ABSTRACT

Title of Document:	HOW ELECTRICAL ENGINEERING STUDENTS DESIGN COMPUTER PROGRAMS
	Brian Adam Danielak, Doctor of Philosophy, 2014
Directed By:	Associate Professor Andrew Elby, Department of Teaching, Learning, Policy, and Leadership

When professional programmers begin designing programs, we know they often spend time away from a computer, using tools such as pens, paper, and whiteboards as they discuss and plan their designs (Petre, van der Hoek, & Baker, 2010). But, we're only beginning to analyze and understand the complexity of what happens during such early-stage design work. And, our accounts are almost exclusively about what professionals do. For all we've begun to understand about what happens in early-stage software design, we rarely apply the same research questions and methods to students' early-stage design work. This dissertation tries to redress that imbalance. I present two case studies — derived from my 10 study participants — of electrical engineering (EE) students designing computer programs in a second-semester computer programming course.

In study 1, I show how analyzing a student's code snapshot history *and* conducting clinical interviews tells us far more about her design trajectory than either method could alone. From that combined data I argue students' overall software designs can be consequentially shaped by factors — such as students' stances toward trusting their code or believing a current problem is a new instance of an old one — that existing code snapshot research is poorly equipped to explain. Rather, explanations that add non-conceptual constructs including affective state and epistemological stance can offer a more complete and satisfactory account of students' design activities.

In study 2, I argue computer science and engineering education should move beyond conceptual-knowledge and concept deficit explanations of students' difficulties (and capabilities) in programming. I show that in doing design students do, say, write, and gesture things that:

- Are outside the phenomenological scope of most (mis)conceptions accounts of programming
- Would be explained differently under frameworks that emphasize manifold epistemological resources. Some student difficulties can be recast as epistemological blocks in activity rather than conceptual knowledge deficits. Similarly, some students' productive capacities can be understood as epistemologically-related stances toward an activity, rather than evidencing particular knowledge of specific computational concepts.
- Would suggest different instructional interventions if teachers attended to the stabilizing aspects — such as epistemological dynamics — that help these episodes of activity cohere for students.

HOW ELECTRICAL ENGINEERING STUDENTS DESIGN COMPUTER PROGRAMS

By

Brian Adam Danielak

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Dedication

To my grandparents. Anna and Josef Danielak endured and survived so that I could be here. Sabina Schwab loved me and made funny faces at me when no one else was looking. Henry Schwab continues to teach me.

I also dedicate this to my brother, Jason. My name may be on the cover of this thing, but without his support it wouldn’t exist.

I love you all.

Acknowledgements

Once, in my high school Economics class, my teacher Mr. Mooney[1] told us to take out a sheet of paper. We were to spend 5 minutes writing down a list of all the people who were responsible for getting us to school that day. I started off slowly. Mom, for waking me up when my alarm clock didn't. Me, for remembering to put gas in the car.[2] Then I thought about all the people who had maintained the car over the years. I thought about the people who had designed the car, when it was just modeling clay and blueprints, before a single piece of metal was bent into a fender.[3] Soon I was writing down the people who made the antifreeze that kept the engine cool, Charles Goodyear for developing vulcanization techniques that make modern tires work, and the guy at Jiffy Lube who told me—and I think I'm quoting here—"normal transmission fluid should be red, like fruit punch; your transmission fluid is brown, like Thanksgiving gravy with metal shavings in it."

I was surprised—having filled the front of the sheet and most of the back—when Mr. Mooney told us to stop. He said the point of the exercise was to show that as a society we are interdependent. That interdependence is key to understanding the very basics of economic theory: scarcity, pricing, trade, comparative advantage. But, I was still stuck on how many people were on my list. Most of them were people I'd never even met. It wasn't that I didn't care about economics; it was that for the first time I realized in a big way how many people were responsible for where I was.

I can't exhaustively thank everyone who helped me make this. Moreover, some people, having read its contents, might wish to dissociate themselves entirely from it. To those who made this dissertation possible, I hope my thanks below will suffice. To those I leave out, please don't let my imperfections reflect on the job you did. Lastly, to those who wanted no public record of having been associated with this, I'm sorry.

I would first like to thank my advisors, Andy Elby and Ayush Gupta. Nearly six years ago I made the decision to leave my Ph.D. program, my school, and my city behind to join Andy's program. It wasn't long until I was working with, joking with, writing with, and being advised by Andy and Ayush. From Andy, I learned to be clever, argue well, and always try to find the good, redeeming features of things. From Ayush, I learned the importance of bringing empathy and compassion to my work and my life. Any graduate student would be lucky to have just one of them as an advisor. I struck gold with both.

My thanks go also to my committee. Ben Bederson graciously welcomed me into Maryland's human-computer interaction lab, all the while offering grounded and focused feedback on my work. Joe Redish has been an upbeat supporter of—and at times loyal opposition to—my research. My thoughts and insights have only grown stronger with his contributions. Jim Greeno has seen my intellectual development from the beginning. It is not an exaggeration to say parts of this dissertation were forged in the crucible of DC rush hour as I tried to navigate traffic and answer his

[1] That's not a pseudonym.
[2] Yes, I counted myself.
[3] Or, for that matter, *re-bent* when mom hit a lightpost in a parking lot.

questions about my research. I thank him for entrusting me with his vehicular safety. I thank him as well because at a dinner in June of 2011 he restored my faith that some people get into academia to try and do good. Finally, Wil Doane helped get me into this crazy mess in the first place. In addition to being a mentor in computer science, Wil has been an unflinching champion of my ideas for as long as we've known each other. He first showed me what passion and innovation can look like in a computer science classroom; I hope this dissertation is a natural extension of those ideals.[4]

I'd like to thank my compatriots in the Physics Education Research Group (PERG) at Maryland. Jessica Watkins, Chandra Turpen, Vashti Sawtelle, and Julia (Svoboda) Gouvea were model post-docs and amazing listeners. I look up to all of them, which at least Julia has confirmed makes her uncomfortable. Tiffany Sikorski has always been impressed with my work, which means so much to me because I think she's a very hard person to impress. Lama Jaber was a constant source of warmth, compassion, and joy. I was very sad to see her leave for Boston, but I'm so honored to have her as a friend. Ben Dreyfus offered a sympathetic ear and ongoing encouragement when I got stuck thinking about students thinking about programming. Luke Conlin was my stalwart conference roommate, intellectual companion, cheerleader from afar, and fellow pillow fort architect. Gina Quan taught me how to hug; her non-hug-related work teaching girls to program Arduino robots has me endlessly excited. And, thanks to PERG's impressive alumni network, Rosemary Russ helped me think through some tough data analysis once I got to Madison.

Eric Kuo and Mike Hull were my teammates in making sense of engineering students making sense of mathematics. Without Eric and Mike taping, analyzing, and dissecting classroom video my work couldn't exist. Eric in particular endured a roundtrip car ride from Boston to Bar Harbor in which I panicked about everything I didn't understand about our research.[5]

Jen Richards, Colleen Gillespie, and Jason Yip have been with me in Science Ed since the beginning. It was a painful change when we stopped taking the same courses, changed advisors, and changed geographies. But, we gave writing feedback to each other, supplied references to each other, and kept each other in good humor when things got hard. I'd like to single out Colleen for being the first one of us to prove this whole dissertation thing was possible. And, I'd like to single out Jen for watching over me and watching out for me. When Jen noticed I was struggling, she talked to me about it — a simple act that got me into engineering education research and led to all of this. In so many ways, I wouldn't be here without her.

It may buck tradition, but I'd also like to thank my undergraduate mentors who have been with me for almost a decade. Robert Daly first convinced me to come to the University at Buffalo. He later, to my delight, became my mentor in English and agreed to supervise my undergraduate thesis. It turned out the ideas in that thesis—while poorly argued and barely cogent—would form the basis of my intellectual life for the past six years.[6] Josephine Capuana backed me from the

[4] And if it's not, I hope you'll forgive me.
[5] I still panic about it. I'm just older now.
[6] I know. I was surprised too.

beginning. As my advisor she supported me no matter what I wanted to do and how many majors I cycled through to do it. She is also the reason I met my second undergraduate mentor, Kenneth Takeuchi. Dr. Takeuchi has backed me, time and time again, since I was a freshman in college. He and Amy Marschilok taught me about mentoring, inspired me to become an academic, and continue to serve as models for what it means to care about students.

I'd like to thank the friends who have been there to prop me up and help me think. The first group is so big it gets its own paragraph. Robert Davis and Anne Davis welcomed me into their home and their family where everyone seemed to me to be strangely excited by what I was working on. Will Davis and Jamie Davis gave C help when I needed it. Russ Davis inspires me with his dedication to serving the country. Nora Deram and Mat Deram, in addition to being improbably lovely people, are both helping to build the future of mathematics. Travis Pond took me to the hallowed ground of the Apollo 1 launchpad so I could witness the engineering marvel of putting a rocket into space. And Nan Pond has been my friend, bandmate, fellow drum major, R convert, transatlantic pen pal, and support system for nearly 15 years. Here's to uncountably infinitely many more.

Al Stein took me to lunch and reminded me that I've been boldly taking chances with my writing since I was in tenth grade. Vanessa Svihla has been my colleague, co-author, and co-conspirator in embracing the importance of design in our work. Jae Vick, Cheryl Medley, Danielle Champney, and Christie Veitch constantly affirmed I could do this, despite my doubts. Heidi Thalman and Adam Lloyd reminded me I am awesome, despite my stern objections to the contrary. And, a ragtag group of Physics Education and Science Education Researchers have been there for me electronically: Warren Christensen, Eleanor Close, Hunter Close, Dedra Demaree, Matty Lau, Sissi Li, Michael Loverude, Sandy Martinuk, Sam McKagan, Rachel Scherr, and Michael Wittmann. I'd especially like to thank Rachel Scherr, Hunter Close, and Michael Wittmann for being so welcoming to me at ICLS 2010.

The people dearest to me get the last word. Mom, thank you for having me. Thank you for raising me. Thank you for loving me. Thank you for doing all of that when it wasn't easy. Also, thank you for that time I called you at 4 am when I was sick.

Dad, thank you for loving me. Thank you for supporting me. Thank you for that time I climbed the tree in our driveway and didn't know how to get back down, so you helped me. And thank you for teaching me how to ride my bike. You might not know it, but almost everything I think about when I think about teaching traces back to that.

Jason, I already dedicated this to you. Stop looking for more credit. It's unbecoming.

And, guys: this is it. This is my Science Book.

Table of Contents

List of Tables

List of Figures

1 Introduction

Seymour Papert's (1980) book *Mindstorms* has a curious section. Or at least I thought so. The book collects years of Papert's thoughts, observations, and experiments with teaching children at school to program. The famous[7] Turtle and LOGO — the language spoken and understood by it — let students express geometrical ideas in the form of procedures and debug procedures using the spatial affordances of geometry. My school did not have LOGO, so I came to see the beauty of programming much, much later in life. But, for many children in Papert's studies, LOGO became a venue for expression and creativity.

The curious section in *Mindstorms* is one in which Papert describes "structured programming," a technique for breaking down large procedures into "mind-sized bites." In short, it's possible to write lots of programs ahierarchically as a list of instructions:

```
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
```

This little program would make a LOGO turtle draw an equilateral triangle. By telling the turtle how much to go forward (in distance), how much to turn (in degrees), and in which direction to turn (in this case right), one can make a turtle trace out this equilateral triangle as a sequence of steps.

But, this program doesn't have much structure. And, as the program grows — say, to accommodate having the turtle draw a simple house — managing the program becomes more complicated. It's possible, in principle, to have dozens or hundreds of simple little statements and it can become more and more difficult to figure out which part of the program drew which part of the picture.

So, I'll propose a modification:

```
TO MAKEATRIANGLE:
      FORWARD 50;
      RIGHT 120;
      FORWARD 50;
      RIGHT 120;
      FORWARD 50;
      RIGHT 120;
```

What I've done is define a *procedure* in LOGO. In a way, the procedure bundles together the list of discrete steps and offers a sense of *structure* to the program. The procedure is now also something I can compose to make more complicated things. Now, for example, I can draw a bunch of triangles that all share a vertex and make a kind of pinwheel:

[7] Bearing in mind that "fame," in the subfield that is computing education, is a relative thing.

```
MAKEATRIANGLE;
RIGHT 120;
MAKEATRIANGLE;
RIGHT 120;
MAKEATRIANGLE;
RIGHT 120;
```

If I apply the same kind of thinking to the pinwheel that I did to the triangle, I might do this:

```
TO MAKEAPINWHEEL:
      MAKEATRIANGLE;
      RIGHT 120;
      MAKEATRIANGLE;
      RIGHT 120;
      MAKEATRIANGLE;
      RIGHT 120;
```

Now imagine two students, both of whom are trying to get the turtle to draw a pinwheel.

Alan’s Code	Ada’s Code

```
;;; Alan's Code
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
FORWARD 50;
RIGHT 120;
```

```
;;; Ada's Code
TO MAKEATRIANGLE:
        FORWARD 50;
        RIGHT 120;
        FORWARD 50;
        RIGHT 120;
        FORWARD 50;
        RIGHT 120;

TO MAKEAPINWHEEL:
        MAKEATRIANGLE;
        RIGHT 120;
        MAKEATRIANGLE;
        RIGHT 120;
        MAKEATRIANGLE;
        RIGHT 120;

MAKEAPINWHEEL;
```

The picture Alan's turtle makes should be identical to the picture Ada's turtle makes.[8] So, if all we care about is the final art, Alan and Ada are making the same thing. They're indistinguishable as students from the perspective of what the turtle draws on paper.

But, clearly Alan's code and Ada's code aren't the same. Ada's program evinces what Papert (1980) would call *structured programming*: pinwheels are composed of triangles, while triangles are composed of straight line segments.[9] Alan's code doesn't

[8] Unless I screwed up, which is possible. If so, that's my fault; don't hold Alan and Ada accountable.

[9] Not to get pedantic on you, but yes, Ada could have written a program that was perfectly syntactically valid — and produced the same picture — while choosing different and possibly bogus procedure names. By renaming functions, for example, Ada could have forced me to say that in her program "unicorns are composed of Cheetos, and Cheetos are composed of dinosaur fingers." And, while *my restatement* would make no

compose things. Every line in Alan's code is a direct instruction to the turtle and there is no sense of hierarchy. Instead, there's just a serial flow of instructions. To think more about the differences between Alan and Ada, let's go back to Papert.

Papert (1980) discusses structured programming in the case of two children. One child, Robert, embraced structured programming. Robert extolled the style: "see, all my procedures are mind-sized bites" (Papert, 1980). And, indeed, programmers today are given similar advice in books on professional programming practice (Martin, 2009). But, Keith was a child who resisted. His programs remained ahierarchical; "featureless" in Papert's words. Debugging was harder for Keith, too, because it was harder to locate problems in a program devoid of structure.

If Keith was rather stubbornly sticking to his ahierarchical style, should a teacher have proactively *instructed* Keith to change? Papert says no. Rather, as Keith encountered troublesome errors debugging his code, he would meet with familiar advice:

> When a child in this predicament asks what to do, it is usually sufficient to say: "You know what to do!" And often the child will say, sometimes triumphantly, some-times sheepishly: "I guess I should turn it into subprocedures?" The "right way" was not imposed on Keith; the computer gave him enough flexibility and power so that his exploration could be genuine and his own. (Papert, 1980)

But, largely, that's where Papert leaves things. As Papert would have it, most children will inevitably run into increasing frustration when they debug. When children hit those problems, it's up to the teacher to patiently remind the student that, in a sense, "we've covered this." After repeatedly hitting this wall most students will see the light — eventually — and come to save themselves from frustration by adopting structured programming.

But what if they don't?

I knew from decades of research in science education that students do not always adopt the practices we want them to. As Hammer (1994) essentially put it, *not all novices go on to develop expertise,* and many cognitive formulations of the novice-expert continuum didn't (and still don't) really explain why. Was structured programming something more than a practice of keeping code tidy and easing debugging? *Yes*, I believed so. Structured programming wasn't the computer science equivalent of "always labeling your units" in science[10]; it reflected a larger view on how we think about procedure and a concomitant willingness to compose big complex ideas out of small atomically-understandable ones (Abelson & Sussman, 1996; Papert, 1980).

Back to Alan and Ada. With all this talk about structured programming, we've mostly been focusing on *code*: how it's structured differently and what affordances and

---

sense, neither LOGO nor the turtle would have a problem interpreting Ada's procedure to make what Ada would call a pinwheel. The dinosaur might have a problem, I guess.

[10] Don't misunderstand me; labeling units is important. The fact that early versions of LOGO didn't drives me *nuts*. But, if you asked a bunch of sophisticated chemists what are the chief intellectual practices of their discipline, I don't think "labeling units" will top the list. If you ask Abelson & Sussman (1996) what the chief intellectual practice of computer programming is, they'll tell you it's managing the complexity of large software systems.

drawbacks result from the design. We still don't have a sense of how Alan and Ada actually built their programs. We can't know, just from these snippets, whether Ada's code started with a compositional structure. Perhaps Ada started just the way Alan did, so there was some intermediate state of Ada's code before she refactored it when it was just as ahierarchical and featureless as Alan's code. Or, perhaps Alan started by *trying* to structure his program, but he ran into some errors. Maybe Alan's literal code is the result of a design retreat late in the game. It's not that he doesn't *appreciate* structured programming; he just couldn't get it to work and, feeling pressured to produce a proper pinwheel, resorted to taking out the structure and leaving just concrete instructions for the Turtle.

I had questions.

Would all students come to structured programming after banging their heads enough against a buggy featureless program? *Maybe*. Would they do it on a timescale we could see? *I wasn't sure*. Was it possible students could get through one, or even multiple semesters of programming without ever embracing structured programming for themselves? *Quite possibly*. Outside of Papert's schools, was it possible for research to not only track but explain why some students used structured programming and other students didn't? Were there Alans and Adas out there in real life whose complex stories could help us understand more about how students learn structured programming?

This dissertation is my stab at beginning to answer those questions. The focus of my study is a programming course for electrical engineering students. I chose that course and population for two reasons. First and foremost, I had preliminary evidence suggesting students were emerging from the course with wildly different mindsets about its relationship to engineering and, by extension, design. One student, Larry, saw structured programming as a natural extension of the way he tried to make sense of the physical world. Another student dismissed the class entirely as "not even a real engineering course." Second, my connections to the school of engineering allowed me relatively easy access to the class.

Over two semesters I observed lectures and interviewed students. In interviews, I was able to use video recordings, screen capture software, and an electronic pen to capture what students said, gestured, typed, and wrote when they programmed. I later augmented the study by tracking students' code histories. I co-developed a simple automated system to create version-controlled repositories of the multi-week projects students worked on for the course. Each time a student compiled, our system captured a snapshot of the entire codebase (called a "commit" in Git, the versioning system we used). With each snapshot we also captured the input students passed to the compiler and any compile-time messages (including errors and warnings) they would have seen.

Table 1 shows a summary of all the participants in my studies and the kind of data I collected.[11] The solicitation process began at the beginning of each semester. During the second class lecture I read a 3-minute IRB-approved speech explaining that I was studying how students understand programming code. Students who expressed interest were invited to participate in interviews, and for each interview they completed they were

[11] The "Inscriptions" column is currently blank because I've been having trouble accessing that portion of my dataset. I hope to resolve my technical difficulties so I can provide inscription counts in later revisions of this dissertation.

paid $15. Interviews were scheduled opportunistically based on when students were available. In total, the data corpus comprises more than 20 hours of clinical interviews and more than 2,500 distinct code snapshots. Of the 10 students who participated, I chose to focus on Rebecca (who appears in both study 1 and study 2) and Lionel (who appears in study 2).

Table 1 – An overview of my research participants and the types of data I collected

Semester	Participant	Interviews[12]	Screencaptures	Inscriptions[13]	Code Snapshots
Fall 2011	Lionel	1	1 session		—
	CJ	1	—		—
	Donna	2	2 sessions		—
	Sam	1	—		—
	Toby	2	—		—
	Will	1	1 session		—
Spring 2012	Isaac	3	1 session		434[14]
	Dana	3	2 sessions		878
	Natalie	4	1 session		262
	Rebecca	5	3 sessions		959

[12] All interviews were videorecorded with the exception of Toby's. Toby requested to be audiorecorded only.

[13] At present, technical difficulties prevent me from reporting an exact count of inscriptions collected.

[14] Due to an unidentified error in Isaac's code snapshot history, we lack data between February 21 and April 12. we know in aggregate what changed in his code between those two dates; we just can't resolve that aggregate change down into individual changes for each time Isaac compiled between those dates.

This dissertation is structured into two studies that analyze data from the corpus. Study 1 asks what we can learn if a student submits a program like Alan’s. In other words, when one student’s final code submission is structured in peculiar ways (or not at all), can research help us retrospectively recover the story of how that design came to be? Study 2 in effect asks what directs and sustains Alan and Ada’s in-the-moment programming activity. I analyze rich data — including talk, gestures, inscriptions, and screen captures — of two real students. The analysis tries to understand what constitutes the approaches they take when they code. Along the way, study 2 offers an illustration of why strictly “conceptual” accounts of students may not be enough to explain certain classes of phenomena in learning to program.

2 Study 1 – Retrospectively analyzing the design of a student's project

2.1 Abstract

This paper focuses on a historically understudied area in computing education: attending to students' *design thinking* in university-level introductory programming courses. We detail the case of Rebecca, a first-year Electrical Engineering student taking a required 2nd-semester programming course in C. Our analysis focuses on two related aspects of Rebecca's code for a multi-week project:

1. The origin, nature, and evolution of unusual structural and behavioral features of Rebecca's code
2. The subtle, yet complex reasons that led Rebecca to make particular design choices in her code

Our data comes from ethnographic observation of Rebecca's class, fine-grained compile-time snapshots of Rebecca's codebase, and semistructured interviews with Rebecca. We first present an analysis of the compile-time snapshots, detailing Rebecca's unusual use of file-scanning loops and her seven-fold repetition of a particular code chunk (once for each day of the week). We then augment that analysis with data from semi-structured interviews with Rebecca, which reveal that affect (Eynde & Hannula, 2006; Hannula, Evans, Philippou, & Zan, 2004) and framing (Hammer, Elby, Scherr, & Redish, 2005; van de Sande & Greeno, 2012) offer substantial explanatory power for understanding why Rebecca made particular design choices.

2.2 Introduction

2.2.1 Studying the software design practices of experts

In 2010, the journal *Design Studies* devoted an entire issue to studies of how professional software engineers design complex systems. The journal featured five different research perspectives on the same dataset: videos of three different professional software engineering teams trying to design a traffic simulator. The existence of the journal issue, and the investigations contained therein, were motivated by what the issue's editors saw as a growing need to know more about how software design gets done in practice. In particular, the editors contended, fields of design studies, interaction analysis, and human-computer interaction don't know enough about how software engineers use representations and collaborative exchanges to organize the beginning phases of a design:

> [d]uring formative design, software engineers spend a great deal of time engaging in creative, exploratory design thinking using pen and paper or a whiteboard—*whether alone or in a small group*. However, not enough is known about how software designers work in such settings. What do designers actually do during early software design? How do they communicate? What sorts of drawings do they create? What kinds of

> strategies do they apply in exploring the vast space of possible designs? (Petre et al., 2010, p. 533; my emphasis added)

Research in that issue takes on a variety of challenges in the study of expert software design practice. One challenge, for example, is that of understanding how engineers process, prioritize, and cope with design requirements. Ball, Onarheim, and Christensen (2010) observed that engineers deploy mixed strategies, developing solutions breadth-first for easy problems and depth-first for more complex problems. They also found that the more complex a requirement became, the more likely engineers were to create speculative simulations (through talk and representations) about how a system might work to solve that problem (Ball et al., 2010).

Looking at design sessions longitudinally, Baker and van der Hoek (2010) explored the shape and trajectory of how ideas generated in the design process develop and relate to one another. Those researchers found that roughly a third of the ideas discussed in a typical design session “were reiterations or rephrasings of previously stated ideas” (Baker & van der Hoek, 2010, p. 604). While it is perhaps frustrating that ideas would be repeated so much, the authors argue such repetition can be viewed as a kind of continual revisitation to make sure a proposed design coheres:

> Rather than representing a failure on the part of the designers, this repetition seems to be a necessary character of successful design sessions. Each time an idea is resurrected it is placed them [sic] in a new context, and compared to different aspects of the system. In this way, a concept of compatible, elegant design ideas is slowly converged upon. (Baker & van der Hoek, 2010, p. 607)

Perhaps most germane to the work presented herein, Jackson (2010) explores the role of *structure* in software design. Jackson’s meaning in using the word “structure” is broad. It can best be described as the arrangement of and relationships between the elements of a system. But, the broadness is deliberate, because Jackson’s overall argument is that the work of software often involves a coordination of different structures, some of which drive the organization of others. In the case of Jackson’s (2010) study, designers worked with various kinds of structured sets to build software that could simulate traffic patterns in a city section. The work of design then, in part, becomes that of defining and coordinating structural relationships at various levels of abstraction:

- The *system-to-be-modeled*, in this case a system of roads and intersections on which one can simulate traffic. Schematically, this system can be structured on the Cartesian plane of a map. Its representation (a diagrammatic map-like network of streets) works to organize how the system-to-be-modeled looks in the real world.
- The *actors* present in the system-to-be-modeled. In this case, that means abstracting from the map to all of the *actors* that simulator software would need to send data to and get data from: including traffic signal units, traffic signal controllers, drivers, and vehicle sensors.

- The *meta-entities* that must exist to execute the actual simulation. Crucially, these abstractions may not be the same as the actors *in* the system.[15] In this case, additional computational entities must be added to carry out a traffic simulation, such as an arrivals model and a simulation clock (Jackson, 2010, p. 559)
- The *programming code* that embodies the data and behaviors of the entities and meta-entities. While to some degree aesthetic, the symbolic structure of code is in many ways driven by decisions about structural relationships in other levels of abstraction.

How engineers develop and interact with even the first three kinds of structural representations can profoundly influence the nature and quality of the code they produce. As Jackson (2010) observes (in notable contrast to (Baker & van der Hoek, 2010)), early negotiations of structure led to design decisions that were *not* revisited later:

> The traffic simulation problem was of a kind unfamiliar to all three design teams. All three quite rightly tried immediately to assimilate the problem to something they already knew. Two teams took the view that the problem was an instance of the Model-View-Controller (MVC) software pattern; the third identified the problem as 'like a drawing program'. Unfortunately, these hasty classifications were inadequate; but in every case they were accepted uncritically and never explicitly questioned. (Jackson, 2010, p. 564)

Ultimately, careful study of how software engineers design shows us the enormous complexity involved in producing effective and thorough code. And, much of that complexity is both afforded and constrained by talk, representational infrastructure, and knowledge of larger-scale solution patterns. It's striking, then, that this research perspective on how programing design work gets done—for example, the detailed ethnomethodological analysis by Rooksby and colleagues on how engineers use a whiteboard (Rooksby & Ikeya, 2012; Rooksby, 2010)—seems largely underrepresented in studies of how students learn to program. Instead, the latest generation of research on student learning in programming has focused much more extensively on questions like *how can we assess and mitigate students' difficulty in programming?* than it has on questions like *how do students learn and display evidence of design thinking in programming?*

---

[15] For example, suppose we were building an AI for chess. The "entities in the system" are the pieces (rooks, queens, knights, etc.) and the actual board. But, those pieces alone aren't enough to create a chess AI. One must also create entities that make decisions, move the pieces, maintain the state of the board, determine the legality of moves, etc. Thus, bishops may be "entities in the system" that know how to move (only along diagonals, up to 8 spaces). But, bishops need another meta-system entity to tell them *when* and *where* to move. That distinction is important because it means simulating chess involves much more than building a board and pieces, even though the board and pieces may constitute the only outward-facing aspects of a chess AI.

2.2.2 The missing attention to students' design thinking

It's difficult to point directly to the absence of a research direction regarding how students design software. Rather, one can instead show that most computing education research focuses elsewhere. In 2004, Valentine conducted a meta-analysis of 20 years of conference papers at SIGCSE[16]. His analysis focused on papers focused on first-year university courses, which as of 2004 comprised about 1/3 of the total papers presented at SIGCSE. Valentine's resulting taxonomy reveals how—in a conference about computer science education—there is at best a minimal focus on how students engage in design thinking. In the decade from 1994-2003:

- About 24% of conference papers were "Marco Polo" studies, which Valentine (2004) describes as teacher- or administrator-centered accounts of describing a new method, course, curriculum, or approach and documenting how well it works.
- Another 42% of papers were classified as either "Nifty" or "Tools," and centered on either innovative assignments to give to students or software tools to augment learning, instruction, and assessment.
- Only 22% of studies were classified as "Experimental," a broad category which includes everything from quasi-experimental comparisons of teaching methods to ethnographic interviews with students solving protocol problems.

Joy et al. (2009) offer a similar, more recent view of the state of computing education research. Their 2009 survey included both conference papers and journals, from which they constructed a taxonomy of over 3,500 papers in two overlapping fields: "education in computer science" and "computers for education" (Joy et al., 2009, p. 112). While Joy et al.'s (2009) classification scheme isn't identical to that of Valentine's (2004), the most ready parallel to Valentine's "Experimental" studies are what Joy et al. call *theoretical pedagogy*:

> The focus of the paper is principally educational, and reports results grounded in education theory (i.e. explicitly references, discusses and applies pure education theory). This category is used for "Learning Psychology" where the "learning" is the predominant focus of theory, such as Bruner and Vygotsky. (Joy et al., 2009, p. 114)

As a percentage of the pool of 3,500 papers, research classified as "theoretical pedagogy" totaled less than 5%.[17] Admittedly, the authors don't disaggregate their final results by "education in computer science" vs. "computers for education." Still, it seems reasonable to assume that if even half the publications were from "computers for education" and none of that half focused on theoretical pedagogy, that still means less than 10% of publications *in* education in computer science foreground learning as a primary phenomenon.

[16] SIGCSE, as used here, is the abbreviation for the annual meeting of the Association for Computing Machinery (ACM) Special Interest Group for Computer Science Education. SIGCSE is one of the premier venues for computer science education.
[17] Joy et al. (2009) don't report raw quantitative numbers, so I estimated this percentage by measuring their data graphic on p. 117.

If one is looking for the kinds of studies that would involve deep, fine-grained analysis of learning via students in action, the pool gets even thinner. In a survey of 79 papers from the International Computing Education Research Conference (ICER)—published between 2003 and 2009—Malmi et al. (2010) found that of the 79% of papers that included any mention of a theoretical framework, 39% were "surveys." (p. 8). "Experimental" papers account for another 15% of papers that explicitly identified a framework. The total number of papers that employed case study methodology, ethnography, phenomenology, *or* phenomenology was 10, or approximately 14% (Malmi et al., 2010, p. 8). But, since Malmi et al. (2010) do not identify those case study/ethnography/phenomenology/phenomenograpy papers by name, it's difficult to assess just how closely those papers hew to their nominal frameworks.

Indeed, one of the most useful collections of research on the psychology of how people design software isn't recent; it's almost thirty years old. In the 1980s, Elliot Soloway, James Spohrer, and others were trying to establish what expert programming behavior looked like (Adelson & Soloway, 1985; Soloway, 1986), and what kept novice programmers from developing expertise (Bonar & Soloway, 1983, 1985; Mayer, 1981; Pea, Soloway, & Spohrer, 1987; Soloway & Spohrer, 1989). One of the most fundamental arguments to emerge from Soloway and Spohrer's work was a refutation of the claim that novices make programming errors because they don't understand the elements of a programming language:

> Our empirical study leads us to argue that (1) yes, a few bug types account for a large percentage of program bugs, and (2) no, misconceptions about language constructs do not seem to be as widespread or as troublesome as is generally believed. Rather, many bugs arise as a result of plan composition problems—difficulties in putting the "pieces" of a program together (see sidebar)—and not as a result of construct-based problems, which are misconceptions about language constructs. (Spohrer & Soloway, 1986, p. 401)

Ultimately, Spohrer and Soloway (1986) found strong support for two kinds of problems that transcend any particular programming language:

> Interpretation Problem: When novices read a programming assignment, they do not always infer the correct interpretation from the specifications.
>
> Composition Problem: Novices may not detect negative interactions between sections of code that are locally correct, but globally incorrect. For example, the code to perform the output may be correct, but in the wrong place in the program. (Spohrer & Soloway, 1986, p. 191)

Despite Spohrer and Soloway's (1986) finding that program composition accounts explain more than do language construct accounts of student error, research since has persisted in documenting students' "misconceptions" about programming language constructs (Fleury, 1991, 1993, 2000; Herman, Kaczmarczyk, Loui, & Zilles, 2008; Kaczmarczyk, Petrick, East, & Herman, 2010). Moreover, in a 2006 paper analyzing the programming behavior of hundreds of novices, Jadud argues that a focus on syntax-level analyses can help improve research and practice: “[b]y identifying and understanding the behaviour of novices learning to program, we hope to build up to later making sound cognitive and constructivist inquiries and recommendations” (Jadud, 2006, p. 80).

In the next section, I argue that a growing syntax- and language-focused trend in computing education research is using bigger and richer datasets to push us farther away from the kind of fine-grained studies that help us understand design thinking in software creation. First, I explain how code-snapshotting systems help us capture changes to students’ code over time. Then, I outline why—in their current usage—such systems fail to help us consider larger design thinking issues that might be at play for students. I conclude my introduction with a claim that we can use the same code-snapshot repositories we’ve been mining for syntactical analyses to look for deeper explanations of why students struggle with software design.

2.2.3 The current state of research on students’ code snapshots

A growing trend in computer science education research is the collection and analysis of code snapshot data—records of the state of and changes to students’ code as they develop it (Jadud, 2006; Rodrigo, Tabanao, Lahoz, & Jadud, 2009; Spacco, Pugh, Ayewah, & Hovemeyer, 2006). Though specific implementations differ, the general strategy in such projects is that a student event (typically compiling code or saving a file) triggers a procedure that creates a record containing the entire content of all of a student’s relevant files, as well as associated metadata (time of save/compilation, for example, and any compiler errors that may have been generated). Mining data from such snapshotting systems has led to large-scale documentation of common student errors (Spacco, Pugh, et al., 2006), the development of compile-time detectors to catch common student errors (Spacco, Strecker, Hovemeyer, & Pugh, 2005), and the proposal of formative assessment models to predict student success (Jadud, 2006; Tabanao, Rodrigo, & Jadud, 2011). Building on existing momentum, some researchers are actively pushing for a continued scale-up of how we collect code snapshot data. One current proposal even calls for creating an international database by collecting snapshot data from thousands of introductory programming students worldwide (Kölling & Utting, 2012).

What’s common to these threads of research is how they mine the data. In most applications of code snapshot[18] research, the aim is to average across events and sessions to arrive at a characteristic measure of a student’s behavior. Jadud (2006), for example, develops the idea of a student’s “Error Quotient,” a measure of how effectively a student addresses compile-time errors in their code.[19] Rodrigo and

[18] What I call “code snapshot” research may be more formally called “online protocol” research (Jadud, 2006; Rodrigo, Tabanao, Lahoz, & Jadud, 2009; Tabanao, Rodrigo, & Jadud, 2011). “Online” here isn’t mean to mean the sense of globally-connected computers, but rather that student materials are collected in a minimally-intrusive way *while students work*.

[19] A useful way of thinking about Jadud’s (2006) error quotient is that the computation believes in second chances. It doesn’t penalize students for making

colleagues extend the idea of error quotient (Rodrigo et al., 2009) and introduce similar measures such as students' "compilation profiles" (Rodrigo et al., 2009), "frustration profiles" (Rodrigo & Baker, 2009), and "error profiles" (Tabanao et al., 2011). Because these measures can be collected in real-time as students code, they offer the potential to identify at-risk students (Tabanao et al., 2011) and respond with early interventions (Jadud, 2006).

But, for all the data these methods collect their focus is primarily what Jadud (2006) calls students' "syntactic" struggles: the challenge of articulating well-formed statements a compiler can properly parse. This focus on syntax-level struggles leads to both a methodological and theoretical trade-off: we can see in detail how students struggle with wording, but we see less of how they struggle with the meaning and intent of their code. Syntactical analyses necessarily ignore the specific *content* of students' code because they abstract meta-information about the event: error type, error location, frequency of error.

As an analogy, suppose that instead of studying computer science students we were studying screenwriting students. Further suppose we have a system to track data whenever students save their screenplays and scripts. For each save, we get a copy of the entire script at that time. We can also run an automated analysis to check whether they've properly formatted slug lines[20], put character names in all-capital letters, properly numbered scenes, etc. We might even know where students are when they write (coffee shop, library, home, etc.). If we mine that data, we could learn something about students' *screenwriting behavior*, but only at the level of how they struggle with screenwriting syntax. Using only error-based data it's much harder to answer questions such as:

- How does this student construct a scene or handle dialog in their script?
- Does the student seem to have a grasp of pacing, story beats, and efficient exposition?
- How does their writing manage and develop character arcs?

We also still can't answer fundamental questions about the context of students' writing processes:

- Do they use particular techniques to "break story" and decompose a narrative into its key beats (index cards? Whiteboard?)
- Do they participate in a writing group?
- How do they respond to people giving them notes on revising a script?

To sum up, because snapshot systems are designed to collect the totality of a codebase at frequent intervals, they offer a rich record of data that captures the results—both major and minor—of students' design decisions. But, computing education research that *uses* code-snapshotting has focused much more on detecting, classifying, and predicting student errors than it has on showing how students progress in programming and design expertise. Nevertheless, snapshot-based research shows tremendous promise. Given that :

---

errors; it penalizes students if they don't fix a given error before compiling the same piece of code *again.*

[20] In a screenplay, a slug line gives information about where and when a scene takes place. For example: "INT. FORTRESS OF SOLITUDE – DAY."

