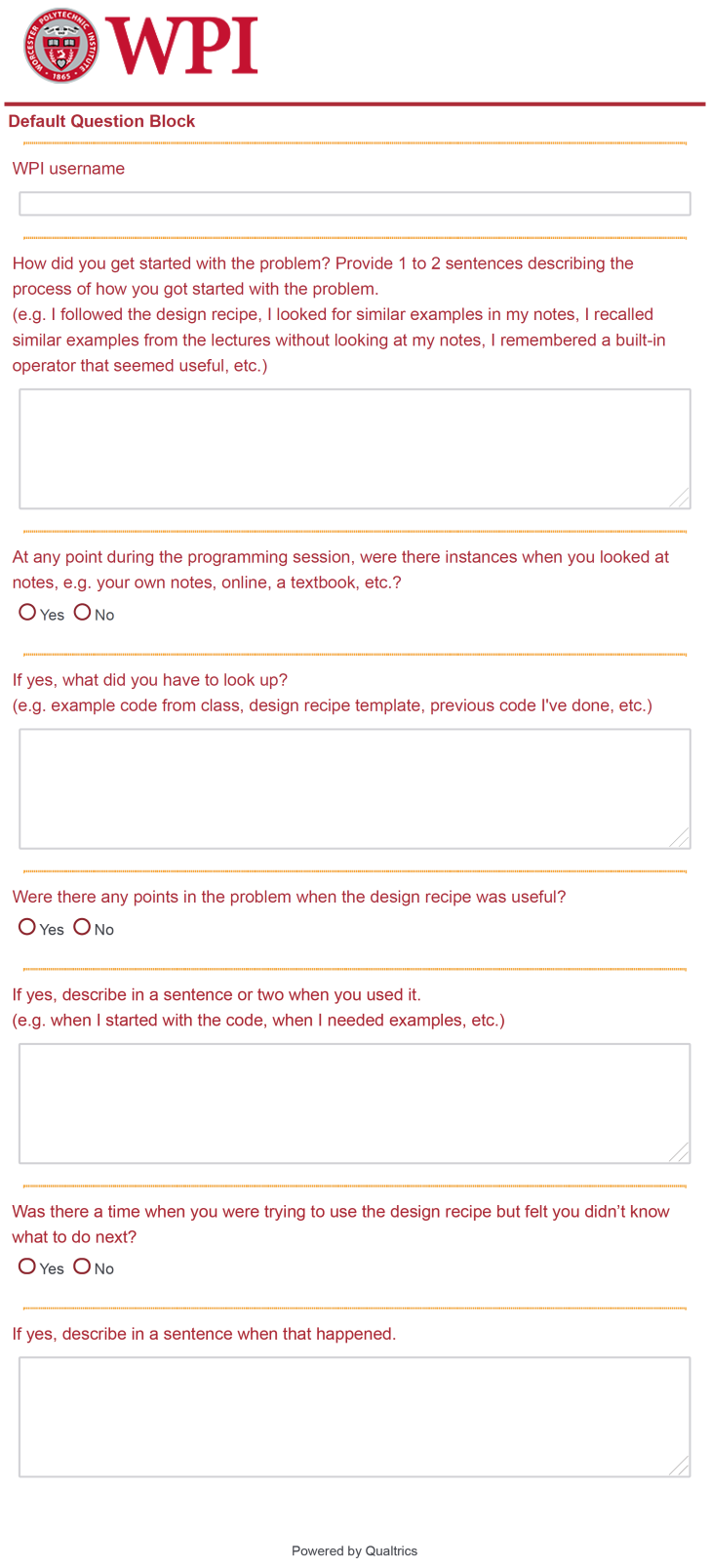


Figure B.4: Instructions for submitting the Adding Machine code and SnagIt programming video.

B.5 Adding Machine post-programming survey



WPI

Default Question Block

WPI username

How did you get started with the problem? Provide 1 to 2 sentences describing the process of how you got started with the problem.
(e.g. I followed the design recipe, I looked for similar examples in my notes, I recalled similar examples from the lectures without looking at my notes, I remembered a built-in operator that seemed useful, etc.)

At any point during the programming session, were there instances when you looked at notes, e.g. your own notes, online, a textbook, etc.?

Yes No

If yes, what did you have to look up?
(e.g. example code from class, design recipe template, previous code I've done, etc.)

Were there any points in the problem when the design recipe was useful?

Yes No

If yes, describe in a sentence or two when you used it.
(e.g. when I started with the code, when I needed examples, etc.)

Was there a time when you were trying to use the design recipe but felt you didn't know what to do next?

Yes No

If yes, describe in a sentence when that happened.

Powered by Qualtrics

Figure B.5: The survey students filled out after working on the Adding Machine problem (administered through the WPI Qualtrics [64] distribution).

B.6 Post-programming survey question 1 responses

Table B.1: Student responses (verbatim) to post-programming survey question 1.

Student	**How did you get started with the problem? Provide 1 to 2 sentences describing the process of how you got started with the problem.**
WPI1-STUD1	I started the problem by defining the definition of the list, and then creating the template for the function. From there, I planned to use the template along with a helper function to do the legwork of the problem.
WPI1-STUD2	I started with data definition first, then template, then actual function
WPI1-STUD3	I started with check expects. Then i tried to find a template to hopefully move off from on the coursera site. I wasnt aware we could use our notes for the whole period.
WPI1-STUD4	I stared at it before I started recording, thinking of different ways to tackle the problem. I then began to write a general body of the code, realizing I was confused still I wrote out a signature/purpose and kept working.
WPI1-STUD5	I did most of the problem off the top of my head
WPI1-STUD6	I tried writing a function that took 2 lists as an argument and built one list with each element that wasn't a zero and appended all theses sub-lists together.
WPI1-STUD7	understood what the concepts needed to do the problem, got confused in what i should put in which function.
WPI1-STUD8	I wrote a description of what I wanted my function to do
WPI1-STUD9	I tried to rush a bit earlier on thinking that I had less time. I did not actually write the template although I had them in my mind. At the very end it struck on my mind that we should use local.
WPI1-STUD10	I tried to recreate a definition that we had done in class before because we have been doing similar exercises throughout the course.
WPI1-STUD11	I tried to write the main function first, but quickly realized that I needed to write some helper functions before continuing.
WPI1-STUD12	Started with a main function and then relized needed several helper functions. started to work on those
WPI1-STUD13	i first started with writing out the purpose of what the function should do, then wrote the check expects then went into writing the function.

Table B.1 continued from previous page

Student	How did you get started with the problem? Provide 1 to 2 sentences describing the process of how you got started with the problem.
WPI1-STUD14	I followed the design recipe. I started with the signature of ListOfNumbers -> ListOfNumbers and a brief description of what I wanted my function to do. Afterwards, I wrote a couple check expects again to know what I wanted my function to do. I didn't recall ever having to write a function that consumed a list of numbers and produce a list of numbers, but I did remember in the last project we had something that consumed a list of ancestors and produced a list of strings. I wrote the template for a list of numbers and worked from there.
WPI1-STUD15	Basic HtDF, Signature, Purpose, Stub, Test Cases, etc
WPI1-STUD16	I started with writhing the check expect, and understant the question.
WPI1-STUD17	I started with the check-expects and after that tried to write a template. After I had the template, I tried to think of a way to change the template to get what i needed to solve the problem.
WPI1-STUD18	I created check expects and then tried to follow the list template.
WPI1-STUD19	I recalled similar examples from class but after I looked up in my notes.
WPI1-STUD20	I first read the problem, then started looking on my notebook to look for the appropriate function for this problem. And also I checked on the brozser for some other function that could help me in this problem
WPI1-STUD21	I followed the HtDF Recipe, and found a list template from my notes.
WPI1-STUD22	I tried to use the recipe. I wasn't clear on which template I needed, so that gave me some trouble when I tried to write the problem.
WPI1-STUD23	I started with the template for a list function, but ended up abandoning that approach. Im not a fan of the templates other than just a very very basic starting point. I never adhere strictly to the template. I just use trial and error and hope that my logic works. I also use check expects to see where the program is going wrong so I can understand how to fix it.
WPI1-STUD24	I started with a template, then moved onto the function. When I got stumped on the function, I moved on to check-expects to help me break down my thinking.
WPI1-STUD25	tried to write out the template and starting HTDP stuff

B.7 Post-programming survey question 2 responses

Table B.2: Student responses (verbatim) to post-programming survey question 2.

Student	At any point during the programming session, were there instances when you looked at notes?	If yes, what did you have to look up?
WPI1-STUD1	No	
WPI1-STUD2	No	
WPI1-STUD3	Yes	i looked up add list template
WPI1-STUD4	Yes	Racket directory for a few functions to make sure I was using them right
WPI1-STUD5	No	
WPI1-STUD6	No	
WPI1-STUD7	No	
WPI1-STUD8	No	
WPI1-STUD9	Yes	the racket documentation
WPI1-STUD10	Yes	I had to look up what expressions to use and went to the how to design functions page.
WPI1-STUD11	Yes	design recipe templates, example code
WPI1-STUD12	No	
WPI1-STUD13	No	
WPI1-STUD14	No	
WPI1-STUD15	Yes	Racket Documentation
WPI1-STUD16	Yes	I want to check is that a function that is exist.
WPI1-STUD17	Yes	Tried to look for previous examples.
WPI1-STUD18	Yes	I looked up certain functions on racket help.
WPI1-STUD19	Yes	notes i took from videos.
WPI1-STUD20	Yes	I looked at previous problems that looks similar to this problem and see how they were solved
WPI1-STUD21	Yes	Templates for lists and helper functions.
WPI1-STUD22	No	
WPI1-STUD23	No	
WPI1-STUD24	No	
WPI1-STUD25	No	

B.8 Post-programming survey question 3 responses

Table B.3: Student responses (verbatim) to post-programming survey question 3.