1. Snapshotting students' code represents a cutting-edge way to resolve the way code—as a design artifact—evolves over time, *but*
2. Code snapshots, as they're currently used, explore neither the totality of a student's design nor the rich context of that design's production, *and*
3. There is a lack of parity between studies of how professionals design software and studies of how students do so,

It seems sensible to ask: *can code snapshots be used—possibly in synthesis with ethnographically-oriented methods—to start studying how students' design thinking plays a role in their introductory programming work?* We believe the answer is yes.

In what follows, we present work that proceeds from an empirical challenge: how can we develop accounts of students' programming activity that explain the form and evolution of their code on a design project? We offer an account of one student—Rebecca—and her experiences and code from a second-semester course on programming concepts for engineers. Using data from both code snapshots and clinical interviews, we explicate both the challenges of studying students' software design processes and the potential for such study to inform accounts of teaching and learning.

2.3 Context and Methods

2.3.1 Context of the study

The data presented in this case are taken from an ongoing IRB-approved study of undergraduate electrical engineering majors undertaken at *Flagship State*, a large, public research institution on the east coast. For two semesters, I have followed a total of 10 students taking "Intermediate Programming Concepts for Engineers." It is the second of a required two-semester course sequence in programming.[21] Students can (and some do) place out of Basic Programming via AP Computer Science credit, but all Electrical and Computer Engineering students must take Intermediate Programming.

Intermediate Programming has two 75-minute lectures per week and a weekly discussion section led by an undergraduate Teaching Assistant (TA). Typical enrollment is between 60 and 80 students per semester. Like Basic Programming, Intermediate Programming is taught using the C programming language, and it incorporates multi-week projects in C as part of its assessment structure. Students must work individually on four projects over the course of the semester, which together comprise 45% of their final course grade.

Grading projects involves running students' compiled code against automated tests that determine whether a student program's output matches the instructor's canonical output. If a student's program completely matches the canonical output, the student receives at least a 90% grade on a project. The remaining 10% are discretionarily allocated "style" points, awarded for things like proper formatting, code commenting, and functional decomposition (Field Notes).

[21] Hereafter, I use "Intermediate Programming" to refer to the second course and "Basic Programming" to refer to the first.

This study centers on “Flights Database,” the second of four projects assigned to Intermediate Programming students during the spring 2012 semester. Students were asked to build a text menu-based program that would let users query information about airports and plan non-stop and one-stop flights between airports. For this project, the instructor gave students three separate text files as source material. The **airports** file contained names of airports and their three-letter abbreviation codes; the **routes** file contained 3-tuples of two airport codes and the route number of a flight flying between them; the **flights** file contained a list of specific flight information (including arrival and departure times) by route number. Crucially, in order to be able to respond to user queries students would need to build a program that could coordinate information across all three files to return an answer.

My analysis details the work of Rebecca, a female first-year electrical engineering major. I focus specifically on Rebecca’s code for finding “one-stop” flights, which the instructor defined as “all pairs of flights that route the user between the departure and arrival airports with exactly 1 stop (i.e., a one-connection flight)” (Flights Database handout, 2012). The one-stop problem is particularly challenging. To solve it successfully students’ code must accept a user’s choice of airports and day, then stitch together routes that involve two separate flights in a way that passes stringent constraints for acceptable layover times.

2.3.2 Methods

This study proceeds from an empirical challenge: how can we develop accounts of students’ programming activity that explain the form and evolution of their code on a design project? The focused form of that challenge for this study is “how can we understand the unconventional design choices embedded in Rebecca’s one-stop flight code?” To answer that question, my study draws from three data streams: ethnographic observation, clinical interviewing, and code snapshot analysis.

For two semesters, I ethnographically embedded myself in the same instructor’s section of Intermediate Programming. My aim throughout was to see what students see in terms of course material, assignment directives, and instruction. In fall 2011 I observed approximately 50% of the course lectures. I also independently completed all class homeworks and three out of the four course projects to more fully understand the course’s assessments. In spring 2012 I continued attending lectures, though less frequently, and began attending select TA-led discussion sections. During both lectures and discussion sections I took field notes while recording ambient audio using a LiveScribe Pulse pen.

Rebecca was one of four students (three female, one male) willing and able to participate in a series of 1-hour outside-of-class clinical interviews during the spring 2012 semester.[22] I interviewed Rebecca five times, and in typical interviews I split

[22]In total, roughly 30 students responded to my initial in-class solicitation to be contacted by email about my study. Of the students I emailed, approximately 8 students responded to my emails to schedule interview times. Of those 8, only four students were able to successfully find interview time slots that fit our respective schedules.

time between asking about her experiences in the course and giving her time in the interview to work on her project code. During each interview, I simultaneously used:

1. A Kodak Zi8 camera for video-recording our interactions
2. A LiveScribe Pulse pen to capture Rebecca's on-paper penstrokes
3. A MacBook Pro (early 2011) with screen-recording software to capture everything on-screen while Rebecca programmed

The final component of data gathering is modeled after Jadud's (2006) system for capturing students' code. My colleagues and I developed software, built around the open-source version control system called Git, that effectively creates an entire copy of a student's code—what we call "snapshots"—every time students invoke the compiler on their code. Our software then sends those snapshots to a secure, researcher-accessible server in real-time as they're created. Consequently, I could plan each interview with Rebecca around up-to-the-minute knowledge of her work—in some cases work she had completed just hours before the interview—and tailor my interview questions to emerging patterns in her code. In total, Rebecca's work resulted in 958 compilation snapshots over the course of the semester.

In what follows, I recount snapshot and interview data to explore the form and history of certain features of Rebecca's one-stop flight code. First, I simulate an analysis of Rebecca's code in the methodological vein of Jadud (2006) and others (Rodrigo, Tabanao, et al., 2009; Tabanao et al., 2011), analyzing only Rebecca's code snapshots. In this first analysis, I focus on the form and evolution of particular design features of Rebecca's code. Then, I present a second, complementary analysis using contextual data from my clinical interviews with Rebecca. In my second analysis I go beyond the snapshots to highlight why, in Rebecca's own words, she made those particular design choices.

2.4 Code Snapshot Data and Analysis

In this section, I simulate what data on Rebecca would look like using an existing method of analysis described in Jadud (2006). Namely, I work at a level of remove from Rebecca herself. I base my inferences only on her actual code, limiting my interpretations about what she may have *intended* to do or *designed* her code to do only to what's supported in the snapshot-by-snapshot record of her code. Through this "snapshot-only" analysis, I show that we can see in great detail how Rebecca's code embodies particular design features that diverge from how an expert might have handled this design task. I restrict my analysis here to just two such features of Rebecca's code:

1. Rebecca's use of repeated file-scanning loops to create a depth-first search algorithm
2. Rebecca's 7-fold duplication of code to handle checking for flights on each day of the week

2.4.1 Rebecca's file-scanning solution is computationally complex

After declaring variables and opening the three provided text files (flights, routes, and airports), Rebecca's one-stop flight code enters a series of conditionally-nested `fscanf()` commands. The purpose of `fscanf()` is to scan through the characters of a file, most often one line at a time. The programmer can specify patterns of text to look for, e.g., “in each line, look for a word, followed by a space, followed by two digits.” `fscanf()` also gives the programmer the flexibility to store such matched patterns. “Once you find a word followed by two digits, store the word in the following location in memory.” What’s interesting isn’t *that* Rebecca used `fscanf()` to read through files like the list of airports. Any suitable solution would need to read through those files. Rather, what’s interesting is *how* she uses `fscanf()` in her design.

Rebecca’s file-scanning logic never persistently stores the contents of the files it reads in. Rather, her program reads through files one line at a time, and it essentially can’t process or act on airport/flight information not in the line currently being scanned. Instead, it’s been designed to copy single patterns temporarily, then rewind the file back to the top and start reading in one-line-at-a-time again.

What’s consequential about Rebecca’s design choice? Computationally, her code has to repeatedly open multiple files and scan them one line at a time in order to coordinate information. So, the task of finding a one-stop flight between two cities becomes a series of repeated, one-line-at-a-time scans of external files:

1. Scan the **airports** file one line at a time to check the correctness of the user’s input (line 20)
2. Scan the file of pairwise airport **routes** one line at a time (line 24)
3. To find possible connection cities, scan the **routes** file one line at a time again (line 38)
4. If a route matches, scan the **flights** file one line at a time to verify whether the time/day constraints are acceptable (one of the following lines depending on the chosen day: 52, 79, 106, 134, 162, 190, 218).

Rebecca’s code is both visually and computationally complex. The multiply-nested blocks can make it difficult for a human reader to follow the code flow, which may have made it challenging for Rebecca to debug her own work. Moreover, nested for- and while-loops increase the complexity space of a program—what computer scientists call the big-O characterization of her algorithm. For every nesting of a scan loop (there are 4 here) Rebecca increases by 1 the degree of a polynomial that represents the execution time of her program. So, from a performance perspective, Rebecca’s design suffers a trade-off in that with each invocation of a scanning loop, we see a geometric increase in the time complexity of her program.[23] In fairness to Rebecca, the point I make here is not about her but about our mode of analysis. From the perspective of a typical software engineer, Rebecca made a decision involving a design trade-off. Still, inferring only from the code we cannot know whether her decision was deliberate. We cannot know from code whether she knew or understood the kind of complexity and performance trade-off she made, nor can we know how she felt about the consequences of the decision. We know only that her final submitted code contained the results of her decision to use nested scanning loops.

23

2.4.2 Rebecca's file-scanning solution ignores an assignment directive

A second feature or Rebecca's `fscanf()` design is that she ignores instructions in the course assignment, which produces a solution that uses abstraction in a non-obvious way. One design solution to the problem of creating a relational database—and one seemingly dictated by the assignment—is to create three separate arrays in the computer's memory:

> To parse the 3 airline flight database files, you will need to declare arrays that will receive all the data. For the purposes of determining array sizes, you may assume there will never be more than 100 airports in the "airports.txt" file, 500 route IDs in the "routes.txt" file, and 3000 flights in the "flights.txt" file. (Flights Database class assignment, 2012)

Presumably, from the instructor's directive, one "will need" to have an array of airports (mapping 3-letter code to full airport name), an array of routes (mapping a pair of airports to a unique routing number), and an array of flights (mapping unique flight numbers to a collection of information about that flight).

Rebecca creates no arrays. Instead, her code attempts to accomplish the same task that a memory-persistent data structure would, only without the persistence. A consequence of Rebecca's approach is that she has no easy way to refer to arbitrary airports, routes, or flights in her code, since her program has no mechanism to store such information persistently. A second consequence is that since she avoids persistent data structures, the complex work her program does to read through each line of each file, in some cases multiple times (above) is repeated every single time a user initiates a query.

Given Rebecca's particular design pattern, we asked the question of whether she may have tried creating arrays before ultimately settling on her scanning-loop solution. The answer, as far as we can tell, is no. We analyzed the history of both Rebecca's main() method and her one-stop flight code module. Our search revealed that no snapshots exist in which Rebecca created arrays—either through dynamically allocating them (through heap memory), or, as the assignment recommended, creating overprovisioned fixed-size arrays on the stack. In other words, at the limit of resolution of our data collection, and within the scope of the code Rebecca typed, she never tried an array solution.[24]

Curiously, we have evidence *outside* of Rebecca's code that suggests she knew, and even perhaps had seen, an array-based design solution. In a file called "notes.txt" contained in her project directory, we see the following lines:

```
think about using: sscanf, array of pointers


his header file!!!
```

[24] If Rebecca had tried an array solution and compiled—whether error-free or not—our automated snapshot collection system would have captured it.

```
-max line lenght: 2000
-max string lenght: 100
-defined true and false
-max airports: 100
-max routes: 500
-max flights: 3000
-min connect time: 60.0
-max connect time: 120
-daily maxk: 254  ???
-char airports[max airports][4]
-char aiport_cities[max airports][max string lenght]
-he has 3D array for routes.... char routes[max routes][2][4]
(notes.txt file, created March 19, 2012)
```

The context of the file is not entirely apparent, because we did not observe lecture on March 19, the day the notes.txt file entered Rebecca's snapshot history. Also, whether "his" refers to the instructor or perhaps another classmate is unclear. —What seems clear, however, is that Rebecca was responding to items she had seen in someone else's header file. Consequently, putting together the notes.txt file with Rebecca's final code submission reveals Rebecca was exposed to a design solution involving arrays, but never implemented it in her code. Thus, a lingering and consequential question remains unanswered: why did Rebecca adopt a solution that defied the directions of the assignment, especially when she'd seen part of a potential design solution that *did* use arrays?

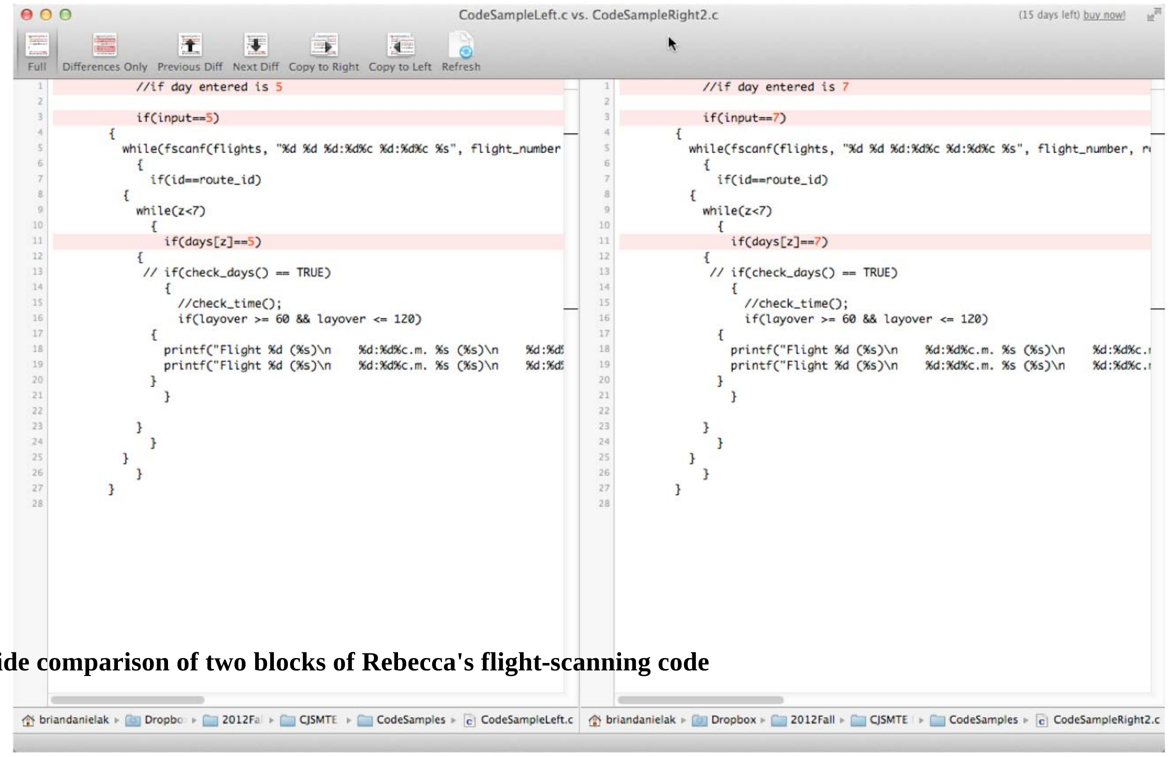
We return to this question in a later section, but first we turn our attention to another unusual feature of Rebecca's work: seven-fold repetition of code.

2.4.3 Rebecca repeats the same chunk of code seven times

A second key feature of Rebecca's code is the almost identical repetition of a single 23-line code chunk seven times (lines 50–240). Because users can run queries by choosing a day to fly (and some flights only run on certain days), students' code must be able to handle each of the seven possible days for when a user would want to fly. In principle, Rebecca's code achieves just that.[25] In practice, her code creates seven different conditional branches—one branch for each day of the week—where the code within each branch is duplicated.

[25] I say "in principle" because Rebecca's code would not compile on my machine. So, in practice, her design contains compile-time errors (and possibly run-time errors). Nevertheless, her code provides ample evidence that she was attempting conditional logic to handle each possible day.

Figure 2-1 represents a side-by-side delta-comparison of two such day-


CodeSampleLeft.c vs. CodeSampleRight2.c
(15 days left) buy now!
Full
Differences Only
Previous Diff
Next Diff
Copy to Right
Copy to Left
Refresh
briandanielak ▸ Dropbo ▸ 2012Fa ▸ CJSMTE ▸ CodeSamples ▸ CodeSampleLeft.c
briandanielak ▸ Dropbox ▸ 2012Fall ▸ CJSMTE ▸ CodeSamples ▸ CodeSampleRight2.c

```
1            //if day entered is 5
2
3            if(input==5)
4        {
5          while(fscanf(flights, "%d %d %d:%d%c %d:%d%c %s", flight_number
6            {
7              if(id==route_id)
8          {
9            while(z<7)
10             {
11               if(days[z]==5)
12           {
13            // if(check_days() == TRUE)
14               {
15                 //check_time();
16                 if(layover >= 60 && layover <= 120)
17             {
18               printf("Flight %d (%s)\n    %d:%d%c.m. %s (%s)\n    %d:%d
19               printf("Flight %d (%s)\n    %d:%d%c.m. %s (%s)\n    %d:%d
20             }
21               }
22
23           }
24             }
25         }
26           }
27       }
28
```

```
1            //if day entered is 7
2
3            if(input==7)
4        {
5          while(fscanf(flights, "%d %d %d:%d%c %d:%d%c %s", flight_number, r
6            {
7              if(id==route_id)
8          {
9            while(z<7)
10             {
11               if(days[z]==7)
12           {
13            // if(check_days() == TRUE)
14               {
15                 //check_time();
16                 if(layover >= 60 && layover <= 120)
17             {
18               printf("Flight %d (%s)\n    %d:%d%c.m. %s (%s)\n    %d:%d%c.
19               printf("Flight %d (%s)\n    %d:%d%c.m. %s (%s)\n    %d:%d%c.
20             }
21               }
22
23           }
24             }
25         }
26           }
27       }
28
```

2-1 – A side-by-side comparison of two blocks of Rebecca's flight-scanning code

specific branches of code. Lines 158–184 of Rebecca's original code are on the left;

lines 214–241 are on the right, and in the figure lines have been renumbered (from 1) to ease comparison. In this delta view, lines that differ are highlighted in pale red, and characters that differ are shown in bold red.

The two code blocks demonstrate just how much code is duplicated for handling user input based on days of the week. Between these two chunks there are only three differences (lines 1, 3, and 11): all of the references to day are changed from 5 (on the left) to 7 (on the right).[26] Moreover, the changes from block to block are patternistic and predictable: the first line of the block is a non-functioning comment, the third line of each block just checks whether the rest of the block should run, while the eleventh line of each block compares an array entry to the day of interest. Everything else is duplicate boilerplate that is essentially repeated 7 times; once for each day of the week. I say "essentially repeated" because, as we'll now explore, there are minute differences between some of the code chunk's seven incarnations.

Repeating code as Rebecca has done can be problematic because each repetition multiplies the number of places she has to examine and modify if she wants to introduce a systematic change. If, for example, Rebecca wanted to change the internal names she gives to scanned-in variables, she has to make that change in

[26] In the text-input files students were given, days of the week were represented as integers (rather than the perhaps more familiar "Tuesday," "Wednesday," etc.).

seven different blocks of code: once for each of the seven days of the week she's hard-coded. And, since any given change may inadvertently introduce an error, increasing the number of places she repeats code also makes the code that much more vulnerable to inconsistently-applied changes.

Indeed, a repeated, inconsistently-applied scan pattern change seems to be exactly what occurred in Rebecca's code history. Between 10:04pm and 10:37pm on March 26, Rebecca introduced a large set of changes to the one-stop flight module. Among those changes Rebecca added the `d_letter` file-scanning-parameter to what would become line 218, but not to what would become line 190. We can reasonably infer Rebecca added this parameter as a way of capturing the "am" or "pm" specifier given the input file's format. Moreover, we can verify through Git that once introduced, Rebecca's omission of the parameter was never modified or corrected. The problem percolated through to her final submitted code.

2.5 Interview Data and Analysis

In the previous section, I described two unusual features of Rebecca's code for searching one-stop flights:

Her use of multiply nested loops that scan through source information files *without* storing the information in those files persistently in long-term memory

The code for handling a user's chosen day, which was essentially the same block of code copied and pasted 7 times

In this section, I offer explanations of Rebecca's design choices by interpreting data from over five hours of clinical interviews I conducted with her. I draw from those interviews to explain how design decisions that might seem unusual to an expert in fact grew rather unproblematically (for Rebecca) as ways of deliberately transferring prior knowledge and designs (which explains feature 1) or coping with a constraint to produce a reliable solution she could trust (which explains feature 2).

2.5.1 Rebecca employed fscanf loops because she was deliberately reusing from an Basic Programming Assignment.

Rebecca's choices become easier to understand when you consider what she said in the interview. When Rebecca initially saw the flights database project, her first reaction was "this is just a lot like our fantasy football project that we did last year" (Interview, March 16, 2012).[27] But, the previous fantasy football project was easier because all of the information she needed was in one file. Flights database, by comparison, fractured necessary information across multiple input files.

Since Rebecca said the fantasy football project was like an easier version of flights database, I asked whether she tried making this project more like the easier one. Her response was an emphatic "Oh yeah! Definitely!" (Interview, March 16, 2012). She said "as soon as we got this project I was like, ah! fantasy football! I'm just gonna go and see how much code I can rework from that and like, use /mmhmm/ in this project" (Interview, March 16, 2012). Specifically, Rebecca went back to her fantasy football code "and I was looking at how I scanned in the information from the

[27] Transcript conventions are shown in Appendix 1.

files, cuz, we haven’t really done anything like that this semester. Uh, scanning in from files before—” (Interview, March 16, 2012). Since the fantasy football draft project was the last in which she’d needed to do file-scanning, she “just, uh, went back to check how I did that /mmhmm/ and then, if I could I copied, but because a lot of the variables were different, uh, like these were more var—less, less variables, and more strings than last year /mmhmm/ uh, I just retyped it out. I just looked at how it was similar” (Interview, March 16, 2012).

My first opportunity to discuss Rebecca’s one-stop flight work was on March 16, 2012, in what would be her third of five interviews that semester. This interview was conducted very early into the time window for the Flight Database project, before Rebecca had done the bulk of her coding. We were discussing her prospective design plans. As Rebecca began explaining how the logic for a one-stop flight search was supposed to work, she described what she saw as one of the central difficulties of the project: the relevant information for answering user queries was spread across multiple files (Interview, March 16, 2012).

As Rebecca explained, something as simple as finding a flight from, say, JFK to BWI “involves scanning through multiple files, because it’s not like one file that has everything conveniently like, there” (Interview, March 16, 2012). When I asked what would make things easier if, hypothetically, all the information she needed *were* in one file, Rebecca responded by appealing to a previous assignment from last semester. In Rebecca’s first-semester programming course (Basic Programming for Engineers), one of several multi-week projects had students create a system for users to conduct a fantasy football draft. The project involved, among other things, topics related to basic file management in the C language, including how to read in a file from disk (Interview, March 16, 2012). Rebecca explained:

> Rebecca: Um, like, cuz when we first got this project, uh, I actually was thinking “oh, well this is just a lot like our fantasy football project we did last year.” /Huh/ We uh, had to scan in, uh, someone had to enter like “I wanna pick a quarterback,” so then you had to scan in and go and look for all the quarterbacks in the file and say “OK, this is the quarterback” and everything. But, in that project we only had the one file that had everyone listed: quarterbacks, runningbacks, wide receiver, in one file. And all the information you needed there /Mmhmm/ So, you could just get it all and compare it all at once with one scanf /mmm/ whereas this you have to, take, uh, you scan in the flights, ah, the flights file. So, then you find the flight number. You have to save the ID from that flight number, use that ID to scan into the routes file /mmhmm/ and then save the routes information and then print it out with the flights information. (Interview, March 16, 2012)

Rebecca’s comments suggest she saw a coupling between the arrangement of the input information and the structure (and complexity) of the computational logic needed to process it. When one fantasy football file contained all of the relevant information (player, position, team, etc.) it could be read in and processed one line at a time. When information was fractured across files (pairs of airport codes in one file, full spell-outs of airport names in another, for example), Rebecca felt she’d need to use information shared across files (such as a route ID) to coordinate a scan across one file with a scan across another (and possibly an additional scan across a third file) before she would have all the necessary information and computations to return a result.

I was interested in the connections Rebecca saw across projects, so I pressed on. When I asked whether she thought about trying to make this project like her fantasy football project, her answer was an emphatic “Oh yeah, definitely!” As she elaborated:

> Rebecca: That was like, as soon as we got this project I was like, ah! fantasy football! I’m just gonna go and see how much code I can rework from that and like, use /mmhmm/ in this project. And, my whole main file, like all those NULL checks and everything, I mean they’re really simple to write, but I just copied ‘em and put ‘em there, cuz, we had the same thing. /Mmhmm/ Um, just changed, like, the names of the files.

As she explained, “reading in files” was a topic covered extensively in Basic Programming—the first course of the sequence—but they hadn’t talked much about it in this semester’s course.

> Rebecca: Because [reading in from files] was a big [Basic Programming] topic we hadn’t talked about it much. /Yup/ So, I just, uh, went back to check how I did that /mmhmm/ and then, if I could I copied, but because a lot of the variables were different, uh, like these were more var—less, less variables, and more strings than last year /mmhmm/ uh, I just retyped it out. I just looked at how it was similar. (Interview, March 16, 2012)
>
> I started doing the scanning, and then I realized like, “I do not have all the information I need in this one file,” so that’s when I realized I would have to make the dummy variable to compare to the other file, so, which, there was no other file in the, um, fantasy football one. So that’s when I realized, I was like OK, so, there’s gonna be other stuff that I’m gonna have to do, and /mmhmm/ I mean, I can still kinda compare cuz it’s still scanning files /yup/, but the idea of the dummy variable and all that, that would not have come from fantasy football, so /sure/.

In sum, then, Rebecca’s repeated, nested scan loops were a structure she deliberately borrowed from a previous semester’s project. By her telling, what seemed obvious was that “scanning in from files” was a topic she’d already covered, which meant she’d already developed a workable solution for how to solve that problem. Thus, she saw the problem of how to coordinate airport information from different files as a new instance of the old problem of reading information in from one file. Her flight database work, accordingly, tried opportunistically adapting a previously working solution to fit the current circumstances.

2.5.2 Rebecca repeated code because she wanted to re-use functionality she could trust

By our interview on April 6, Rebecca had already completed and submitted her code for the flights database project. When I looked at the final form of her code for finding one-stop flights I noted an unusual pattern described in section 2.4.3 above: she had a code chunk repeated almost character-for-character 7 times. In the interview, this section is what Rebecca referred to as "my obnoxiously long part of my code" (Interview, April 6, 2012):

> Rebecca: So, the way I did it was really long and probably, there was probably like a much easier way, but I just did a giant if—if statements {swings cursor from line 50 to line 63} If they wanted to fly on Monday /OK/ I went through and checked to see if the route ID was the same {wiggles cursor across line 54}, and if it did, I went through to che—uh, I made a check_days function {wiggles cursor across line 60} uh, I ended up commenting that out cuz I didn't end up /mmhmm/ finishing it. But, uh, my check_days function worked, it just didn't work completely with the code /OK/ (Interview, April 6, 2012)

As I scrolled the screen to look at each of the repeated blocks of code, Rebecca elaborated:

> Rebecca: And this is why my code, I feel like, is not uh, concise enough, or, I don't really, I forget the word they use, but uh /{inaudible}/ it's very long because I couldn't figure out if I should do a while loop or whatever /Uh-huh/ But, so I was just like, I know this way should work if I get everything else right, that uh, just go through, if input's 1, if input's 2 and just do the same thing in each of 'em just /Mmmhmm/ check for, "oh, if days is 2, if days is 1" instead of, like—Cuz I probably could have done, like, maybe a giant while loop, um, to try and, and if, while, inputs something, uh, then you check to see whatever i is. But, I could, I didn't—couldn't figure out how that would work, so I just did the same thing six times.
>
> Interviewer: So, in, in each one of these it's like, looks, and I'm not sure about this, but it looks like the way you wrote it—so this {highlights line 79} is pretty much the same in all of them, right? /Yes/
>
> Interviewer: So's this one {highlights line 81} /Yes/ this one {highlights line 83} Here's where it's different {highlights line 85}
>
> Rebecca: Yes, because it just checks if it's a 2 instead of a 1.
>
> Interviewer: OK. Um. /And then everything else is still the same/ Layover's still the same. OK.
>
> Rebecca: Yeah. So that's why it's prob—it's not, uh, the neatest code or whatever, because it's the same thing six times. (Interview, April 6, 2012)

Given Rebecca's assertions that her code wasn't neat, I asked what, if anything she might change if she hypothetically had another week to work on the project.

> Rebecca: Um, first I'd try and get it to make sure it worked completely /Ahh, OK, yeah/ this way, {laughs}, uh, and then, if I had the week after that whatever, I'd probably go through and see if I could figure out a way to make it concise-r because he likes uh, neat, as, like, code that's, uh, easy for the user to see /uh-huh/ I guess. Uh, I forget what, I keep forgetting what the word he used was at the beginning of the year, but uh, just very concise and, uh, this is [a] very expanded {laughs} way of coding, but, it made sense to me at the time and I was just like "I just want something that makes sense right now." /Right/ So, that I can actually work with and have an idea.
>
> Interviewer: Um, OK. So, so it would take you some extra thinking to figure out /Mmmhmm/ how to break this down into /Yes/ smaller stuff /smaller code/ Do you feel like you've had a lot of practice doing that, or like?
>
> Rebecca: Uh, a little. Like, but, a lot of times in [Basic Programming] they didn't really mention too much about being concise. They were just like "if you can do it, do it" {laughs} /OK/ So I usually stuck to what made sense to me /Right/ uh, to turn the projects in. (Interview, April 6, 2012)

In summary, Rebecca's "expanded way of coding" was a way of expressing ideas in code that, in her own words, "made sense" to her. Moreover, her Basic Programming course seemed, to her, to set expectations that functionality comes first; "neatness" second. If she hypothetically had more time to work on the project, her first priority would be to get her existing code working. Consequently, Rebecca's repetition of code can be understood as a kind of pragmatic solution to a difficult problem: choosing which computational techniques were best for accomplishing a complex goal. Moreover, her approach was shaped by the fact that her Basic Programming course historically valued a philosophy of "if you can do it, do it" (Interview, April 6, 2012). Ultimately, those factors seem to be what led Rebecca to choose repeating code that made sense to her over the difficult-to-envision alternative of a "giant while loop."

2.6 Discussion

When we consider evidence from both code snapshot histories and clinical interviews with Rebecca, we can draw several plausible conclusions about Rebecca's patterns of software development on this project. Moreover, those conclusions can spur larger considerations about theorizing student learning and considering alternative instructional strategies that may be more responsive to student needs. Below, I synthesize and discuss patterns in Rebecca's development.

2.6.1 Rebecca’s key design decisions are made early, persist through to her final submission, and carry consequences

On both her non-stop flight code and her one-stop flight code, key structural features of how Rebecca’s code works never meaningfully change after Rebecca introduces them. In particular, Rebecca’s use of fscanf() patterns (as opposed to storing the data persistently in memory) occurs extremely early in the work history of the project. The fscanf() patterns initially appear in Rebecca’s flight-checking code on March 19, the third of what would be 286 chronological snapshots we have comprising Rebecca’s work on the project.[28] We note several consequences that arise *because* these design choices were made early and persisted to the final code.

First, Rebecca’s *early* choice not to persistently store data likely forced her into the complicated scanning loop logic she developed *later* to solve the one-stop flight problem. A bit of explanation may help here. Because the database students were building has multiple uses, including uses that find flights for users, to be successful Rebecca had to write functionality that found both non-stop and one-stop flights for users. But, Rebecca initially designed each part of her code—airport listings, flight listing, non-stop flight search, and one-stop flight search—with its own internal fscanf() loops, as opposed to having the four of them each refer out to a separate module that could handle reading in data. The result is that none of her individual modules share file-scanning code, and repeat data look-ups necessarily have to be handled by creating another nested scan loop. A design where code with a common function is pasted repeatedly into separate modules contrasts strongly with one in which separate modules all refer out to a single external piece of code.[29]

We can understand Rebecca’s development as involving a kind of design inertia. Because she decided from the start to scan information using fscanf() separately in each module, her later code was also beholden to that commitment. The end result is that increasing the problem complexity a small amount (the challenge of handling one-stop flights versus non-stop flights) forces her to write much more code that is itself more complex in its flow of control. And, when we consider her seven-fold repeated code for checking flight availability on given days, there are similar consequences. The decision—whether intentional or not—to create seven different conditional paths forces Rebecca to write and maintain much more code over time as the project evolves. As we suggested in section 2.4.3, having a large number of

[28] The project was distributed to students on March 5 and was due March 28, but the bulk of Rebecca’s work on the project (284 of the 286 snapshots) occurs beginning March 19.
https://github.com/TLPLEngineeringEdResearch/Rebecca/compare/e0a7caab4eb2d8401a0e50e8c6c6d8b514e9054e...0e8a5c606ef30f3b404024fdfad17d8d50e32456

[29] As an analogy, imagine a supermarket where every unique type of item on the shelf has a cash register specific to that item next to it. To buy an item shoppers have to check-out individual items as they move through the store. It’s a pain for shoppers because shoppers have to swipe their cards at dozens of separate item-specific registers across the store just to pay for a single grocery run. It’s a pain for store-owners because now there are literally hundreds of individual registers to manage, maintain, and upgrade.

locations to keep in sync is likely what made Rebecca vulnerable to one of the errors she ultimately introduced into her code.

2.6.2 Her design decisions may be influenced by framing and overzealous transfer

Rebecca said in our interviews that she explicitly tried to reuse solutions from the previous semester's fantasy football project. Our code snapshots help us understand the extent to which that's true. Rebecca's first compile for her flights database work begins with this commented line of code:

```
//Rebecca Wells's Fantasy Football Team Maker: Project 1
```

Thus, our interview with Rebecca corroborates the snapshot evidence that Rebecca directly copied code from a prior project. Moreover, Rebecca's behavior of copying her old fantasy football code suggests several implications.

First, to extend our point in section 2.6.1, Rebecca's "design inertia" actually traces as far back as the decisions she made during the first project of her first semester of programming. That is, by copying code (and particularly file-scanning logic) from an old project, she was incorporating core functionality that she designed when she first learned to program. Admittedly, code reuse isn't in and of itself a problem in software development. Parson and Saunders (2004), for example, have written on cognitive heuristics that keep professional software engineers from reusing code, even when reusing and extending existing software artifacts is the best course of action on a software project. So, the concern isn't that Rebecca reused code, but rather the matter of what cognitive dynamics were at play that directed her choice to reuse that code. In that regard, the constructs of framing and transfer may help us better understand Rebecca's activity.

Framing, as it has been applied in contexts such as physics education (Elby & Hammer, 2010; Hammer et al., 2005; Scherr & Hammer, 2009) and mathematics education (van de Sande & Greeno, 2012) concerns how participants understand the social and intellectual activities in which they're engaged. Of particular relevance to studying Rebecca is the notion of *epistemological framing*, which van de Sande and Greeno (2012) summarize as

> participants' understanding of kinds of knowledge that are relevant for use in their activity and the kinds of knowledge, understanding, and information they need to construct to succeed in their activity (e.g., what kind of information would count as a solution to the problem they are working on). (van de Sande & Greeno, 2012, p. 2)

Rebecca's decision to copy code from fantasy football implicitly reflects her orientation toward what kinds of knowledge (the course topic of scanning information from files) are relevant to solving the problem. More broadly, note that in Rebecca's interview she explains that her primary objective is to get a solution that works, which stands in contrast to having a design priority like having a solution that is elegant, or one that transparently manages complexity. That commitment again reflects a manifestation of epistemological framing as “what would count as a solution,” where for Rebecca what counts is a solution that works.

Additionally, Rebecca saw the flight database project as a new instance of a prior problem: fantasy football. Consequently, she consciously adapted past solution patterns, because the flights database project looked like it contained problems she had already solved in previous code. Rebecca’s deliberate reuse of old code can be readily understood as an example of what Schwartz, Chase, and Bransford (2012) call “overzealous transfer”:

> Of particular concern are situations where students transfer skills, knowledge, and routines that are effective for the task at hand but may nevertheless be sub-optimal in the long run because they block additional learning. We will call this *overzealous transfer* (OZT)—people transfer solutions that appear to be positive because they are working well enough, but they are nevertheless negative with respect to learning what is new. (Schwartz et al., 2012, p. 206)

In short, when students overzealously transfer prior knowledge as Rebecca did, “they may believe they are doing the right thing, and without appropriate feedback they cannot know otherwise” (Schwartz et al., 2012, p. 206).

In Rebecca’s case, the constructs of framing and overzealous transfer together let us describe why she would have repurposed a solution that she felt was adequate, even to the exclusion of the topics being taught in class and the explicit directions in the project brief. By framing the flights database problem as a new instance of an old problem, Rebecca treated it as she did the old problem. But, she arguably transferred *too much* of the old code’s structure; so much that she had to introduce even more complexity into her code just to make the transferred parts work properly under the new constraints. And, because her framing of the task seemed to privilege a philosophy of “if you can do it, do it” (Interview, April 6, 2012), her primary goals were to get her program to work, by whatever means she could understand and trust.