Student	Were there any points in the problem when the design recipe was useful?	If yes, describe in a sentence or two when you used it.
WPI1-STUD1	Yes	Using the template to operate on the list.
WPI1-STUD2	No	
WPI1-STUD3	Yes	when i started the code
WPI1-STUD4	Yes	Clarity from signature/purpose
WPI1-STUD5	No	
WPI1-STUD6	Yes	signature and purpose statemnt
WPI1-STUD7	No	
WPI1-STUD8	Yes	when I started
WPI1-STUD9	Yes	The basic template was useful
WPI1-STUD10	Yes	It helped me set up how to write the function correctly.
WPI1-STUD11	Yes	i used examples to help me understand the problem more clearly
WPI1-STUD12	No	
WPI1-STUD13	Yes	started with the check expects helped and also writing out the purpose of the function.
WPI1-STUD14	Yes	I used it when I started the code and tried to come up with possible helper functions for my code. Additionally, It told me what I wanted my output to look like.
WPI1-STUD15	Yes	The entire code, untill I considered helper functions
WPI1-STUD16	Yes	I was kind of fallowing the recipe.
WPI1-STUD17	Yes	I used to start writing the functions but got stuck.
WPI1-STUD18	Yes	When I started the code it was helpful to understand what I was doing
WPI1-STUD19	Yes	I started the code without writing the signature and purpose and started to releaase i should do it with recipe

Table B.3 continued from previous page

Student	**Were there any points in the problem when the design recipe was useful?**	**If yes, describe in a sentence or two when you used it.**
WPI1-STUD20	Yes	As i mentioned before I checked on my browser the design recipe to help me in this problem
WPI1-STUD21	Yes	The check-expects got me thinking about how to approach the problem.
WPI1-STUD22	Yes	Signature Purpose Stubs Check-expects were all the parts of the recipe that I used
WPI1-STUD23	No	
WPI1-STUD24	No	
WPI1-STUD25	Yes	helper function layout

B.9 Post-programming survey question 4 responses

Table B.4: Student responses (verbatim) to post-programming survey question 4.

Student	**Was there a time when you were trying to use the design recipe but felt you didn't know what to do next?**	**If yes, describe in a sentence when that happened.**
WPI1-STUD1	No	
WPI1-STUD2	No	
WPI1-STUD3	Yes	When i was trying to figure out the diction for the small details of the problem
WPI1-STUD4	Yes	I sort of started with it, but got lost because I felt like the template didn't work very well.
WPI1-STUD5	No	
WPI1-STUD6	Yes	OI tried attacking the problem from another angle.
WPI1-STUD7	No	
WPI1-STUD8	Yes	
WPI1-STUD9	Yes	how to implement local made me thinking
WPI1-STUD10	No	
WPI1-STUD11	No	
WPI1-STUD12	No	
WPI1-STUD13	Yes	it was hard to get the function right
WPI1-STUD14	Yes	I just couldn't think of a way I could combine certain parts of a list. I should have checked the racket site, but I felt too strapped for time to look through the site. Looking back on it, it probably would have been worth it.
WPI1-STUD15	Yes	Defining the template
WPI1-STUD16	No	
WPI1-STUD17	Yes	Got stuck when I had to write the function.
WPI1-STUD18	No	
WPI1-STUD19	Yes	I need to look in my notes to see if i was doing it right
WPI1-STUD20	No	
WPI1-STUD21	Yes	I didn't know if I should use a helper function or not.
WPI1-STUD22	Yes	the template

Table B.4 continued from previous page

Student	**Was there a time when you were trying to use the design recipe but felt you didn't know what to do next?**	**If yes, describe in a sentence when that happened.**
WPI1-STUD23	Yes	I stopped using the design recipe (except for signatures and purposes)
WPI1-STUD24	No	
WPI1-STUD25	Yes	got stuck on how to get the program to work dispite trying to follow HTDP process

Appendix C

Program Design Skills Evolution Study: Instruments and Data

C.1 Interview Questions

C.1.1 Code-writing exercises and reviewing homework solutions

1. Was the problem statement clear to you when you read it? [No] What was unclear and how did you go about figuring out what the problem asked for?

2. [For your main function(s), *or* For each of your helper functions,] why did you choose these examples/test cases? (May expand on this with an example, perhaps using a different homework problem as an example) Can you describe what scenarios your examples/test cases address? Do you think you've covered all possible scenarios of the input in your examples/test cases? What other examples/test cases do you think can you add and why would you add them?

3. We are interested in your thought processes when solving this problem. 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? In what way (if any) did solutions to those problems influence your work on this problem? For example, they gave you ideas for how to structure the code, which constructs to use, *etc.*

5. Did you feel stuck at any point while working on this problem? Did any aspects of the problem stand out as more challenging than others? What were these and why were they challenging?

6. Describe the approach that [your code takes *or* each of your solutions (for multiple solutions) take] to solving the problem. [If they just read the code, re-prompt.] Give a more general description of how the code processes the input to produce the output.

C.1.2 Solution comparison or ranking multiple solutions

1. What differences do you notice between the solutions? Can you identify any strengths and weaknesses in each of the solutions?

2. Given these solutions for this programming problem, which of these do you prefer and why?

3. Given these solutions for this programming problem, can you think of another way of solving the problem? How would you do it?

4. Given these solutions for this programming problem, is there a solution that you find confusing or difficult to understand? Can you identify/describe what makes the solution confusing or difficult to understand: is it using constructs you haven't seen used before, is the general structure something you have not encountered before, or other issues that were not mentioned?

C.1.3 Class, course content, curriculum

1. Are the program design techniques taught in class helpful to you when solving a programming problem on your own? Were the program design techniques taught in class helpful to you when solving this problem? What about these techniques have been helpful/not helpful in your programming?

2. [When you don't use the program design techniques taught in class on a given problem, what approach do you take instead? *or* Did you use other program design techniques that weren't taught in class?] How is this approach helpful to you?

3. Are there particular programming problems you have been given in class that you have experienced a difficult time solving? What are these problems? For each problem, can you identify/describe what makes the problem difficult to solve for you: is it using terminology/language you can't understand, is the input/expected output expressed vaguely, can you not recall previous examples that can help you solve the problem, can you not recall constructs that can help you solve the problem, or other issues that were not mentioned?

C.1.4 Programming language

1. Are there any constructs/commands of the programming language that you find difficult or confusing to use? What are these?

2. 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, or other issues that were not mentioned?

C.2 Full Homework Problem Sets

The problems used in the sessions are in **red**. For session 1, *Problem 8* was used for the solution interview and *Problem 4* was used for comparing alternative solutions. For session 2, *Problem 6* was used for both the solution interview and for comparing alternative solutions.

C.2.1 Homework 3 problems

If you watch TV, you're probably pretty sick of watching television ads, especially now that the presidential campaigns are in full swing. The typical 30-minute TV program consists of 22 minutes of programming and 8 minutes of commercials. In this assignment, you'll create data to represent information about television ads, and write programs that process lists of ads.

1. An `Ad` consists of whether or not the type of the ad is political (as opposed to a product ad), the name of the product or politician the ad is for, the duration of the ad (in seconds), the cost to produce the ad (in thousands of dollars), whether or not the ad is to be aired nationally (as opposed to locally), the time of day that the ad is to be aired (either daytime, primetime, or off-hour), and the number of times the ad is to be aired.

   Write data definitions and provide examples of data for `Ad` and `ListOfAd`. The name of your struct should be **ad**. Make sure you define the fields in your struct in the order given above. You should use the data type `Boolean` to represent the fields for political/product and national/local.

2. Write the templates for `Ad` and `ListOfAd`.

3. Write a function `count-political-ads` that consumes a list of ads and produces the number of ads in the list that are classified as political ads.

4. Write a function `any-ads-for?` that consumes a list of ads and a String representing a product name or politician's name, and produces a Boolean. The function returns true if the list contains any ads for the given product or politician.

5. Write a function `primetime-ads` that consumes a list of ads and produces a list of all the ads airing in primetime.

6. Write a function `politicians-sponsoring-ads` that consumes a list of ads and produces a list of strings. The list that is produced contains the names of the politicians who have political ads (it's OK if the resulting list contains duplicate names).

7. Write a function `air-cost` that consumes an Ad and produces a Number. The number produced is the cost of airing the ad, which is determined as follows: airtime is sold in 30-second-spot blocks (so if the duration of the ad is 15 seconds, it would cost half the stated amount, etc.). The cost of a 30-second primetime ad for a national market is $100,000. The cost of a 30-second primetime ad for a local market is $5000. A discount of 20% is applied to the cost if the ad is

aired in the daytime, and a discount of 50% is applied if the ad is aired during off-hours; the discounts apply to both national and local ads. Finally, the air-cost is multiplied by the number of times the ad is to be aired. (You may assume that all airings of an ad occur in the same market and at the same time of day.)

8. Write a function `campaign-air-cost` that consumes a list of ads and the name of a politician, and produces the total air-cost of all political ads for that politician.

9. Write a function `total-ad-cost` that consumes an Ad and produces the total cost of the ad. The total cost is the sum of the cost of producing the ad and the cost of airing the ad.

10. Write a function `expensive-ads` that consumes a list of ads and a Number. The function produces a list of those ads for which the total ad cost exceeds the given number.

C.2.2 Homework 5 problems

A river system can be represented as a hierarchy. For example, a list of some of the tributaries that feed into the Missouri River includes the Jefferson, Sun, Yellowstone, Madison, and Gallatin Rivers. The Jefferson, in turn, is fed by the Beaverhead and Big Hole rivers. The Yellowstone is fed by the Gardner, Shields, and Boulder rivers, and so on. In this set of exercises you will create a data definition for a river and its tributaries, and write programs that answer questions about the quality of the water in the rivers.

Assume that for each river, measurements of the river's pH and DO (dissolved oxygen) levels are available. Such measurements are taken at the confluence of the rivers (the point at which the tributaries converge). pH levels can range from 0 (most acidic) to 14 (most alkaline). The normal range for bodies of water are 6.5 - 8.5. DO is measured in milligrams per liter (mg/L). DO levels are dependent on many factors, including water temperature, salinity, atmospheric pressure, aeration, and bacterial levels. Dissolved oxygen levels can range from less than 1 mg/L to more than 20 mg/L depending on how all of these factors interact.

1. Provide data definitions for a river system. For each river in the hierarchy, you should record the following information: the name of the river, the pH of the water, the DO in mg/L, and a list of the tributaries (rivers) that feed into the river. The name of your struct should be **river**. Make sure you define the fields for a river in the order given in the description.

2. Provide an example of a river system that starts with a single river and consists of at least two levels in the hierarchy below that. You may use the example given above for the Missouri River, if you wish. (You may make up numbers for pH and DO — for these exercises we're not concerned about the accuracy of the information, just that you can provide a correct model for the information.)

3. Provide the templates for your data definitions.

4. Develop a function `list-acidic-rivers` that consumes a river system and produces a list of string. The function returns a list of the names of rivers in the system that have a pH level lower than 6.5.

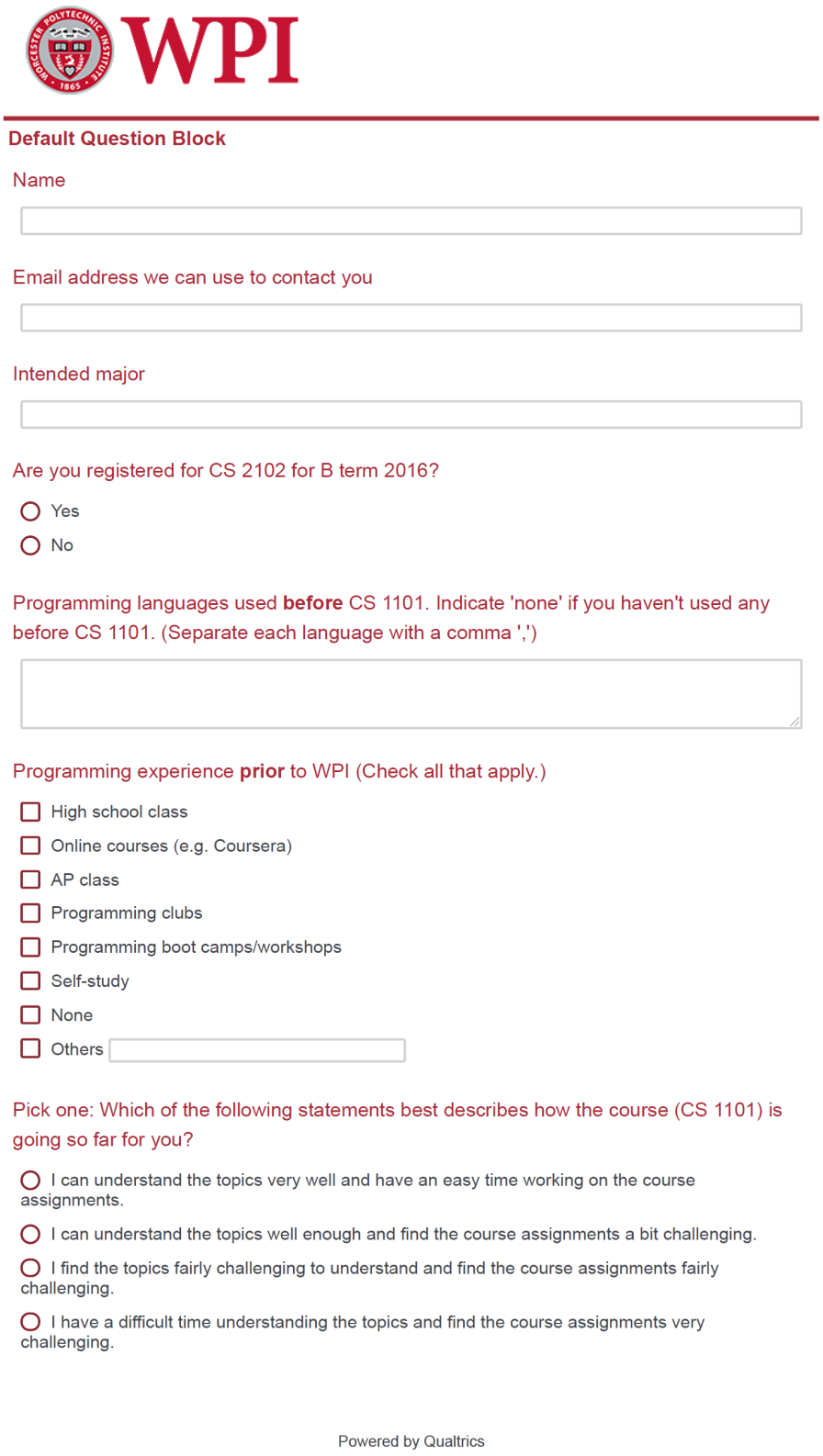
5. Acid rain can lower the pH of water in a river system. Develop a function `lower-all-ph` that consumes a river system and produces a river system. The river system that is produced is the same as the original, except that the pH of all the rivers in the system has been lowered by 0.1.

6. Cold-water fish such as trout and salmon are more vulnerable to low DO levels than are warm-water fish. Although they can survive for a short time in water with a DO level of less than 5 mg/L, they will die in water below 3 mg/L. Develop a function `cold-water-fish-warning` that consumes a river system and produces a string. If all rivers in the system have DO levels at 5mg/L or above, the string produced is "OK". If any of the rivers in the system have a DO level below 3 mg/L, the function produces the string "Deadly". Otherwise, the string produced is "Marginal". You'll need to use helpers here. Think about where you'll want to use the templates.

7. Write a function `find-subsystem` that consumes the name of a river and a river system and produces either a river system or false. The function returns the portion of the original river system that has the named river as its root. If there is no river in the system with the given name, the function returns false.

8. Use `find-subsystem` to write a function `count-acidic-tributaries-of` that consumes a river system and the name of a river. The function produces a count of the number of tributaries of the named river which have a pH < 6.5. You may assume the named river exists in the river system. You should count the acidic tributaries only, (don't include the named river itself in your count).

C.3 Study Recruitment Survey



WPI

Default Question Block

Name

Email address we can use to contact you

Intended major

Are you registered for CS 2102 for B term 2016?

- ○ Yes
- ○ No

Programming languages used **before** CS 1101. Indicate 'none' if you haven't used any before CS 1101. (Separate each language with a comma ',')

Programming experience **prior** to WPI (Check all that apply.)

- ☐ High school class
- ☐ Online courses (e.g. Coursera)
- ☐ AP class
- ☐ Programming clubs
- ☐ Programming boot camps/workshops
- ☐ Self-study
- ☐ None
- ☐ Others

Pick one: Which of the following statements best describes how the course (CS 1101) is going so far for you?

- ○ I can understand the topics very well and have an easy time working on the course assignments.
- ○ I can understand the topics well enough and find the course assignments a bit challenging.
- ○ I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- ○ I have a difficult time understanding the topics and find the course assignments very challenging.

Powered by Qualtrics

Figure C.1: The survey that volunteer students filled out to express interest in participating in the study (administered through the WPI Qualtrics [64] distribution).

C.4 Student Survey Responses

Table C.1: Student responses to the participant recruitment survey (verbatim)

Student	**Intended major**	**Programming languages used before CS 1101**	**Programming experience prior to WPI**	**How is the course going so far for you?**
WPI2-STUD1	CS	A little bit of Java	AP class, High school class, Online courses (e.g. Coursera)	D
WPI2-STUD2	Computer Science/ Game Development	Python	High school class	B
WPI2-STUD3	Computer Science	C++	Online courses (e.g. Coursera)	B
WPI2-STUD4	Computer Science	Java, C++	AP class, High school class	C
WPI2-STUD5	CS	Javascript, CSS, HTML	Programming clubs	B
WPI2-STUD6	Computer Science	Java, Python, Ruby	Self-study	B
WPI2-STUD7	Computer Science	Python, JavaScript, Java, HTML5, CSS, PHP	Self-study, AP class, High school class, Online courses (e.g. Coursera)	B
WPI2-STUD8	Computer Science	Python	None	A
WPI2-STUD9	Computer Science	None	None	C
WPI2-STUD10	Computer Science	HTML, CSS, PHP, Jquery	Self-study, Programming boot camps/workshops, High school class	B
WPI2-STUD11	CS	None	None	B
WPI2-STUD12	Computer Science	Java, C++	Self-study, High school class	B
WPI2-STUD13	Bioinformatics	Racket	High school class	A

Legend for the survey question, "*How is the course going so far for you?*" (rightmost column):

- **A**: I can understand the topics very well and have an easy time working on the course assignments.
- **B**: I can understand the topics well enough and find the course assignments a bit challenging.
- **C**: I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- **D**: I have a difficult time understanding the topics and find the course assignments very challenging.

C.5 Instructor Recruitment Survey

Validating a framework of students' program-design skills

We are inviting you to participate in our study to understand the nuances that HtDP instructors (you) consider when assessing how well students are doing at program design via HtDP.

To this end, we will ask you to rate two students' explanations of how they went about designing their solution to a programming problem. Students' explanations are captured through think-aloud transcripts – we had students think out loud while working. Each transcript is accompanied by the student's actual code submission to provide context to the transcripts. The activity will take 45-60 minutes of your time (sometime between now and mid-January).

We'll ask you to rate the students on their demonstration of 4 design skills we've identified, using a 5-point scale. We'll also ask you to justify your rating; these explanations are critical for us to understand what factors and nuances HtDP instructors use. We'll also ask about any criteria you use that our questions omitted.

1. Name *

2. Email we can use to contact you about this study *

3. Your current school/university/institution affiliation *

4. How many years have you taught/been teaching HtDP-based courses?

   Mark only one oval.

   - ◯ About a year or less
   - ◯ 2 to 3 years
   - ◯ More than 3 years

5. At what education levels have you taught HtDP-courses in? (Select as many as applicable) *

   Check all that apply.

   - ☐ grades 9-10 (students aged roughly 13-15; lower high school in US)
   - ☐ grades 11-12 (students aged roughly 16-18; upper high school in US)
   - ☐ College/University-level
   - Other: ☐ _____

6. How far do your classes go into the HtDP curriculum? *

   We're asking this to gauge your HtDP teaching experience, so if you've used HtDP in multiple course formats, answer based on the longest format in which you feel comfortable with the material.

   Mark only one oval.

   - ◯ Lists of atomic data (numbers, strings, etc)
   - ◯ Lists of structs
   - ◯ Trees
   - ◯ Mutual recursion
   - ◯ Beyond mutual recursion (generative recursion, set!, etc)
   - ◯ Other: _____

7. Did you ever attend an HtDP/TeachScheme! teacher-training workshop *

   Check all that apply.

   - ☐ Yes
   - ☐ No -- I am self-taught in the material
   - ☐ No -- I learned the material as a student
   - Other: ☐ _____

Figure C.2: The survey that volunteer HTDP instructors filled out for participation in the study.