2.7 Conclusion

In closing, even with a complete snapshot history and over five hours of clinical interviews, it’s still an enormous challenge to thoroughly understand one student’s design trajectory on a project. The challenge is both methodological and theoretical. Through interviews we know what Rebecca was thinking retrospectively, but we still work at a remove from understanding what she was thinking in the precise moments she wrote her code. Conversely, our snapshots provide a detailed record of the state of her code every time she compiled. But, our snapshots are limited by the kinds of materials they can track and the frequency with which they’re taken. We don’t know, for instance, what Rebecca might have been talking to others about while sitting in a dorm lounge working on her project. And, because Rebecca wouldn’t always compile frequently, some snapshots detail large changes to the code made over long periods of time—hours or days—which can limit our ability to precisely determine what happened, and when.

From a theoretical perspective, we find our attempts to understand her design decisions involve appeals to constructs of a finer granularity and dynamism than are typically considered when studying first-year programming courses. Understanding why fscanf() loops dominate Rebecca’s flight search code, for example, pushes us to consider the effects of how Rebecca framed the task and the subsequent overzealous transfer that resulted. And, understanding why code gets repeated seven times pushes us to consider how emotion couples with sense-making in students’ design decisions. In Rebecca’s case, a time-pressured preference for code that “makes sense right now” translated to a design of explicitly repeating procedures instead of elegantly abstracting them.

3 Study 2 – What directs and sustains students' in-the-moment programming activity?

3.1 Introduction

> Although imaginative and carefully designed, the Harvard course taught that there is only one right way to approach the computer, a way that emphasizes control through structure and planning. There are many virtues to this computational approach (it certainly makes sense when dividing the labor on a large programming project), but Lisa and Robin have intellectual styles at war with it. Lisa says she has "turned herself into a different kind of person" in order to perform, and Robin says she has learned to "fake it." Although both women are able to get good grades in their programming course, they represent casualties of this war. Both deny who they are in order to succeed. (Turkle & Papert, 1991, p. 165)

Over 20 years ago Sherry Turkle and Seymour Papert called for epistemological pluralism in computing education. Their argument — in my view — had multiple parts:

1. Programming a computer involved a shift in how one thought about the nature knowledge from a propositional *what is* to an imperative *how to* (Abelson & Sussman, 1996; Papert, 1980).
2. Historically, there was a tendency to treat computers and the act of programming them as an extension of the same Western logico-deductive knowledge tradition that fueled science.
3. Recent scholarship, often at the intersection of post-structuralism and social studies of science, had been troubling the idea that knowledge-work in science proceeded solely by way of logical deductions (Keller, 1983; Latour, 1987; Traweek, 1988)
4. Data from professionals suggested programming, too, didn't proceed solely by way of the structured-planning approach emergent from Western logico-deductive knowledge traditions. There were many ways of knowing and constructing knowledge in programming.
5. A viewpoint of *epistemological pluralism* — embracing multiple kinds and ways of knowing — was informing the field of science studies but not, to a large degree, how education researchers thought about or taught computing.
6. The lack of support for diverse ways of knowing in computing classrooms demonstrably hurt students.

To Turkle and Papert (1991), the result of this culturally-rooted epistemological tension was a war. More specifically, it was a war where students of computing were the casualties, and the aggressor was a distributed failure to recognize, embrace, and support diverse ways of knowing. While I think their strong-form characterization of a "war" no longer applies, their consideration of epistemological issues is still relevant two decades later. An understanding that computing does in fact involve diverse ways of knowing should continue to inform research and discussions on learning

This paper tries to model what directs and sustains students' in-the-moment activity when they program. Its focus is on early-stage program design; analyzing how what students say, do, write, and gesture *even before they type code* can help us improve theories of cognition and activity in learning programming. Using a "revelatory case study" (Yin, 2009) of two students in an introductory programming course, I argue the following:

1. Students' early-stage design activity reveals patterns outside the explanatory scope of misconception-based accounts of cognition.
2. For a subset of phenomena, we can recast students' productive capacities and their difficulties in terms of epistemological stances.
3. Such recastings are analytically powerful. Beyond Rebecca and Lionel, they could potentially explain the diversity of practices I saw other students take up.
4. As evidenced by work in science and math education, dynamic epistemological models can offer a lens for reforming assessment and instruction.

3.2 Literature Review

I begin by exploring the historical legacy of Turkle and Papert's (1991) remarks on epistemological pluralism in computing. I'll then describe what I see as a well-intentioned obstacle: the research-based preoccupation with students' misconceptions in computing education. These two stra.

3.2.1 Computing culture should support a diversity of ways of knowing

Turkle & Papert (1991) describe a tension between individuals and a kind of cultural collective. In this tension, individual students and their personally-identified ways of knowing were in conflict with the larger culture of computing and its manifested ways of knowing. Lisa, for instance, was a Harvard student who identified as a poet:

> Lisa experiences language as transparent, she knows where all the elements are at every point in the development of her ideas. She wants her relationship to computer language to be similarly transparent. When she builds large programs she prefers to write her own smaller "building block" procedures even though she could use prepackaged ones from a program library; she resents the latter's opacity. (Turkle & Papert, 1991, p. 164)

Turkle and Papert connect Lisa's programming practices to Lisa's personal view of knowledge. She writes building block procedures because of a continuity between herself as knower-of-her-own-poetry and that of knower-of-her-own-programs. Across both contexts, Lisa *resents opacity* because it is an obstacle to her knowing. But, that resentment slows her progress on projects in the class.

In part because of her commitments to transparency and the difficulty they created, Lisa changed her approach to programming. She ultimately had to, in her words, “be a different kind of person with the machine” (Turkle & Papert, 1991, p. 164). Turkle and Papert explain:

> She had been told that the “right way” to do things was to control a program through planning and black-boxing, the technique that lets you exploit opacity to plan something large without knowing in advance how the details will be managed. Lisa recognized the value of these techniques — for someone else. She struggled against using them as the starting points for her learning. Lisa ended up abandoning the fight, doing things “their way,” and accepting the inevitable alienation from her work. (Turkle & Papert, 1991, p. 164)

The phrase “inevitable alienation” is particularly damning. One might accuse Turkle and Papert (1991) of hyperbole, but research from other disciplines suggests they’re not overstating the case. Identity alienation for epistemological reasons — or some variation of them — is playing out in other subjects as well.

Connections and tensions between identities, learning, and personal ways of knowing are not unique to computing education. Wortham (2006), for example, detailed instances of social identification and academic identity as *jointly emerging* in a combined high school English and History class. Nasir and colleagues explored how different learning environments afford access to identity, domain practices, and self-expression (Nasir & Cooks, 2009; Nasir & Hand, 2008). To single out one study, Nasir & Hand (2008) followed players “from the [basketball] court to the classroom.” They found the court afforded players a sense of role on the team, chances to be expressive through play, and “access to the domain [of basketball] as a whole” (Nasir & Hand, 2006, p. 147). Comparatively, the mathematics classroom offered little (if any) sense that students were on a team, a much narrower role for them as having either right or wrong answers, and no strong sense of self-expression or creativity through activity. These markedly different contexts led not only to differences in kinds of activity but, Nasir and Hand (2006) argue, divergent *practice-linked identities* for the players. In a sense, players had a different identity on the court than they did in the mathematics classroom because who they were was strongly coupled to a sense of their relationship with a practice.

Boaler’s work on learning and identity in mathematics classrooms (Boaler & Greeno, 2000; Boaler, 1998, 2000, 2002) offers resonant findings. In particular, Boaler (2000) strongly echoes the student voices in Turkle and Papert (1991). Where Turkle and Papert’s aggressor is the dominant computing culture, Boaler describes a beast with similar effects in the form of “school mathematics” — the *monotonous, meaningless* (Boaler, 2002, pp. 383–384) imposition of routinized mathematical procedures in ways that seem far removed from the real world. Of school mathematics, Boaler writes:

> School mathematics, for many of them, was *of another world* and to fully engage in that world, students needed to suspend their knowledge of the real world, suppress their desire to interact with others, and strive to reproduce

> standard procedures that held little meaning for them. (Boaler, 2002, p. 392 emphasis in original)

Schoenfeld (1988, 1991) had already argued school mathematics could and did depart substantially from the core of mathematical thinking. Boaler established that such school mathematics could alienate learners (Boaler, 1998, 2000) and influence their feelings about pursuing mathematics after secondary school (Boaler & Greeno, 2000).

Most recently, Danielak, Gupta, and Elby (in press) extended those findings in engineering education. In their 3 ½ year case study of an electrical engineering undergraduate named "Michael," those authors argued Michael's practice of sense-making in engineering coursework was strongly coupled to his identity. Michael's passion for deeply understanding concepts, much like Lisa's (Turkle & Papert, 1991) desire to understand every element of her program, did not mesh neatly with the larger culture of his courses. But, because Michael's practice of sense-making entwined with his identity, forces that pushed against sense-making alienated him. That tension was particularly pointed when Michael described his father's resistance to Michael's way of thinking:

> [My dad said] "you're just an undergraduate. Nobody expects undergraduates to understand how anything works. That's why you go to graduate school." I was like "look, you know. I'm gonna be an unhappy person if I have to....I have life goals other than to just get a good grade on the exam. Other things are important to me." (Danielak et al., in press)

And, much like Turkle and Papert's (1991) Lisa, Michael learned to curtail parts of his identity to get by in class:

> I think the reason [pursuing deep learning] actually hasn't affected my GPA is because I view learning as a hobby. So, as with any hobby, you shouldn't let it interfere your GPA. But it *is* one of my hobbies, and I *do* enjoy learning, I just—up to the point where I get my grades done {raises eyebrow}. (Danielak et al., in press)

Ultimately, research suggests the identity-epistemology tensions Turkle and Papert (1991) described in computing resonates with those found in related disciplines. And, what such accounts — particularly those of Boaler and others (Boaler & Greeno, 2000; Boaler, 1998, 2000, 2002; Danielak et al., in press) — have in common is a culturally-sanctioned "right way" (Turkle & Papert, 1991, p. 164) of thinking or knowing that alienates some students. Avowedly such ways of knowing have use and value. Turkle & Papert themselves acknowledge "there are many virtues" to black-boxing and top-down design in programming (Turkle & Papert, 1991, p. 165). But,

1. When a preponderance of results from science studies suggests a plurality of ways of knowing among practicing scientists and programmers, and
2. When research from education shows that the inflexible imposition of one way of knowing above all else is alienating students, then

3. It’s worth reconsidering whether a narrow sense of what counts as knowing might still be hindering improvements in computing education.

In the next section, I turn to misconceptions research in computing education. In keeping with observations 1–3, I argue research approaches that inflexibly privilege canonical knowledge do so at the expense of other productive knowledge and ways of knowing that students have.

3.2.2 Misconceptions research in computing education tends to ignore students’ productive knowledge

In the past three decades, educational research has had a marked focus on students’ misconceptions in programming. It’s a focus with a sensible origin. Students get things wrong in programming — often systematically so — and in ways that seem resistant to instruction. The cause of those errors is theorized to be something cognitive, whether it’s a “bug” (Pea et al., 1987; Pea, 1986; VanLehn, 1990) a “misconception” (Bayman & Mayer, 1983; Bonar & Soloway, 1985; Clancy, 2004; Gal-Ezer & Zur, 2004; Herman et al., 2008; Kaczmarczyk et al., 2010), a “belief” (Fleury, 1993) or a “student-constructed rule” (Fleury, 1991, 2000). Instruction should try to identify, address, and correct these misconceptions (Clancy, 2004) because they can represent barriers to learning.

How that line of thinking and research becomes problematic is two-fold. First, when taken in total the alleged brokenness of student knowledge begins eclipsing all else in studying the cognition of learning to program. In other words, most cognitively-focused educational research in computer science treats students as having varied degrees of deficiency with respect to canonical knowledge. Below is an unordered, partial sampling of topics about which researchers have documented students’ misconceptions. Note across the list the variation in both the grain sizes of students’ misconceptions and the programming languages in which they manifest:

- Objects in object-oriented programming (Holland, Griffiths, & Woodman, 1997)
- Algorithms and data structures (Danielsiek, Paul, & Vahrenhold, 2012; Paul & Vahrenhold, 2013)
- Programming statements in BASIC (Bayman & Mayer, 1983)
- Programming in Java (Fleury, 2000)
- Programming in Pascal (Fleury, 1993)
- Parameter-passing (Fleury, 1991)
- Arrays in Java (Kaczmarczyk et al., 2010)
- Objects in Java (Kaczmarczyk et al., 2010)
- Algorithms and computational complexity (Trakhtenbrot, 2013)
- Boolean logic (Herman et al., 2008)
- The efficiency of algorithms (Gal-Ezer & Zur, 2004)
- The Build-Heap algorithm (Seppälä, Malmi, & Korhonen, 2006)
- Hashtables (Patitsas, Craig, & Easterbrook, 2013)
- The correctness of programs (Kolikant & Mussai, 2008)

Clancy (2004) provides a comprehensive overview of this line of research, though in the past decade it has only grown. Indeed, roughly half the articles above were published in the ten years since Clancy's overview.

Identifying and removing barriers to student learning seems like a good thing. So, it should follow that cataloging student misconceptions and developing remedies for them should also be a good thing. But, the logical implication isn't that clean. In some cases students display productive, useful knowledge that's either ignored or outright criticized by researchers. Aligning students toward canonical knowledge makes sense, but doing so at the expense of—or in direct contradiction to—useful ways of knowing seems undesirable at best. Next, I expand on two examples from misconceptions research in programming. Specifically, I show how and why I think a misconceptions focus in programming casts aside students' useful intuitions and understandings.

The first example comes from Kaczmarczyk et al. (2010). Part of that study involved giving students snippets of Java code and asking students to diagram (or pseudo-code) how the information would be stored in memory. Below I have reproduced the code for Problem 2 (Kaczmarczyk et al., 2010, p. 110):

```
Cheese[] cheeses = new Cheese[4];
Meat[] meats = new Meat[2];
Turkey turkey;
Ham ham;
RoastBeef roastBeef;
boolean lettuce = true;
boolean tomato = true;
SauceType sauceType = new SauceType();
int numMeat;
int numCheese;
```

In diagramming this information, a student in the study makes a mistake:

> Student3 makes incorrect assumptions about connections between variables to the extent that the student makes a mistake concerning the types of the variables. As a result, the student places Objects of different types in an array whose type matches none of them: "And so because there's two arrays, cheese and meats, uh, all those turkey and ham and roast beef are gonna be sorted into the meats array." (Kaczmarczyk et al., 2010, p. 110)

The researchers are correct in the sense that **`turkey`** and **`ham`** and **`roastBeef`** will *not* be sorted into the **`meats`** array. First, there is no code here that places **`turkey`** and **`ham`** in the array; there is only code that declares them as variables. Moreover, as written, an attempt to place **`turkey`** and **`ham`** and **`roastBeef`** into the array would fail. Because of type restrictions in Java, only objects of class Meat (or objects that inherit from class Meat) can go in the array. **`turkey`** and **`ham`** and **`roastBeef`** are, perhaps confusingly, references to object instances of classes **`Turkey`** and **`Ham`** and **`RoastBeef`**, so in the current snippet they cannot enter an array of type **`Meat`** because (1) they don't yet exist as objects and (2) even if they did exist, their types don't match the array's type. The authors call this misconception *semantics to semantics,* which occurs "when the student inappropriately assume[s] details about the relationship and operation of code samples, although such information was neither given nor implied" (Kaczmarczyk et al., 2010, p. 110).

Again, the researchers are right that the student is failing to describe the code in a way consistent with canon.[30] But, in their non-canonical thinking Student3 evidences potentially productive insights about design. Precisely *because* there is no code stating that **`turkey`** and **`roastBeef`** and ham are sorted into the array, the student is *inferring* that to be true. And, while that behavior is not what's happening, it *would* be sensible to design a program where specific instances of classes **`Turkey`** and **`Ham`** and **`RoastBeef`** could go into an array of type **`Meat`**. To do so, a designer could define **`Turkey`** and **`Ham`** and **`RoastBeef`** as inheriting from **`Meat`**.

Student3 has an idea about a relationship between entities where that relationship is not specified in the code. Kaczmarczyk et al. (Kaczmarczyk et al., 2010) focus only on the downside of the idea: the student fails to display a proper understanding of how arrays work in Java. Moreover, the student might be prone to similar mistakes of inferring information that does not actually exist in code. But, there is also an upside of this idea. Because Student3 is thinking about real-world propositions like turkey and ham being kinds of meat, they might be prepared to appreciate and discuss an object-oriented way to put turkey in a **`Meat`** array. But, that possibility is speculative conjecture. We can't know for certain whether Student3 could be tipped into a productive object-oriented design activity around the meats example because that question was not a focus of the research.

My second example of research that criticizes students' non-canonical understandings comes from Bonar and Soloway's (1983) study of Pascal programmers, part of which is discussed in Pea (1986). A student in Bonar & Soloway's study was asked to "Write a program which reads in ten integers and prints the average of those integers" (Bonar & Soloway, 1983, p. 12). In pseudo-code, she wrote:

```
Repeat
(1) Read a number (Num)
    (1a) Count := Count + 1
(2) Add the number to Sum
    (2a) Sum := Sum + Num
(3) until Count :=10
(4) Average := Sum div Num
(5) writeln (‘average = ‘,Average)
```

The interviewer then asked whether (1a) and (2a) were "the same kinds of statements." That interchange is reproduced here:

> Interviewer: Steps 1a and 2a: are those the same kinds of statements?
>
> Subject: How's that, are they the same *kind*. Ahhh, ummm, not exactly, because with this [1a] you are adding - you initialize it at zero and you're

[30] In this case, canon is the specifications and operations of the Java language and its compilers.

> adding one to it [points to the right side of 1a] which is just a constant kind of thing.
>
> Interviewer: Yes
>
> Subject: [points to 2a] Sum, initialized, to, uhh Sum to Sum plus Num, ahh - thats [points to left side of 2a] storing two values in one, two variables [points to Sum and Num on the right side of 2a]. That's [now points to 1a] a counter, that's what keeps the whole loop under control. Whereas, this thing [points to 2a] was probably the most interesting thing…about Pascal when I hit it. That you could have the same, you sorta have the same thing here [points to 1a], it was interesting that you cold have, you could save space by having the Sum re-storing information on the left with two different things there [points to right side of 2a], so I didn't need to have two. No, they're different to me.
>
> Interviewer: So - in summary, how do you think of 1a?
>
> Subject: I think of this [point to 1a] as just a constant, something that keeps the loop under control. And this [points to 2a] has something to do with something that you are gonna, that stores more kinds of information that you are going to take out of the loop with you. (Bonar & Soloway, 1983, p. 12)

Pea's (1986) interpretation? "Here, again, we see the student believing that the programming language knows more about her intentions than it possibly can" (p. 32).

As before, this student has an idea about relationships in code. Pea (1986) see the downside of her idea: believing PASCAL can understand shades of programmer intent when, in fact, it cannot. And again, that downside is real. It could cause trouble for this programmer later on if she expects PASCAL to interpret her intent and it cannot.

In defense of the student, the question — as asked — is vague. Are those statements the same *to whom and in what way?* Pea (1986) treats the data as though she meant "the same to PASCAL." Indeed, maybe she did, in which case his interpretation has traction. But, another interpretation is that she meant to herself, or to someone else reading the code. Those statements might not be the same *to her* because she treats (1a) as having a function of controlling iteration while (2a)'s job is to combine two numbers into a new sum.

These two purposes, which for the sake of description I'll call *keeping control* and *totaling up* are, in a sense, different. The PASCAL compiler (and runtime) does not differentiate them, but humans can. And, humans may well *want* to differentiate them. diSessa (1986) describes exactly this kind of differentiation as a consequence of separating the structural understanding of a programming language from a functional understanding of a language. As an example, he discusses the structure/function difference with respect to variables:

> The structural aspects of a variable in a computer language are given primarily by the rules for setting their values and for getting access to their values. These rules apply in all contexts. In contrast, a variable's functions might

> vary. Sometimes they might be described as “a flag” or more generally, as “a communications device.” At other times a variable might function as “a counter,” “data,” or “input.” (diSessa, 1986, p. 202)

The student in Bonar & Soloway’s (1983) study did not show evidence of understanding the structural similarities between (1a) and (2a) in her pseudo-code. And, those authors as well as Pea (1986) justly insist that similarity is important for students to understand. From a conceptual standpoint, seeing the structural similarity constitutes a part of “knowing” PASCAL. But, even if knowing PASCAL were not the goal, seeing the similarity helps one to take the perspective of a computing agent that has no means for discerning programmer intent. Such perspective-taking may help students avoid mistakes that arise from over-assuming what a computer “understands.”

The student did show evidence of understanding a functional difference between (1a) and (2a), but Pea (1986) does not remark on that kind of understanding at all.[31] Again, I claim this oversight is part of a subtle but observable trend in programming misconceptions literature. While, or perhaps because research has been so preoccupied warring with students’ problematic knowledge, it has sometimes failed to recover the productive knowledge (or resources for building it) students have. In this example, the student already has a grasp that syntactically similar statements could serve different conceptual purposes. Hypothetically, the student might use that information in design by calling up the “ = + 1” syntactical template when the situation seems to demand *keeping control*, while calling up “ = + number” when *totaling up* is the goal. And, the idea that structurally identical symbol templates can serve different functional and conceptual purposes fits precisely in line with Sherin’s (2001) theory of symbolic forms. For example, the symbolic forms *parts-of-a-whole* and *base+change* have different conceptual schemata. Parts-of-a-whole refers to the contributions of component entities while base±change describes a kind of accumulation. Specifically, the terms in base±change “play different roles” (Sherin, 2001, p. 534) But, the two distinct conceptual schemata share what I would argue is the same concrete symbol template of how to write an expression: = + .

The problem, for learning to program, comes in needing to fluidly interpret and write code in languages that may demand incommensurable, or at least distinct, conceptual schemata. As I show later in Table 4, three current programming languages make remarkably different use of the plus sign (+) as an operator. Crucially, some of the entailing ways to make sense of how + works in those languages don’t exist in Sherin’s (2001) catalog of conceptual schemata. In other words, I would argue there are conceptual ways a programmer may need to think about interpreting or writing a = + symbol template that Sherin doesn’t

[31] Also glossed over is, to me, another important difference: a programmer might not know in advance which numbers are being passed in to the sum statement. So, in advance the programmer can say nothing about how the value of **`Sum`** will change as the loop iterates. In contrast, the programmer knows exactly how the value of Count will change with each loop iteration.

enumerate.[32] To follow that implication, the canonical body of knowledge about which programmers must reason is itself fractured, because different languages design their operations around different symbolic and conceptual metaphors.

To return to misconceptions, what drives research on students' misconceptions is largely a need to get students to program computers and reason about computation in ways that are canonically correct. And, those are assuredly worthwhile goals. But, as I've argued, we can already identify cases where a narrow misconceptions focus is silent about or dismissive of students' useful intuitions. We can also identify the further problems whereby unilateral emphasis on one language's canon opposes the vocabulary of symbolic forms (and diverse conceptual schemata) expert programmers ultimately need (Sherin, 2001) Taken in total, that silence, dismissiveness, and narrow view of refining knowledge perpetuates a deficit-focused discourse about student's knowledge in computing.[33] Such a perspective also fails to address what aspects of students reasoning might get broken by attempts to "fix" such "misconceptions".

It's important to note that misconceptions research in computing didn't always treat students' non-canonical knowledge this way. I begin the next section by backtracing to some of the earliest work on students' cognitive "bugs". There, we find researchers talking more explicitly about what's useful in students' non-canonical knowledge—a stance largely absent in modern computing misconceptions research.

3.2.3 Not all cognitive programming bugs imply a problem with the student

What's curious about Pea's (1986) comments on Bonar & Soloway (1983) is that Pea drew different conclusions from the data than Bonar and Soloway did. Pea (1986) emphasized that when the student thought two semantically-equivalent assignment statements in PASCAL were different, it was a problem of egocentrism: "students assume that there is *more* of their meaning for what they want to accomplish in the program than is actually present in the code they have written" (p. 30). In other words, Pea treated the data as a fairly clear example of a class of bugs. Specifically, he saw a bug class in which students simply assumed the interpreter or runtime could infer the code author's shades of intent.

Bonar and Soloway (1983), by contrast, were less quick to make inferences either about the nature of the bug or about the intervention it entailed. Rather than jump to definitive conclusions, they were circumspect:

> It is not clear exactly how to react to the bugs we have uncovered in novice understanding of programming. In some cases it may be appropriate to design new languages or constructs. Often, better instruction would take care of the problem. The intent of our studies is to better understand the source of the

[32] One of the most obvious is the movement from seeing " = + " as a statement of equality to seeing it as the storage of a sum to a variable.
[33] Worse still, in my view, is that there is a pattern of researchers failing to discuss or acknowledge students' productive knowledge *even when their own data supports it*.

> mismatches and misconceptions that cause novice bugs. Only once a bug is uncovered and understood are we ready to create a remedy for that bug. (Bonar & Soloway, 1983, p. 12)

That the authors would even consider developing new constructs or languages is a noteworthy distinction. Rather than assume in toto that students with "bugs" had wrong knowledge, the authors instead suppose cognitive bugs have plausible origins worth designing around. Moreover, they treat students' divergence from canon as *an opportunity for research to learn from students*. Precisely because students saw functional differences (cf., diSessa, 1986) in semantically-equivalent PASCAL statements, Bonar and Soloway (1983) reflect that perhaps programming languages should be more expressive:

> We find it quite interesting that novices seem to understand the role or strategy of statements more clearly than the standard semantics. Such roles discussed here include "counter variable," "running total variable," "running total loop," and "first, then rest loop". (See Soloway et al [1982b] for a detailed discussion of novice looping strategies.) Much work in programming languages is concerned with allowing a programmer to more accurately express his or her intentions in the program. Perhaps we can learn something from novices here - our programming systems should support recording the roles the programmer intends for various statements and variables. (Bonar & Soloway, 1983, p. 12)

Again, what's noteworthy here is that rather than treating students' non-canonical views as a burden for instruction, Bonar and Soloway instead see them as an opportunity for programming language designers to make languages better.

That insight—that instruction and design can meet novices where they are—carries through to their final remarks about studying and analyzing novice programming knowledge:

> The experience and understanding of a novice are available for analysis. In particular, our results suggest that the knowledge people bring from natural language has a key effect on their early programming efforts. Our work suggests that we need serious study of the knowledge novices bring to a computing system. For most computerized tasks there is some model that a novice will use in his or her first attempts. We need to understand when it is appropriate to appeal to this model, and, when necessary, how to move a novice to some more appropriate model. (Bonar & Soloway, 1983, p. 13)

I assume Bonar and Soloway structured their final line deliberately. If so, their phrasing has three consequences:

1. Appealing to novice's existing models gets precedence. That is, understanding how to leverage novices' existing knowledge comes first in research.

2. Changing the models novices have comes next, and “when necessary.”
3. They speak of “how to move a novice to some more appropriate model,” which does not necessarily entail the removal or destruction of novice’s existing models.

Taken together, these points convey a sense of how Bonar and Soloway view learning and instruction in computing. Instruction explicitly includes appeals to prior models and knowledge students might already have. Learning, meanwhile, involves the movement *when necessary* to more appropriate models of computation. As I explain in the next section, such a view exactly aligns with a particular branch of constructivism, where cognition is viewed as the complex activation of manifold resources for thinking and knowing.

3.2.4 Examples motivate the need for contextual-sensitivity in modeling programming cognition

Let’s begin with two motivating examples. My aim with these examples is to show how a practicing programmer might employ specific, distinct conceptual models to reason locally about a piece of code. First, consider variants of the PASCAL statements from Bonar and Soloway (1983), where an assignment statement worked to increment a value and store the result back to that value. In Table 2 below, I imagine five different ways of writing a programming statement. Each of the five ways is semantically equivalent to the others (or near enough for explanatory purposes). And, in each example I add a layer of specificity to the syntax. I also propose a corresponding interpretation of how I might apply interpret and think about a given statement.[34]

[34] I assume for simplicity’s sake that all variables and functions in my code samples evaluate to floating-point numbers that can be combined under addition. I also assume, crucially, that all statements are in the same hypothetical language and that the variable names connote nothing to the computational interpreter.

Table 2 – Five different interpretations of semantically equivalent programming statements in the same language

Statement	Programming Syntax	How I might think about it
1	x = x + d	An incrementer or accumulator
2	x = x + update	Updating the value of x
3	x_position = x_position + update	Updating the x-position with uniform velocity
4	x_position = x_position + update(t)	Acceleration on the x-position
5	x_position = x_position + update(…)	Generalized x-position update (update could be any function of any number of parameters: random noise, low-level hardware functions getting mouse or keyboard input, etc.).

I stress that the third column is about *how I might think about* each programming statement. By no means am I making normative claims about how one *ought* to think about it, or whether the statement actually does what its names might suggest it does. Rather, "How I might think about it" reflects the kind of local meaning or interpretation I might attach to such a statement when I work with it, given my understanding of its role and context.

Consider specifically statements 1 and 4. Statement 1 could very well be an incrementer in some kind of iterative code. If I had to debug code that employs statement 1, what I might do is exploit my knowledge of what d is (does it hold a constant value? Is it 1?) and try inserting intermediate print statements into the iterative code. Doing so, I can inspect textually how values change with each iteration. In statement 4's case, it could very well be that the position-update code animates images on a screen. If there's a problem with that code, one of the easiest ways I might notice is that the acceleration seems off in the graphics. Consequently, my debugging might call upon knowledge I have from kinematics. I might try inserting code that draws a dot at the object's on-screen position with each iteration, leaving a trail I can visually inspect. I could then look at the path of the object's trajectory and the spacing patterns between successive dots as a first-pass test of whether my code achieves the motion I want.

To be clear, it's not just that graphics might improve my efficiency in debugging a statement. Rather, applying an interpretive frame that treats an assignment statement as a kinematic position update lets me *use conceptual knowledge from physics* to diagnose and fix problems in my code. If I view the assignment statement as saying something about the motion of an object (cf., Hammer, 1994, p. 165), a field of knowledge and concomitant techniques from physics becomes available to me to think with. But, if I focus only on the semantic-level equivalence of statements 1 through 5, there would be no obvious reason for me to access what I know about physics in order to reason about the code.

My second example concerns statements that are syntactically-similar, rather than semantically-similar. I proposed that the statements in Table 2 all came from the same language. But, another phenomenon comes into play when different languages use the same symbology for conceptually different operations. Table 3 shows examples of what look like semantically-equivalent operations, but in fact are not.

Table 3 – Three syntactically-similar statements with very different semantics

Statement	Programming Syntax	Language	How I think about it
1	i = i + 1	C	**Increments** i by 1
2	w = w + “ly”	JavaScript	**Appends** “ly” to the string w
3	p = p + geom_point()	R (ggplot2)	**Composes** a layer of points onto a plot p

The catch here is that the plus operator takes on different roles in different languages because of how those languages define its use. Statement 1 increments a number in C; Statement 2 appends the letters “ly” to a string; Statement 3 adds a layer of points to a statistical graphics plot. These different kinds of operations become even more apparent and consequential when, for example, such statements are repeated. In the R/ggplot2 code below,[35] I’m using multiple reassignment statements to compose a statistical graphics plot. The layered creation of a plot invites a very different kind of conceptual interpretation than, say, repeatedly accumulating numbers into a running sum:

```
p <- InitializeGgplot_1w()
  p <- p + GrandMeanLine(owp)
  p <- p + GrandMeanPoint(owp)
  p <- p + ScaleX_1w(owp)
  p <- p + ScaleY_1w(owp)
  p <- p + JitteredScoresByGroupContrast(owp, jj)
```

One obvious reason for thinking about this code with a different interpretive frame is that numeric addition is commutative; composing a plot is not necessarily commutative.[36] So, despite the syntactic similarity, reassignments that compose a plot using reassignment (as above) does not obey the same rules as reassignments for a running total. But, even within this code block, statements that look alike perform operations of a different nature. While some expressions (e.g, “+ GrandMeanLine(owp)”) add a visual layer to a plot, others modify features of the plot (e.g, “+ Scale_X(owp)”, which adjusts the scales on the plot’s x-axis to fit the numeric range of the data).

Given these motivating examples, it seems sensible to think there’s utility in a programmer having different conceptual metaphors available to think about and work with code. Example 1 shows that choosing to apply knowledge from physics to a

[35] In R, the assignment operator can be written as a directional arrow. The symbol <- (“less-than, hyphen”) indicates the value on the right side of the symbol is being assigned to the variable on the left side of the symbol. As used, the symbol itself is semantically equivalent to having written an equals sign (“=“).

[36] To convince yourself that plot composition isn’t commutative, imagine a scale function that squares up the aspect ratio of a plot and another scale function that sets the aspect ratio to 1.5:1. Applying the square function last produces a square plot; applying it first produces a rectangular plot. String concatenation is also not necessarily commutative. “cat” + “dog” evaluates to “catdog,” while “dog” + “cat” evaluates to “dogcat.”

piece of code can change the cognitive nature of debugging. There, debugging a position-update statement becomes, in part, reasoning kinematically about the properties of motion trails.[37] Example 2 shows that across languages, programmers might have to deploy different conceptual metaphors to reason about statements in a locally-consistent way. Knowing that plots in ggplot2 can be composed layer-by-layer with reassignment is crucial if you're trying to write or understand code that creates statistical graphics. But, I would argue that thinking about `□ = □ + □` as "compose new layer onto plot" can and does appeal to different kinds of knowledge when compared to thinking about □ = □ + □ as "include this addend in the sum," which itself can and does appeal to different kinds of concepts when compared to thinking about □ = □ + □ as "increment the counter."

Stepping back, I can build the following argument

1. Programming can be helped by applying conceptual models to code, particularly when relevant domain-knowledge structures can advantageously transform a problem (example 1)
2. But, conceptual models don't work all the time for all statements. Because languages are designed differently, the same syntax can actually correspond to very different operations in code (example 2). And, that's true both within and across languages.
3. Consequently, it makes less sense to treat conceptual models as right or wrong, and more sense to treat them as differentially advantageous for thinking about what a piece of code does. (Thinking a "+" implies numerical addition isn't *globally wrong* in JavaScript, but it won't explain why 1 + "1" yields "11" as a result.)
4. It seems plausible that successful programmers, when reasoning about or writing code, are able to dynamically access or deploy conceptual models that are advantageous given the context (language, syntax, surrounding code). Certainly such a supposition is in line with work suggesting students have resources for thinking conceptually and dynamically about how mathematics models real-world situations (Izsák, 2004; Sherin, 2001)
5. To model how programmers think with conceptual models, a suitable framework should be able to account for the dynamic, context-sensitive deployment of conceptual knowledge.
6. To model how programmers develop expertise, a suitable framework should be able to describe higher-order phenomena. Such phenomena include explaining how programmers come to have conceptual models or generate new ones, why they decide to deploy them, and how programmers consider which conceptual model (i.e., which way to think about code) is appropriate.

Taken together, these assertions propose criteria for how we might strive to model cognition in programming. Our modeling frameworks should be context-dependent, dynamic, and capable of explaining where conceptual models come from. They should also be able to account for phenomena that are not themselves

[37] For comparison, consider the argument that ringing an aircraft speedometer with physical markers changes the nature of cognition when pilots work to land a plane (Hutchins, 1995b).

conceptual, including what directs the use of certain kinds of conceptual knowledge. In the learning sciences, such models already exist and have proven useful and productive for thinking about thinking.

3.2.5 Manifold models of cognition explain context-dependence and the growth of expertise

In 1993, a pair of articles in the learning sciences staked a strong claim for viewing knowledge as a network of pieces, isolated enough to be locally triggered but trainable enough to fire in larger concerted patterns (diSessa, 1993; Smith, diSessa, & Roschelle, 1993). Informed in part by agent-based accounts of cognition (Minsky, 1986) and complex systems models (diSessa, 2002), the central tenets of an "in-pieces" approach hold that knowledge is emergent from interacting primitives, rather than unitary and monolithic. An example from Smith, diSessa, and Roschelle helps illustrate the point.

The authors show that we might think of a rubber band as a different conceptual entity depending on context. In several different situations—wrapped around a newspaper, pulled taut as a string, spun to store energy in a toy plane propeller—we intuitively think about the rubber band's physical behavior differently: one as a negligible part of the newspaper's point mass, another as a transverse pendulum (likely) obeying Hooke's Law, and the third as a torsional spring. Those differences in intuitive thinking reflect the contextual dependency of what we know about the physical world:

> In each of the rubberband examples, various pieces of intuitive physical knowledge describe the mechanism at work: the rubber band binds the newspaper, grips the jar lid, and acts a source of springiness for the bobbing object. Although a mapping cannot be made from the rubberband to scientific entities, it is quite easy to map these qualitatively distinct physical processes to scientific entities and laws. For example, instances of binding almost always map to a practically rigid body. Likewise, gripping maps to friction forces, and springiness maps to Hooke's law. This suggests that applicability can depend directly on our intuitive knowledge—knowledge that exists prior to any formal scientific training (Smith et al., 1993, p. 144).