C.6 Instructor recruitment survey responses

Table C.2: Validation study recruitment survey responses of the 7 participating HTDP instructors

Instructor	Affiliation[1]	No. of Years Teaching HTDP	Education level taught	How far (topic) do you teach HTDP?	Previously attended HTDP workshop?
INSTRUCTOR1	College/ University	>3 years	Postsecondary	Beyond mutual recursion (generative recursion, set!, etc)	No - self-taught
INSTRUCTOR2	College/ University	>3 years	Postsecondary	Beyond mutual recursion (generative recursion, set!, etc)	Learned material as a TA
INSTRUCTOR3	College/ University	>3 years	Postsecondary	Trees	Yes
INSTRUCTOR4	High school	>3 years	Grades 9-10, 11-12	Lists of structs	No - self-taught
INSTRUCTOR5	College/ University	>3 years	Grades 11-12, Tertiary	Beyond mutual recursion (generative recursion, set!, etc)	Yes
INSTRUCTOR6	Grades 6-12	>3 years	Grades 9-10, 11-12	Beyond mutual recursion (generative recursion, set!, etc)	Yes
INSTRUCTOR7	College/ University	>3 years	Postsecondary	Trees	Yes

[1] Actual institution names redacted for anonymity

C.7 Instructor worksheet

C.7.1 Page 1: Instructions and skill descriptions

Instructions

You are given 2 student transcripts. Each transcript has an accompanying code output and problem statement for reference. The transcripts detail student responses during think-aloud sessions about their program-design activities, knowledge, and processes. More specifically, the think-aloud transcripts contain students' verbalizations while they are developing a solution to the Rainfall problem.

On the next page, we've provided a sheet through which we'd like you to rate students' demonstration of four core design skills, using the scale below the table. Base your ratings on both of the provided artifacts (transcript and code output). For each of the scores you gave, explain what made you pick that number (as opposed to one higher or lower); we want to understand what nuances you think about or consider when rating students on each criterion:

Program-design skills	Description
Methodical choice of tests and examples	Ability to write tests/examples that capture details about the problem. Understanding of the individual/collective purposes of the tests/examples (e.g. the kinds of tests/examples written, reasoning around the selection of tests/examples, or justification of why tests/examples are interesting cases).
Composing expressions within function bodies	Ability to correctly compose expressions, whether built-in or user defined (e.g. correctly composing built-in functions or user-defined functions) to build function bodies. Understanding of code syntax and the underlying semantics (e.g. relationships between return values and calling contexts).
Decomposing tasks and composing solutions	Ability to identify tasks in a given problem, decompose a program into relevant tasks, and compose task-solutions. For example, associating tasks with specific variables or functions in the solution, and combining those elements appropriately to solve the intended problem. Understanding the logical relationship among identified tasks in the context of the solution.
Leveraging multiple artifacts/representations for functions	Use of the various artifacts for functions (e.g. contract/signature, purpose, examples, test cases, code), through the components of the HtDP design recipe. Articulation of how the different artifacts interact and relate to each other.

Rating scale:

- 0: None
- 1: Little to none
- 2: Fragmented, but present
- 3: Strong
- 4: Applies beyond current problem (e.g. Talks about a plausible application of the skill outside the context of the current problem)

Note:

As you're reading through the transcripts, please note sections, sentences, or phrases in the transcripts that you think support or witness to your explanations of your ratings. You can do this by either:

(a) highlighting and making margin notes on the transcripts or
(b) writing the paragraph number(s) alongside your explanations in the worksheet.

Please use a **blue/black pen** when filling in the worksheets and marking the transcripts. This is so text will come through when scanning the worksheets and transcripts for submission to the researchers. You have two options for submitting the completed worksheets and marked transcripts:

1. You may **scan** the worksheets and marked transcripts (most preferably into PDF) and send them to **kfisler@cs.brown.edu**.
2. You may send the worksheets and marked transcripts by **postal mail** to:
   Kathi Fisler
   Computer Science Department
   Box 1910, Brown University
   Providence, RI 02912-1910
   USA

Figure C.3: Page 1 of the instructor worksheet. This page provides instructions on how to complete the worksheet and the descriptions of the skills to be scored.

C.7.2 Page 2: Skill ratings and descriptions

Scoring

Scorer name:

Student number: **Session/Transcript Number:**

1. If you were to assign your own score from 0 (lowest) to 4 (highest) to rate the quality of student's work and solution based on their code and transcript, what score would you give and why? (This helps us calibrate our rating metrics with what you would use independently as an HtDP instructor)

2. Write your ratings of the student's demonstration of each skill and the explanations of your ratings in the table below.

Skills	Rating	Explain your rating
Methodical choice of tests and examples		
Composing expressions within function bodies		
Decomposing tasks and composing solutions		
Leveraging multiple representations of functions		

3. If your grading would have considered skills/factors that weren't covered in our four design skills, briefly describe them here.

Figure C.4: Page 2 of the instructor worksheet. Instructors fill this page in with their ratings of a student's design skills and their justification for their ratings.

C.8 Instructor worksheet responses

Design skill shorthands are: **MTE** = *Methodical choice of tests and examples*, **CFB** = *Composing expressions within function bodies*, **DTC** = *Decomposing tasks and composing solutions*, and **LRF** = *Leveraging multiple representations of functions*

C.8.1 INSTRUCTOR1

Worksheet Question	WPI2-STUD3	WPI2-STUD6
Overall quality of student's work	The student didn't create any tests in advance. They seem to know how to use the template (though they refer to in-class examples, rather than the HtDP terminology). They seem to have some ability to reason why the program doesn't give the right answer, but they do not explore this with more examples. They eventually realized they needed a help function 'average', but ran out of time before exploring this	Good work in generating examples and subtasks. But I can't give them a 3, since they didn't solve the problem :). This student seemed to misunderstand the problem, since they posited wrong values as the desired answers for some of their tests.
MTE	Tries an example with two positive numbers first, but doesn't try a single positive number or an empty list. I'd rate this skill as present, but not strong enough even to count as 'fragmented'. If you'd said 'rudimentary' as the description of a 2, I might have gone for that.	Good choice of examples. But did not write them down as test cases. Did not have a correct understanding of the problem See paragraph (1): "because 8 plus 5 is 13 plus 1 is 14, -2 is 12 and divide that by 4 is 3 okay."
CFB	Composes expressions plausibly. Shows reasoning about relating observed behavior to the program text	The code they wrote mostly mirrors their intentions.
DTC	Eventually got to 'average', but ran out of time	Discovered the need for 'average' early on, but gave up on it too soon.
LRF	Wrote contract and purpose for main function, but not examples or test cases.	Has contract and purpose statement for main function, but no examples or test cases. No contract or purpose statement for local function.
Other skills/ factors you would have considered	I would have required a contract and purpose statement for the inner function	I would have required contract and purpose statement for the local function.

C.8.2 INSTRUCTOR2

Worksheet Question	WPI2-STUD3	WPI2-STUD11
Overall quality of student's work	1 - But this is giving no credit for the insight at the end (par. 17) of the transcript, otherwise it could be a 2. The student is doing some of what they should be doing (signature, purpose, eventually starting a helper function) and shows some knowledge (how to traverse a list), but doesn't yet have the skeleton of a solution written down, and isn't applying all the tools, in particular tests and examples, that they should be.	3 - This is good, but there are (at least) two function defects: 1) it keeps going past -999 because the negative check (line 27) comes before the -999 check. 2) It crashes if the input list contains 0, because that case is not handled by the cond (line 26-28). Given a finer scale, I would also deduct for incomplete test coverage (line 28 is unreachable) and bad style (unnecessary accumulator, too many cases in the main function).
MTE	1 - No examples in the code, but tries one interactively (par. 8), and then reduces it (par. 9).	2 - Tests cover a few edges cases, but omit important cases, such as -999 followed by more numbers, or a list that mixes negatives and non- negatives. It's clear from the transcript (par. 29-30) that testing was an afterthought.
CFB	3 - Doesn't seem to have any trouble with this. Though some of the reasoning in par. 6 might show a lack of understanding of accumulators? And that may have led to passing (first alon) as the accumulator (line 10).	3 - Doesn't seem to have any trouble with this. (I'm having trouble imagining why a student would do what is required to score a 4, though.) One low-level issue is not understanding the difference between 'list' and 'cons' (par. 26).
DTC	2 - Big problem here, and on the code alone, this would rate a 1, not 2. But the transcript shows the student figuring out the decomposition (par. 17) before running out of time.	3 - The student right away goes to a flawed recursive strategy (par. 7), realizes quickly it won't work, and then applies a not-very-good strategy (accumulator, par. 10). But somehow (maybe par. 13?) they figure out that the list-traversing function has to be a helper, which leads to a workable decomposition.
LRF	1 - Writes a function signature (line 1, par. 2), but otherwise doesn't do this.	2 - Wrote signatures and purposes for helpers, but the purpose for 'check-number' (line 22) got out of date (par. 17). Didn't actually leverage examples/tests when designing the code.
Other skills/ factors you would have considered	My grading is more about applying process than about the kinds of design skills in your rubric; my goal is to induce students to follow the process even when it feels mismatched with their understanding (because they feel they don't need it, or because the problem feels too forbidding). So I give points for: meaningful function name, (correct) signature, (correct, succinct) purpose statement, example adequacy (this problem requires at least 4), test coverage, an explicit strategy statement, and correctly following a template (when that is the strategy). I also grade for style, e.g., failure to factor out redundancy, magic numbers, inappropriate choices of data types, etc. In this case, the local helper 'rainfall' (line 7) needs a better name and an accumulator statement (loop invariant). Perhaps thinking about an accumulator statement would have helped the student understand that their approach is sooner.	My grading is more about applying process than about the kinds of design skills in your rubric; my goal is to induce students to follow the process even when it feels mismatched with their understanding (because they feel they don't need it, or because the problem feels too forbidding). So I give points for: meaningful function name, (correct) signature, (correct, succinct) purpose statement, example adequacy (this problem requires at least 4), test coverage, an explicit strategy statement, and correctly following a template (when that is the strategy). I also grade for style, e.g., failure to factor out redundancy, magic numbers, inappropriate choices of data types, etc. In this case, I would just want the helpers to have better names.