The in-pieces approach to modeling cognition has been used, among other things, to explain how experts reason about fractions and decimals (Smith et al., 1993), how students reason about forces in physics (diSessa & Sherin, 1998; diSessa, 1993; Hammer, 1996; Sherin, 2001), how students construct and evaluate algebraic representations of physical situations (Izsák, 2004), and how knowledge transfers across contexts (Hammer et al., 2005; Wagner, 2006). Because its starting assumption is that knowledge is fragmented, knowledge-in-pieces can account for wide variations of how people—particularly novices—use knowledge on a moment-to-moment basis. In other words, because it assumes knowledge is local, it can still explain the kinds of globally-inconsistent ways people might reason about physical situations (diSessa, 1993). As a framework, an in-pieces approach ultimately argues that models of concept replacement and good/bad criteria for knowledge should be supplanted by a learning model of alignment/refinement of prior knowledge and the consideration of knowledge as productive/unproductive.

Hammer and colleagues have worked to extend the in-pieces approach to explain how students' epistemological activity—how they orient toward knowledge and knowing in a context (Hammer et al., 2005; Hammer & Elby, 2002, 2003). Specifically, those authors use two core theoretical constructs to explain students' stances toward knowledge and knowing:

- *Epistemological Resources* are the epistemological equivalent of diSessa's phenomenological primitives (p-prims). Resources, the authors propose, are the atomic units involved in how people cognize about the source of knowledge, the nature of knowledge, and epistemological activities (Hammer & Elby, 2003; Louca, Elby, Hammer, & Kagey, 2004).
- *Epistemological Frames* are the emergent result of subsets of resources acting in concert. Drawing from both Goffman's (1974) sociological notion of frame as structures of expectations and subsequent work on framing in discourse (Tannen, 1993), epistemological frames are a participant's local answer to the question "what is it [specifically, what knowledge activity] that's going on here" (Goffman, 1974, p. 8).

Resources can frames can interact in activity settings to produce larger-scale patterns called "epistemological coherences" (Rosenberg, Hammer, & Phelan, 2006) where evidence from data suggests that a network of discrete cognitive units can nonetheless give rise to stable cognition.

A useful conceptual metaphor for resources and framing is to think about a lecture hall with different sets of lights: a spotlight in the back to highlight a lecturer, chalkboard lights, house lights, and a projector. Each light has an individual brightness, but the lights are only controllable at the per-bulb level. In that sense, they are atomic. But, light patterns interact with one another to create a field of lighting for the room. Thus, lights are a bit like resources, and different light configurations can correspond to different, sociologically-stable uses of the room.

Table 4 – Using room lighting configurations to think about epistemological framing

What is it that's going on?	Houselights	Spotlight	Chalkboard	Projector
On-stage monologue	Low	ON	Low	OFF
Presenting slides	Low	ON	Low	ON
Working through equations	Low	ON	High	OFF
Students discussing with each other	High	OFF	Low	OFF
Watching a movie	Low	OFF	Low	ON
Cleaning the room after a movie	High	OFF	High	OFF

To convince yourself of the sociological stability of these configurations, imagine you were watching a movie when suddenly the house and chalkboard lights came up to full intensity. It would jar you out of the experience. You might wonder whether something was wrong: is there an emergency? Should you evacuate?

The metaphor isn't perfect. Ontologically, for example, resource and framing theory propose these constructs exist not in the world but in the minds of individuals. But, the metaphor is quite useful for understanding how individual elements—in this case lights—can work in concert to create and sustain a stable frame. And, crucially for a cognitive system, the metaphor lets us account for variation in activity. The same lecture hall can become a window onto one person's thoughts (monologue), a site for instruction (working through equations at the board), a shared space for collaboration (student discussion), or a place in need of repair (cleaning up) simply by varying the intensities of lighting banks in particular ways.

An example helps ground this in-pieces approach to epistemology. In Russ, Coffey, Hammer, and Hutchison (2008), the authors describe the situation where an elementary student reasons about why an empty juice box collapses when you suck on the straw. One student gives what the authors deem to be an excellent mechanistic account of why the juice box collapses. But, as the teacher seems to steer the discussion toward vocabulary—in this case, "pressure"—the student clearly pulls back from her mechanistic reasoning, seems much more diffident, and claims that pressure is hard to explain. That example highlights the disconnect between doing science as knowing vocabulary and doing science as reasoning mechanistically. Moreover, it strikingly highlights that a student who by all accounts produced an excellent explanation of how pressure works was left nonetheless with the impression that pressure was hard to explain.[38]

3.3 Methods and Theoretical Commitments

3.3.1 Student population, course background, and selection

This paper focuses on Electrical Engineering (EE) students from Flagship State, a large public research institution in the mid-Atlantic. The students I studied were taking Intermediate Programming, the second semester of an introductory programming course taught in C. The course was exclusively for EE majors and taught by EE faculty, and its enrollees were typically first-year or second-year EE majors. Its model was two 75-minute lectures and a 1-hour teaching assistant-led discussion section for students each week. Course topics included:

- Strings
- Pointers
- Dynamic Memory Allocation
- Testing
- Debugging
- Hash tables
- Trees
- Linked Lists
- Abstract Data Types
- Functional Decomposition

[38] In fairness to the teacher, I'm trying to focus on the result of the interaction and far less on the intent the teacher might have had. The student still left with a sense that science might be about vocabulary, even if that's not the view the teacher would espouse or was trying to enact in the moment.

Students in the course had varying degrees of experience with programming. Partly, that variation is because students with Advanced Placement (AP) Computer Science credit could place out of Basic Programming, the first semester C course. So, some students in Intermediate Programming came from technical magnet schools where they may have already had one or more years of programming in multiple languages, while other students may have been first-time programmers with only one semester of programming experience: Basic Programming.

I studied the same course, Intermediate Programming, during the fall 2011 semester and the spring 2012 semester. During fall 2011 I ethnographically observed over 50% of the course lectures, and during spring 2012 I sat in on several discussion sections. At the beginning of each semester I solicited student participants for semi-structured clinical interviews. Of the students who responded to the solicitation, I scheduled interviews opportunistically with students given my resources as the sole researcher on the project. In total, I worked with a cohort of 6 students in fall 2011 and 4 students in spring 2012.

My specific interest was in how students design computer programs. By that, I mean I wanted to know not just how students program, but whether and how they structured programs, how they did (or did not) try to incorporate modularity into their programs, and how the final form of a program reflected what they had learned about how to manage complexity through software. In the sections that follow I outline the palette of methods I used to pursue those questions.[39]

3.3.2 Studying design as a complex phenomenon of disciplinary practice

Practicing professionals make complex use of talk, gesture, and representational artifacts in physics (Gupta, Hammer, & Redish, 2010; Kaiser, 2005; Ochs, Gonzales, & Jacoby, 1996); chemistry (Stieff, 2007); field biology (Hall, Stevens, & Torralba, 2002); civil engineering (Stevens & Hall, 1998); structural engineering (Gainsburg, 2006); mechanical and electrical engineering (Bucciarelli, 1994; Henderson, 1999); and architectural design (Hall et al., 2002; Hall, 1999). Given that:

1. Science and engineering education research has made progress by looking for continuities in how novice learners develop disciplinary practices (Gupta et al., 2010; Hall & Stevens, 1995; Smith et al., 1993; Stevens & Hall, 1998), and
2. Emerging research on software engineering reveals that early-stage software design involves complex inscriptional, discursively, and epistemic practices,

it seems striking that there is no contemporary body of research, comparable to studies of expert practice, that looks at students' software design practices. In other words, we have every reason to believe that expertise in software design involves complex practices, but little (if any) research on what productive capacities students

[39] The language in sections 3.3.2 and 3.3.3 is taken with slight modification, from "Studying students' early-stage software design practices," a paper I authored, along with William Doane, and submitted to the 2014 International Conference of the Learning Sciences (ICLS).

have for them. Finding those productive design capacities requires a shift from questions such as *how can we assess and mitigate students' difficulty in programming?* toward questions such as *how do students learn and display evidence of design thinking in programming?*

Rephrasing research questions asked of professional software engineers (Petre et al., 2010, p. 533) and instead treating *students* as designers, I ask:

- What do students actually do during early stage software design work?
- What does students' exploratory design thinking look like?
- How do students communicate?
- What sorts of drawings do students create when they design software?
- What kinds of strategies do students apply in exploring the vast space of possible software designs?

3.3.3 Deploying methods to capture the complexity of early-stage design work

The methods below form the core of my developing program to study students as software designers. None of the methods below are new; all have been used in prior educational research. What *is* new, I believe, is the opportunity to combine them all under the umbrella of understanding what happens when students design software. For each of the four students in my spring 2012 cohort I captured multiple streams of data.

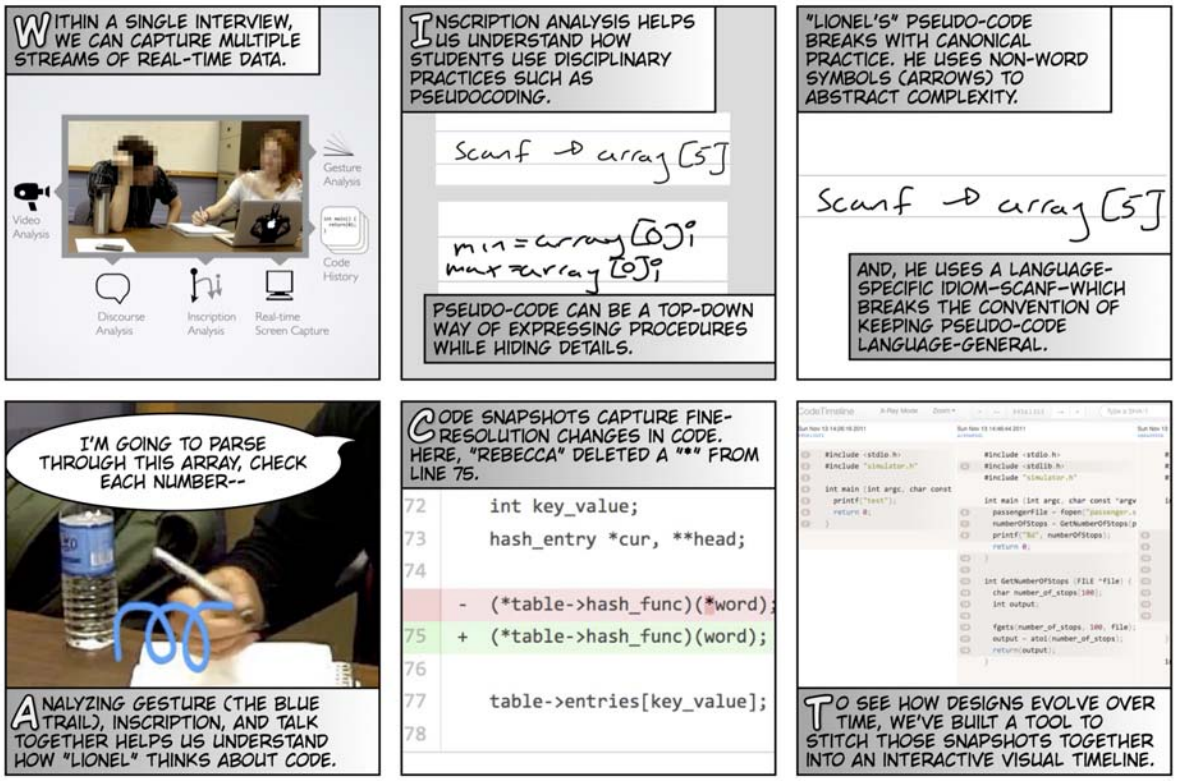
- I collected their **code history**. By this, I mean I preserved frequent snapshots in time of what students' code looks like. Research has already shown code snapshotting to be a useful method for understanding large-scale patterns of student error (Jadud, 2006; Rodrigo et al., 2009; Spacco, Hovemeyer, et al., 2006). And, the resolution of snapshots is extremely fine: Spacco et al. are able to capture the contents of a file each and every time a student saves it to disk. But, that research takes an aggregate view: it identifies large-scale error patterns at the expense of detailed naturalistic understandings of why students make those errors. Moreover, it's primarily used to identify what *mistakes* students make, which is distinctly different from a research orientation that considers the negative *and* positive consequences of students' design choices. A currently untapped advantage of collecting code history data, then, is that while we historically use it to aggregate *across programmers* it nevertheless also gives us fine-grained individual or team-based histories of how designs evolve.
- I conducted **clinical interviews** with them. Clinical interviews have proven historically useful in understanding the substance of students' knowledge and the nature of conceptual change (diSessa & Sherin, 1998; Duckworth, 2006; Ginsburg & Opper, 1988; Ginsburg, 1997). Crucially for Computer Science Education (CSEd) research, clinical interviews can tell us about students' epistemologies—how they view knowledge and knowing in a discipline (Hofer & Pintrich, 1997)—which can affect how they approach and adopt that discipline's practices (Hammer, 1989, 1994; Lising & Elby, 2005). Moreover, because my interviews were videotaped and analyzed from perspectives of

interaction analysis (Goodwin, 2000; Jordan & Henderson, 1995), they offer rich evidence of the substance of students' design practices

- I analyzed their in-interview **inscriptions** when they designed — what they wrote, how they wrote it, and how those writings got used. Evidence from both science studies (Hall et al., 2002; Henderson, 1999; Hutchins, 1995a; Kaiser, 2005; Latour, 1990; Ochs et al., 1996) and educational research (Hall & Stevens, 1995; Lehrer, Schauble, Carpenter, & Penner, 2000; Stevens & Hall, 1998) highlights the centrality of inscriptions to disciplinary practice in science and mathematics. Ethnomethdological data from studies of professional software engineers supports the same result: inscriptional practice is central to how professional engineers design software (Rooksby & Ikeya, 2012; Rooksby, 2010). And, since part of the inscriptional environment when designing software is the computer itself, I captured and analyzed what happened on-screen as students design programs.
- I paid close attention to students' in-interview **gestures**. Perspectives of gestural analysis hold that gestures can not only support or extend thinking, they can also communicate entirely different information than what's being said (Goldin-Meadow, 2003). Moreover, perspectives from cognitive anthropology and embodied cognition studies argue that bodily motion *is itself* cognition (Hall & Nemirovsky, 2012; Hutchins, 1995a; Nemirovsky, Rasmussen, Sweeney, & Wawro, 2012). For example, when students describe a part of code by moving their hand across an imaginary row of items and tapping each item, their bodies convey information we can interpret about how they understand iteration.

Figure 3-1 presents a visual overview of some of these methods and modes of analysis. In particular, the first (top-left) panel depicts how we deploy these data collection methods during a clinical interview:

- a voice recorder captures speech (often a redundancy in case another recording device fails)
- a LiveScribe Pulse pen digitizes written inscriptions, allowing us to play back what was written in time
- a videocamera records data for knowledge analysis, interaction analysis, and gestural analysis,
- an in-interview computer tracks code history and its screen-capturing software records real-time activity.


WITHIN A SINGLE INTERVIEW, WE CAN CAPTURE MULTIPLE STREAMS OF REAL-TIME DATA.
Video Analysis
Gesture Analysis
Code History
Discourse Analysis
Inscription Analysis
Real-time Screen Capture
INSCRIPTION ANALYSIS HELPS US UNDERSTAND HOW STUDENTS USE DISCIPLINARY PRACTICES SUCH AS PSEUDOCODING.
Scanf → array [5]
min = array [0];
max = array [0];
PSEUDO-CODE CAN BE A TOP-DOWN WAY OF EXPRESSING PROCEDURES WHILE HIDING DETAILS.
"LIONEL'S" PSEUDO-CODE BREAKS WITH CANONICAL PRACTICE. HE USES NON-WORD SYMBOLS (ARROWS) TO ABSTRACT COMPLEXITY.
Scanf → array [5]
AND, HE USES A LANGUAGE-SPECIFIC IDIOM—SCANF—WHICH BREAKS THE CONVENTION OF KEEPING PSEUDO-CODE LANGUAGE-GENERAL.
I'M GOING TO PARSE THROUGH THIS ARRAY, CHECK EACH NUMBER--
ANALYZING GESTURE (THE BLUE TRAIL), INSCRIPTION, AND TALK TOGETHER HELPS US UNDERSTAND HOW "LIONEL" THINKS ABOUT CODE.
CODE SNAPSHOTS CAPTURE FINE-RESOLUTION CHANGES IN CODE. HERE, "REBECCA" DELETED A "*" FROM LINE 75.
CodeTimeline
X-Ray Mode
Zoom
TO SEE HOW DESIGNS EVOLVE OVER TIME, WE'VE BUILT A TOOL TO STITCH THOSE SNAPSHOTS TOGETHER INTO AN INTERACTIVE VISUAL TIMELINE.

```
72      int key_value;
73      hash_entry *cur, **head;
74
   -    (*table->hash_func)(*word);
75 +    (*table->hash_func)(word);
76
77      table->entries[key_value];
78
```

```
#include <stdio.h>
#include "simulator.h"

int main (int argc, char const
  printf("test");
  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include "simulator.h"

int main (int argc, char const *argv
  passengerFile = fopen("passenger.
  numberOfStops = GetNumberOfStops(p
  printf("%d", numberOfStops);
  return 0;
}

int GetNumberOfStops (FILE *file) {
  char number_of_stops[100];
  int output;

  fgets(number_of_stops, 100, file);
  output = atoi(number_of_stops);
  return(output);
}
```

Figure 3-1 – An comic-based overview of my methods for capturing students' design practices

3.3.4 Developing a revelatory case of students' early-stage design work

In the empirical work that follows I explore what productive knowledge and resources students have for structuring programs. I begin with Lionel, a male student in Intermediate Programming, fall 2011. First, I propose the idea that Lionel connects "designerly" (Archer, 1979) stances across contexts. I analyze his retrospective account of modifying a bike and compare it to his retrospective accounts of how he programs in Intermediate Programming. Then I detail Lionel's multi-stage cascading approach to structuring a program solution in an interview. The argument throughout is that from bike design to code design Lionel evinces knowledge and stances that productively help him solve problems.

Next I transition to Rebecca, first explaining the epistemological difficulty she felt when coding, then showing how she circumvented that block by structuring a program piece using gesture and talk. I argue that what Rebecca and Lionel have in common are resources for handling design obstacles as they emerge. Where the two students differ is in their in-the-moment judgments about what sorts of knowledge and activities are appropriate to work through a design problem.

I argue that taken together, Rebecca and Lionel constitute what Yin (2009) calls a "revelatory case," by which I mean my investigation arises from "an opportunity to observe and analyze a phenomenon previously inaccessible to social science inquiry." In my case, I argue the phenomenon I'm observing is what engineering students do in the early stages of design work on a program. Its inaccessibility is demonstrated by the almost complete paucity of studies that draw on real-time (e.g., video records) of how students begin work on a complex program. Contrary to being “representative” or “typical,” my revelatory case does not aim to generalize. Rather, its value is in phenomenological richness. As Yin (2009) writes, and I think others including Erickson (1986) would agree, such case studies are worth doing “because the descriptive information alone will be revelatory.”

3.4 Lionel’s approach to design and to programming

3.4.1 While modifying a bike Lionel decomposed tasks, explained solutions to himself, and created intermediate design representations

Prior to entering my study, Lionel had already earned a Bachelor’s degree in Business Information Technology at a different university. We began the interview by discussing why he re-matriculated to earn an electrical engineering degree.

> Interviewer: How did you, um end up coming out of your business program and your business experience and deciding that electrical engineering was what you wanted?
>
> Lionel: Um, well, since I was Business Information Technology, basically all my job was doing was writing SQL code and, you know, short statements like that. I also worked at a help desk for—I guess I worked at Department of Homeland Security. And, pretty much besides the SQL statements I worked with this one program and whenever somebody had a problem with it they’d come to me. And basically that—that’s what my whole degree was—was going down that track and then that’s really not what I wanted to do: just sitting at a desk, you know people coming to me being like “oh, you gotta do this for me.” So, I was like, alright. Or, I have to fix a problem for them, pretty much. And that’s really what I didn’t want to do. I wanted to do something more hands-on where I could, you know, I’d be given a project where I’d do something on my own, create something on my own. And, um, also my Dad and my uncle were both electrical engineers. And my brother became a computer scientist. Cu—so I’ve always been around, you know, that type of environment, where my dad’s always building something and, you know, I always see it and I’m like, you know, that’s very interesting. I’d like to do something like that too. So it’s really just uh, a mixture of bad experience with my previous job and being around company that likes to do that sort of stuff. (Interview 1 of 1, October 17, 2011)

Lionel apprenticed building things with his father first, taking on small, highly-directed roles in projects.

> Lionel: Be-before I started my degree, you know, I really didn’t know how to do any of this stuff. So, I would just kinda help my dad out a little bit. He’d be

> like "Oh, here," you know, "to, to do this you have to do exactly this." You know, "solder this here," or, you know, "drill this," blah blah blah. He'd tell me exactly what to do. So, I just kind of followed along. (Interview 1 of 1, October 17, 2011)

Presumably in those projects Lionel was at a periphery and direction came from his father. More recently, though, Lionel had struck out on projects of his own. As Lionel said, "now that uh, my dad kinda showed me how to do certain typical types of things, I've tried to become ||more adventurous|| |{air quotes}| /Sure/ and do things on my own" (Interview 1 of 1, October 17, 2011).

One of Lionel's recent adventurous projects was adding a motor to a beach cruiser bicycle. The gist of the project was simple: using a low-priced motor he found online, Lionel realized he could mount the motor to an inexpensive bicycle and make a motorized bike. With one exception, Lionel was solely responsible for the project from start to finish.[40] He bought the parts, he mounted the motor, and he maintained it as a working vehicle. But, in practice the project was not straightforward. In the next subsections I show at length how Lionel addressed the emergent challenges of modifying[41] his bike with a motor. My analysis focuses on three features of Lionel's story that I argue carry over to his programming approach:

1. When faced with ambiguity, Lionel worked to decompose large tasks into smaller subtasks he could understand.
2. Throughout the task, Lionel was metacognitive about the state of his design. His constant refrain was "*how* am I gonna make this work?"
3. Before committing work and materials to a solution, Lionel created intermediate designs whose suitability he could evaluate.

I argue in later sections that these three features of Lionel's approach—decomposing subtasks, explaining procedures to himself, and explicitly planning prior to committing time and resources to a final product—have strong resonances with the approach Lionel took to while programming.

3.4.1.1 Lionel had to decompose Step 1, which simply said "Install the Engine"

Lionel bought an inexpensive motor online with the plan of attaching it to a cheap bike. As he discovered, though, frugality came with a hidden cost: poor instructions. Attaching the motor to the bike was not at all a straightforward process.

> Lionel: it was funny. When I, when I got the motor /Mmmhmm/ Um, oh, it—like I said it was like cheaply made and whatnot and it came with instructions, but um, no exaggeration at all, step 1 said "install the engine". And that was it. {Laughs}
>
> Interviewer: Huh

[40] Lionel told me that the only part his father helped him with was showing him how to use a Dremel power tool to smooth, sand, and cut edges (Interview 1 of 1, October 17, 2011).

[41] Often called "modding", for short.

> Lionel: And then step 2 was, um, I guess it was somethin like “put the gas tank on.” It really didn’t tell me anything. It was kinda like “OK, those are the obvious steps, but *how* do I do that? So I kinda had to figure stuff out on my own. /So/ yup
>
> Interviewer: Did it—how did it end up going? Was it a lot of trial and error? Like, did they even have pictures of how it was supposed to look when it /There, there/ was installed?
>
> Lionel: There are maybe two or three pictures, but they were black and white pictures printed out on like a, I guess like a crappy printer. So they were kinda hard to see. Um, so I guess my first step was to go look online at better quality pictures. Unfortunately nobody really—I was hopin to find you know, like a YouTube video: this is how I did it.
>
> Interviewer: Mmmhmm.
>
> Lionel: I couldn’t really find somethin like that, so I just looked at pictures. (Interview 1 of 1, October 17, 2011).

Lionel recognized that having an attached motor is an obvious goal state, but that goal state was poorly-specified by “crappy pictures” that were hard to interpret. Moreover, the states in between start and goal were undocumented and not obvious. Lionel’s subsequent search for materials that showed a step-by-step progression came up empty. Faced with a poorly-specified goal-state and no intermediate guidance, Lionel’s response was to force himself to think carefully about how he would proceed.

3.4.1.2 In making modifications Lionel frequently asked himself “how am I gonna make this work?”

Faced with the challenge of interpolating between “crappy pictures,” Lionel thought hard about what to do.

> Lionel: I spent, I guess the, I guess the first five hours I just spent kinda staring at it like, “hmm, *what* am I gonna do?” Like, “*how* am I gonna make this work?” (Interview 1 of 1, October 17, 2011)

Ultimately, he was struggling with more than just poor instructions. Even if he did figure out how the motor was supposed to be mounted, he still faced another problem: the motor was designed for a mountain bike.

> Lionel: I had to make a few modifications on my own. Because I guess it’s built for uh, a mountain bike. And the tubes on mountain bikes are, y’know like *that* thin {makes circle with right thumb and index finger}, but on my beach cruiser there’s one tube {widens thumb and forefinger} that’s pretty thick where I had to mount the engine.

> Lionel: And since the engine mount was built for a motorized bike it was really small. So I kinda had to like devise my own plan on *how* to, y'know, add a bigger mount.
>
> Interviewer: Mmmhmm
>
> Lionel: So, that was, that was my first problem. That was, y'know the first step was to put the motor on I'm thinkin, well, how am I gonna put the motor on if, y'know the, the specs don't even align. So, I guess spent y'know five hours just thinkin', like, "what am I gonna do?"
>
> Lionel: Finally I came up with a plan that seems to be working. So. And then, I just kinda went from there. I'd put it together, look at the picture and say "OK, this piece looks like that piece, and y'know I guess it looks like it attaches that way, so I'm gonna try that." And just put it together and, y'know if it fit, it fit. If it didn't, then I'd, y'know I'd be like "OK, maybe that was wrong. Lemme, lemme work on somethin' else on it." So. (Interview 1 of 1, October 17, 2011)

Asking himself *how* to do something helped bring sub-problems into focus. Within a larger mod task (viz., adding a motor to a bike), Lionel in effect entered a nested task: modding a mountain bike engine mount to fit a beach cruiser. Modding the mount itself was entirely undocumented, so Lionel's solution in part was to look for affordances in the physical structure of parts to see what he could attach. Crucially, the environment afforded Lionel rapid, iterative feedback. It wasn't hard to tell when a design choice worked because "if it fit, it fit."

3.4.1.3 Lionel used intermediate representations to consider alternative designs

Lionel grew increasingly frustrated as he saw how difficult it would be to mount the motor. Crucially, part of his tactic around that frustration was to resolve to keep trying and thinking of possible solutions.

> Lionel: So, I guess I had to give myself, like a few hours just to cool down and then actually think "OK, well maybe, you know I've already bought it, so, I'm gonna continue on with it. So I had to think, you know, OK, what's my next step. *How* could I make this work? You know, I—I thought of a few different ideas. You know, I'd think of one idea and think "OK, would this work? It might, but let me think of another idea." So I'd come up with another idea and say, "would this work?" You know, so once I got a few ideas I would just kind of, in my head picture it. Say, "OK, this is how it would mount, or this is how it would mount. Which one would be safer? Or will one of these not even work?"
>
> Interviewer: Right.
>
> Lionel: So, I guess after, you know, awhile of deciding which one was better, you know, which one would be easier and safe at the same time /mmhmm/ so

> then, that's when I made my decision, and just kinda, went for it to see if it would work. And luckily it did {laughs}. (Interview 1 of 1, October 17, 2011)

Before committing to a design, Lionel forced himself to evaluate alternatives first. Naively, he could have gone with the first idea that came to him. Instead, even if he thought an idea might work he pushed himself to consider others first. Then, with a collection of possible designs he evaluated what he saw as their suitability in his head. He applied criteria as he evaluated, trying to balance the ease with which a mounting scheme might work with a concern for the safety of the user—in this case, himself. Again, at least in Lionel's telling, those steps happened *before* committing to a final implementation. In other words, Lionel generated and evaluated multiple configurations with an eye toward both the builder of and user of those configurations before settling on a path to take.

In sum, I note three key features of Lionel's approach to the design challenges of modifying his bike. First, he repeatedly asked himself "how am I gonna do this," forcing himself to think in terms of a plan or procedure to fill in the gaps between the instructional pictures. Second, he proposed ideas, tried to evaluate their feasibility, and continued pressing for alternate solutions ("would this work? It might, but let me think of another solution"). Third, so much of his planning took place before he ever mounted the motor. By his account, he fiddled with pieces, envisioned multiple ways the engine could mount, and evaluated their feasibility before ever taking concrete steps to mount the motor.

3.4.2 Lionel's approach to programming resonates strongly with the design stance he held when working on his bike

3.4.2.1 Lionel thinks about steps at a chalkboard

Later in the interview I asked Lionel to compare his experiences in Basic Programming with those he was having in Intermediate Programming. His response revealed a great deal about how he starts projects in Intermediate Programming: he starts at a chalkboard.

> Lionel: Well, [Basic Programming] I already learned the basics of the class so I know the basics of where to start my code. And then, obviously the projects in this class [Intermediate Programming] are harder than last class, but I have the basics of, "OK, well, I know how a program runs. So how can I translate that into my program, for, for [Intermediate Programming]?" And, yeah, so I guess I'll sit down, like, yeah, and I'll have like a chalkboard and I'll write things out, and say OK well first, you know, {sweeps hands] not even think about the code. Just, in {air quotes} English words, so you know, my project's gonna do this. You know, first I need to make this calculation, then I'm gonna print this. And then I'm gonna make this calculation. Et cetera. And I'll sit down and write that out, in the order that it needs to be done. And then from there I'll go back and think, "OK, what's the code, to, you know, make this calculation or print that?"

Interviewer: So, when you start a project, you don't start at a computer, it sounds like you kinda start at your chalkboard.

Lionel: Right. Exactly. Yeah, cuz I've tried to start a project at a computer, and you know I'll write a—I'll start writing a few lines of code, but then I just really zone in on what I'm doing and kinda lose sight of the whole picture.

Lionel: So then I'll write this code for, you know, this one little section of the project, and then after I've done that I'll look back and say "Oh, yeah, I need to write the whole project," but then this little code doesn't actually fit into my project, so it's kinda, you know I guess I get tunnel vision when I start at a computer. (Interview 1 of 1, October 17, 2011)

For Lionel, the chalkboard is a pivotal object in his design activity. It holds the top-down "English words" description of his program. In so doing, it serves as the earliest durable representation of his program's intended hierarchy and relationships. That function is important, because without it Lionel is apt to "lose sight of the whole picture" of what his code is supposed to do. Moreover, as the next section reveals, the board continues to help Lionel orient his activity as he digs into the specifics of how his program will work.

3.4.2.2 Lionel copies pseudo-code into the computer even though it won't compile

As Lionel described the role of the chalkboard, I initially thought there was a distinct separation between the chalkboard and the computer as activity objects. The chalkboard, I assumed, was where Lionel sketched out design plans while the computer was where he wrote code. When I probed Lionel based on my hunch, I discovered Lionel's activity wasn't cleanly separable into design on the board / code on the computer.

Interviewer: Did you, when you start thinking about the code, do you end up usually filling it in on your board, or will you actually go straight over to the computer and do it?

Lionel: Um, on my board, I'll do, I guess like ||pseudo|| |{air quotes}| code /mmhmm/ uh, you know, first on the board it would only say "this function calls this function, this function calls this function."

Interviewer: Mmmhmm

Lionel: So then on the board I'd write down, "alright, well this function" um, I dunno, for instance I'll, I'll say you know, I'm workin with a function that checks to make sure that it's on the—that I'm moving *on* the board between spaces 1 and 24.[42] So then, you know I'll write down like, pseudocode, write

[42] At that point in the semester, Lionel's class was working through a Backgammon project, so the spaces and moves he's referring to pertain to programming a basic Backgammon game.

> "OK, well I'm gonna start. I'm gonna have a for-loop that will go from 1 to 24, and then here I'll, you know I'll have an if() statement that says if it's between you know, 1 and 24 it's valid, if not it's invalid." So, that'd be my pseudocode, and then when I actually—I'd, I'd try to write that out for the entire program, or as much as I could.
>
> Interviewer: Filling out the tree, you mean.
>
> Lionel: Right, fillin out the tree. And then I'd go to my computer, so that at least I already had a baseline of what to work with. And then, on my computer I'd, you know I'd write out the pseudocode, I'd ac—I'd literally write out the pseudocode, even though obviously it wouldn't compile and actually work.
>
> Interviewer: Mmmhmm
>
> Lionel: So, then, from the pseudocode, as, as I read down on my computer what the pseudocode said, I'd actually try to write the actual code that would work. (Interview 1 of 1, October 17, 2011)

Lionel blurred my initial chalkboard / computer distinction in two ways. First, his overall "English words" plan of functions calling functions evolved on the chalkboard to include computational idioms, including if-statements and for-loops. But, then, according to Lionel, he would copy that pseudo-code as-was into the computer, knowing full well it wouldn't compile. These observations—that Lionel writes computational pseudo-code on the board, then copies that same code into the computer—may seem trivial, but I contend they aren't.

Because of the choices Lionel makes, neither chalkboard nor computer has a distinct, privileged kind of syntax. Rather, "English words" and pseudo-code can peacefully co-exist on the chalkboard, and non-working pseudo-code gets deliberately typed into the computer before it's expanded out into "actual code." Put another way, Lionel is capable of distinguishing between "English," "pseudo-code," and "actual code," but all three forms of expression are a part of his design process. It's perfectly acceptable for him to have something other than "actual code" exist in intermediate representations of his design.

3.4.3 Gestural, inscriptional, and verbal in-interview evidence reveals Lionel's resources for structuring a program

To probe Lionel's retrospective account of how he programs, I gave him an in-interview task to work on. In this section, I argue Lionel's development process on that task moved rather linearly through what I call a "representational cascade."[43] My schematic for that cascade — along with the features and affordances in each layer of

[43] As I'll explain later, I think it would be a mistake to assume that all development activity proceeds linearly and top-down in the manner Lionel's does. Not only do I think Lionel's clean linearity is rare, I also think it's not normative. So, my point here isn't to stress that Lionel moved linearly, monotonically through the cascade. Rather, my point is to show that he moved between different representational levels *at all*.

the cascade — is depicted in Figure 3. What I show here is that actually, the final code Lionel produces is in fact one representation of a procedure that actually existed in different forms and modalities before it became the final code in Figure 3.

In the analysis that follows, I explore two of those modalities—verbal description and written pseudo-code—and explain why each is crucial to understanding the final produced code. Ultimately, I argue there is a need to explain *how* Lionel creates and moves fluidly between these markedly different cascade layers. Specifically, I attempt to answer the question of *Why does he see the upper layers (verbal code, hand-written pseudocode, and handwritten source code) as legitimate activity?* My candidate answer is that Lionel's approach to programming is in part stabilized by epistemological resources that allow him to treat the upper layers of the cascade as valid, productive knowledge expressions in programming. Such an answer is strongly in line with the findings from sections 3.4.1, where we see Lionel deliberately making use of other intermediate design representations.