C.8.3 INSTRUCTOR3

Worksheet Question	WPI2-STUD3	WPI2-STUD11
Overall quality of student's work	I would give this student a score of 1.25 out of 4. I think my scores are probably drawn from the context of the fourth-year college PL class that I'm taking, where some students are having almost exactly this kind of difficulty. A score of 1 is essentially a mercy score, suggesting that the student clearly worked hard, but that the ingredients of success are mostly missing.	I would give this student a 3.5. The student fails to handle -999 correctly; specifically, positive numbers after the -999 are not ignored, because of the ordering of the 'cond' clauses. Code coverage testing reveals this bug (and indeed, I would probably not have spotted it if I hadn't quite accidentally had test coverage enabled). Other than this, the final code looks very solid.
MTE	1 - The student failed to write down any test cases at all. They did try the program on a few inputs, but were missing test cases that were *simple* enough to aid in the thinking process. They were also completely missing the complex test cases.	2 - The student produces reasonable test cases, but the think-aloud makes it clear that these were an afterthought , rather than a part of the design process.
CFB	3 - The student doesn't appear to have trouble composing expressions; the code is syntactically solid, and code such as (/ (rainfall (rest alon) (first alon)) (+ 1 acc)) is doing exactly what the student wants, even if it isn't actually helpful in this case.	3 - The student seems to have no problem composing expressions within function bodies. There were some syntactic issues, but the student largely overcame these, though the use of 'list*' is a little unsettling; a 'cons' would have been the simple and correct choice here.
DTC	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 hammer to hold, and whether it can do the whole job. Until you know the patterns well, you don't know their limitations.	2 - The student did not immediately decompose the problem into its parts, but got there eventually. It was fascinating to see how the student started out with a helper function with a "random name", and moved from a function that adds numbers to one that just returns a list of the non-negative ones. This suggests that the mental cost of initiating a helper function makes it easier to re-purpose one than just to come up with another one.
LRF	2 - The purpose statement looks solid, and the signature is there, and clearly the first informs the second and the second informs the choice of template, so that parts of the design recipe are working. However, things fall apart in the decision not to write test cases as a means of thinking about the problem.	2 - This student uses the signature to drive the template, but fails to develop test cases as a means of understanding the problem.
Other skills/ factors you would have considered		

C.8.4 INSTRUCTOR4

Worksheet Question	WPI2-STUD6	WPI2-STUD11
Overall quality of student's work	2 - One huge untestable local function – better to have access to inputs without local Stuck on 1 example	3 - It works, has check-expects of reasonable coverage, signature [illegible], purpose – one wrong Should test helpers
MTE	2 - Sticks with same long example throughout Never goes smaller and introduces features one by one	3 - See check-expects – small and build up
CFB	3 - Rainfall helper uses many helper functions	3 - Uses helper functions
DTC	2 - Has accumulator, little evidence of decomposition beyond that	3
LRF	1 - Connections seem very superficial Did not affect results in my opinion	2 - Note purpose to check-number is wrong and name choice is poor
Other skills/ factors you would have considered	Poor use of test cases is striking, #1 issue	Testing thoroughness – should have test with two -999's to demonstrate they work, or at least data after -999: (list 12 -999 3 4 -999)

C.8.5 INSTRUCTOR5

Worksheet Question	WPI2-STUD6	WPI2-STUD11
Overall quality of student's work	0 - Justification: The student has failed to follow the design recipe in several ways. First, the student did not write any tests in their program. Furthermore, in the transcript the student failed to carefully choose tests. For example, the student never considered what the result ought to be if the list (i.e., alon0) is empty. Second, the student chose to use an accumulator, but never explicitly stated the role/invariant of the accumulator nor explicitly stated in the transcript how the accumulator is to be exploited. Third, the student did not spend enough time analyzing the problem in order to understand how to properly solve it and to discover interesting tests. Fourth, the student fails to truly grasp that if -999 is encountered then what is left of the list need not be processed. Instead of stopping the processing of the list at -999, the student continues to use it to compute the answer. Finally, the student is using programming constructs that are not needed. Specifically, the student used cond to in all-negative? despite the fact that no decision needs to be made. Clearly, the student does not clearly understand that cond ought to used only when a decision is to be made.	3.7 - Overall, the student has successfully used divide-and-conquer to find relevant subproblems/tasks that need to be solved. The student has then successfully put the pieces together to formulate an answer to the rainfall problem. The solution could be improved, but I consider that an optimization and, therefore, I am more lenient in grading the lack of optimizations.
MTE	0 - Despite thinking out loud about a couple of examples, the student never translated those thoughts to actual tests in the code developed.	3.2 - The student failed to think of examples/tests that: 1. Contained faulty and non-faulty readings before -999 2. Contained readings after -999 3. Test the auxiliary functions: check-number and add-number Otherwise, the student used examples to guide the development as seen in paragraphs 24, 27, and 29 of the transcript.
CFB	1 - The student does have a program free of syntax errors and does demonstrate some knowledge of function composition. For example, they call the length function to compute the returned answer. They also demonstrate some understanding about how to process a list when they recursively process the rest of the list. This knowledge, however, is incomplete given that they formulate no answer for when the list is empty.	4 - The student has clearly leveraged problem analysis and understanding of syntax to produce a program that does not have syntactical mistakes and that is faithful to their design decomposition of tasks. The output of functions that solve a subproblem are effectively used in the rainfall function as input to other functions such as /, cons?, and add-number.
DTC	0 - The student did not formulate the tasks that need to be accomplished, neither in their code nor in their transcript. The student focuses on a single task: the sum of nonnegative numbers before -999. This task, however, is poorly done as the student did not carefully think about what is needed to compute the average of these numbers.	4 - The student has correctly composed functions to solve subproblems. Three useful subproblems/tasks are identified: summing a list of numbers, shortening the input list to rainfall, and getting the length of a list of numbers. The functions to solve these subproblems are effectively used in the rainfall function.
LRF	0 - See my answer to question 1.	3.5 - The student failed to properly analyze the quality of he submitted solution. Specifically, (check-number alon) is computed three times instead of using a local expression to define a variable to compute this value once and use the variable three times. There is also the curious choice of using list* instead of using cons. The transcript did not shed any light on the rationale behind this choice other than some misguided notion that adding a "*" has fixed things in the past. Being unable to explain this choice indicates that the student cannot fully explain how or why their solution works.
Other skills/ factors you would have considered	None.	None.

C.8.6 INSTRUCTOR6

Worksheet Question	WPI2-STUD6	WPI2-STUD11
Overall quality of student's work	1-2 - The code is incorrect, inadequately tested, and not structured well. However, the student has written signatures and purpose statements, and the definitions are well-formed. I'd call it a 2 (= C or C-) for a HS student, maybe 1 for college?	2.5 - Definitely at least 2 and at most 3. Good decomposition, function form follows data, and the main function is correct for many inputs. My reservations about a 3 are the incorrectness on lists with nums after -999, the testing gaps that allowed it, and the purpose statement and name in check-number.
MTE	1 - Student not using check-expect to automate tests rules out 3 or 4. Student knows about testing and uses it to observe program behavior, but isn't systematic.* *See, e.g., pars. 6, 8, and my comments on par. 11 of transcript.	2 - Student includes tests on the main function, but not helpers. Tests address some, but not all, cases.
CFB	2 - Student is making well-formed but incorrect body expressions. Student makes use of APS without a clear reason and loses sight of expressions' purpose.	2 (2.5?) - Mostly good skills here; higher levels of nesting may still be too much (see "check-number") but the student has 2 clear, sensible functions.
DTC	1 - Student misses the "obvious" decomposition (average = sum/length of list), but does recognize filtering as a sub-task (even if not implemented right).	3 - Clearly recognized the main task divisions in the problem, and the resulting program structure makes sense.
LRF	2 - Little use of examples/tests – especially for understanding the problem. Accurate use of signatures and purpose statements, but not fully realized.	2.5 or 3 - Student knows the design-recipe steps but doesn't get the idea that concrete examples are a tool for understanding the problem before coding.
Other skills/ factors you would have considered	See my notes on the source and transcript; that should mostly cover this Q. I'm also interested in the student's communication and organization skills, and try to distinguish (and separately address) the student's awareness and understanding of the design tools, and disposition towards using these.	Same comment as I made re: student 6: communication skills. Also, proofreading and revision, this student could have eliminated many of the remaining problems in the code by making a proofreading pass, and checking whether names, writing, and code all make sense.

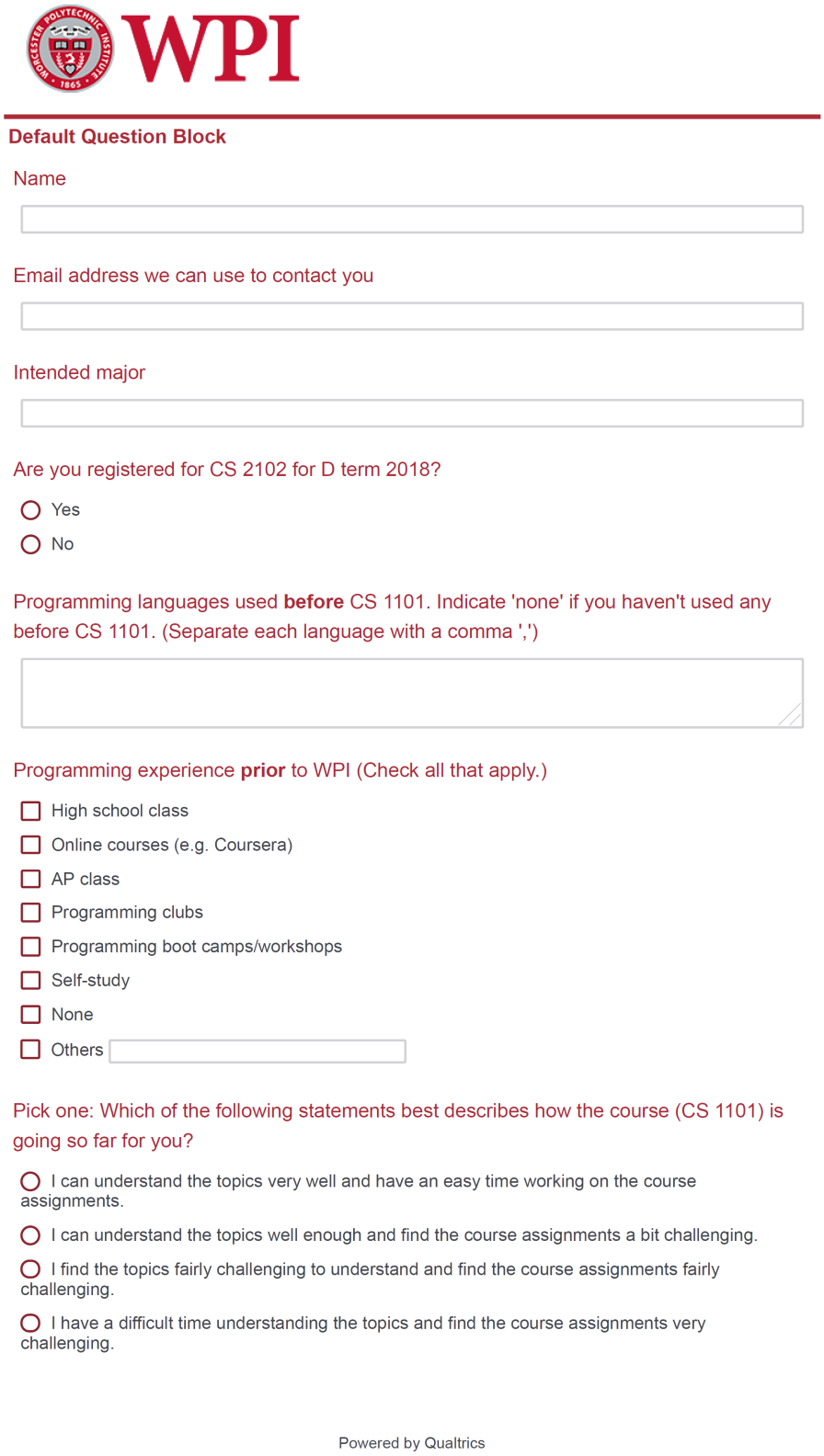
C.8.7 INSTRUCTOR7

Worksheet Question	WPI2-STUD3	WPI2-STUD6
Overall quality of student's work	3	3+
MTE	2 - Not enough examples when approaching the problem. Interview parts 8, 9, 15. Examples and tests used later in reasoning.	3-4 - Student uses examples and tests to analyze the problems. (Interview 3, 5, 6, 8, 11)
CFB	2-3 - Good composition in terms of syntax.	3 - Student should try more ways of decomposing the problem. Syntax is good.
DTC	3 - Rainfall function includes the rainfall with accumulator. Average function is defined as a separate function, but it is not used.	3 - All-negative function is placed after the testing for -999. Decomposing is partially used.
LRF	3 - Average function has a wrong signature/contract, but the explanation in interview was good attempt.	3-4 - Good usage of purpose/ signatures. Have lots of examples.
Other skills/ factors you would have considered		

Appendix D

Study on Task- and Code-level Thinking: Instruments and Data

D.1 WPI Study Recruitment Survey



WPI

Default Question Block

Name

Email address we can use to contact you

Intended major

Are you registered for CS 2102 for D term 2018?

- ○ Yes
- ○ No

Programming languages used **before** CS 1101. Indicate 'none' if you haven't used any before CS 1101. (Separate each language with a comma ',')

Programming experience **prior** to WPI (Check all that apply.)

- ☐ High school class
- ☐ Online courses (e.g. Coursera)
- ☐ AP class
- ☐ Programming clubs
- ☐ Programming boot camps/workshops
- ☐ Self-study
- ☐ None
- ☐ Others

Pick one: Which of the following statements best describes how the course (CS 1101) is going so far for you?

- ○ I can understand the topics very well and have an easy time working on the course assignments.
- ○ I can understand the topics well enough and find the course assignments a bit challenging.
- ○ I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- ○ I have a difficult time understanding the topics and find the course assignments very challenging.

Powered by Qualtrics

Figure D.1: The survey that volunteer WPI students filled out to express interest in participating in study 5 (administered through the WPI Qualtrics [64] distribution).

D.2 WPI Student Survey Responses

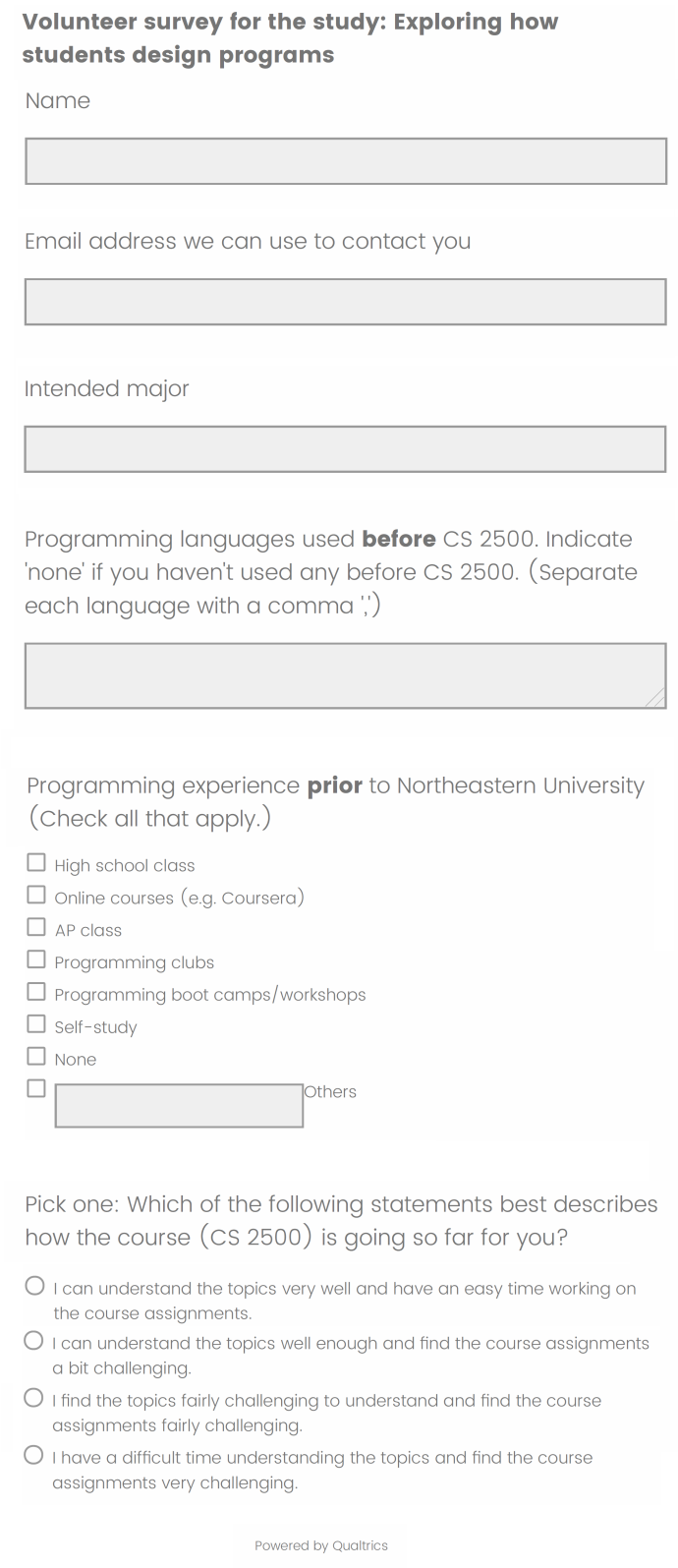
Table D.1: WPI student responses to the participant recruitment survey (verbatim)

Student	**Intended major**	**Programming languages used before CS 1101**	**Programming experience prior to WPI**	**How is the course going so far for you?**
WPI3-STUD1	Robotic engineering	Robot c, c++	High school class, Online courses (e.g. Coursera), AP class, Programming clubs, Self-study	A
WPI3-STUD2	BME	None	None	B
WPI3-STUD3	RBE	none	None	B
WPI3-STUD4	Possible cs and imgd double major	Java	Online courses (e.g. Coursera), Self-study	A
WPI3-STUD5	Robotics Engineering	Arduino, C, Java	High school class, Self-study	A
WPI3-STUD6	Mechanical Engineering and Robotics Engineering	C-based programming (Robot C)	High school class	B
WPI3-STUD7	CS	None	None	D
WPI3-STUD8	RBE	none	None	D
WPI3-STUD9	Computer Science	C, & C++, Java, Python	None	A
WPI3-STUD10	robotics	c++, javaswift	High school class, Online courses (e.g. Coursera), AP class, Self-study	B
WPI3-STUD11	Robotic Engineering	Arduino, Robot c, python, java	High school class, Programming clubs	B
WPI3-STUD12	Mechanical Engineering	Java, Python, Pascal	AP class, Programming boot camps/workshops	A

Legend for the survey question, "*How is the course going so far for you?*" (rightmost column):

- **A**: I can understand the topics very well and have an easy time working on the course assignments.
- **B**: I can understand the topics well enough and find the course assignments a bit challenging.
- **C**: I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- **D**: I have a difficult time understanding the topics and find the course assignments very challenging.

D.3 NEU Study Recruitment Survey



Volunteer survey for the study: Exploring how students design programs

Name

Email address we can use to contact you

Intended major

Programming languages used **before** CS 2500. Indicate 'none' if you haven't used any before CS 2500. (Separate each language with a comma ',')

Programming experience **prior** to Northeastern University (Check all that apply.)

- ☐ High school class
- ☐ Online courses (e.g. Coursera)
- ☐ AP class
- ☐ Programming clubs
- ☐ Programming boot camps/workshops
- ☐ Self-study
- ☐ None
- ☐ Others

Pick one: Which of the following statements best describes how the course (CS 2500) is going so far for you?

- ○ I can understand the topics very well and have an easy time working on the course assignments.
- ○ I can understand the topics well enough and find the course assignments a bit challenging.
- ○ I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- ○ I have a difficult time understanding the topics and find the course assignments very challenging.

Powered by Qualtrics

Figure D.2: The survey that volunteer NEU students filled out to express interest in participating in study 5 (administered through the WPI Qualtrics [64] distribution).

D.4 NEU Student Survey Responses

Table D.2: NEU student responses to the participant recruitment survey (verbatim)

Student	**Intended major**	**Programming languages used before CS 2500**	**Programming experience prior to NEU**	**How is the course going so far for you?**
NEU1-STUD1	Computer Science	Java, C++, C#, Python	Self-study, Classes at prior community college[1]	A
NEU1-STUD2	Computer Engineering & Computer Science	Java, G-code, Python, C++, Matlab, SQL, Visual Basic	High school class, Online courses (e.g. Coursera), Self-study, internship	A
NEU1-STUD3	Computer Science and Design	none	None	A
NEU1-STUD4	Mathematics and Physics	Python (only very basic)	None	A
NEU1-STUD5	Computer Science and Math	Python	Programming boot camps/workshop, Self-study	B
NEU1-STUD6	Computer Science/ Design	java (not very skilled)	High school class	B
NEU1-STUD7	Data Science	none	None	B
NEU1-STUD8	Business admin and political science combined	HTML	Self-study	B
NEU1-STUD9	Computer Science and Music Technology	none	None	C
NEU1-STUD10	Chemistry	Beginner C++, Basic Unix, and Basic Matlab	Online courses (e.g. Coursera), Self-study	C

Legend for the survey question, "*How is the course going so far for you?*" (rightmost column):

- **A**: I can understand the topics very well and have an easy time working on the course assignments.
- **B**: I can understand the topics well enough and find the course assignments a bit challenging.
- **C**: I find the topics fairly challenging to understand and find the course assignments fairly challenging.
- **D**: I have a difficult time understanding the topics and find the course assignments very challenging.