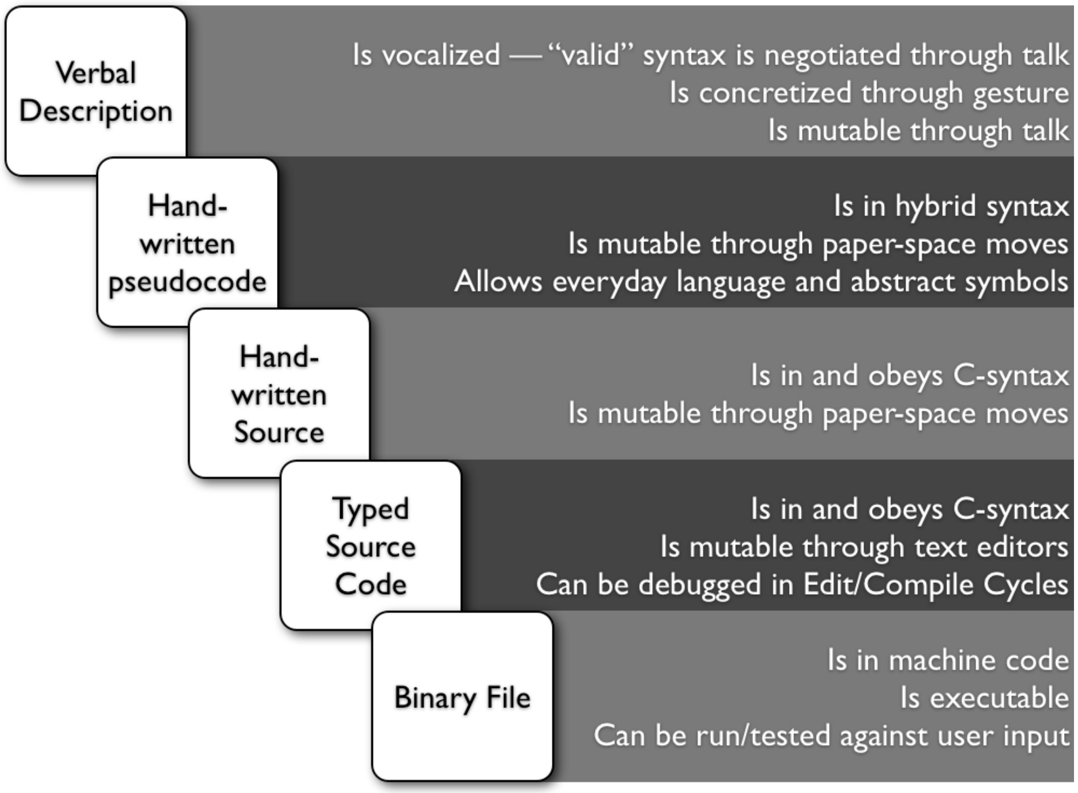

Verbal Description
Is vocalized — "valid" syntax is negotiated through talk
Is concretized through gesture
Is mutable through talk
Hand-written pseudocode
Is in hybrid syntax
Is mutable through paper-space moves
Allows everyday language and abstract symbols
Hand-written Source
Is in and obeys C-syntax
Is mutable through paper-space moves
Typed Source Code
Is in and obeys C-syntax
Is mutable through text editors
Can be debugged in Edit/Compile Cycles
Binary File
Is in machine code
Is executable
Can be run/tested against user input

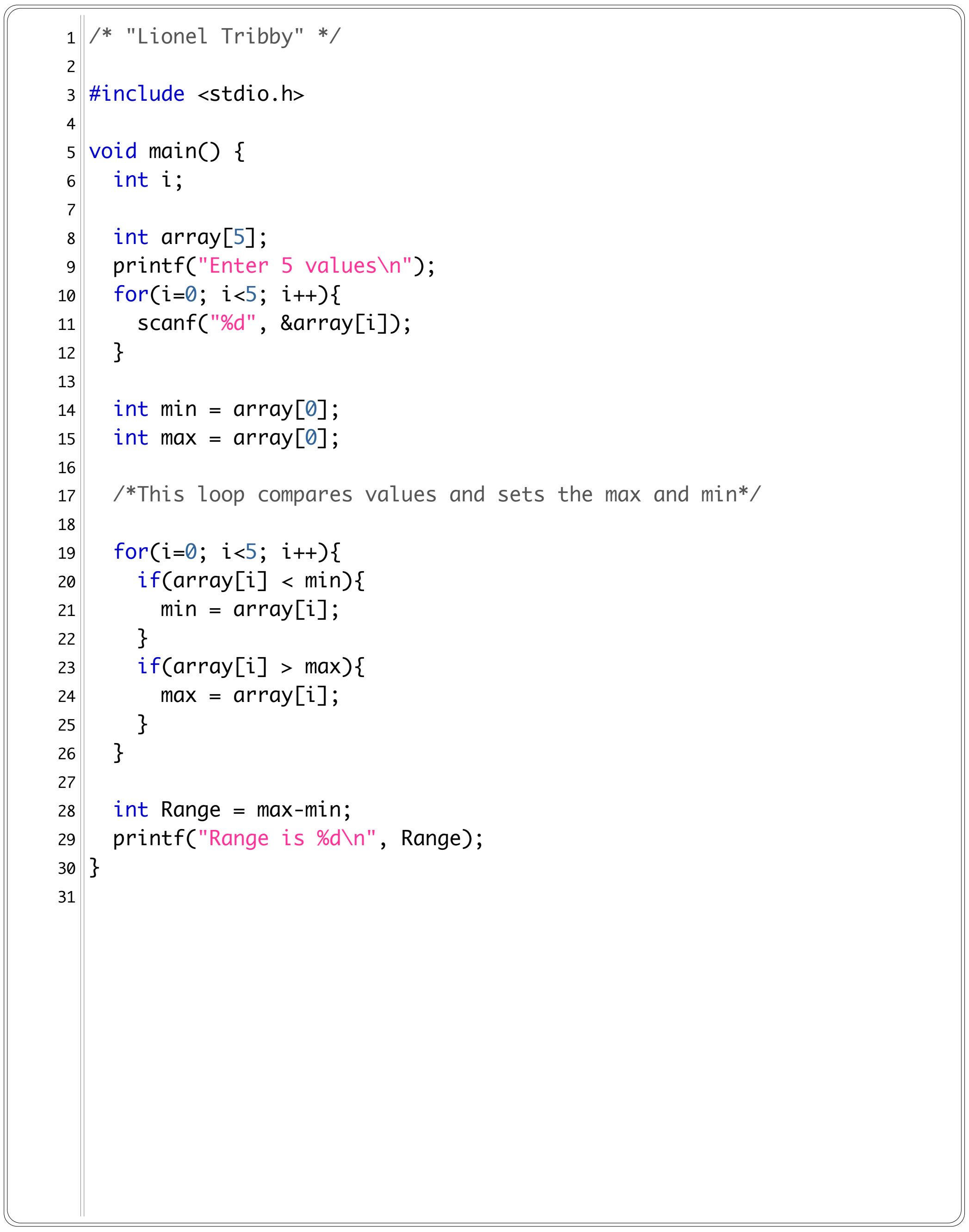
Figure 3-2 – An overview of Lionel's representational cascade and the features and affordances of each representational layer.

3.4.3.1 The range-finding prompt and Lionel's final source code

About 35 minutes into our interview, I gave Lionel the programming task reproduced below:

> Suppose you want to write a program that finds the range of a set of numbers. So, the program would take 10 numbers as input, then it would compute the range as the difference between the largest number and the smallest number.
>
> How would you write such a program? (Interview prompt, October 10, 2011)

The prompt was deliberately vague about certain constraints. It did not prescribe *how* the program should get input (Should the programmer expect the user might type it in, or feed in a formatted file?). It also did not prescribe how the program should provide the result (should the programmer print the result on screen? Store it to a file? Return the result to be used by another function?) The reasons for vagueness were in part because I wanted to see what assumptions students might make during their design process. Some solutions, I reasoned, might decouple the procedures of getting input, computing range, and returning output. Others might simply hard-code a procedure for getting keyboard-based input. I wanted to prescribe no specific approach up front if I didn't have to. I have reconstructed his original source code from screen-capture footage of his in-interview programming:



```
 1 /* "Lionel Tribby" */
 2 
 3 #include <stdio.h>
 4 
 5 void main() {
 6   int i;
 7 
 8   int array[5];
 9   printf("Enter 5 values\n");
10   for(i=0; i<5; i++){
11     scanf("%d", &array[i]);
12   }
13 
14   int min = array[0];
15   int max = array[0];
16 
17   /*This loop compares values and sets the max and min*/
18 
19   for(i=0; i<5; i++){
20     if(array[i] < min){
21       min = array[i];
22     }
23     if(array[i] > max){
24       max = array[i];
25     }
26   }
27 
28   int Range = max-min;
29   printf("Range is %d\n", Range);
30 }
31 
```

Figure 3-3 – Lionel's final source code for the range-finding prompt

Let's explore some top-level features of Lionel's code. First, Lionel chose to have a user enter text directly using the keyboard (lines 9–12), and his program displays the range result as printed text (lines 28–29). His strategy for finding the maximum and minimum is to implement a procedure that iterates through the array of numbers and compares the current number to the globally stored max and min.[44] If

[44] I think of this as a "king of the court" procedure. The current value stays on as an extreme if and only if it beats the current "king" for that extreme. Otherwise, the

the current number (*array[i]*) “beats” the global min (by being lower than *min*), its value replaces that of the global min. Similarly, if the current number exceeds the global max, its value replaces that of *max*. Before the procedure starts, Lionel assigns the first number of the array as the value for both extrema.

3.4.3.2 Lionel’s verbal description of the program

Lionel chose to start the task on pencil and paper. After scribbling a line about scanning input into an array (which I discuss in a later section), Lionel stopped and began explaining out loud to me.

challenger is replaced and the contest resumes with a new challenger until there are no more challengers.

Lionel: so I’m thinkin, my general concept is I’m going to parse through this array, check each number /Mmmhmm/ and I’ll have two variables, min and max, and, first I’ll set both min and max to the first variable in array, er, yeah first number in array /Mmmhmm/ and, I’ll parse through the rest of the array, checking to see if,

Lionel: Umm, you know if—I’ll s—I’ll start with max, for example. /Mmmhmm/ And I’ll check to see if the number in that section of the array is bigger than max. /uh-huh/ And if it is, I’ll set max to that new number. /OK/ And then I’ll do the same thing for minimum. And then at the very end, I’ll have my minimum and max and then I’ll just subtract ‘em.

Interviewer: OK

Lionel: So that’s my main concept. (Interview, October 17, 2011)

This verbal description is in some ways bound by the least stringent rules of expression. If we're hard-nosed about syntax, Lionel's speech stops and starts (line 5 into line 6), which leans on the unspoken understanding that ideas in talk are under negotiation. Stopping mid-sentence and leaving that sentence unfinished is a valid and acceptable move in conversational space. Moreover, there is a strong mapping between Lionel's temporal language and the intended flow-of-control for the program. "First I'll set," "then I'll do," "and then at the very end" all describe when instructions will be run.

Finally, Lionel's instructions begin at the high level, and in some cases implementation details are completely left out. For example, Lionel does not specify precisely how he'll "parse through the array." At this stage of the cascade, it seems sufficient for him to explain *that* he plans to parse through, but now how. What Lionel *does* do is gesture while describing "parsing". This gesturing, I argue, creates a concrete representation of iteration and incrementation — the gestural marker of a loop.

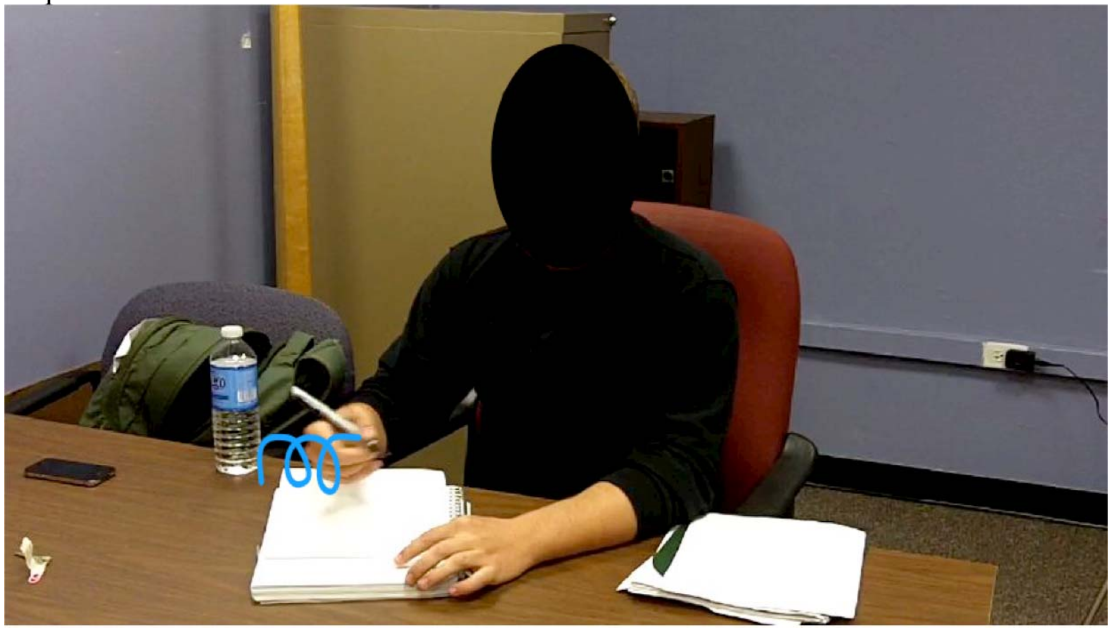


Figure 3-4 – Lionel makes a cycloid/helix gesture as he says "parse through this array"

Contrast how Lionel specifies "parsing" through the array with how he describes "checking" its contents. First, Lionel explains that he'll "check each number" (line 2), decides that he has underspecified how "checking" happens (line 5–6), and finally makes "checking" relatively more explicit (lines 7–8). Once checking for the maximum has been made explicit (lines 7–8), Lionel says "and then I'll do the same thing for minimum" (line 9). Line 9 thus becomes the verbal equivalent of a repeated, parameterized procedure. It is, in a sense, the talk-space seed of a function. Conversationally, both participants understand that Lionel saying, "then I'll do the same thing for minimum" (line 9) refers to a procedure defined elsewhere in the conversation, not a new set of instructions.

3.4.3.3 Lionel's pseudo-code

Immediately following the verbal description above, Lionel begins creating what he calls his "pseudo-code," narrating it as he writes. There are four key features in Lionel's talk (below) that I discuss in my analysis. Below is an overview of each feature and what importance I think it has:

1. Lionel iterates over his initial verbal description by fleshing out some implementation specifics — structures that further specify *how* something in a program is to be done.[45] This iteration is important because it suggests that his subgoals — and criteria for satisficing them — change from one micro-coherence to the next.
2. Reconciling what Lionel *narrates* with what Lionel *writes* reveals peculiar mismatches. The mismatches suggests a productive capacity: that Lionel can *background* certain details within a micro-coherence by omitting them from the written code while still maintaining an understanding of how his program will work. Put another way: the inscriptional object in a microcoherence captures what Lionel thinks is important within that microcoherence, but the inscription does not necessarily represent the totality of his conceptual understanding.
3. At one point in his code, Lionel writes "same for min," which we can reasonably infer means he intends to repeat a procedure for calculating the minimum. That line encapsulates the precursors of functional abstraction: Lionel not only recognizes procedures that share the same structure, he can background the specifics (point 2) of code he repeats.

[45] It's a limitation of my writing style that in this document I use "iteration" both to describe aspects of Lionel's design process and as a formal term of art in computer science. In both cases, though, the conceptual meaning is the same: the incremental, rule-based set of transitions to the state of an object.

Lionel: So it’ll be scanf, um, and I’ll have my for-loop, and I’ll just have my int-i, and I’ll go from zero uh, zero to, four actually cuz it’ll be zero, one, two, three, four, yeah, zero to four. /Mmmhmm/

Lionel: Umm, and I’ll have, oh, up top here, to set the, right—right after the scanf /Mmmhmm/, I’ll just say min and max equal the first—mm, hiccup, sorry

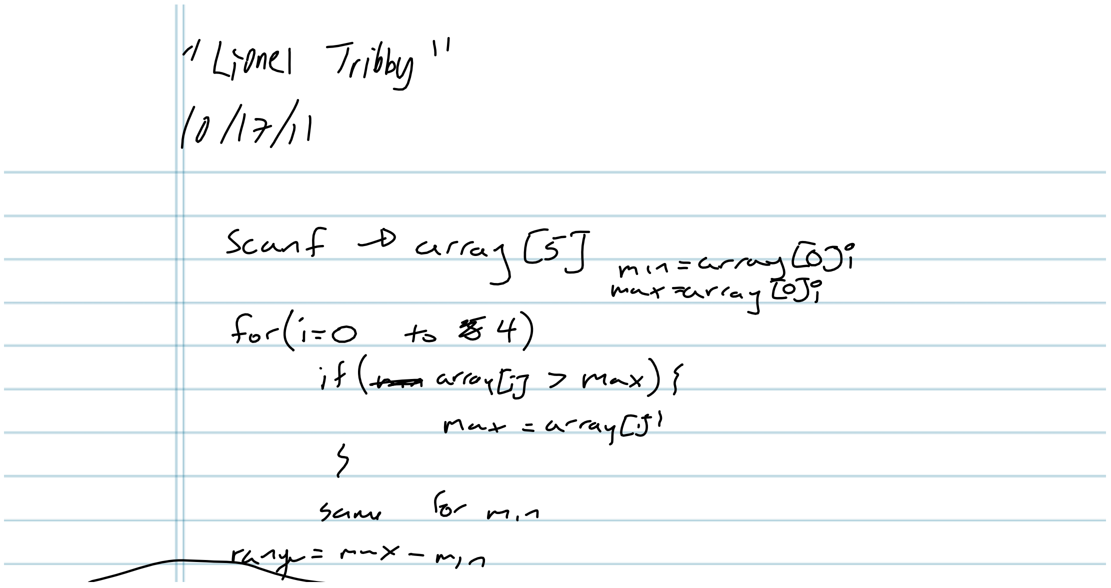
Interviewer: It’s OK.

Lionel: {laughs} the first number in the array. So I’ll say min and max equal array of zero. And that’s just setting them so that I have something to compare to, I guess. /uh-huh/ I think that will work. Yeah.

Lionel: So, so then, as I parse through my array of five variables, I’ll say “if min,” er no, I’ll say “if array of i, you know of, of that, if, uh, the number in the array that I’m looking at is greater than max, then max equals that number, array of i” /Mmmhmm/ and then I’ll do the same thing, same for min. And then at the very bottom, I’ll have, uh, range equals max minus min.

Lionel: So that’s my pseudo-code, just so that *I* understand what’s goin’ on in my head.

First, Lionel's talk iterates over his initial verbal description by fleshing out

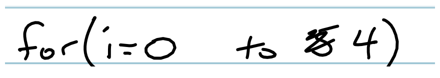


```
"Lionel Tribby"
10/17/11

Scanf -> array[5]        min=array[0];
                         max=array[0];
for(i=0  to ~~3~~ 4)
      if(array[i] > max){
            max = array[i];
      }
      same  for  min
range = max - min
```

implementation details — programming structures that further specify *how* something is to be done. Compare here how Lionel instantiates ideas of iteration in Table 1 below.

Table 5 – Comparing Lionel's words and actions across different stages of "programming" over the same putative section of his program. I have bolded some language-specific syntax.

Design Coherence	What he says	What he does
Verbal Description	"I'm going to parse through this array, check each number"	
Narrated pseudocode	"I'll have my **for-loop**, and I'll just have my **int-i**, and I'll go from zero uh, zero to, four actually cuz it'll be zero, one, two, three, four, yeah, zero to four.	 `for(i=0  to ~~3~~ 4)`

In his narrated pseudo-code, he introduces the C-specific syntax of “for-loop” and “int-i,” which handle iteration and indexing, respectively. Moreover, he counts in order to specify the index bounds of the loop (zero to four).

Second, it’s actually somewhat complicated to address the question “how do Lionel’s inscriptions align with what he says?” He *says* “I’ll have a for-loop”, but the visual evidence shows that he hasn’t bounded his for-loop body in curly-braces.[46] If he had done so, his for-loop preamble would end with an opening curly brace; it doesn’t (Table 1 graphic above). Also, for matching, his for-loop should have concluded with a closing curly brace; it doesn’t. Nevertheless, his verbal evidence — particularly him saying “int-i” — suggests that he understands details that variables need to be declared. Together, these mismatches suggest that features (viz., denoting iteration and setting its bounds) are important enough to be included in this representation. Other specifics, including proper declaration of variables, are things Lionel understands but does not insist on writing.[47] Together, and combined with later evidence of Lionel’s successful working program, these observations suggest that ignoring the details is actually productive for Lionel.

Third, Lionel’s talk reveals productive capacities for dealing with design work. For example, he describes the guts of his program through brief perspective-taking (lines 14 – 18 above). As a reminder, Lionel’s procedure is what I refer to as a “king of the court” approach: if the number currently-being-inspected exceeds the reigning champ, the champ is dethroned and replaced by the number currently-being-inspected. But, the way Lionel articulates that process actually uses multiple senses of

[46] As a language, C typically uses curly braces to bound instructions that span multiple lines. So, a typical for-loop would be written like this, where the loop preamble is bounded by parenthesis and the loop body is bounded by curly braces (bolded):

```
for(int i=0; i < maximum_value; i++) {
      // Do something
}
```

[47] One might argue that Lionel is simply being sloppy and forgetting to include details, even though he said them. The point I’d make here is that even if he is being sloppy, I would think it unfair to say, for example, that Lionel has conceptual issues with declaring variables. His talk (and many later iterations of his design) suggests he understands that declarations are necessary for working with variables in C. Moreover, even if he is being sloppy on the particular point of variable declarations, the overall character of this pseudo-code is still starkly different from that of the full, working source code solution he’ll ultimately produce. So, even if we propose that Lionel has some unintended tolerance — such as sometimes overlooking missing variable declarations — for assessing whether his pseudo-code satisfices, there is still strong evidence that his criteria for evaluating his work in pseudo-code stage are markedly different from the criteria he’ll apply to later iterations of design.
no one would confuse the pseudo-code skeleton here for Lionel’s fleshed-out, fully-working code later. So, even if Lionel *meant* to put them in and forgot, he still judged this pseudocode as satisficing his subgoal

"I," where it's at times actually not clear whether "I" means "I the programmer" or "I the program."[48] For example, the I in "I'll say 'if the array of i" could very well be the programmer, where by "I'll say" he means "I the programmer will write a conditional test." But, that phrase is immediately followed by "if, uh, the number in the array that I'm looking at is greater than max", where I believe we may be seeing Lionel taking the perspective of the program itself.[49] If anything, the seemingly untroubled, interchangeable use of "I" here might suggest that to do the work of programming one can occupy a hybrid space of both being the program and being the programmer at the same time.

Lionel concludes this procedure with the invocation "same for min." Picking up the thread of hybridity, this phrase could mean, "I the programmer will write a similar procedure for min" or "I the program will perform similar operations to determine min." That Lionel leaves his pseudocode there ("so that's my pseudocode", line 20) suggests that whatever perspective it might be written from, this designed artifact satisfices his pseudo-code production subgoal. And, that satisfaction is remarkable because if "same for min" is an appropriate place-holder, it suggests Lionel has ways of seeing that procedure as *similar enough* in structure to the procedure already laid out as to not require further specification. This is an important point: if a precursor to functional abstraction is recognizing repeated code, Lionel has actually recognized repeated code *before ever writing it*. As we'll see later, he ends up writing this procedure out explicitly, which means that whatever answer we might come up with as to why Lionel doesn't abstract this to a function *can't* be that he doesn't see the repeated computation; to the contrary, repeating core parts of the max computation was built into his design from the start.

3.4.4 We can trace continuities in Lionel across different kinds of design activities

We can trace patterns in Lionel's design activity by asking questions about the intermediate design artifacts he creates. Specifically, we can pay attention to *what* representations he creates, *where* they live, and *how* he uses them. Table 3 outlines those relationships for the three contexts discussed: Lionel's bike modding project, Lionel's retrospective account of his programming practices, and Lionel's in-interview work on the range-finding task. Table 3 helps support three key points:

1. There is a specific consistency to Lionel's design activity across contexts. All activities—given the limits of Lionel's self-reports—*involved the generation and use of intermediate design representations*. Those representations may at times have lived in Lionel's head, on paper, or in electronic format.
2. A thorough account of Lionel's in-interview programming activity cannot and should not ignore those intermediate representations and how he used them.

[48] To keep your head from exploding, I remind you that the lower-case version of "i" in Lionel's talk actually refers to the subscript for indexing an array.
[49] My point here is that technically, Lionel can never inspect an array element himself; he can only write code that does that. You might disagree, in which case we get to have a really fun discussion over beer about how we think programming is a case of "distributing cognition" (Hall, Wright, & Wieckert, 2007).

3. Because intermediate representations are a feature of all three design contexts, it seems likely that what stabilizes their creation and use is tied to something deeper or more global than just conceptual knowledge in a domain such as computer programming.

Table 6 – Comparing Lionel's design activity across contexts

Context / Question	Modding the bike	Retrospective programming account	In-interview task
***What* intermediate representations did he create?**	Possible configurations for mounting the engine	Relationships of functions to each other; pseudo-code for how functions would work	A verbal description; narrated pseudo-code; hand-written pseudo-code; hand-written source code
***Where* do the intermediate representations live?**	In Lionel's head	On a chalkboard and, later, in the text editor	In Lionel's talk and gestures; on paper; in the text editor
***How* did he use those intermediate representations?**	To compare mounting configurations for ease and safety	To "see the whole picture" of a project	To "understand what's goin on in my head" when structuring his code

3.5 *Rebecca's approach to programming*

In this section, I analyze data from a different student — Rebecca. I offer this data as an example of what developing design expertise might look like. If Lionel's approach to programming uses high-level work (e.g., his pseudo-code) to cascade down the specifics of syntax, it seems reasonable to ask whether other students use pseudo-code in the same way. And, if they do not, it seems worth asking how uses of pseudo-code differ and what that difference might tell us about students' approaches to programming.

In Rebecca's case, the data I'll present begins with her stuck on the third of four projects in the course: the "iTunes" project.[50] My analysis will expand on the following points:

1. To understand Rebecca's struggle on the iTunes project and her approach to programming during that struggle, we need to understand some of her prior experiences in programming. In our fourth interview Rebecca recalls a pivotal

[50] In this project, students were given a series of text files — representing the user's music collection — that contained formatted albums and song titles. The task was to create a program to read in the music information, create a database representation of it, and allow the user to transact with that database. Typical transactions might be asking for a listing of all the albums in a user's library or creating a playlist by allowing the user to choose specific songs.

project experience from her Basic Programming course. At the cost of hours of work, half her codebase, and a good grade, Rebecca discovered that her sense of how her program should work did not match what was expressible in C. I analyze Rebecca's recollection because it sits at the intersection of self-efficacy and disciplinary practice. Rebecca lost confidence and felt "demoralized" because her attempts at expressing a procedure did not match the constructs and underlying workings of the C language. Moreover, the fallout from that experience persisted into Intermediate Programming. By our fourth interview she had considered dropping Intermediate Programming entirely because of her low grades. "My *biggest* problem" in programming, she said, "is I think I have the logic, my logic just doesn't transfer to code. Or I don't know *how* to transfer it correctly" (Interview 4 of 5, April 6, 2012).

2. Rebecca has the capacity to make sense of program designs. As she tries to understand why the instructor would prescribe an array of pointers to store track titles in the music server, Rebecca makes progress by reasoning about what she knows about pointers and arrays. Like Lionel, Rebecca can use gestures, talk, and inscriptions (viz., pseudocode) to express how she might intend for her program to work. Rebecca also has a grasp of the conceptual issues — from a computational and programming perspective — relevant to the part of her design she's struggling with. But, Because Rebecca's approach to programming so strongly orients toward producing valid, working syntax, it may be limiting Rebecca from seeing other ways in which she can make progress on her design. In other words, Rebecca *can* do things like write pseudo-code to plan, but she doesn't tend toward those behaviors when she's stuck. I'll offer an explanation based on epistemological dynamics to explain why that's the case.

In what follows, I first present point 1 to give readers a sense of how Rebecca's prior experiences may inform her approach to programming. I then outline the particular part of the iTunes project where she feels "stuck," discussing point 2. Before I jump into that analysis, I *strongly* urge readers who are not familiar with pointers or dynamic memory allocation to refresh themselves on the subject.

3.5.1 Rebecca's struggles on the iTunes project make sense in light of some prior programming experiences she had

At the time of this study Rebecca was a first-year electrical engineering major who had gone to high school in rural Maryland. Her university courses in Basic and Intermediate Programming were the first experiences she had doing any programming at all, let alone in the C language. Though she felt academically at ease in high school, a tough project in her first-semester Basic Programming course shook her confidence in programming. In the data presentation and analysis that follows, there are four sub-points I'd like to make about Rebecca's development in programming.

1. Prior to the semester I worked with her, Rebecca had difficult experience trying to understand arrays on a Basic Programming course project.
2. That difficult experience had two pieces to it that I believe relate to personal epistemology:

    a. Rebecca came to feel that her idea for a solution wasn't something expressible in C.
    b. When she got help from a friend who was "very smart in programming," she felt like she typed the code he told her to into her project without really coming to understand it.
3. Reflecting on it in our first interview, Rebecca pinpoints the experience of that array project as demoralizing and having a lasting effect on her confidence in programming.
4. Epistemological issues continue to factor into Rebecca's confidence in programming. By our fourth interview, Rebecca was getting low grades in Intermediate Programming. She had strongly considered dropping the course. Trying to articulate her "biggest problem," Rebecca explained that she felt like she understood the theory behind the code examples in class but had difficulty transferring her logic into code.

3.5.1.1 Rebecca had a difficult experience on a Basic Programming arrays project

In our first interview, I asked Rebecca about how her experiences in Basic Programming compared to those she was having in Intermediate Programming. She explained that in that first semester, she "kinda got lost on one part" — arrays — "that was key to the rest of the year."

> Rebecca: [I]n [Beginning Programming] we had three projects, and the first one, uh, we did—I did discuss it with other students, and we all worked together kind of. But that one I felt much more like I had a grasp on everything I was doing. Whereas the second project, I would stare at the code and be like "ohmygosh, I don't even understand what, like, it's asking me to do." Like, or how to make it do what it needed to do, which was to like store in numbers in arrays, change them, and add 'em together and stuff. (Interview 1 of 5, February 10, 2012)

I asked Rebecca if she felt she had a better grasp by the time she turned in the project.

> Rebecca: No. Cuz, there was a friend on my floor, who—very smart at programming, he's had like years of experience. And he gave me a lot of pointers and tips doing it, so I felt like I was just using him a lot instead of actually learning it, which was a downfall for me.
>
> Interviewer: Huh. So, how, I mean. I feel like sometimes you could—you might ask somebody for help, and then they show you something and then you might be like oh, you've learned it now /yes/, but it sounds like you're saying /no/
>
> Rebecca: Just, I don't, like, I don't blame him at all because he helped me, but, it was kind of more of a, he showed me how to do it, I was like {mimes exaggerated typing, in an almost singsong voice} *OK, sure, I'll type that code*, but I don't really, I never really, I don't know I never made the connection. (Interview 1 of 5, February 10, 2012).

3.5.1.2 Rebecca's "terrible" array experience has epistemological components

In our second interview a week later when we discussed debugging, Rebecca returned to that experience in her Basic Programming class:

> Rebecca: My array project, my—the second project we did last semester, that I had a big error. I had to delete like half the code I worked on. Like, just cuz, like I had said last week, I didn't do very well on arrays. So.
>
> Interviewer: Oh. Was it cuz of one thing, or was it like a repeated—
>
> Rebecca: It was kind like a general, like, what I was trying to do didn't work with C.
>
> Interviewer: Do you remember what it was you were trying to do?
>
> Rebecca: I don't know exactly, but uh, I just remember, like, whatever I was tryin to do, cuz I had my friend help me, he's like "you can't do this this way," like, it just, C doesn't recognize whatever I was trying to do.
>
> Interviewer: Huh. So that was like, it sounds like it's almost kind of a case where your, your plain English of what should happen /yeah/ can't actually translate—it's not even that you're not sure how it's—
>
> Rebecca: It's like, it was impossible.
>
> Interviewer: It was never—
>
> Rebecca: It was never allowed.
>
> Interviewer: There is no word for that /yeah/ in this language.
>
> Rebecca: I {laughs}, that was back then, so. That was always fun.
>
> (Interview 2 of 5, February 17, 2012)

Epistemologically, two patterns stand out in Rebecca's recollections. First, as Rebecca remembers it, one of her core problems was that her visions for how her program should accomplish things were "impossible" and "never allowed" in C. Second, in order to get past that obstacle, she ultimately typed code a friend told her to without feeling like she understood it. As this analysis unfolds, I think these two points give us a crucial starting point to understand Rebecca's practice. The experience underscored a gulf between ideas Rebecca wanted to invoke and the syntax she needed to realize them in C. And, her way of dealing with that gulf was to make an end-run around it by taking and using code she didn't fully understand.[51]

It's worth noting that whether her friend's code worked to spec[52] is immaterial to the two epistemological issues I raise. First, if the code she copied worked, Rebecca would get a good grade, but she wouldn't understand how her project functioned. And, if it didn't work, she would get a bad grade *and* she wouldn't understand how her project functioned. Ultimately, no matter how the code performed, Rebecca would likely be in a tough spot if she ever had to maintain, revisit, or refactor the code she copied. Second, no matter how her copied-code project functioned, it represented *someone else's* articulation of how a procedure should go. That is, the copied code was an artifact of someone else bridging the gulf between high-level idea and articulation in C. Rebecca still found difficulty traversing that gulf.

3.5.1.3 Rebecca's array project experience had a lasting effect on her confidence in programming

As that interview drew to a close, I wrapped it up with what had become standard protocol: asking whether there was anything else she wanted to talk about.

> Rebecca: I think I'm good, I—just
>
> Interviewer: Anything else on your mind?
>
> Rebecca: Sorry I'm not the greatest programmer {laughs}
>
> Interviewer: No, hey hey hey. That's {shakes head}, first of all, I mean, um, how come you're apologizing?
>
> Rebecca: I dunno. It's a study, and I'm probably not the best test subject. (Interview 2 of 5, February 17, 2012)

Rebecca's spontaneous apology spawned a discussion about why she feels diffident in programming. For her, that diffidence traced its way back Basic Programming. Feeling lost and stuck on that array project was more than just a tough experience; it was a turning point:

> Rebecca: When I got lost I think is when I started losing that confidence thing. And then I never really got it back, so.

[51] I say this without any intended judgment (or pejorative connotations) about what Rebecca did. Given the assessment structure of the course, where 90% of a student's grade came entirely from whether the program worked to spec, she may have felt that getting a good grade by using code she didn't understand was preferable to getting a bad grade with code that didn't work.

[52] Here, I'm using "worked to spec" as a shorthand for "worked in such a way that it passed all of the input/output tests the instructor would run to determine 90% of a student's grade."

> Interviewer: When was it you started to get lost?
>
> Rebecca: Halfway through last semester, like with the arrays in the second project. So…. I guess it was like, demoralizing, kind of, and, like, like I don't want to sound cocky or arrogant or anything, but in high school I did very well in academics. And, so, I never really, like anything that I had to work for, I worked for for a short period of time, got it, and like I'd understand it, and if I didn't, like it was really easy for someone usually to explain it to me. And, I felt like I didn't really get that after I lost everything on that second project, I was like, I wasn't able to build myself back up out of that, which probably stems to why, not as confident in the class. (Interview 2 of 5, February 17, 2012)

As I interpret how this array project experience may be playing a part in Rebecca's later approaches to programming, I stress that my analysis is about Rebecca's perceptions and recollections.[53] It's possible that her ideas for how her project should work wouldn't have been implementable in any language, let alone in C. And, while I think that possibility is unlikely, I have to concede that I only know what Rebecca recalled; I wasn't present for the experiences she's describing. Nevertheless, the question for my research is not "can we know what really happened in that experience?" Rather, my question is, "how might Rebecca's *image* of that experience — with its specific epistemological facets — be playing a part in her approach to programming now?"

Part of the answer to that question may be in her quote above. She talks about losing her confidence and "never really [getting] it back." Later, she says "I felt like I didn't really get that after I lost everything on that second project." Because of the peculiarities of speech, I had a difficult time parsing Rebecca's statement about losing everything on that second project. It wasn't clear to me whether there was a pause between "get that" and after, but the meaning of "that," and in turn the meaning of the sentences, depends on the antecedent of "that." So what she said could have two consequentially different meanings:

1. "I feel like I didn't really get **that**. After I lost everything I lost everything on that second project, I was like, I wasn't able to build myself back up out of that." In this version, "that" may mean the topic—arrays—or the project as a whole. I might then interpret Rebecca's speech as saying this project was a turning point because of the particular topic. Her being "unable to build myself back up out of that" reflects a kind of debt that might have piled up when later assessments depended cumulatively on that early topic. Her failure to grasp arrays was thus cumulatively punishing because arrays formed a core part of many projects that followed.

[53] Due to deliberate study limitations I don't have access to the actual code from that project. To be as conservative as possible in our data collection and respect the privacy of participants, we asked participants to grant us access only to folders that would contain their Intermediate Programming projects. Outside code/data, including code from prior semesters, was excluded from the scope of our code snapshot data collection.

2. “I feel like I didn’t really get **that** after I lost everything on the second project. I was like, I wasn’t able to build myself back up out of that.” In this version, “that” may mean her ready insight into topics or her facility in understanding when friends explain them to her. The distinction I draw in this second interpretation is “that” refers to a kind of continuity in her identity—someone for whom school always came easy. The array project disturbed that continuity because it was an instance in which things no longer came easy to her. And, the experience may have been so troubling that she was never able to re-establish being able to easily grasp things in programming. The result, then, is that she couldn’t build herself—in the sense of her continuous identity as one who quickly grasps school topics in general and programming topics in particular—back up after the array project.

Distinguishing between these possible meanings of Rebecca’s speech might seem pedantic. But, I think the bifurcation is important for theorizing about learning and instruction in computing. If Rebecca’s difficulty was about the topic of arrays, a plausible narrative is that the difficulty of that concept—and the assessments that continued to depend on it—caused a run of bad grades, and the bad grades shook Rebecca’s confidence. But, if Rebecca’s difficulty comes from troubling her sense that she has a ready understanding of programming, the attack on her confidence is more complex. It’s not just that she doesn’t get arrays, it’s that she feels she doesn’t readily understand other new code she’s seeing in the course—even code that doesn’t strongly rely on arrays. In this view, the confidence Rebecca speaks of isn’t just a product of getting good grades; it’s part of a feedback loop in which grades, confidence, and a feeling of ready insight into programming are all components.

That potential coupling of grades, confidence, and feelings about ready insight into code matters because it further pushes the discussion toward epistemological considerations. It is why, for example, I was paying such careful attention to Rebecca’s recollection of copying down code her classmate told her. Instead of helping her confidence in programming by potentially boosting her grade, the copying may have diminished her confidence because she couldn’t understand why someone else’s code worked.

3.5.1.4 Epistemological issues continued to interact with Rebecca’s confidence in Intermediate Programming

Rebecca started our fourth interview with a warning to me that she might “rant” about programming.

> Interviewer: What would you rant about? Go ahead.
>
> Rebecca: Ohh god. Um, I am not doing well {laughs} in the class right now.
>
> Interviewer: OK
>
> Rebecca: And it’s not good, so. Yeah, like, we just had our last project due, which I did a lot better on than the first project
>
> Interviewer: Uh-huh

> Rebecca: Which is making me really happy. Um, but, everything about programming just upsets me, because there's the two guys, like I had mentioned before who, uh, help me
>
> Interviewer: Mmmhmm
>
> Rebecca: Uh, one of them, he—like I worked every day from the day we got this project on this project, and my project still didn't turn out perfect. He wrote his twice in one day because he lost the whole thing. And, his was perfect. And, I dunno—it just irks me *so* bad that that, it can be that much of a disparity between people and programming. (Interview 4 of 5, April 6, 2012)

In her telling, Rebecca wasn't the only person upset by the relative ease with which this student programmed.

> Rebecca: we didn't say it to him, but some of us were like, it's, we just found it like, so, like aggravating. Like, cuz there's me and another girl who, on my floor who, she's uh, never had programming experience either—
>
> Interviewer: Uh-huh
>
> Rebecca: And we're just like it gets so aggravating that it comes easy, like to him, and that he does it well, but, that's just—he ju—he is very good at school. He's very smart. So. (Interview 4 of 5, April 6, 2012)

Note here that Rebecca is connecting the idea that programming came easy to this student to the notion that he was "very good at school." In this moment, the statement seems to suggest Rebecca has a *fixed mindset* {Dweck and colleagues} about programming expertise. While it would be naïve to say Rebecca believed — globally — that programming came easily to those who were "good at school," it's nonetheless true she saw a discrepancy between how easy programming was for her and her friend and how easy it was for this "very smart" classmate.

As we kept talking about this discrepancy, Rebecca offered further evidence that struggling with programming troubled her academic identity. In high school, she "didn't have to study a lot" because "a lot of things came relatively easy" (Interview 4 of 5, April 6, 2012). But, her experiences in Intermediate Programming pushed her into a new position. As she said,

> now I'm on the other end and I'm realizing how aggravating it is…. And, I don'—I guess it's just a shock, cuz I do not like not being good at things {laughs}. So..." (Interview 4 of 5, April 6, 2012).

Again, Rebecca's comments stress the frustration of feeling like programming comes easily to some people but not to her.[54] And, they are charged with emotion. She

[54] Or, at least, at that moment, in that semester, it did not seem to be coming easily to her.

speaks of being “on the other end,” connoting an idea of a structure whose poles have people who “get” programming on one end and those who don’t on the other. It’s “aggravating” to be on that end, and a “shock” to be there for the first time.

I asked Rebecca to reflect on whether she could ever remember seeing classmates in the position she was now, where something seemed to come easily to Rebecca but not to them.

> Rebecca: Um. I mean, there was a—there’s a friend of mine, who, from my high school, who, I, I mean just always got math. And, uh, she was struggling in math a lot.
>
> Interviewer: Mmmhmm
>
> Rebecca: But her attitude was, uh, “I don’t need math. I’m gonna be an English major.”
>
> Interviewer: OK
>
> Rebecca: So, she didn’t even care about it
>
> Interviewer: Oh, OK
>
> Rebecca: Whereas, like, I was like, I’m an electrical engineer major, I actually need this. So. Like that’s the only time I think I’ve ever felt like someone’s had that feeling that I probably did.
>
> Interviewer: Right.
>
> Rebecca: And, I mean I’m sure there are people that I—uh, that I didn’t know, or didn’t realize they had that feeling. But, it’s just, interesting to be on the other end. {6 second pause}
>
> Rebecca: Umm. Just. Ugh. I’ve like, debated withdrawing from the class. And, like, I’ve talked to three different advisors or whatever, and they’re like “don’t do it. It’s not—you can still bring your—you can bring your grade up. And even if you don’t you have freshman forgiveness,[55] so.”

Rebecca, unlike her friend, couldn’t afford *not* to care about Intermediate Programming; it was a required course for her electrical engineering major. But, the combination of feelings she had—consisting partly of a feeling that programming did not come easily to her—pushed her to think about withdrawing from the course anyway. And, Rebecca was not in the habit of withdrawing from classes. Intermediate

[55] In the interview, Rebecca explained that Freshman Forgiveness was a policy extended to students during their first 24 credit-hours at the university. If a student gets a grade she’s unhappy with, she can retake that course and her second final grade will replace the first.

Programming was the first and only class where she was experiencing this level of difficulty (Interview 4 of 5, April 6, 2012).

It would be tempting to think Rebecca struggled in Intermediate Programming because she didn't work hard enough. But, by her own assertions she *did* put a considerable amount of effort on projects. On Project 2, for instance, she said "I worked every day from the day we got this project on this project, and my project still didn't turn out perfect" (Interview 4 of 5, April 6, 2012). Figure 5 below shows Rebecca's activity over time from March 5, the day Project 2 was handed out, to March 28, the day it was due:

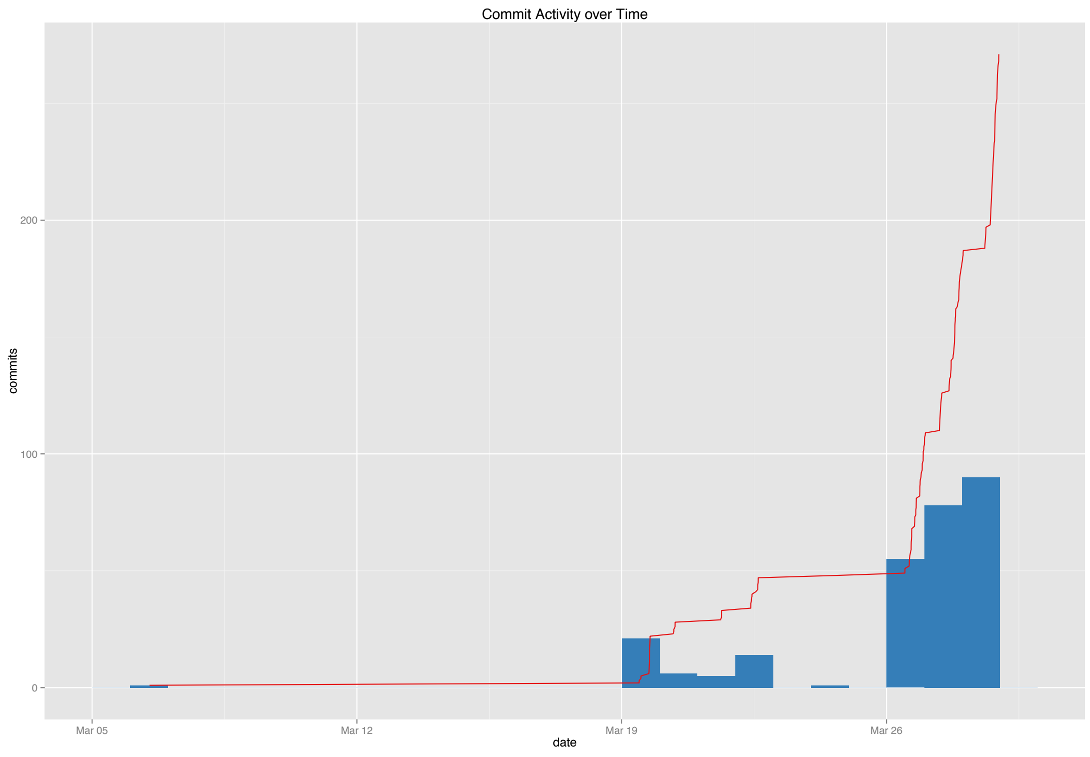


Figure 3-5 – Rebecca's compilation activity over time for Project 2. Each compile a student initiates creates a commit, so long as there has been a change to the underlying code. The height of each bar maps to the number of commits recorded that day. The red line charts *cumulative* commits over time. The red line is steepest in periods of frequent activity and shallowest in periods with little or no compile activity.

From the data we can tell Rebecca was compiling Project 2 code almost every day between March 19 and March 26. Moreover, roughly two thirds of compilations for the entire project happened in the last 3 days leading up to the deadline. So, at the very least our data shows she was actively compiling her code for 8 out the last 10 days.

Rebecca registered no commits between March 12 and March 19. There were also other days for which no commits were recorded. One possible cause for those zero values is our data collection system was faulty. A second possible explanation is that Rebecca was not working. But, a third explanation is that Rebecca was working but not compiling. This third explanation actually seems most plausible. It fits the pattern of Rebecca's work on project 1, where she wrote dozens of lines of code before ever compiling (Interview 2 of 5, February 17, 2012). It also accounts for

Rebecca's lack of compilations despite her claim that she worked on Project 2 every day.

So, we can be fairly certain Rebecca was spending time on projects.[56] Another possibility for why Rebecca was struggling might be that she wasn't studying. To address that conjecture, we have to step back to when I asked Rebecca whether other classes — classes other than Intermediate Programming — had ever pushed her close to withdrawing from them. She said no.

> Rebecca: the thing is, I *don't* know how to like, study for programming. Cuz like, I look at code and stuff, and I try, and I work every day on it, but like, even when—I dunno, it just, does not come naturally.
>
> Interviewer: So, what is it like when you study it? What do you usually try to do?
>
> Rebecca: Uhh, a lot of times, like, I'll read through the notes we took in class, like, cuz, what we're doing now—we have a project, uh, we're doing linked lists, and like, malloc'ing data, which is, uh, *you* storing it in, as you go. And, like, I get the theory and everything behind it, and, like it makes sense what it does, but, I just like try and look at his code examples and stuff, and try and replicate it, but I don't know what's happening right now, but it's not doing it correctly. I just, like, my *biggest* problem, I realized, in programming, is I think I have the logic, my logic just doesn't transfer to code. Or I don't know *how* to transfer it correctly.
>
> Interviewer: So, um. when you're spending time studying, um, do you spend most of it on your computer trying to actually program or—
>
> Rebecca: A lot of it is on my computer and a lot of it is also trying to just look at his code and notes in class. Uh, that's probably about split, 50/50. Uh, because, sometimes I'll just end up staring at my computer screen, like, "what do I even type in to try?" So, I try and go look at his examples, so.
>
> (Interview 4 of 5, April 6, 2012)

There are two striking features of this exchange.[57] The first is that Rebecca felt she didn't know how to study for programming. The second striking feature of this exchange is how Rebecca elaborated on what it might mean for programming to come naturally to her.

---

[56] The specific nature of *what* Rebecca's work looked like on Project 2 is the subject of the first study of this dissertation, so I don't expand upon it substantially in this study.

[57] There are actually three striking features. The one I left out above is that Rebecca describes malloc'ing as *you* storing in data. I think a great deal of Rebecca's difficulty with malloc specifically hinges on what she means by "you" storing it in.

Rebecca feels programming doesn't come naturally to her because she routinely finds herself unable to transfer her "logic" to code. As a follow-up to Rebecca's answer, I asked what "logic" meant to her in this case.

> Rebecca: Like, like how to get the—like, if I need to get a certain data, like I have a process of how I want to get it, I ju—and I'm like, "Oh, you would just do this," but can the *computer* do that? Can I—and it's my—me being able to tell the computer to do that is where I get lost. (Interview 4 of 5, April 6, 2012)

The tension Rebecca is describing is one I see as strongly epistemological. First, it's epistemological in the sense that it reflects a struggle to articulate knowledge of *how to do something*. Papert (1980) and Abelson & Sussman (1996) would describe such a struggle as a pertaining to *procedural epistemology*: the struggle concerns a particular kind of knowledge work in which the goal is not to describe what is, but rather how to. The second sense in which it's an epistemological struggle is closer to how Hammer, Elby, and colleagues use the word epistemological: what does students' activity reveal about how they orient toward knowledge and knowing in a discipline (Hammer et al., 2005; Hammer & Elby, 2002, 2003; Scherr & Hammer, 2009). In this second sense, we can observe that Rebecca defines her difficulty in terms of the creation of proper code. We can also pose the question "if Rebecca struggles to transfer logic to code, what directs her activity as she *tries* to transfer logic to code?"

Later in the interview, she compared the iTunes project — on which she was stuck — to Project 2.[58] In so doing, she offered another telling idea about the relationship between her logic and "syntax":

> Rebecca: Like, cuz, I guess I feel like I did a lot better on the last project because it was so similar to stuff that I'm used to /Mmmhmm/ um, whereas all this is brand new and I still don't have any of the syntax down yet. /Mmmhmm/ Like, if-statements and while-loops, they make sense to me, whereas I'm still like, trying to grasp at this stuff.

By "this stuff," the context of the conversation suggests Rebecca was referring to dynamic memory allocation and malloc, which we had been discussing.

It seems sensible to think if-statements and while-loops might "make sense" to Rebecca because their syntactical form aligns strongly with their computational operation.[59] By that, I mean in principle the designers of C could have chosen *any*

---

[58] Project 2 did not require students to use malloc() to dynamically allocate memory. Project 3 did.

[59] To borrow terminology from Sherin (2001) and rebut Saussure (1986) at the same time, the symbol template for an if-statement is strongly connected to its conceptual schema because we so often use everyday language in a manner concordant with an if-statement's branched control flow. In this instance, the relationship between signifier and signified is the *opposite* of arbitrary. The similarity between the syntax

symbol to represent branched control flow, but they chose a word — if — that invites connections to its meaning in everyday speech. If-statements in C split control flow based on a Boolean expression's value; while-loops allow continued iteration until a Boolean expression turns up false. Both structures have everyday analogues with similar conceptual properties. A parent might say, "if you're staying out past 9, let me know," or, "while we're gone, can you make sure Sadie doesn't eat anything from the garbage?" If a child will be home by 8, there is no need to let that parent know. Similarly, it's reasonable to expect that once the parents return, the responsibility of keeping Sadie out of the garbage no longer falls solely to the child.[60] The point of this observation is that Rebecca may feel certain kinds of programming structures "make sense" not simply because they were covered in introductory programming, but because something about their syntactical form offers an affordance for thinking about how they work.

3.5.2 Rebecca has productive capacities for making progress on software design work

In our fourth of five interviews, Rebecca was having trouble making progress on Project 3, which the instructor described as an "online music server."[61] "Like, I am so lost, like I don't even know where to start," she said. We discussed why she was stuck. Then, with my help, we began to work through what Rebecca identified as a core cause of her confusion. In brief, the assignment contained a diagram showing the data scheme students were required to use to store data about the music (Figure 8 below). Specifically, the scheme required an array of pointers — each entry of which pointed to an array of characters — as a mechanism for storing data about song titles for the music server. Rebecca's work to resolve her confusion offers evidence for her productive capacities in designing programs. In this section, I argue:

1. Rebecca made sense of the array-of-pointers design[62] using talk and gesture. She was able to explain both why an array of pointers *should* exist in C as well as suggest candidate syntax for what might create such an array.
2. When I proposed we pretend her candidate syntax worked, Rebecca articulated through gesture, talk, and written pseudo-code how part of her design would incorporate that syntax.
3. After the interview, Rebecca resumed work on the project and incorporated a version of the design she developed in the interview into her code.

---

for an if-statement and an understanding of what that statement does is by design, because C itself was a designed language.

[60] Some parents might object that family members should *always* be on the lookout for Sadie rooting through the trash, that these parents were just emphasizing the point. And, reasonable children might in general try to heed that. But, what makes computers computers is their lack of qualms about abandoning any and all responsibility to keep Sadie from the trash once the parents return.

[61] Students in my study colloquially referred to this project as the "iTunes project," so "the iTunes project" and "Project 3" all refer to this project.

[62] By "design scheme," here, I mean the instructor-provided diagram indicating how students were to structure their data for Project 3.

3.5.2.1 Rebecca made sense of the array-of-pointers design using talk and gesture

The figure below shows the instructor-provided diagram for how students should structure their data.

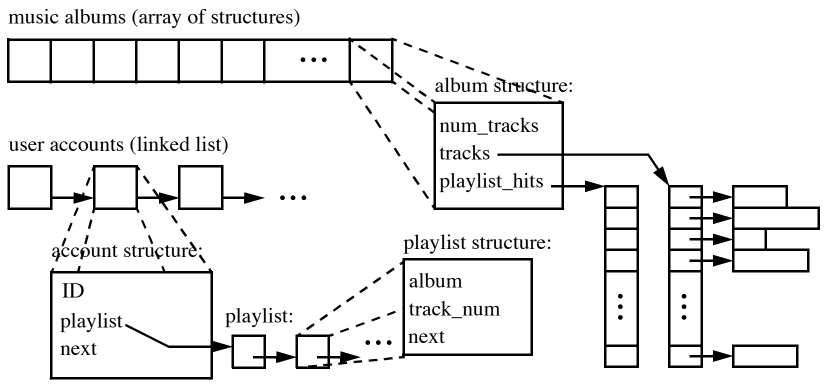


Figure 1: Recommended data structures for implementing the on-line music server.

integer that specifies the number of tracks contained in the album. The "tracks" field is a pointer to a pointer array that holds the title of each track. The pointer array contains one pointer for each track in the album, so it has "num_tracks" entries. Each pointer in the pointer array is a pointer to a character array that holds a string containing the track title. Lastly, the "playlist_hits" field is a pointer to an array of integers, one per track (so there are "num_tracks" integers in this array). Each integer in this array records the number of users that have included the corresponding track in their playlists.

Figure 3-6 – The instructor required students to use this data arrangement for storing information in the music server. The scheme uses an array of pointers to represent the track names of an album.

Rebecca would have seen this diagram before because the project assignment was given out 11 days prior to our interview. But, the data I present is the first time I had any access to how she was thinking about it.

At the time of the interview Rebecca felt stuck on handling issues with *music albums* — the top-left and top-right structures depicted on the diagram. The assignment required that "all of the data structures described [in this diagram]", including music albums and all of the data they referenced, "must be allocated dynamically" (Instructor-provided Project 3 description). So, Rebecca and I started discussing what parts of that scheme she understood. Table 3 below presents what she said and what she did while beginning to explain her understanding.[63]

63

Table 7 - Rebecca describes an array of structures using gestures (Interview 4 of 5, April 6, 2012)

What she says	What she does
Like, the—the thing that makes sense to me that I, uh, got right now, is that, to malloc the array, to {bounds a wide space with her hands} make the array a certain length, do that.	
And then I know that each of the {chops out three invisible boxes in the air} I guess nu—characters—I—uh, I'll call 'em characters.	
Like, each of the spots in the array holds a structure and each of the structures holds three things.	(gestures obscured by the computer ☹)

Rebecca's gestures create a kind of virtual object that looks much like the music albums array in the diagram. As she chops out spaces for each of the "characters," those spaces are in series and fit — more or less — within the larger area she bounded with her when she said "to malloc the array." The spatial subdivision is key because it offers supporting evidence that in some way, Rebecca might have been thinking about the album structures themselves as being *contained* in or by the array. And, that spatial containment metaphor resurfaced later when we discussed the array of pointers, which I'll get to in a moment.

As Rebecca explained her understanding to me, I took to writing down what I understood her to be saying on paper:

> Interviewer: OK. So you have, so here's what you just said. You said {writes} "malloc the array" um, "each element has a structure," and then "each structure has" you said—
>
> Rebecca: Has three parts
>
> Interviewer: OK. Has three parts. OK.
>
> Rebecca: Yeah. So, and the first part I can do 'cuz it's an integer {laughs}. The first part is just the number of tracks on the album. So—
>
> Interviewer: Oh, oh. So, of the three parts [[I'm sorry
>
> Rebecca: Yes]]
>
> Interviewer: I [[{untillegible}
>
> Rebecca: Oh yeah, sorry]]
>
> Interviewer: So you're saying, um, and then—and then this is number of tracks, which is an integer, right? /yeah/ OK (Interview 4 of 5, April 6, 2012)

Figure 9 below shows what I wrote as we worked:

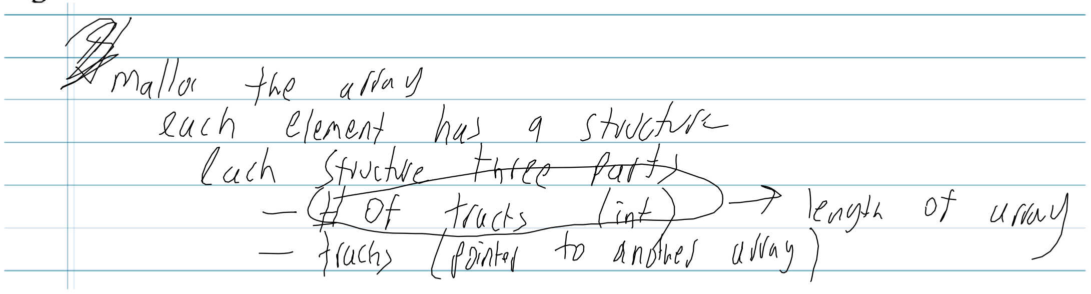
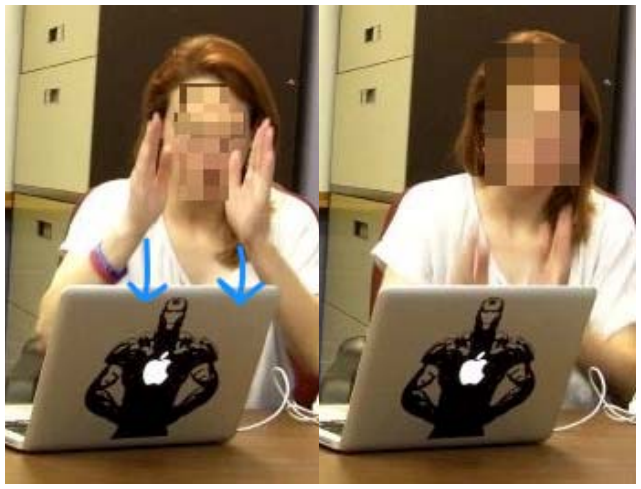
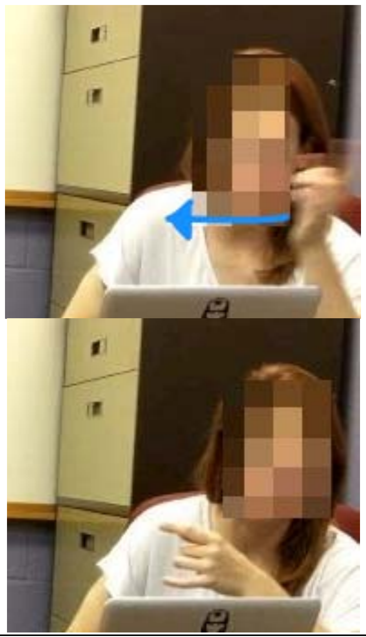

malloc the array
each element has a structure
each structure ~~three parts~~
— # of tracks (int) → length of array
— tracks (pointer to another array)

Figure 3-7 – I write down what I understand as Rebecca explains the overall data structure of albums to me

It wasn't the first time — either in my series of interviews with Rebecca or in that interview in particular — that I tried to write down what she said. But, I highlight it here because it became part of my activity as we worked together in this episode. Rebecca would articulate something, often through talk and gesture, and I would try to write down and offer back to her what I understood her to be explaining.

As we continued, we landed on trying to understand the "tracks" portion of the album data structure. Storing tracks was where Rebecca felt she was getting "lost":

Table 8 – Rebecca says gets lost on the array of pointers pointing to track names (Interview 4 of 5, April 6, 2012)

What she says	What she does
Rebecca: And, the next one is where I get lost, on the tracks part. Because it's a pointer {2 sec pause} to {hands make a vertical cylinder in front of her; note that the array is represented on the assignment as a narrow, vertical rectangle}—	
it looks like another array that's ||pointing|| |{left hand crosses from her left to right, at chest height, index finger extended in the direction of motion; this is the same direction (from Rebecca's perspective) in which the pointer array on paper points to each track name}| to the names.	

Rebecca's gestures for the array of pointers again spatially mimic the depiction of the array on the page: a vertical cylinder. The reason I note the orientation of her array gestures is that for practical purposes arrays as represented by bits in C have no orientation in space that the programmer could know, much less care about. An array cannot be parallel to the ground, nor can it be oriented toward the ceiling.[64] But, the assignment *depicted* the album array as horizontal, and so too did Rebecca's gestures

[64] The computational hardware representing the array does have an orientation in space, but that's clearly not what is at issue here. The whole point of the array is to be an abstraction away from the soldered transistors, capacitive plates, flipped switches, vacuum tubes, tinker toys, or other physical means that store the state of the array.

for it. The assignment *depicted* the pointer array as vertical, and so too did Rebecca's gestures for it. In other words, the observation that Rebecca's gestures align with the assignments pictures might suggest that she is doing more than thinking about arrays in the abstract. Rather, some elements of the representational features of the page have made their way into her activity and, arguably, her cognition.[65]

Rebecca continued on to try thinking through why an array of pointers was part of the design.

> Rebecca: But, I've always been confused as to why you—I guess, I always—I was always like, when we learned pointers, I was like "why do you need pointers when you could just call it the name? Why do you need two names for it?" But, I think, what—at least what I'm seeing here maybe is like, this is just another array /Mmmhmm/ because you can't put more than one character in each element of the array, so that is just an array of pointers that point to strings. And the strings are the names.
>
> Rebecca: So, that would make sense. So you would—the tracks would point to the array, and so you could do the pointer of that {1.5 sec pause} The pointer to the array, an array—the array of like 1 would be a pointer to track 1's name.
>
> Interviewer: Mmmhmm
>
> Rebecca: So. OK, so—see that part makes sense now. I just—no idea how I would ever access that. Or, store it, I guess, is the better word. (Interview 4 of 5, April 6, 2012)

What "confused" Rebecca has an understandable origin. Most of the work she would have been doing until that point would have used named variables. For example:

```
int age;
float weight;
char *name;
```

If age, weight, and name are already-named pieces of data, why would something ever need to refer to them indirectly? It would be as if we had to call me "The second-born son of the second-born daughter of Henry" for some reason. I already have a name. Why do I need another name that uses the names and relationships of my forebears to refer to me when historically my given name has perfectly fine for all referential purposes?

[65] One could argue that gesture is possibly just undirected flailing, and that the correlation between Rebecca's gestures and the page is weak at best — the sort of thing you'd expect to happen some percentage of the time anyway under the assumption that the pictures on the page have nothing to do with how she's thinking about it. I concede that it's *possible* the correlation isn't structurally meaningful. But, I think the correlation is meaningful given how many other times her gestures align with canonical written representations of computational structures and processes.

Rebecca's way of finding sense in pointers was to observe that you can't put names — each of which is defined as an array of characters — into an array because "you can't put more than one character in an array." To verify I was understanding her idea correctly, I tried to restate it:

> Interviewer: OK, so you were saying the reason you—when you're tryin to convince yourself like—you originally you were thinking "why don't you just put all the names [[here
>
> Rebecca: Yeah]]
>
> Interviewer: And you were [[saying the reason is
>
> Rebecca: saying, no]]
>
> Interviewer: because==
>
> Rebecca: ==That makes sense now. Yeah, you can't put more than one character in an element. And, it's a whole name of a song, so you would need to point—that element would just {gesture obscured by my laptop} be a pointer to a name. OK. (Interview 4 of 5, April 6, 2012)

The latching and overlapping turns of talk above offer strong evidence that Rebecca's explanation aligned with my interpretation. And again, notions of *containment* come into play. Rebecca says, "you can't *put* more than one character *in* an element" in part because you're talking about "a *whole name* of a song" (my emphasis added here). If Rebecca has a capacity for thinking about the canonical concept of type-restrictions on arrays, it's manifesting here as a part-physical metaphor, evoked through both talk (the quote above) and gesture (Table 4), in the service of making sense of a design decision.

As she continued, Rebecca discerned that because the input data contained the number of tracks on each album, she could use that number to define how long the pointer array needed to be. But, something was still troubling her.

> Rebecca: So. {4 sec pause} That would {6 sec pause}
>
> Interviewer: What're you thinkin?
>
> Rebecca: Just that, um, I'm just tryna think, cuz like that makes sense, but I don't know how to get to that. Cuz, how do you make a pointer point to a random array that you just made? Uh, I guess {4 sec pause}
>
> Rebecca: Uh, p—the pointer syntax I need—I would need to go back and look at them because now you have an array, each array—my issue—cuz like now that makes sense. Like, it's just how do I get to that point of making that work?
>
> Interviewer: [[So

> Rebecca: And this is what]] I mean by my logic /OK/ I get the logic and the theory behind it but I don't know how to actually put it into C.
>
> Interviewer: OK, so in other words, what makes ||sense|| |{air quotes}| to you know /Mmmhmm/ that I guess didn't before is that *that* {gestures to the instructor's structural diagram on screen} is a—that's an understandable arrangement /yes/ for the data
>
> Rebecca: Yes.
>
> Interviewer: But then, how do I [[make it
>
> Rebecca: make it]] Yeah {nods}. (Interview 4 of 5, April 6, 2012)

When Rebecca said "I don't know how to get to that," I take "how to get to that" to mean *how to write the code for that procedure*. We had just agreed that design-wise, she could use a number provided in the input data to define the length of the pointer of array. But, she was stuck because of her difficulty translating the "logic" we developed into code.

From the outside, what Rebecca and I had done together was something I think crucially important to the design of a program. We worked to understand how someone else chose to structure their data.[66] We tried to decide for ourselves whether and how those structures made sense. Along the way we created ephemeral objects of that sense-making in the form of talk and gesture as well as the durable artifacts of my inscriptions. But, none of that work resulted in C code.

For Rebecca, the lack of certainty about *how to make* our ideas in C code was a marked concern. And, again, that concern makes sense. We had just convinced ourselves that we could understand *why* an array-of-pointers. We even figured out *how* we could use the input data to provide the length of the arrays we would need to make. But we had not laid out how to declare the array *and* preserve a reference to it ("how do you make a pointer point to a random array that you just made?). Without that piece — a piece Rebecca was unsure of and felt she would need to look up — our solution clearly wouldn't work.

Absent that piece, Rebecca tried thinking of possible ways to fill in the missing syntax.

> Rebecca: Cuz, I'm like, like I'm trying to think right now of things that I would try /Mmmhmm/ but I don't know what I would try first because, all I'm thinking right now is that I know tracks is going to be a pointer. /Mmmhmm/ So you make that a pointer.
>
> Interviewer: OK.
>
> Rebecca: But I'm not sure where it would point to, because you don't have that array made yet. So I {3 sec pause} so you'd have to, I guess you could

[66] That the "someone else" was in this case the instructor is important for larger power dynamic considerations.

> make the array, and then make it point to the array, but {3 sec pause} Cuz you have to make an array of pointers.
>
> Interviewer: Mmmhmm
>
> Rebecca: And I don’t know how to do that. Because I’ve made arrays of characters and integers and stuff before /OK/ but never pointers, so.
> (Interview 4 of 5, April 6, 2012)

Rebecca was seeing a temporal problem: how could she reference an array that didn’t exist yet?[67]

Rebecca started exploring ways to solve that problem of getting tracks to point to the array of pointers.

> Rebecca: But I’m not sure where it (tracks) would point to, because you don’t have that array made yet. So I {3 sec pause} so you’d have to, I guess you could make the array, and then make it point to the array, but {3 sec pause} Cuz you have to make an array of pointers.

> Interviewer: Mmmhmm
>
> Rebecca: And I don’t know how to do that. Because I’ve made arrays of characters and integers and stuff before /OK/ but never pointers, so.
> (Interview 4 of 5, April 6, 2012)

Rebecca landed on a potential solution: make the array first, then make tracks point to the array. But, with that solution came another problem: how do you make an array of pointers?

3.5.2.2 Rebecca could propose candidate syntax and build pseudocode around it.

Rebecca came up with an idea to create an array of pointers, but she wasn’t sure if it would work.

> Rebecca: if I were to d—when I declared the array, if I were to—I dunno if you even can, like, whenever we—we do ints, and then it would be like star-p /Mmmhmm/ and then star-p would be the pointer /ok/ I don’t know if you can

[67] As a crude analogy, imagine trying to write your own will. If you think forward in time, you might later have children to whom you’d like to bequeath your assets. But how can you enumerate those children in the will now, before you’ve had them? You need your will to refer to something that does not exist at the time you write the will. Extrapolating forward and rather ludicrously, we could imagine you are always capable of producing children. So, up until the day of your death, there is always the seeming risk that your will might need to refer to a beneficiary who does not yet exist. Recognizing this subtle problem as Rebecca did is an act I consider to be sense-making, even though she as-yet still did not have a solution.

do like int star-p ||bracket-bracket|| |{makes square brackets with hands}| and that would be an array or not.

Interviewer: [[OK

Rebecca: But I don't know]] if you can do that.

Notice what's happening here. Rebecca is taking patterns that she knows work:

```
int age;      // creates an integer variable called age
int ages[35]; // creates an array called ages,
              // 35 integers long
int *p;       // creates a pointer-to-an-integer;
              //     said pointer is called p
```

and considering a pattern that might work:

```
int *p[35]; // Rebecca is not exactly sure what this will do
```

Notice further that in this exchange, Rebecca wondered "if you can do that." Determining the referent of *that* was, again, consequential. One possibility was Rebecca meant "int *p[35]" might not be the proper syntax for creating an array of 35 pointers-to-ints. If so, Rebecca was struggling with her aforementioned problem of not being able to translate her logic to code. Another possibility was Rebecca meant "I don't know if there exists *any* syntax in C to create an array of pointers." So, I followed up on this latter interpretation. We both agreed it *should* be possible to create an array of pointers, it was indeed just a question of whether that syntax accomplished it (Interview 4 of 5, April 6, 2012). So, that left the former interpretation: it looked like Rebecca was unsure how to "transfer her logic to code."

We both agreed creating an array of pointers should be possible. So, I asked Rebecca to suppose her candidate syntax worked. She readily agreed.

Interviewer: OK. So then, um, what if we just pretended for a minute, that that, like==

Rebecca: ==That that works==

Interviewer: ==That that worked==

Rebecca: ==OK==

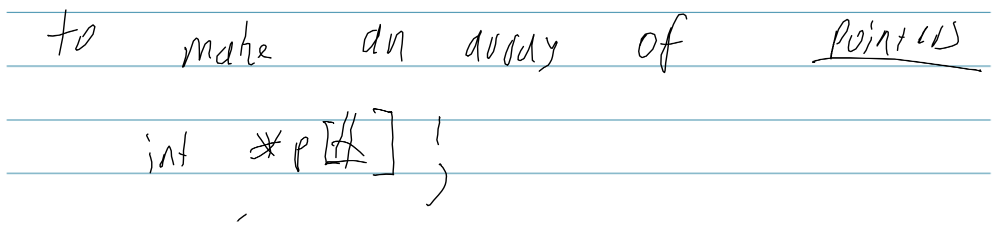
Interviewer: ==OK. So then, um. {begins writing} So then you might write like to make an array of {stops writing} actually what would we call this? This is==

Rebecca: ==array of pointers, I guess==

Interviewer: ==OK. {under breath} ||pointers|| |{writes "pointers"}|. It'd be, well. So. The—the one we had was like ||int, star p, um, it would be some number|| |{writing}|, um, and I guess that’d be it in order to just declare /Yes/ that right? OK.

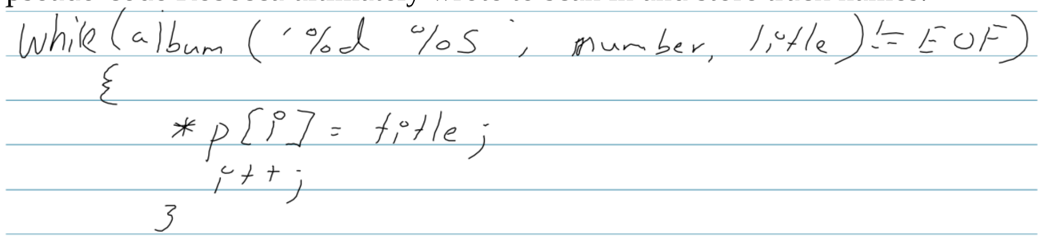
gure 3-8 – The candidate pseudo-code Rebecca dictated to me for declaring an array pointers

In this exchange note every turn boundary is latched. Once I bid for pretending, Rebecca accedes. When I’m unsure what to call the array, Rebecca finishes my sentence. Rebecca rapidly approves the final product before I even finish my last sentence. Figure 10 below is the syntax I wrote in the exchange.



```
to make an array of pointers
      int *p[#] ;
```

If that syntax created the array-of-pointers, the next challenge for storing track names was how to access elements in array-of-pointers. Figure 11 presents the pseudo-code Rebecca ultimately wrote to scan in and store track names.



```
while(album('%d %s', number, title)!=EOF)
    {
        *p[i] = title;
        i++;
    }
```

Figure 3-9 – Rebecca’s pseudo-code for scanning in and storing track names

The next section will show the rich talk and gestures that accompanied Rebecca’s generation of this pseudo-code. I argue activity in the form of gesture and talk constituted and sustained Rebecca’s in-the-moment approach to programming.

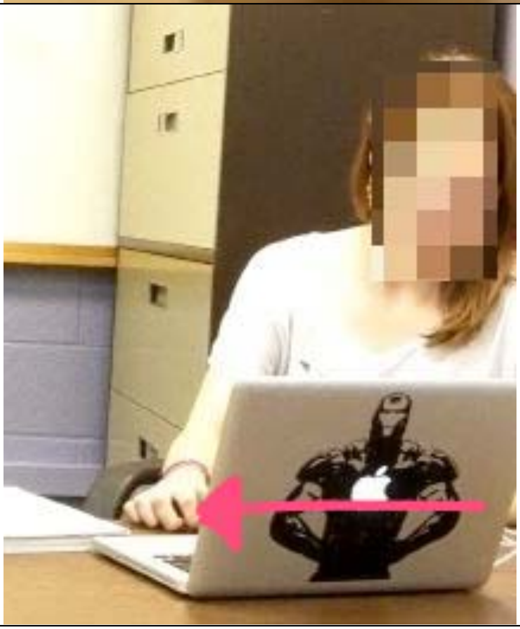
3.5.2.3 Rebecca’s talk and gestures constituted and sustained her in-the-moment approach to programming

Throughout this section I detail the speech and gestures that accompany Rebecca’s pseudocode production. Along both modalities I saw evidence of a marked shift in Rebecca’s orientation. It was a shift from being hunched, silent, thinking over the keyboard to being animated, open, and gestural toward me. It was also a shift from seeming diffident, uncertain, and hedging in speech to seeming confident, assured, and able to dispatch my questions. Taken in their totality across the episode, these subtleties constitute evidence of an approach that strongly differs from those Rebecca had when discussing her struggles in the course (section 3.5.1).