[1] Actual school name redacted for anonymity.

Appendix E

Exploratory Study on Planning: Instruments and Data

E.1 CRS-BROWNU Pre-Assessment Problems

E.1.1 Programming Problems

Palindrome Detection Modulo Spaces and Capitalization

A palindrome is a string with the same letters in each of forward and reverse order (ignoring capitalization). Design a program called `is-palindrome` that consumes a string and produces a boolean indicating whether the string with all spaces and punctuation removed is a palindrome. Treat all non-alphanumeric characters (*i.e.* ones that are not digits or letters) as punctuation.

Examples:

`is-palindrome("a man, a plan, a canal: Panama")` is true

`is-palindrome("abca")` is false

`is-palindrome("yes, he did it")` is false

Sum Over Table

Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths. Design a program `sum-largest` that consumes a table of numbers and produces the sum of the largest item from each row. Assume that no row is empty.

Example:

`sum-largest([list: [list: 1, 7, 5, 3], [list: 20], [list: 6, 9]])` is `(7 + 20 + 9)`

Adding Machine

Design a program called `adding-machine` that consumes a list of numbers and produces a list of the sums of each nonempty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Example:

`adding-machine([list: 1, 2, 0, 7, 0, 5, 4, 1, 0, 0, 6])` is `[list: 3, 7, 10]`

E.1.2 Ranking Problems

Below you are given problem statements followed by multiple solutions to each problem. Assume that the solutions are correct; ignore any small deviations in behavior. Also assume that any missing helper functions are defined in the obvious way. Finally, ignore stylistic differences in naming. Instead, focus on the structure of the solutions.

For each problem, rank the solutions in order (from most to least) of your preference. You are allowed to have ties. Below the ranking grid, explain why you picked that ordering, mentioning briefly all solutions in your response.

The solutions are labeled A, B, etc. Indicate your ordering by selecting the appropriate radio button for each solution. For instance, selecting "1st" for B, "2nd" for A and C, and "3rd" for D means you liked B the most, followed by A and C (tied), followed by D.

Remember to explain your choice!

Rainfall

Design a program called rainfall that consumes a list of real numbers representing daily rainfall readings. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the nonnegative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). Assume that there is at least one nonnegative number before -999.

Example:

`rainfall([list: 1, -2, 5, -999, 8])` is 3

Solution A:

```
fun rainfall(l :: List<Number>) -> Number:
   fun helper(rds :: List<Number>, total :: Number, days :: Number) -> Number:
      cases (List) rds:
         | empty => total / days
         | link(f, r) =>
            ask:
               | f == -999 then: total / days
               | f < 0 then: helper(r, total, days)
               | otherwise: helper(r, total + f, days + 1)
            end
      end
   end
   helper(l, 0, 0)
end
```

Solution B:

```
fun rainfall(l :: List<Number>) -> Number:
  fun sum-rfs(shadow l :: List<Number>) -> Number:
    cases (List) l:
      | empty => 0
      | link(f, r) =>
        ask:
          | f == -999 then: 0
          | f < 0 then: sum-rfs(r)
          | otherwise: f + sum-rfs(r)
        end
    end
  end
  fun count-days(shadow l :: List<Number>) -> Number:
    cases (List) l:
      | empty => 0
      | link(f, r) =>
        ask:
          | f == -999 then: 0
          | f < 0 then: count-days(r)
          | otherwise: 1 + count-days(r)
        end
    end
  end
  total = sum-rfs(l)
  days = count-days(l)
  total / days
end
```

Solution C:

```
fun rainfall(l :: List<Number>) -> Number:
  fun cleanse(shadow l :: List<Number>) -> List<Number>:
    cases (List) l:
      | empty => empty
      | link(f, r) =>
        ask:
          | f == -999 then: empty
          | f < 0 then: cleanse(r)
          | otherwise:
            link(f, cleanse(r))
        end
    end
  end
  actual = cleanse(l)
  total = sum-of(actual)
  days = actual.length()
  total / days
end
```

Length of Triples

Design a program called `max-triple-length` that consumes a list of strings and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings. Also assume we are given:

```
data Triple: triple(a, b, c) end
```

Example:

`max-triple-length([list: "a", "bb", "c", "dd"])` is 5

Solution A:

```
fun max-triple-length(l :: List<String>) -> Number:
   fun break-into-triples(shadow l :: List<String>) -> List<Triple>:
      link(triple(l.first, l.rest.first, l.rest.rest.first),
         ask:
            | is-empty(l.rest.rest.rest) then: empty
            | otherwise: break-into-triples(l.rest)
         end)
   end
   triple-lengths = map(lam(t): string-length(t.a) + string-length(t.b) + string-length(t.c) end,
      break-into-triples(l))
   max-of(triple-lengths)
end
```

Solution B:

```
fun max-triple-length(l :: List<String>) -> Number:
   fun break-into-triples(shadow l :: List<Number>) -> List<Triple>:
      link(triple(l.first, l.rest.first, l.rest.rest.first),
         ask:
            | is-empty(l.rest.rest.rest) then: empty
            | otherwise: break-into-triples(l.rest)
         end)
   end
   triples-as-nums = map(string-length, l)
   triple-lengths = map(lam(t): t.a + t.b + t.c end, break-into-triples(triples-as-nums))
   max-of(triple-lengths)
end
```

Solution C:

```
fun max-triple-length(l :: List<String>) -> Number:
   shadow l = map(string-length, l)
   fun helper(shadow l :: List<Number>, max-so-far :: Number, prev-2 :: Number,
      prev-1 :: Number) -> Number:
         cases (List) l:
            | empty => max-so-far
            | link(f, r) =>
               prev-3 = prev-2 + f
               helper(r,
                  if prev-3 > max-so-far: prev-3 else: max-so-far end,
                  prev-1 + f,
                  f)
         end
   end
   helper(l.rest.rest.rest,
      l.first + l.rest.first + l.rest.rest.first,
      l.rest.first + l.rest.rest.first,
      l.rest.rest.first)
end
```

Shopping Discount

An online clothing store applies discounts during checkout. A shopping cart is a list of the items being purchased. Each item has a name (a string like “shoes”) and a price (a real number like 12.50). Design a program called checkout that consumes a shopping cart and produces the total cost of the cart after applying the following two discounts:

- if the cart contains at least 100 worth of shoes, take 20% off the cost of all shoes (match only items whose exact name is "shoes")
- if the cart contains at least two hats, take 10 off the total of the cart (match only items whose exact name is "hat")

Assume the cart is represented as follows:

```
data CartItem: ci(name :: String, cost :: Number) end

type Cart = List<CartItem>
```

Example:

`checkout([list: ci("shoes", 25), ci("bag", 50), ci("shoes", 85), ci("hat", 15)])` is 153

Solution A:

```
fun checkout(cart :: Cart) -> Number:
   shoes = filter(lam(c): c.name == "shoes" end, cart)
   shoe-cost = sum-of(map(lam(c): c.cost end, shoes))
   shoe-discount = if shoe-cost >= 100: shoe-cost * 0.20 else: 0 end

   hats = filter(lam(c): c.name == "hat" end, cart)
   hat-count = hats.length()
   hat-discount = if hat-count >= 2: 10 else: 0 end

   init-cost = sum-of(map(lam(c): c.cost end, cart))
   init-cost - shoe-discount - hat-discount
end
```

Solution B:

```
fun checkout(cart :: Cart) -> Number:
   fun helper(ct :: Cart, total :: Number, shoe-cost :: Number, hatcount :: Number) -> Number:
      cases (List) ct:
         | empty =>
            shoe-discount = if shoe-cost >= 100: shoe-cost * 0.20 else: 0 end
            hat-discount = if hat-count >= 2: 10 else: 0 end
            total - shoe-discount - hat-discount
         | link(f, r) =>
            new-total = total + f.cost
            ask:
               | f.name == "shoes" then: helper(r, new-total, shoe-cost + f.cost, hat-count)
               | f.name == "hat" then: helper(r, new-total, shoe-cost, hat-count + 1)
               | otherwise: helper(r, new-total, shoe-cost, hat-count)
            end
      end
   end
   helper(cart, 0, 0, 0)
end
```

E.2 CRS-WPI Pre-Assessment Problems

Programming with Lists

Up until now, all of the programs you have had to write needed a straightforward traversal of the input data. Often, we need to write programs that combine multiple tasks on the same data. Then we have to think about how to organize the code. This organizational task is called *planning*.

For this part of the assignment, we're asking you to write two programs that involve multiple tasks. We will use these to set up upcoming lectures, so the goal is for you to think about how to do this, as much as to produce code. These will be graded on the correctness of answers that they produce.

Create a class called `Planning` and put both of the following methods in that class.

- Write a program called `rainfall` that consumes a `LinkedList<Double>` representing daily rainfall readings (double is the type for real numbers in Java). The list may contain the number -999 indicating the end of the data of interest. Produce the average of the nonnegative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list (representing faulty readings). If you cannot compute the average for whatever reason, return 1.

  For example, given a list containing `(1, -2, 5, -999, 8)`, the program would return 3.

- Write a program called `maxTripleLength` that consumes a `LinkedList<String>` and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three strings.

  For example, given a list containing `("a", "bb", "c", "dd")`, the program would return 5 (for "bb", "c", "dd").

  You don't have to actually concatenate the strings to solve this, but if you want to, you can do this with `+`, as follows `"go " + "goats"`

Also provide an `Examples` class with up to four test cases for each of these two methods. We will not run either of these for thoroughness against broken implementations, but we are interested in seeing what cases you would choose to check within four test cases.

E.3 CRS-BROWNU and CRS-WPI Post-Assessment Problems

> *Note: The post-assessment problems for both* CRS-BROWNU *and* CRS-WPI *are similar; the problems were adapted to the programming language used in each course. Here we show the problems in Pyret, the language used in* CRS-BROWNU*; a similar version in Java was used with* CRS-WPI.