First, Rebecca suggested she could write the pseudo-code as we talked.

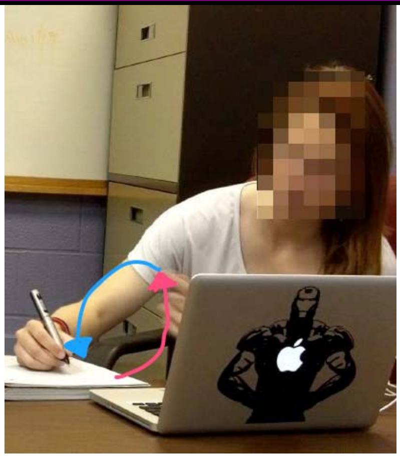
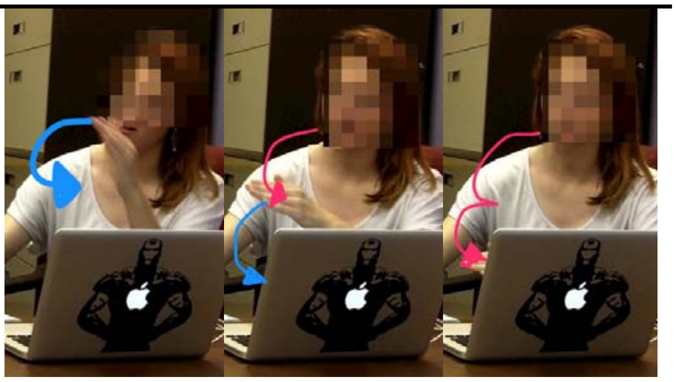
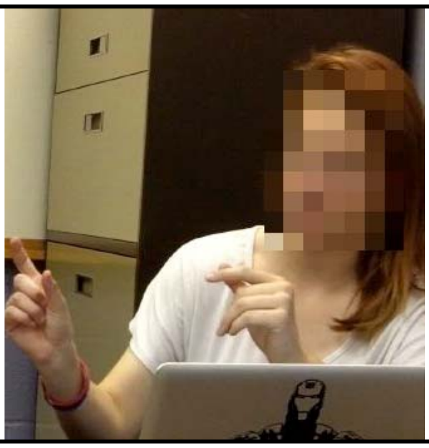
Table 9 – Rebecca starts talking out pseudo-code and asks for the pen to write it

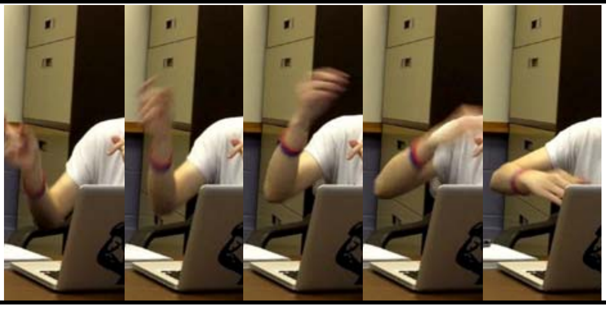
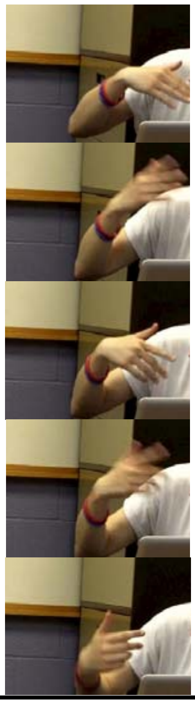
What she said	What she did

Uh, well, what I was think—like if I wanted to like, ‖save it‖ |{pinches right hand, thumb and index finger a few inches apart}| or whatever, /Yeah/	
I could make a while loop, ‖scan in the‖ |{drags hand across the table}| —scan in the data from the album, /OK/ {interviewer begins writing} uh, which is, [[I can write it if you want	

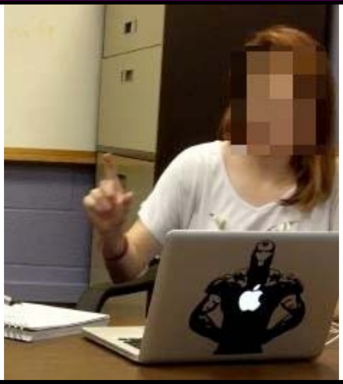
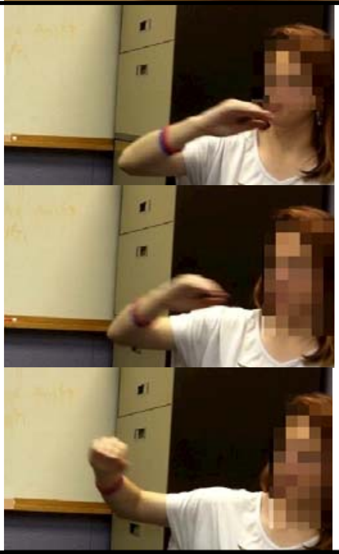
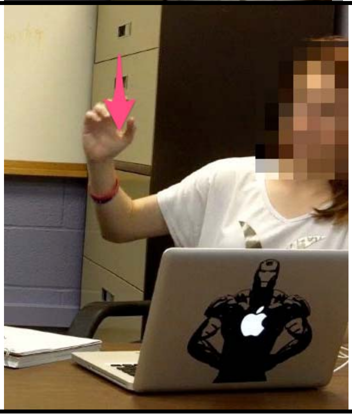
Rebecca then took the pen and started describing her thinking while she wrote.

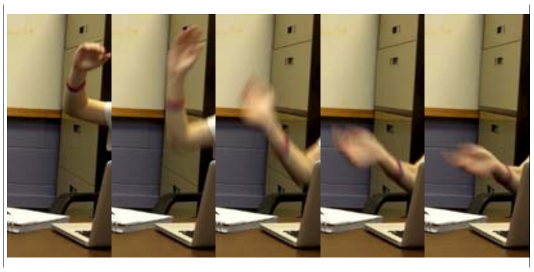
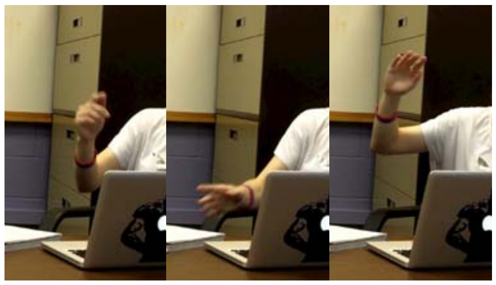
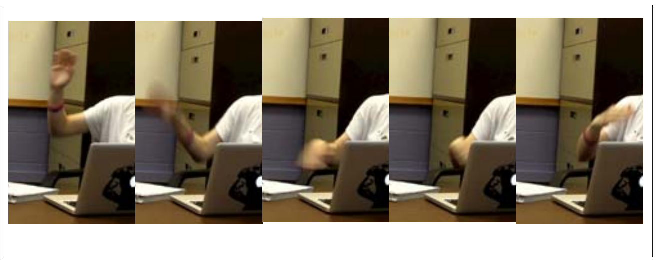
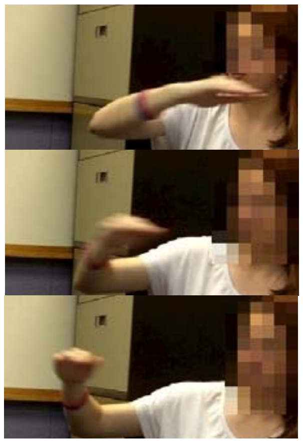
Table 10 – Rebecca's verbal/gestural overview of looping through the input track titles (Interview 4 of 5, April 6, 2012)

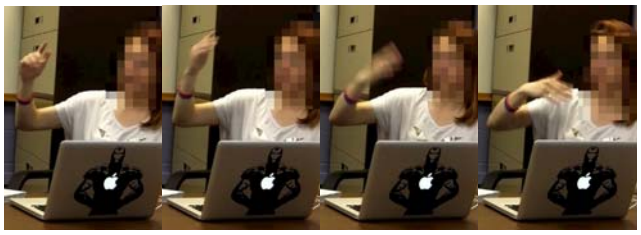
Panel #	What she said	What she did
1	And then my thinking at least, is you should be able to, um, say that "star p of i" /mmhmm/ equals, uh, the title, and then you just do i++, so then it'll ||move to the next one|| |{makes looping gesture with left hand}| /OK/	
2	and you just keep ||saving each of the pointers|| |{left hand makes horizontal chops in the air, like rungs down a ladder}|	
3	in the array ||to a title|| |{right hand makes pinching motion, left-hand points to it}|	

4	||And|| |{right hand makes a large loop counterclockwise (from Rebecca's perspective)}|	
5	||you just increment by 1|| |{right hand makes a cycloid to her right, in a plane parallel to the back wall, like counting dots on a line}| until you reach the end of file.	

"Like, that would make sense to me," Rebecca said. I asked whether the while loop would get a fresh line of text when it runs again.

Panel #	What she said	What she did
1	Yeah, because]] uh, ||what the while loop does|| |{pinches right thumb and forefinger}|	
2	||is it reads in the line|| |{right hand, palm down, chest height, slides out to the right}|	
3	and then ||once it reaches|| |{right hand rises, lowers in a chop}|	

4	the end of ||character it'll|| |{right hand rises, swoops down in a crescent, index finger extended, looping back up}|	
5	go back down—it'll ||do the while loop|| |{right hand moves back down toward table}| ||and then|| |{right hand moves back up above shoulder-height, palm-down}|	
6	||go back|| |{right hand creates a loop down and back up}| down to the next thing—line,	
7	it'll ||read in the line|| |{right hand scans rightward at shoulder height}|	

8	||until it|| |{pinches right thumb and forefinger, swipes right hand back to the left, palm down}| reaches the end of file	

In panel 4, notice that Rebecca's gesture of swooping back actually *precedes* her saying "go back down." In other words, her body articulates the idea of cycling back before she actually speaks it. A similar argument could be made for panel 8, where Rebecca's gesture for a process hitting the bottom (hand swooping and hitting an inflection point) precedes her saying "reaches the end of file"

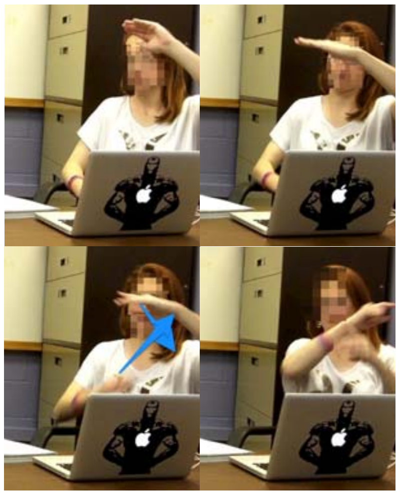
I tried to sum up my understanding of Rebecca's description:

Interviewer: So as you step through this loop /mmhmm/ {points to i++} i keeps going up by one==

Rebecca: ==Yes. (Interview 4 of 5, April 6, 2012)

As I thought about that, Rebecca continued explaining and gesturing:

Table 11 – Rebecca concludes her visual and gestural explanation for scanning in track titles (Interview 4 of 5, April 6, 2012)

Panel #	What she said	What she did
1	And the ||lines keep going down|| |{left hand horizontally chops the air, creating "ladder rungs"}|	(Gesture partially obscured by computer ☹)
2	so, ||that way,|| |{left hand raises up to her head, palm down and parallel to the table, and it is replaced by right hand, index finger extended and pointing to her left}|	

3	||the first line|| |{right hand scans across to her right}|	
4	is going to ||be|| |{right hand rotates to become pinched thumb and forefinger, palm out}| , ||uh the p—|| |{right-hand wiggles}|	
5	element zero /OK/ uh ||the second|| |{hand swoops slightly up, then curves down and locks in in a position below the prior one}| one'll be element one	

3.6 Conclusion

In pulling together concluding ideas, I revisit the points I established at the end of section 3.1.

3.6.1 Students' early-stage design activity reveals patterns outside the explanatory scope of (mis)conceptions accounts

Lionel's strategy of working at a whiteboard, keeping himself at a top-level of planning, and copying pseudo-code into a computer isn't explained by appealing to "concepts" in computer science, which in many research accounts are simply mappings of content in computer science curricula. I demonstrate this gap in Appendix 4, where the "conceptual features" of Lionel's code reveal little (if anything) about the resources involved in Lionel's design process.

Rebecca, in a similar fashion, has lots of productive knowledge for thinking about arrays (section 3.5.2), but at the time of the interview none of her code reflected that knowledge. Moreover, Rebecca was able to sense-make (Danielak et al., in press) about an array-of-pointers design, where her sense-making was again a complex activity that would be poorly accounted for in CSEd frameworks that appeal to misconceptions. In Rebecca's case, the situation is particularly paradoxical if we consider conceptual knowledge about a topic to naively be something students have or don't have.

Suppose Rebecca had the requisite conceptual knowledge for thinking about an array of pointers. It's a puzzle, then, to explain solely via conceptual knowledge why she was so frustrated with programming and felt stuck when she began Interview 4. Why hadn't the knowledge she possessed—relevant to solving the problem at hand—manifested already some way? Why was she stuck?

The alternative is to assume that as of Interview 4 she didn't have the conceptual knowledge for thinking about the problem at hand. But, that doesn't make sense either. If Rebecca didn't know how to think about an array of pointers, why was she able to do it so well during the interview? Both possibilities—that either Rebecca had or did not have conceptual knowledge about an array of pointers—lead to fairly non-sensical conclusions under naïve have/don't have assumptions of conceptual knowledge. Yet, again, the majority of conceptual knowledge research in CSEd today is silent on the issue of degree when it comes to conceptual knowledge: students either have a mental model that matches canonical function or they don't.

3.6.2 Rebecca had epistemological resources to support expert-like practices, but she framed those practices differently

Consider these summative statements made by Lionel and Rebecca on how they use pseudo-code in their work.

Table 12 – Comparing Lionel's and Rebecca's views toward pseudo-code

Lionel	Rebecca
"on my computer I'd, you know I'd write out the pseudocode, I'd ac—I'd literally write out the pseudocode, even though obviously it wouldn't compile and actually work." (Interview 1 of 1, October 17, 2011)	"So, I can't just type in "if the white piece reaches in," I—the, the C language, I guess, putting it in those, their terms, their terminology into programming language." (Interview 2 of 5, February 17, 2012)

As illustrative examples, they capture a fundamental difference in practice between Lionel and Rebecca. Lionel worked in pseudo-code on his computer, even going so far as to copy it into his source-code files despite the fact that it "obviously wouldn't compile and actually work." Rebecca also wrote pseudo-code, and she even wrote it into her source code files. But, her local sense of the activity of writing pseudo-code was different. It came to represent a discontinuity with the final code she was trying to write because there was nonetheless a gap between writing in English and writing C in “their terms.”

These diverging stances take the same practice—in this case, pseudo-coding—and situate it within a different kind of epistemological coherence. For Lionel, the coherence associated with pseudo-coding is productive and optimistic. Copying non-working pseudo-code into the computer is, in a colloquial sense, all part of the plan. It’s a legitimate step toward creating a final working program and, crucially, it reflects his own understanding of what’s supposed to happen in his code. For Rebecca, by contrast, writing pseudo-code is a fallback. Pseudo-code is what she writes “if I know what I want to put underneath of it,” by which she means if she’s not sure how to flesh out a loop or other control structure. Rather than being part of a planful coherence, the practice of pseudo-coding for Rebecca gets triggered as part of a stopgap coherence; a contingent measure at times associated with a kind of diffidence.

Crucially, Rebecca has intellectual resources for pseudo-coding. Moreover, her practice of writing pseudo-code to flesh out a loop is the plausible beginning of “method stubbing,” the process by which one might defer implementing the guts of a function and instead create a simple stopgap. For example, the function below will always print 28 Fahrenheit, which is fairly useless as far as temperature-getting functions go:

```
getOutsideTemperature <- function(sensor) {
      print(“28F”)
}
```

But, the advantage of having *something* in the function body, no matter how trivial, is enormous. Now, because **`getOutsideTemperature`** is defined and takes arguments, other functions can safely call it. It also offers a diagnostic output—by printing “**`28F`**” out—that we can rely on if we ever need to make sure the function was called. At the time of creating **`getOutsideTemperature`**, a programmer may have no idea how the temperature will be obtained but will nonetheless need to be able to write code that depends on it getting the temperature. Stubbing is, in such cases, a highly productive thing for a programmer to be able to do. But, stubbing is as much a strict practice as it is an epistemological move, because it represents a decision that an entity-to-be-known can be, for the moment, underspecified so it can be incorporated into a system before its behavior is defined.

Rebecca is already capable of simple stubbing. It’s part of an epistemological coherence that’s triggered when she doesn’t know how to flesh out a control structure. So, from a constructivist standpoint (Smith et al., 1993), she has knowledge of something that can be further refined into software engineering knowledge. Indeed, much of the surrounding data from Rebecca suggests that she would have been served well if an instructional intervention had taught her some simple ways to stub out functions so she could call them even if she hadn’t defined their bodies yet. To my knowledge, she was never exposed to stubbing formally, which is unfortunate considering how much it might have led to improving her confidence in programming.

Lastly, consider Rebecca's episode in reasoning about an array of pointers (section 3.5.2). As I show, Rebecca clearly had resources for reasoning about the sensibility of an array-of-pointers design and for predicting array-of-pointers syntax by extrapolating from patterns she had seen before. And, when I explicitly suggested that we suppose a candidate syntax works, Rebecca was able to fluidly articulate the procedure she would build around that syntax (section 3.5.2.3). Rebecca knew things, but her knowledge was deployed in such a highly contextual way as to be sensitive to what kind of knowledge-activity she thought she was supposed to be doing. It's thus sensible to model that set of phenomena from an epistemological standpoint, rather than a strictly conceptual one. Rebecca had certain kinds of knowledge, but part of my in-interview intervention was to establish a frame where *supposing syntax worked* was a valid part of programming. Once we mutually negotiated that frame there was a strong microcoherence of Rebecca stably explaining her code. But, crucially, my intervention wasn't about introducing conceptual content but rather establishing a kind of knowledge game—supposing—that one could play as part of programming design.

3.6.3 Students displayed a diversity of approaches to programming in the moment

The diversity of programming approaches I saw students take is far greater than just the space spanned by Lionel and Rebecca. Within and across my data, I continually observed students saying and doing things that not only distinguished them from each other, but constituted phenomena I haven't seen described in prior CSEd research. I saw students employ clever tests for debugging, create their own makeshift debugger, and discuss at length whether certain programming constructs represented natural ways of thinking. For illustrative purposes, I'll discuss three such examples. While I don't analyze them at length, I include them here as proof of principle of the kinds of phenomena we're not currently capturing that nonetheless have a strong effect on students' programming approaches. Moreover, these "in the wild" (Hutchins, 1995a) practices are further evidence of the kinds of knowledge that could be further refined into student expertise (Smith et al., 1993). So, not only do our research accounts miss these kinds phenomena, by missing them they preclude research and practice from building off them.

3.6.3.1 Isaac used a thoughtful debugging strategy that code snapshots alone could never capture

When Isaac was debugging an error in his checkers game code, he made use of a clever test to discover he had reversed array index subscripts. He had become uncertain about which of the two subscripts corresponded to what he thought of as the x-coordinate on the board and which controlled the y-coordinate. So, he took a piece whose x-y position he knew and instructed his program to print it as a "7" on the board. Then, he told it to increment the first index of that known piece by 1, leave the second index alone and print the resulting piece on the board as "8". Finally, he told his program to increment the second index of the known piece by 1, leave the first piece alone, and print the resulting piece on the board as "9". By inspecting the visual output, Isaac could tell which subscript controlled which position because he knew where 8 and 9 were relative to the unchanged 7. When he reflected on his work in the interview, he explained that what was tricky was realizing that the coordinate system for the checkers board didn't work the way standard coordinate systems in math worked; in the checkers program the indices were reversed.

Because Isaac had been set up as a code-snapshot participant, we have a snapshot record of his work for that portion of the interview. But, the only snapshot-visible changes are the reversal of the indices and the accompanying test cases. The narrative surrounding Isaac's particular debugging text, including his reflection that coordinate systems in programming are "trickier" than those in math, is entirely invisible to the snapshot record. If I hadn't interviewed him I never would have captured it.

3.6.3.2 Dana created her own debugging environment

Dana was working on her checkers program and needed to debug some errant behavior. In class and in discussion students had been exposed to the GNU Debugger (GDB), so in the interview I asked Dana if she'd like to use it. She said she didn't feel comfortable using GDB. Instead, using my MacBook, Dana decided to create three separate instances of a terminal. In one instance, she had her checkers program source code open. In the second, she had the source code for a set of test inputs she was developing to try to pinpoint the error. The third window was dedicated to compiling and running the code. Since the program was designed to print an ASCII (text) representation of the board at each turn, the dedicated program window was Dana's visual output for which pieces were where at any state in the game.

The important finding from this episode is how Dana deliberately structured her environment to support her activity. At the lowest level, she could have tried to do everything in a single terminal window. But, if she did that she'd have to constantly switch contexts. Because students in the course used a terminal version of emacs, she had no easy way to simultaneously view her code while compiling it. GDB would have been in many respects the right tool for that problem: it would let her set breakpoints, step through iterations of code, and inspect variable values as she did so. But Dana chose not to use GDB.

Instead, Dana exploited the fact that Terminal.app on Mac OS X 10.7 can have multiple instances of itself open at the same time on the same screen. With three terminals running, she could persistently have her source code visible and executing at the same time. With one glance pattern she could move from the board's visual output to the test to the program source and back again. Collectively, the field of displays enabled a kind of programming that was substantially different—in terms of its representational affordances—from using GDB or a single terminal instance. Moreover, a snapshot-history-only research approach would have never had access to how or why Dana (re)configured her development environment.

3.6.3.3 Toby said recursion was the "hardest programming way to think"

When I gave Toby a code sample task in an interview, he immediately commented on the fact that it used recursion. Recursion was, as he said, "the hardest programming way to think." At one point his exact phrasing, quoting the film *Ice Age*, was that "recursion is bad juju." The task in question was adapted from an example in Abelson and Sussman (1996) that approximates the square root of a number using an elegant recursive approach.

Toby’s first explanation for why recursion was bad was that it’s a completely unnatural way to think. He cited examples from cooking by saying, in effect, “Nobody ever stirs by saying stir once, and if it’s not stirred, stir once. They say stir 100 times.” He cited simple operations—including incrementing a number—where using recursion to push and pop a stack seemed needlessly complex. One of his most damning indictments of recursion was that his class introduced it as a solution to generate numbers in the Fibbonacci sequence. Of course recursion works for examples that are mathematically recursive, he reasoned, but outside of those obviously contrived examples it was bad joo joo when compared to understandable, sensible iteration.

The fuller account of Toby’s resistance to recursion—and slight change of heart after an instructional moment—is beyond the scope of what I can present here. Suffice it to say that Toby’s feelings toward recursion—rooted in a sense of it being an unnatural way to think—strongly directed his approach to programming. The approach was so strong, in fact, that during our first interview he wasn’t fully able to produce working code examples of why recursion was absurd despite his vehement conviction that it was. In other words, he maintained that recursion was bad and inefficient even though every time he tried to demonstrate its inferiority he made mistakes in his code and his examples didn’t work.

3.6.4 Dynamic epistemological models can offer a lens for reforming assessment and instruction.

Historically, a recognition of in-pieces cognitive dynamics in science education (diSessa, 1993; Hammer, 1994) has led to more direct research on how those frameworks inform instruction (Hammer & Elby, 2003; Hammer, 1996; Louca et al., 2004). What I describe here is research still at the formative end of generating models to explain cognition. And, because the focus of the research was explicitly on learning and not instruction, I advise prudence in trying to draw certain kinds of specific recommendations from the work I present.

Those restrictions being said, we can think carefully about what this work means for instruction. For practical purposes, let’s refine terminology so we can talk more specifically and precisely about components of teaching. Below are my working definitions for teaching components:

- How should instructors change the way they act in the classroom (*instruction*)?
- How should this research inform the set of intended learning outcomes instructors prepare (*curriculum*)?
- How should we change the way we measure or otherwise go about ascertaining what students know (*assessment*)?

The clearest implications of my work are in instruction and assessment.

From an instruction perspective, it may help teachers to know that students can enact different epistemological stances as part of their programming practice. As stated, that finding is in principle not new (cf., Gaspar & Langevin, 2007). What is new, I think, is the characterization that these stances evidence epistemological resources that can be tapped for learning (cf., Hammer & Elby, 2003). That sense-making about program design choices or computational concepts is a worthwhile activity, for example, is a message instructors can send to students that resonates with larger findings in engineering education research (Danielak et al., in press). Moreover, instructors can be more sensitive to what kinds of knowledge-activity students think they're doing, particularly when making epistemological moves like *supposing* might help students work through design problems.

My findings also speak to assessment. In the course I studied, there were few (if any) assessment instruments that would have revealed the kinds of knowledge I document Lionel and Rebecca having in this study. At no point in my observational data of the course were students asked to explain a design choice or reason about competing solutions to handle a problem. At no point were they encouraged to deeply sense-make about the conceptual content of the course, particularly the traditionally tricky topics of pointers and pointer arithmetic. At no point in my data did the instructor model the kind of sense-making Rebecca did in her interviews, a fact which she noted at one point as distinguishing the course from Basic Programming, its first-semester counterpart. Because the course never explicitly asked students to sense-make, reason about design choices, or explain designs, it never used information about what knowledge students had about those things to inform its instruction. It couldn't. Moreover, because the course never explicitly asked students to sense-make, reason about design choices, or explain designs, I was left to conclude such things were not prioritized learning outcomes of the course. To be clear, all courses establish a kind of focus by deciding what won't be covered, and I don't fault the instructor at all for running a course where design knowledge wasn't an enacted learning outcome. But, I'm also left to wonder: why wasn't it? Given that even beginning students can think about software design, and given that engineering as a discipline fundamentally involves design, shouldn't it have been?

4 Conclusion

If there is one overarching finding from this dissertation, it's that students clearly have resources for thinking about designing programs and a diversity of approaches to programming in the moment. Below, I explain the kind of diversity I saw in programming approaches. Then, I conclude with a discussion of what my research might mean for assessment.

4.1 We should think carefully about what students' programming design knowledge means for assessment

My research helps us document and model the kinds of knowledge students have. In so doing, it points out kinds of information that the course's assessments were apt to miss:

1. How students frame or otherwise approach the task of programming in the moment
2. The role of different kinds of prior experience in stabilizing (or potentially destabilizing) certain kinds of frames
3. In-the-moment practices—including talking out solutions, sketching out debugging strategies, and writing out pseudo-code—that display students' competence and sense-making
4. The code history that traces how designs evolve, including how students start projects and what parts of their designs become dead-ends

Together, those four points are relevant for formative assessment (Black & Wiliam, 1998) in introductory programming. (1) and (2) point us toward what students think they're doing when they're programming. Analogous work from science education tells us that students' sense of the kind of knowledge activity they're enacting matters for learning and assessment (Russ et al., 2008). For example, knowing early on that students are blindly copying code or randomly trying syntax gives instructors a chance to intervene. But, my research suggests intervention can't just be about stopping a bad behavior: knowing that copy-paste behavior is happening is different from knowing *why* it's happening.

Formative assessment has to be about diagnosing causes, not just identifying symptoms. Otherwise, we run the risk of ignoring or even harming students' productive knowledge. Take the example of copy-paste behavior, as documented by Gaspar and Langevin (Gaspar & Langevin, 2007). One reason we may see copy-paste behavior, as Study 1 demonstrates, is that students might see situations as new instances of already-solved problems. In professional practice, seeing old solved problems in new situations can be productive. Such insights can, for example, direct engineers to use a pre-built library of functions instead of building their own. But, another related reason for copy-paste may be efficiency. For Rebecca, copying and pasting code was very fast; it required only a few quick keystrokes. In the short run, Rebecca's choice to copy the code was a faster, less demanding, more trustworthy route to go on than was abstracting the code to a function. When we consider students trying to see common problems in new scenarios and solve such problems efficiently, "copy-paste" becomes a symptom rather than a root cause. Epistemological frameworks, then, can inform assessments by offering explanations that aim at the root cause of certain behaviors.

Points 3 and 4 complement knowledge analysis by drawing focus to artifacts, practices, and history. As data from Lionel shows (Study 2), a crucial part of his design process involves artifacts and activities that were only distally knowable to the assessments he got in class. Lionel's instructor had no direct access to:

- Lionel's whiteboard
- the hours Lionel may have spent working out designs in chalk
- how Lionel might have talked out design features
- how Lionel's initial "pseudo-code" evolved into his final design.

In Rebecca's case, commented-out code in her final project submission were perhaps the only clues at all that she tried abstracting some flight day-checking procedures into functions (Study 1). Without Rebecca's history, the instructor would have had no reliable way of knowing:

- Rebecca began her flights database project by copying code from an earlier project
- Rebecca's original solution for day-checking evolved from a seven-fold conditional structure, one for each day
- Rebecca may have tried abstracting day-checking procedures, even creating a function called check_days.

To sum up: traditional assessments can tell us what students finally produce but not how they produce it, when they produce it, why they produce it, or what the production process was. In the summative assessment structure of the course, final student products were the only products submitted to the instructor. My research shows that what's happening in the interstices—before typing, between compiles, away from the computer—can be captured and, in principle, analyzed and acted upon by instructors.

4.2 What We Might Change About Classroom Practice

What follows is my speculation about how we might specifically change classroom practices in light of my research findings. I move from recommendations I think are most strongly supported by my data to recommendations that align with my findings but are more expansive, and thus less strongly supported. As my recommendations broaden, I try to offer not only an instructional recommendation but a concomitant question for research.

4.2.1 Instructors could look beyond content to understand student difficulties

I think first and foremost, instructors have to have a willingness to recognize that the difficulties they think they're seeing in students are difficulties they may be viewing through a lens of content. For example, after seeing a student struggle an instructor might say "the student doesn't know assignment statements." But, those difficulties almost certainly have a deeper explanation. I say "almost certainly," where what I mean is that there are a number of patterns we've identified as common novice errors, but most research stops before asking why that's a common novice error. With rare exceptions (Fleury, 1991, 2000), the computing education community doesn't encourage asking the question *why might a student be doing this?* and *what might this tell me about the way a student is making sense of this?* By contrast, my research suggests those orientations are ripe, low-hanging fruit for instructors to take on. I think one of the first things as instructor could ask is, *why would a student be thinking that this is the appropriate thing to be doing?* And that holds true whether the thing in question is writing the statement this way, or interpreting the code this way, or enacting this kind of programming activity this way.

Moreover, we know from research on metacognition (Schoenfeld, 1987, 1992) that thinking beyond content opens the palette of kinds of interventions an instructor can make. Schoenfeld (1987), for example, came to encourage metacognition in his classroom by doggedly asking student groups what they were doing, why they were doing it, and how they hoped it would lead them to a solution. Students ultimately came to internalize such strategies and spent far less time floundering along solution paths that weren't ultimately productive. Might the same idea be true in introductory programming courses? To find out, we could begin researching in earnest how often and in what ways instructors model metacognition for their students. Currently, I would argue, we don't know whether and how instructors do so.

4.2.2 Instructors could use code history to inform interventions

When, for example, a student comes to office hours with a problem on a project, an instructor needs to quickly come to grips with the state of the students' project, the logic of their design, where they're stuck, why they're stuck, and what might best help them. That's no small task. But, given my analysis I have strong reason to believe that having code history available to instructor could change both the nature of coming to terms with a student's project and the conversation with a student that results. If an instructor can see the evolution of a student's code, at the very least they could see where the student started, how the code was growing, and where the student was working most recently. Even if the instructor had never before seen the code (or its history) until that moment in office hours, having both available changes the kind of view the instructor can get of the code and the specifics of the intervention that might result. In Rebecca's case, for example, it might have been an opportunity to explore why her check_days abstracted function was failing on her flights database project.

Having code-snapshot capabilities also changes the kind of research we can do. First and foremost, a result of this dissertation was to create a freely-available, lightweight, open-source framework for capturing and visualizing students' code histories.[68] So, the most basic kind of study would involve deploying that framework from the instructional side of a course and exploring what happens, as, for example, Hurd (2013) has done. One could ask questions of how instructors build code snapshotting into their course, how such information could or did inform assessment, and how such information could or did change the nature of instructional interactions with students. Was it for the better? If so, how do we know? If not, how do we know?

[68] https://github.com/briandk/gitvisualizations

4.2.3 Instructors could establish a norm of asking why a design choice makes sense

The most far-reaching implication of the research I present here is that instructors should establish a norm in their classes where anyone, at any point, for any piece of code, is allowed to ask *why does this make sense as a design choice? Why is that an obvious choice to make? How does that work?* Consider a parallel example from mathematics education. In Don Saari's calculus class at UC Irvine, he "invokes the principle of what he calls "WGAD"—"Who gives a damn?" (Bain, 2004, pp. 38–39). Bain explains:

At the beginning of his courses, he tells his students that they are free to ask him the question on any day during the course, at any moment in class. He will stop and explain to his students why the material under consideration at that moment—however abstruse and minuscule a piece of the big picture it may be—is important, and how it relates to the larger questions and issues of the course. (Bain, 2004, p. 39)

What I'm suggesting is even broader than that, because it's not just a question students can ask of professors; it's a question *anyone* can ask of *anyone*. I see it as the programming and design extension of a sociomathematical norm (Yackel & Cobb, 1996), and I think it could lead to collaborative sense-making. That is, I'm asking for an accepted cultural practice that is also itself a design practice, whereby an instructor can help create a safe, stable space for a community to be reflective and critical about design choices. I also think it's an idea that leads to others, such as letting, if not requiring, students review one another's code. When students are forced to reckon with someone else's code and understand their design decisions, they're also forced to justify their own decisions. And establishing "why that design choice?" or "how does that make sense?" as a norm provides reciprocal opportunities for students, not just instructors, to improve the code of others.

The research questions that come out of such an idea would include:

- How can an instructor satisfactorily establish norms about design in a classroom?
- What does it look like when students collaboratively sense-make about a program's design choices?
- What does it look like when students are asked to reflect on their own design choices?
- How might introducing a code review component into a course change the way students approach design? How might it improve students' conceptual understanding? How might it improve the quality of the code students produce?

4.3 Final Remarks

How students design programs matters for learning and instruction in engineering. It matters because finished code reflects what students know about design, whether or not instructors capture such information. It matters because students have resources for learning about and engaging in design; whether or not curriculum, instruction, and assessment choose to tap into those resources. It matters because design should be an intellectual thread that runs through all engineering courses. That thread shouldn't stop when we introduce students to programming.

5 References

Abelson, H., & Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). Cambridge, Mass: MIT Press.

Adelson, B., & Soloway, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering, SE-11*(11), 1351 – 1360. doi:10.1109/TSE.1985.231883

Archer, L. B. (1979). Whatever became of design methodology. *Design Studies*, *1*(1), 17–18.

Bain, K. (2004). *What the best college teachers do*. Cambridge, Mass: Harvard University Press.

Baker, A., & van der Hoek, A. (2010). Ideas, subjects, and cycles as lenses for understanding the software design process. *Design Studies*, *31*(6), 590–613. doi:10.1016/j.destud.2010.09.008

Ball, L. J., Onarheim, B., & Christensen, B. T. (2010). Design requirements, epistemic uncertainty and solution development strategies in software design. *Design Studies, 31*(6), 567–589. doi:10.1016/j.destud.2010.09.003

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, *26*(9), 677–679. doi:http://doi.acm.org/10.1145/358172.358408

Black, P., & Wiliam, D. (1998). Inside the Black Box: Raising Standards Through Classroom Assessment. *Phi Delta Kappan, 80*(2), 139–44.

Boaler, J. (1998). Open and closed mathematics: Student experiences and understandings. *Journal for Research in Mathematics Education*, *29*(1), 41–62.

Boaler, J. (2000). Mathematics from another world: Traditional communities and the alienation of learners. *The Journal of Mathematical Behavior*, *18*(4), 379–397. doi:10.1016/S0732-3123(00)00026-2

Boaler, J. (2002). The development of disciplinary relationships: Knowledge, practice and identity in mathematics classrooms. *For the Learning of Mathematics, 22*(1), 42–47.

Boaler, J., & Greeno, J. G. (2000). Identity, agency, and knowing in mathematical worlds. In J. Boaler (Ed.), *Multiple perspectives on mathematics teaching and learning* (pp. 171–200). Westport, CT: Ablex Pub.

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 10–13. doi:10.1145/567067.567069

Bonar, J., & Soloway, E. (1985). Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, *1*(2), 133.

Bucciarelli, L. L. (1994). *Designing engineers*. Cambridge, Mass: MIT Press.

Clancy, M. (2004). Misconceptions and Attitudes that Interfere with Learning to Program. In S. Fincher & M. Petre (Eds.), *Computer Science Education Research* (pp. 85–100). London, UK: RoutledgeFalmer.

Danielak, B. A., Gupta, A., & Elby, A. (in press). The Marginalized Identities of Sense-Makers: Reframing Engineering Student Retention. *Journal of Engineering Education*.