For each problem, write two solutions, where each solution solves the problem using a different approach. You should also determine which solution structure you prefer. Specify your preference with a brief discussion of why.

Approaches count as different if they cluster at least some subtasks of the problems differently (like we saw for the Rainfall solutions); merely syntactic differences, such as replacing an element-based for-loop with an index-based one, don't count as different. It has to be a different decomposition of the tasks (*i.e.* compositions of different plans).

In the end, if after racking your brain you simply can't think of two truly different ways of doing one of these problems, submit the two most different versions you can.

Programming Problems

A personal health record (PHR) contains four pieces of information on a patient: their name, height (in meters), weight (in kilograms), and last recorded heart rate (as beats-per-minute). A doctor's office maintains a list of the personal health records of all its patients.

```
data PHR:
   | phr(name :: String,
         height :: Number,
         weight :: Number,
         heart-rate :: Number)
end
```

The BMI Sorter

Body mass index (BMI) is a measure that attempts to quantify an individual's tissue mass. It is commonly collected during annual checkups or clinic visits. It is defined as:

```
BMI = weight / ( height * height )
```

A simplified BMI scale classifies a value below 18.5 as "underweight", a value at least 18.5 but under 25 as "healthy", a value at least 25 but under 30 as "overweight", and a value at least 30 as "obese".

Design a function called `bmi-report`—

```
fun bmi-report(phrs :: List<PHR>) -> Report
```

—that consumes a list of personal health records (defined above) and produces a report containing a list of names (not the entire records) of patients in each BMI classification category. The names can be in any order. Use the following datatype for the report:

```
data Report:
   | bmi-summary(under :: List<String>,
                 healthy :: List<String>,
                 over :: List<String>,
                 obese :: List<String>)
end
```

Data Smoothing

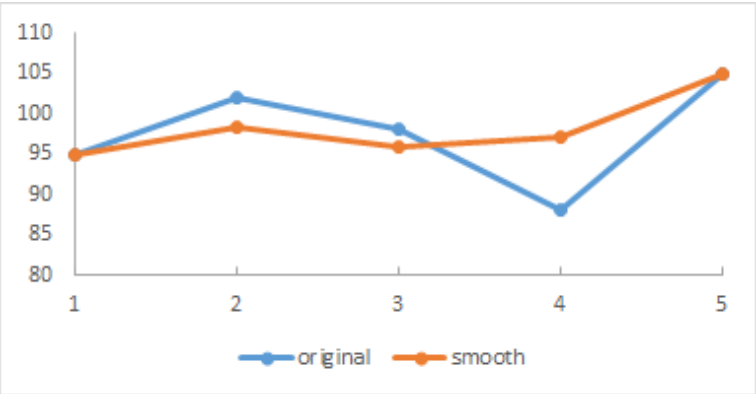
In data analysis, *smoothing* a data set means approximating it to capture important patterns in the data while eliding noise or other fine-scale structures and phenomena. One simple smoothing technique is to replace each (internal) element of a sequence of values with the average of that element and its predecessor and successor. Assuming that extreme outlier values are an aberration caused, perhaps, through poor measurement, this averaging process replaces them with a more plausible value in the context of that sequence.

For example, consider this sequence of `heart-rate` values taken from a list of personal health records (defined above): `95 102 98 88 105`

The resulting smoothed sequence should be: `95 98.33 96 97 105`, where:

- 102 was substituted by 98.33: `(95 + 102 + 98) / 3`
- 98 was substituted by 96: `(102 + 98 + 88) / 3`
- 88 was substituted by 97: `(98 + 88 + 105) / 3`

This information can be plotted in a graph such as below, with the smoothed graph superimposed over the original values.



Design a function `data-smooth`—

```
fun data-smooth(phrs :: List<PHR>) -> List<Number>
```

—that consumes a list of PHRs and produces a list of the smoothed `heart-rate` values (not the entire records).

Most Frequent Words

Given a list of strings, design a function `frequent-words`—

```
fun frequent-words(words :: List<String>) -> List<String>
```

—that produces a list containing the three strings that occur most frequently in the input list. The output list should contain the most frequent word first, followed by the second most frequent, then the third most frequent. Break ties by putting the shorter word (by length in characters) first. You may assume that the three most frequent words will have different length. You may also assume that the input will have at least three different words.

Earthquake Monitoring

Geologists want to monitor a local mountain for potential earthquake activity. They have installed a sensor to track seismic (vibration of the earth) activity. The sensor sends measurements one at a time over the network to a computer at a research lab. The sensor inserts markers among the measurements to indicate the date of the measurement. The sequence of values coming from the sensor looks as follows:

```
20151004 200 150 175 20151005 0.002 0.03 20151007 ...
```

The 8-digit numbers are dates (in year-month-day format). Numbers between 0 and 500 are vibration frequencies (in Hz). This example shows readings of 200, 150, and 175 on October 4th, 2015 and readings of 0.002 and 0.03 on October 5th, 2015. There are no data for October 6th (sometimes there are problems with the network, so data go missing). Assume that the data are in order by dates (so a later date never appears before an earlier one in the sequence) and that all data are from the same year. Also, assume that every date reported has at least one measurement.

Design a function daily-max-for-month—

```
fun daily-max-for-month(sensor-data :: List<Number>, month :: Number) -> List<Report>
```

—that consumes a list of sensor data and a month (represented by a number between 1 and 12) and produces a list of reports indicating the highest frequency reading for each day in that month. Only include entries for dates that are part of the data provided (so don't report anything for October 6th in

the example shown). Ignore data for months other than the given one. Each entry in your report should be an instance of the following datatype:

```
data Report:
   | max-hz(day :: Number, max-reading :: Number)
end
```

Appendix F

Additional Files, Figures, and Entries

F.1 NEU Design Recipe Course Web Page

The Design Recipe

Data	Functions
Definition	Signature
Interpretation	Purpose Statement
Examples	Tests
Template	Code

How To Use The Design Recipe

The design recipe is here to help you design programs properly. When beginning to solve a new problem, the first question to ask yourself is: do I need a new type of data? If so, follow the steps in the **Data** column for every new type of data you need. When writing functions, follow the steps in the **Function** column for every function you write.

Note that "do I need a new type of data?" can often be answered with the question "can I write the signature of the function I want to write?"

Below are helpful questions to ask yourself when tackling a specific step of the DR:

Definition

- What do I need to keep track of?
- How many cases are there?
- In each case, do I need to keep track of more than one thing? (if so, I need a struct or list)
- In each case, how do I encapsulate what I need to keep track of? (number? some other struct? list? etc.)

Interpretation

- What about this data is potentially unclear/needs to be explained?
- Are there units? (grid vs. pixel, for example)

Examples

- Have I covered every case of the data definition?
- Do I have relevant examples for testing for the functions I know I need?

Template

- How many cases of the data are there?
- How do I tell them apart?
- What data can I pull out at the top level of those cases?
- Are the pieces of data I can pull out complex with their own template that I should call to **(possibly self referential)**?

Signature

- What pieces of data does this function need to take in?
- What are their types?
- What does this function output and what is its type?

Purpose Statement

- What is this function doing?

Functional Examples/Tests

- Have I covered every case?
- Have I covered edge cases?
- Do the tests I have convince me my function is properly written?

Code

- Can what I'm trying to achieve be broken down into multiple steps (i.e. should I use helpers)?
- What template do I use?

Figure F.1: The NEU course web page on the design recipe

F.2 WPI IRB Approval Letters

WORCESTER POLYTECHNIC INSTITUTE

Worcester Polytechnic Institute IRB# 1
HHS IRB # 00007374

12 September 2016
File: 16-187

Re: IRB Expedited Review Approval: File 16-187 "Developing Narratives to Explore Factors in Planning Computer Programs"

Dear Prof. Fisler,

The WPI Institutional Review Committee (IRB) approves the above-referenced research activity, having conducted an expedited review according to the Code of Federal Regulations 45 (CFR46).

Consistent with 45 CFR 46.116 regarding the general requirements for informed consent, we remind you to only use the **attached stamped approved consent form** and to give a copy of the signed consent form to your subjects. You are also required to store the signed consent forms in a secure location and retain them for a period of at least three years following the conclusion of your study. You may also convert the completed consent forms into electronic documents (.pdf format) and forward them to the IRB Secretary for electronic storage.

The period covered by this approval is 12 September 2016 until 11 September 2017, unless terminated sooner (in writing) by yourself or the WPI IRB. Amendments or changes to the research that might alter this specific approval must be submitted to the WPI IRB for review and may require a full IRB application in order for the research to continue.

Please contact the undersigned if you have any questions about the terms of this approval.

Sincerely,

Kent Rissmiller
WPI IRB Chair

100 INSTITUTE ROAD, WORCESTER MA 01609 USA

Figure F.2: WPI IRB approval letter for the original study

WORCESTER POLYTECHNIC INSTITUTE

Worcester Polytechnic Institute IRB# 1
HHS IRB # 00007374

26 February 2018
File:16-0187MR

RE: Modification to IRB File 16-0187 "Developing Narratives to Explore Factors in Planning Computer Programs"

Dear Prof. Fisler,

The WPI Institutional Review Committee (IRB) approves the modification submitted to application 16-187 "Developing Narratives to Explore Factors in Planning Computer Programs" dated 12 February 2018 and approves the modification to reduce the number of sessions with students from six to two, contained the study within one course (cs1101) instead of two (1101 and 2102).

Consistent with 45 CFR 46.116 regarding the general requirements for informed consent, we remind you to only use the **attached stamped approved consent form** and to give a copy of the signed consent form to your subjects. You are also required to store the signed consent forms in a secure location and retain them for a period of at least three years following the conclusion of your study. You may also convert the completed consent forms into electronic documents (.pdf format) and forward them to the IRB Secretary for electronic storage.

This study is also approved for renewal and extends the study to **25 February 2019** unless terminated sooner (in writing) by yourself or the WPI IRB. Amendments or changes to the research that might alter this specific approval must be submitted to the WPI IRB for review and may require a full IRB application in order for the research to continue.

If the research is to continue past 25 February 2019 a renewal, application must be filed with the IRB in sufficient time for approval before 25 February.

Please contact the undersigned if you have any questions about the terms of this approval.

Sincerely,

Kent Rissmiller
WPI IRB Chair

100 INSTITUTE ROAD, WORCESTER MA 01609 USA

Figure F.3: WPI IRB approval letter for modifications to the original study