Danielsiek, H., Paul, W., & Vahrenhold, J. (2012). Detecting and Understanding Students' Misconceptions Related to Algorithms and Data Structures. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 21–26). New York, NY, USA: ACM. doi:10.1145/2157136.2157148

diSessa, A. A. (1986). Models of Computation. In D. A. Norman & S. W. Draper (Eds.), *User centered system design: new perspectives on human-computer interaction* (pp. 201–218). Hillsdale, N.J: L. Erlbaum Associates.

diSessa, A. A. (1993). Toward an Epistemology of Physics. *Cognition and Instruction*, *10*(2/3), 105–225.

diSessa, A. A. (2002). Why "Conceptual Ecology" is a good idea. In M. Limón & L. Mason (Eds.), *Reconsidering conceptual change: issues in theory and practice* (pp. 29–60). Dordrecht ; Boston: Kluwer Academic Publishers.

diSessa, A. A., & Sherin, B. L. (1998). What changes in conceptual change? *International Journal of Science Education*, *20*(10), 1155–1191. doi:10.1080/0950069980201002

Duckworth, E. R. (2006). *"The having of wonderful ideas" and other essays on teaching and learning* (3rd ed.). New York: Teachers College Press.

Elby, A., & Hammer, D. (2010). Epistemological resources and framing: A cognitive framework for helping teachers interpret and respond to their students' epistemologies. In L. D. Bendixen & F. C. Feucht (Eds.), *Personal epistemology in the classroom: theory, research, and implications for practice* (pp. 409–434). Cambridge, UK ; New York: Cambridge University Press.

Elliott Tew, A. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner* (Ph.D.). Georgia Institute of Technology, Ann Arbor. Retrieved from ProQuest Dissertations & Theses Full Text. (873212789)

Elliott Tew, A., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 97–101. doi:10.1145/1734263.1734297

Elliott Tew, A., & Guzdial, M. (2011). The FCS1: a language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 111–116). New York, NY, USA: ACM. doi:10.1145/1953163.1953200

Erickson, F. (1986). Qualitative methods in research on teaching. In M. C. Wittrock (Ed.), *Handbook of research on teaching* (3rd ed., pp. 119–161). New York : London: Macmillan ; Collier Macmillan.

Eynde, P., & Hannula, M. (2006). The Case Study of Frank. *Educational Studies in Mathematics*, *63*(2), 123–129. doi:10.1007/s10649-006-9030-8

Fleury, A. E. (1991). Parameter passing: the rules the students construct. In *Proceedings of the twenty-second SIGCSE technical symposium on Computer* *science education* (pp. 283–286). New York, NY, USA: ACM. doi:10.1145/107004.107066

Fleury, A. E. (1993). Student Beliefs about Pascal Programming. *Journal of Educational Computing Research*, *9*(3), 355–371. doi:10.2190/VECR-P8T6-GB10-MXJ5

Fleury, A. E. (2000). Programming in Java: student-constructed rules. *SIGCSE Bull.*, *32*(1), 197–201. doi:10.1145/331795.331854

Gainsburg, J. (2006). The mathematical modeling of structural engineers. *Mathematical Thinking & Learning*, *8*(1), 3–36. doi:10.1207/s15327833mtl0801_2

Gal-Ezer, J., & Zur, E. (2004). The efficiency of algorithms--misconceptions. *Computers & Education*, *42*(3), 215–226.

Gaspar, A., & Langevin, S. (2007). Restoring “coding with intention” in introductory programming courses. *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education*, 91–98. doi:10.1145/1324302.1324323

Ginsburg, H. P. (1997). *Entering the child’s mind: the clinical interview in psychological research and practice*. Cambridge ; New York: Cambridge University Press.

Ginsburg, H. P., & Opper, S. (1988). *Piaget’s Theory of Intellectual Development* (3rd ed.). Englewood Cliffs, N.J: Prentice-Hall. Retrieved from http://lccn.loc.gov/87017353

Goffman, E. (1974). *Frame Analysis: An Essay on the Organization of Experience*. New York: Harper & Row.

Goldin-Meadow, S. (2003). *Hearing gesture: how our hands help us think*. Cambridge, Mass: Belknap Press of Harvard University Press.

Goodwin, C. (2000). Action and embodiment within situated human interaction. *Journal of Pragmatics*, *32*(10), 1489–1522. doi:10.1016/S0378-2166(99)00096-X

Gupta, A., Hammer, D., & Redish, E. F. (2010). The Case for Dynamic Models of Learners’ Ontologies in Physics. *Journal of the Learning Sciences*, *19*(3), 285. doi:10.1080/10508406.2010.491751

Hall, R. (1999). Following mathematical practices in design-oriented work. In C. Hoyles, C. Morgan, & G. Woodhouse (Eds.), *Rethinking the Mathematics Curriculum* (pp. 29–47). London: Falmer Press.

Hall, R., & Nemirovsky, R. (2012). Introduction to the Special Issue: Modalities of Body Engagement in Mathematical Activity and Learning. *Journal of the Learning Sciences*, *21*(2), 207–215. doi:10.1080/10508406.2011.611447

Hall, R., & Stevens, R. (1995). Making space: A comparison of mathematical work in school and professional design practices. In S. L. Star (Ed.), *The cultures of computing* (pp. 118–145). Oxford, UK: Blackwell Publisher.

Hall, R., Stevens, R., & Torralba, T. (2002). Disrupting representational infrastructure in conversations across disciplines. *Mind, Culture & Activity*, *9*(3), 179–210.

Hall, R., Wright, K., & Wieckert, K. (2007). Interactive and Historical Processes of Distributing Statistical Concepts Through Work Organization. *Mind, Culture & Activity*, *14*(1/2), 103–127. doi:10.1080/10749030701307770

Hammer, D. (1989). Two approaches to learning physics. *The Physics Teacher*, *27*(9), 664–670. doi:10.1119/1.2342910

Hammer, D. (1994). Epistemological beliefs in introductory physics. *Cognition and Instruction*, *12*(2), 151–183. doi:10.2307/3233679

Hammer, D. (1996). Misconceptions or P-Prims: How May Alternative Perspectives of Cognitive Structure Influence Instructional Perceptions and Intentions? *Journal of the Learning Sciences*, *5*(2), 97–127. doi:10.1207/s15327809jls0502_1

Hammer, D., & Elby, A. (2002). On the form of a personal epistemology. In B. K. Hofer & P. R. Pintrich (Eds.), *Personal epistemology: The psychology of beliefs about knowledge and knowing* (pp. 169–190). Mahwah, N.J: L. Erlbaum Associates.

Hammer, D., & Elby, A. (2003). Tapping epistemological resources for learning physics. *The Journal of the Learning Sciences*, *12*(1), 53–90. doi:10.2307/1466634

Hammer, D., Elby, A., Scherr, R. E., & Redish, E. F. (2005). Resources, framing, and transfer. In J. P. Mestre (Ed.), *Transfer of learning from a modern multidisciplinary perspective*. Greenwich, CT: IAP.

Hannula, M., Evans, J., Philippou, G., & Zan, R. (2004). Affect in Mathematics Education–Exploring Theoretical Frameworks. Research Forum. *International Group for the Psychology of Mathematics Education*, 30.

Henderson, K. (1999). *On line and on paper: visual representations, visual culture, and computer graphics in design engineering*. Cambridge, Mass: MIT Press.

Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Proof by incomplete enumeration and other logical misconceptions. *Proceeding of the Fourth International Workshop on Computing Education Research*, 59–70. doi:10.1145/1404520.1404527

Hofer, B. K., & Pintrich, P. R. (1997). The development of epistemological theories: Beliefs about knowledge and knowing and their relation to learning. *Review of Educational Research*, *67*(1), 88–140. doi:10.3102/00346543067001088

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding object misconceptions. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (pp. 131–134). New York, NY, USA: ACM. doi:10.1145/268084.268132

Hurd, A. (2013, March). *Assessment in an Introduction to Programming Course*. Presented at the 16th Annual Course Technology Conference, San Diego, CA, USA. Retrieved from http://www.slideshare.net/CengageLearning/andrew-hurd-assessment-in-an-intro-to-programming-course

Hutchins, E. (1995a). *Cognition in the Wild*. Cambridge, Mass: MIT Press.

Hutchins, E. (1995b). How a cockpit remembers its speeds. *Cognitive Science*, *19*(3), 265–288. doi:10.1016/0364-0213(95)90020-9

Izsák, A. (2004). Students' Coordination of Knowledge When Learning to Model Physical Situations. *Cognition & Instruction, 22*(1), 81–128.

Jackson, M. (2010). Representing structure in a software system design. *Design Studies, 31*(6), 545–566. doi:10.1016/j.destud.2010.09.002

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2006 international workshop on Computing education research - ICER '06* (p. 73). Canterbury, United Kingdom. doi:10.1145/1151588.1151600

Jordan, B., & Henderson, A. (1995). Interaction analysis: Foundations and practice. *Journal of the Learning Sciences*, *4*(1), 39. doi:10.1207/s15327809jls0401_2

Joy, M., Sinclair, J., Sun, S., Sitthiworachart, J., & López-González, J. (2009). Categorising computer science education research. *Education and Information Technologies*, *14*(2), 105–126. doi:10.1007/s10639-008-9078-4

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 107–111. doi:10.1145/1734263.1734299

Kaiser, D. (2005). *Drawing theories apart: the dispersion of Feynman diagrams in postwar physics*. Chicago: University of Chicago Press.

Keller, E. F. (1983). *A feeling for the organism: the life and work of Barbara McClintock*. San Francisco: W.H. Freeman.

Kolikant, Y. B.-D., & Mussai, M. (2008). "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education*, *18*(2), 135.

Kölling, M., & Utting, I. (2012). Building an open, large-scale research data repository of initial programming student behaviour. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (pp. 323–324). New York, NY, USA: ACM. doi:10.1145/2157136.2157234

Latour, B. (1987). *Science in action: how to follow scientists and engineers through society*. Cambridge, Mass: Harvard University Press.

Latour, B. (1990). Drawing things together. In M. Lynch & S. Woolgar (Eds.), *Representation in Scientific Practice* (1st MIT Press ed., pp. 19–68). Cambridge, Mass: MIT Press.

Lehrer, R., Schauble, L., Carpenter, S., & Penner, D. (2000). The interrrelated development of inscriptions and conceptual understanding. In P. Cobb, E. Yackel, & K. McClain (Eds.), *Symbolizing and Communicating in Mathematics Classrooms: Perspectives on Discourse, Tools, and Instructional Design* (pp. 325–360). Mahwah, N.J: Lawrence Erlbaum Associates.

Lising, L., & Elby, A. (2005). The impact of epistemology on learning: A case study from introductory physics. *American Journal of Physics*, *73*(4), 372. doi:10.1119/1.1848115

Louca, L., Elby, A., Hammer, D., & Kagey, T. (2004). Epistemological Resources: Applying a New Epistemological Framework to Science Instruction. *Educational Psychologist*, *39*(1), 57–68.

Malmi, L., Sheard, J., Simon, Bednarik, R., Helminen, J., Korhonen, A., … Taherkhani, A. (2010). Characterizing research in computing education: a preliminary analysis of the literature. In *Proceedings of the Sixth international workshop on Computing education research* (pp. 3–12). New York, NY, USA: ACM. doi:10.1145/1839594.1839597

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Upper Saddle River, NJ: Prentice Hall.

Mayer, R. E. (1981). The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys (CSUR)*, *13*, 121–141. doi:10.1145/356835.356841

Minsky, M. L. (1986). *The Society of Mind*. New York: Simon and Schuster.

Nasir, N. S., & Cooks, J. (2009). Becoming a Hurdler: How Learning Settings Afford Identities. *Anthropology & Education Quarterly*, *40*(1), 41–61. doi:10.1111/j.1548-1492.2009.01027.x

Nasir, N. S., & Hand, V. (2008). From the court to the classroom: Opportunities for engagement, learning, and identity in basketball and classroom mathematics. *Journal of the Learning Sciences*, *17*(2), 143–179. doi:10.1080/10508400801986108

Nasir, N. S., & Hand, V. M. (2006). Exploring Sociocultural Perspectives on Race, Culture, and Learning. *Review of Educational Research*, *76*(4), 449–475.

Nemirovsky, R., Rasmussen, C., Sweeney, G., & Wawro, M. (2012). When the Classroom Floor Becomes the Complex Plane: Addition and Multiplication as Ways of Bodily Navigation. *Journal of the Learning Sciences*, *21*(2), 287–323. doi:10.1080/10508406.2011.611445

Ochs, E., Gonzales, P., & Jacoby, S. (1996). "When I come down I'm in the domain state": grammar and graphic representation in the interpretive activity of physicists. In *Interaction and Grammar* (pp. 328–369). Cambridge: Cambridge University Press.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books. Retrieved from http://lccn.loc.gov/79005200

Parsons, J., & Saunders, C. (2004). Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering*, *30*(12), 873 – 888. doi:10.1109/TSE.2004.94

Patitsas, E., Craig, M., & Easterbrook, S. (2013). On the Countably Many Misconceptions About #Hashtables (Abstract Only). In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 739–739). New York, NY, USA: ACM. doi:10.1145/2445196.2445443

Paul, W., & Vahrenhold, J. (2013). Hunting High and Low: Instruments to Detect Misconceptions Related to Algorithms and Data Structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (pp. 29–34). New York, NY, USA: ACM. doi:10.1145/2445196.2445212

Pea, R. D. (1986). Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36.

Pea, R. D., Soloway, E., & Spohrer, J. C. (1987). The Buggy Path to the Development of Programming Expertise. *Focus on Learning Problems in Mathematics*, *9*(1), 5–30.

Petre, M., van der Hoek, A., & Baker, A. (2010). Editorial. *Design Studies*, *31*(6), 533–544. doi:10.1016/j.destud.2010.09.001

Rodrigo, M. M. T., & Baker, R. S. J. d. (2009). Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the fifth* *international workshop on Computing education research workshop* (pp. 75–80). New York, NY, USA: ACM. doi:10.1145/1584322.1584332

Rodrigo, M. M. T., Tabanao, E., Lahoz, M. B. ., & Jadud, M. C. (2009). Analyzing Online Protocols to Characterize Novice Java Programmers. *Philippine Journal of Science*, *138*(2), 177–190.

Rooksby, J. (2010). "Just try to do it at the whiteboard": Researcher-participant interaction and issues of generalisation. In *Proceedings of the Studying Professional Software Design (SPSD) Conference*. San Diego, CA, USA.

Rooksby, J., & Ikeya, N. (2012). Collaboration in Formative Design: Working Together at a Whiteboard. *IEEE Software*, *29*(1), 56 –60. doi:10.1109/MS.2011.123

Rosenberg, S., Hammer, D., & Phelan, J. (2006). Multiple Epistemological Coherences in an Eighth-Grade Discussion of the Rock Cycle. *Journal of the Learning Sciences*, *15*(2), 261–292. doi:10.1207/s15327809jls1502_4

Russ, R. S., Coffey, J. E., Hammer, D., & Hutchison, P. (2008). Making classroom assessment more accountable to scientific reasoning: A case for attending to mechanistic thinking. *Science Education*, *93*(5), 875–891. doi:10.1002/sce.20320

Saussure, F. de. (1986). *Course in general linguistics*. LaSalle, Ill: Open Court.

Scherr, R. E., & Hammer, D. (2009). Student Behavior and Epistemological Framing: Examples from Collaborative Active-Learning Activities in Physics. *Cognition & Instruction*, *27*(2), 147–174. doi:10.1080/07370000902797379

Schoenfeld, A. H. (1987). What's all the fuss about metacognition? In *Cognitive Science and Mathematics Education* (pp. 189–215). Lawrence Erlbaum Associates.

Schoenfeld, A. H. (1988). When good teaching leads to bad results: The disasters of "well-taught" mathematics courses. *Educational Psychologist*, *23*(2), 145–166. doi:10.1207/s15326985ep2302_5

Schoenfeld, A. H. (1991). On mathematics as sense-making: An informal attack on the unfortunate divorce of formal and informal mathematics. In *Informal reasoning and education* (pp. 311–343).

Schoenfeld, A. H. (1992). Learning to Think Mathematically: Problem Solving, Metacognition, and Sense-Making in Mathematics. In D. Grouws (Ed.), *Handbook for research on mathematics teaching and learning* (pp. 334–370). New York: MacMillan.

Schwartz, D. L., Chase, C. C., & Bransford, J. D. (2012). Resisting Overzealous Transfer: Coordinating Previously Successful Routines With Needs for New Learning. *Educational Psychologist*, *47*(3), 204–214. doi:10.1080/00461520.2012.696317

Seppälä, O., Malmi, L., & Korhonen, A. (2006). Observations on student misconceptions—A case study of the Build – Heap Algorithm. *Computer Science Education*, *16*(3), 241–255. doi:10.1080/08993400600913523

Sherin, B. L. (2001). How students understand physics equations. *Cognition and Instruction*, *19*(4), 479–541.

Smith, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. *The Journal of the Learning Sciences*, *3*(2), 115–163.

Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, *29*, 850–858. doi:10.1145/6592.6594

Soloway, E., Bonar, J., & Ehrlich, K. (1983). Cognitive Strategies and Looping Constructs: An Empirical Study. *Commun. ACM*, *26*(11), 853–860. doi:10.1145/182.358436

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the Novice Programmer*. Hillsdale, N.J: L. Erlbaum Associates.

Spacco, J., Hovemeyer, D., Pugh, W., Hollingsworth, J., Padua-Perez, N., & Emad, F. (2006). Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses. In *ITiCSE '06: Proceedings of the 11th annual conference on Innovation and technology in computer science education*. ACM Press.

Spacco, J., Pugh, W., Ayewah, N., & Hovemeyer, D. (2006). The Marmoset project: an automated snapshot, submission, and testing system. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (pp. 669–670). New York, NY, USA: ACM. doi:10.1145/1176617.1176665

Spacco, J., Strecker, J., Hovemeyer, D., & Pugh, W. (2005). Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*. St. Louis, Missouri, USA.

Spohrer, J. C., & Soloway, E. (1986). Alternatives to construct-based programming misconceptions. *ACM SIGCHI Bulletin*, *17*, 183–191. doi:10.1145/22339.22369

Stevens, R., & Hall, R. (1998). Disciplined perception: Learning to see in technoscience. In *Talking Mathematics in School: Studies of Teaching and Learning* (pp. 107–150). Cambridge, U.K: Cambridge University Press.

Stieff, M. (2007). Mental rotation and diagrammatic reasoning in science. *Learning and Instruction*, *17*(2), 219–234. doi:10.1016/j.learninstruc.2007.01.012

Tabanao, E. S., Rodrigo, M. M. T., & Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research* (pp. 85–92). New York, NY, USA: ACM. doi:10.1145/2016911.2016930

Tannen, D. (Ed.). (1993). *Framing in Discourse*. New York: Oxford University Press.

Trakhtenbrot, M. (2013). Students Misconceptions in Analysis of Algorithmic and Computational Complexity of Problems. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (pp. 353–354). New York, NY, USA: ACM. doi:10.1145/2462476.2465604

Traweek, S. (1988). *Beamtimes and lifetimes: the world of high energy physicists*. Cambridge, Mass: Harvard University Press.

Turkle, S., & Papert, S. (1991). Epistemological Pluralism and the Reevaluation of the Concrete. In I. Harel, S. Papert, & Massachusetts Institute of Technology

(Eds.), *Constructionism: research reports and essays, 1985-1990*. Norwood, N.J: Ablex Pub. Corp.

Valentine, D. W. (2004). CS educational research: a meta-analysis of SIGCSE technical symposium proceedings. *SIGCSE Bull.*, *36*(1), 255–259. doi:10.1145/1028174.971391

Van de Sande, C. C., & Greeno, J. G. (2012). Achieving Alignment of Perspectival Framings in Problem-Solving Discourse. *Journal of the Learning Sciences*, *21*(1), 1–44. doi:10.1080/10508406.2011.639000

VanLehn, K. (1990). *Mind Bugs: The Origins of Procedural Misconceptions*. Cambridge, Mass: MIT Press.

Wagner, J. F. (2006). Transfer in Pieces. *Cognition & Instruction*, *24*(1), 1–71. doi:10.1207/s1532690xci2401_1

Wortham, S. (2006). *Learning Identity: The Joint Emergence of Social Identification and Academic Learning*. Cambridge: Cambridge University Press.

Yackel, E., & Cobb, P. (1996). Sociomathematical Norms, Argumentation, and Autonomy in Mathematics. *Journal for Research in Mathematics Education*, *27*(4), 458–477. doi:10.2307/749877

Yin, R. K. (2009). *Case study research: design and methods* (4th ed.). Los Angeles, Calif: Sage Publications.

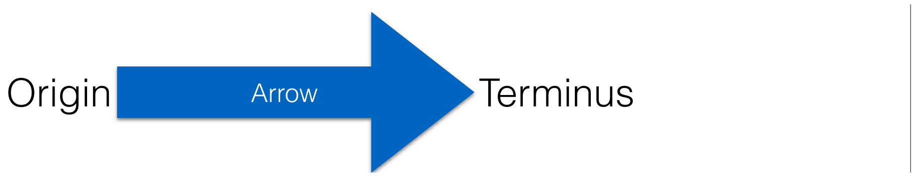
6 Appendix 1 – Transcript conventions

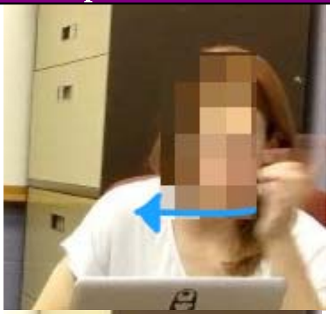
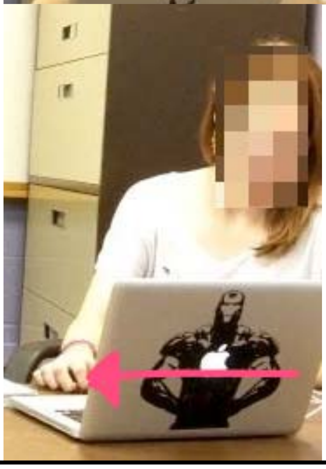
- Turns are not numbered, but they are blank-line-delimited
- Short interjected speech that does not interrupt a speaker's turn is bounded /by slashes/
- Matching double brackets show the [[onset and termination]] of overlapping talk across turns.
- *emphasized speech* is bounded by asterisks
- Parenthetical clarifications by the analyst appear in parentheses. (He considered but rejected square brackets.)
- matching double equals signs mark turn boundaries== ==with minimal or no audible silence (also known as latching turns)
- Where gestures don't overlap speech they are in-lined by curly braces when they happen {folds arms, having made his point}. All gestures enacted *by* a speaker appear within that speaker's turn unless otherwise noted {smiles, self-satisfied at having made this important clarification} {audience scoffs}.
- When gestures happen *during* speech, the speech is presented first and bounded by double pipes. Gestures that happen simultaneously with such speech are bounded by curly braces, nested within double pipes, and immediately follow the speech they overlap. For example:

  Throwing chainsaws ||*up*|| |{throws chainsaw}| is easy. Just be careful when they come ||*down*|| |{catches chainsaw for a punctuated finish}|.

7 Appendix 2 – Visual conventions for gestures

A challenge of this dissertation was trying to use still-frames to convey motion and animation. Where I could, I annotated pictures with arrows to show trajectories of movement:



Description	Example
Blue arrows project forward to show action that will occur but hasn't yet in the frame you're looking at. The origin is where a hand *is*; the terminus is where it *will be* shortly.	
Pink arrows show action that already occurred before the frame you're looking at. The origin is where a hand *was*; the terminus is where it *is*.	

8 Appendix 3 – Transcript of Rebecca's pseudo-code episode *without* gesture codes

Interviewer: OK. So then, um, what if we just pretended for a minute, that that, like==

Rebecca: ==That that works=

Interviewer: ==That that worked==

Rebecca: ==OK==

Interviewer: ==OK. So then, um. So then you might write like to make an array of {stops writing} actually what would we call this? This is==

Rebecca: ==array of pointers, I guess==

Interviewer: ==OK. Pointers. It'd be, well. So. The—the one we had was like int, star p, um, it would be some number, um, and I guess that'd be it in order to just declare /Yes/ that right? OK. And then you're saying, how would you like access an element of it?

Rebecca: Uh, well, what I was think—like if I wanted to like, save it or whatever, /Yeah/ I could make a while loop, scan in the—scan in the data from the album, /OK/ uh, which is, I can write it [if you want

Interviewer: Sure, go ahead]]

Rebecca: Um, so, it was==

Interviewer: [[Oh, sorry, just have to]

Rebecca: [Oh]

Interviewer: It's /Oh, OK/ one of those annoying ticks, but it works best cause of the camera if, ah, /Ah, OK/ if that's pointing away from your hand

Rebecca: [OK]

Interviewer: [That's fine]

Rebecca: Um, there was album, and then, so there was, uh, percent d, and percent s, and then give those names, just, I'll just call it number, and title. /Mmmhmm/ And so, at least, my—until it does not equal EOF, and then my thinking at least, is you should be able to, um, say that

Rebecca: “star p of i” /mmhmm/ equals, uh, the title, and then you just do i++, so then it’ll move to the next one /OK/ and you just keep saving each of the pointers in the array to a title /Mmmhmm/ And you just increment by 1 until you reach the end of file.

Interviewer: OK==

Rebecca: ==Like *that* would make sense to me.

Interviewer: So then, like, what would go in p of one would be the first title we read in==

Rebecca: ==Yes,

Interviewer: uh, would, when the while loop runs again, does it get a fresh line [[line from that

Rebecca: Yeah, because]] uh, what the while loop does is it reads in the line and then once it reaches the end of character it’ll go back down—it’ll do the while loop and then go back down to the next thing—line, it’ll read in the line until it reaches the end of file

Interviewer: So as you step through this loop /mmhmm/ i keeps going up by one==

Rebecca: ==Yes.

Interviewer: Uh

Rebecca: And the lines keep going down so, that way, the first line is going to be, uh the p—element zero /OK/ uh the second one’ll be element one (Interview 4 of 5, April 6, 2012)

9 Appendix 4 – Conceptual knowledge frameworks in computing don't tell us much about how Lionel structured an in-interview program

In this section, I explore Lionel's work in solving an in-interview programming task. First, I reproduce the prompt Lionel was given. Next, I show the final source code of his solution to the problem, written in C. In the analysis that follows, I move outward to explore the context of that code's production.

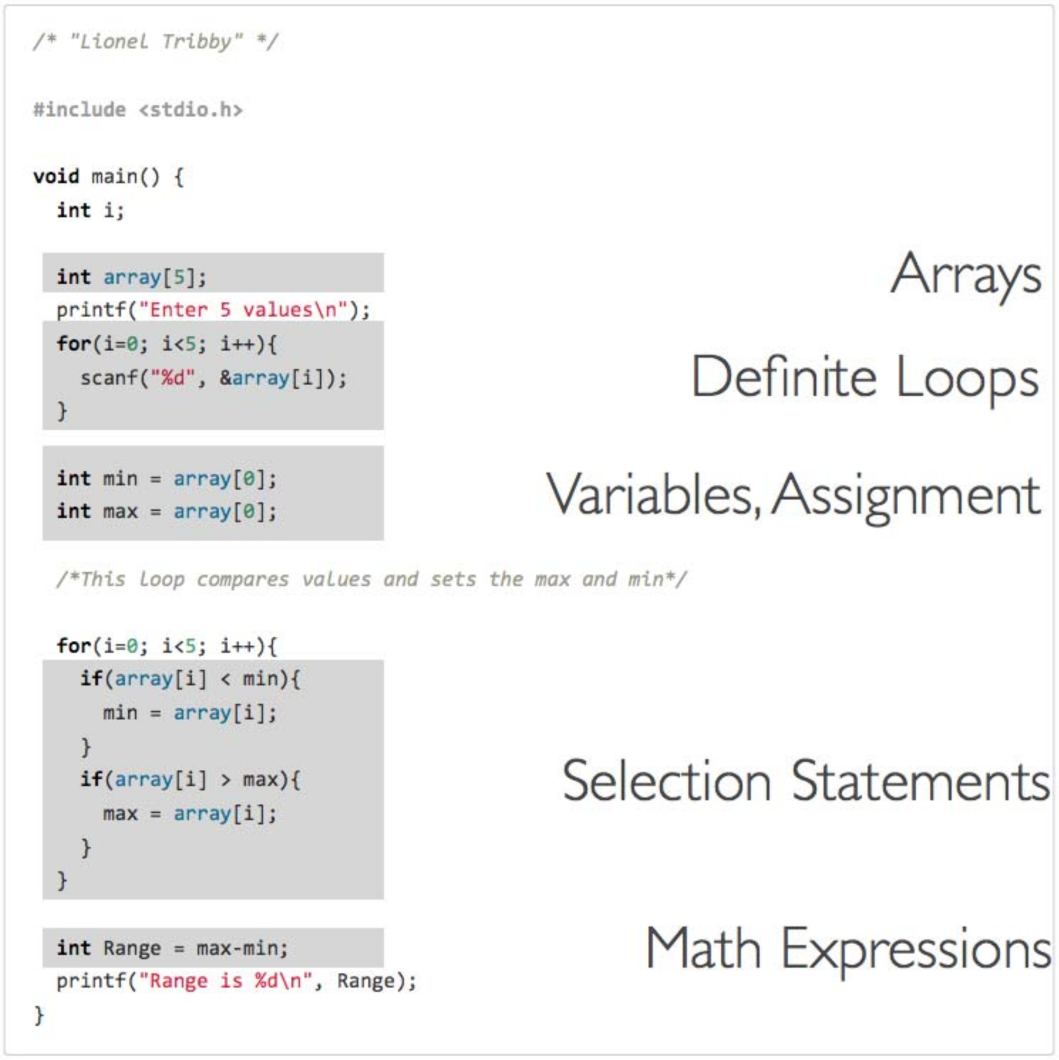
My choice to start with the code first is deliberate. I'm trying to mimic what the instructor of a typical programming course might see: the final submitted form of a student's code. This approach is admittedly a bit backwards, since the data I show after I present Lionel's submission is all about what happened *before* the code reached its final form. But, I think this approach useful because it invites us to explore an artifact first, then raise questions about the conditions of its production. In that sense, my presentation has the anthropological feel of understanding a found object, which I think is a faithful way of looking at what many instructors (not to mention researchers, and of course active software developers) face in their day-to-day work.

9.1 Analyzing Lionel's Solution From a Conceptual Knowledge View

To infer conceptual knowledge, I'll use Elliott Tew's (2010) conceptual categories for procedural programming in first-semester computer science (CS1) courses. My reasons for doing so are:

1. Elliott Tew (2010) sought concepts that were language-agnostic. That is, the concepts Elliott Tew (2010) identified are not idiomatic to just one language. Rather, they are general ideas implemented in nearly every major procedural language taught to undergraduates (Java, Python, C-based languages, Scheme).
2. Elliott Tew's (2010, pp. 23–27) selection process was extensive. She began with curricular documents, moved to canonical texts, then ultimately conducted a thematic analysis on emerging categories to produce her final list of concepts.
3. To date, Elliott Tew's (2010) identification of CS1 concepts is the only work I know of that has been carried forward to create, administer, and validate concept inventories in CS1 (Elliott Tew & Guzdial, 2010, 2011)

Figure 2 below presents an overview of the conceptual features I identified in Lionel's code. Even in a short, relatively simple program Lionel exhibits four of the ten concept categories Elliott Tew (2010) identifies. Moreover, he uses each of them "correctly," so to speak, in that not only does his program compile, it also properly finds the range of the numbers it's given. It's also worth noting that Lionel's code displays consistent, helpful use of whitespace (indentation) as well as informative comments. While these attributes aren't conceptual, per se, they are the sorts of features an instructor might consider in assigning a grade to this program.



```
/* "Lionel Tribby" */

#include <stdio.h>

void main() {
  int i;

  int array[5];
  printf("Enter 5 values\n");
  for(i=0; i<5; i++){
    scanf("%d", &array[i]);
  }

  int min = array[0];
  int max = array[0];

  /*This loop compares values and sets the max and min*/

  for(i=0; i<5; i++){
    if(array[i] < min){
      min = array[i];
    }
    if(array[i] > max){
      max = array[i];
    }
  }

  int Range = max-min;
  printf("Range is %d\n", Range);
}
```

Figure 9-1 – Highlighting the conceptual features of Lionel's code given Elliott Tew's (2010) outline of CS 1 concepts

10 Appendix 5 – Neverly-Asked Questions (NAQs)

10.1.1 Neverly Asked Questions (NAQs) about my conceptual analysis of Lionel's code

Didn't Elliott Tew (2010) define those concepts as a way of creating a concept inventory?
Yes. In her work, I believe each concept was represented by at least one multiple-choice question (MCQ) with carefully-chosen distractors. In each MCQ a student would see a snippet of pseudo-code and be asked to choose from a list of responses. The structure was explicitly modeled after Force Concept Inventory (FCI) work in physics education.

So, if her intent (and research) was about making a concept ***inventory****, aren't you misusing her ideas of concepts?*
What makes you say that?

Well, first off, she used categories of concept to create concrete instances of code, which she tests students on…
That's my understanding, yep.

But you used a concrete instance of code — Lionel's code — and inferred the existence of concepts in it. So, that's not the same thing as what Elliott Tew (2010) did.
I agree I'm not doing what she did, but I don't think that observation necessarily undermines the usefulness of what I *am* doing. If the concepts she identifies really are general concepts, then it would seem silly to think we couldn't find concrete instances of code that exemplify them. I think I might rephrase your objection: "how can I trust that you properly identified concepts in Lionel's code?" My answer to that is, again, yes, I have no coding scheme from Elliott Tew to apply. But, the code Lionel wrote is there and my candidates for assigning concepts are there: my results are open to inspection and challenge.

OK. Fine. But what "conceptual knowledge analysis" is there to speak of? You just made a slide and identified which parts of his code might correspond to which concepts.
What I did was assign concepts to canonically-correct — that is, syntactically valid parts of Lionel's code that also work properly to make the program produce the right results — sections of Lionel's code. I think your objection is about my inferential basis: "why does the presence of those code chunks — and my tagging certain chunks as aligning with Elliott Tew's (2010) concepts — imply that Lionel has those concepts in his head?"

I accept your amendment. So, what basis do you have for saying "this concept is in Lionel's mind?"

I feel like I'm on the same playing field as the concept inventory folks, though in some ways I'm setting a modified standard for validity. Elliott Tew (2010) says this about validity:

> validity is the evidence that assures us that questions about a particular concept are indeed measuring that concept. For instance, a question about arrays should require a student to have knowledge about arrays, but should not require knowledge about another concept, such as recursion. In addition, it is important that the question cannot be answered correctly without knowledge of arrays. (Elliott Tew, 2010, p. 10)

I don't know that I share her assumptions about validity. But, what specifically might be at issue here is "does the presence of a for-loop in code imply the student has knowledge of definite loops?" My answer is, "if not that, then what?"

But couldn't I argue that the for-loop could be there because he copied and pasted it from the internet? That he wouldn't have had to know anything about definite loops to do that?
Yes, you could. My rejoinder to that is in section 3.3.3.

10.1.2 Neverly-asked questions about Lionel's verbal pseudo-code description

Why do you so pedantically analyze such a short, small snippet of conversation?
Because by Lionel's own assertion what was said constitutes his "main concept" (line 12) for the program. That is, by his own account he has just expressed the entire top-level design for the program. But, he's done it all all without writing a single additional line of code.

So?
If anything, this episode suggests that Lionel can and does have ways of expressing a process or procedure that don't rely on writing code. More importantly, I think, is that he views it as a legitimate activity to express ideas at a level *above* the particularities of syntax and implementation.

What do you mean "views it as a legitimate activity?"
I think here I'm appealing to my interpretation of Goffman's (1974) *frames*: a participant's understanding of "what is it" that's going on in a social situation. Lionel takes up my bid for him to explain what he's thinking, but he does so first by explaining his "general concept" for the program.

Right. But, I mean, he's just doing what you ***asked*** *him to do. I don't understand how any of this relates to Goffman or frames. That just seems like a needlessly complicated way to explain he did what you asked.*
OK, let's step back for a second. Suppose you've just started going to group therapy. One of the things group therapy environments often stress is that you should try to

talk about how *you feel*, which can be a hard transition for people. Group therapy participants may be used to saying "you always do [X]" when in a fight with someone. And, if prompted by a group leader to discuss how they feel about troublesome instance, a participant may adopt that kind of language. "My partner always yells at me if I'm out late," for example.

Group therapy can encourage inward reflection in a way that reshapes how participants talk about themselves and their feelings. So, after some time in group therapy our hypothetical participant may change the way they orient toward questions. If asked "how are you feeling about the fight you and your partner had?" our hypothetical participant may now say "I feel like I don't have the freedom to see the people I want to see." The focus in speech shifts from the *other*—in this case a partner—to the *self*. Moreover, the substance of the speech is about feelings, rather than behavior.

I think these reshapings of speech are neither trivial nor incidental. If we had transcripts of our hypothetical person's early sessions in group therapy and compared them to those of later sessions, I think we would not be surprised to see a marked change. When asked the same question—in this case "how are you feeling about [X]," the structure and character of our participant's response changes after therapy.

My point here is that an everyday sense of how participants' speech patterns change in therapy primes our intuition for thinking more broadly about social phenomena. A cynical view of our participant would still have to acknowledge that, observably, the structure of their response to the question "how does that make you feel?" changes after therapy. I think a sociolinguistic stance would be more generous: our participant is *orienting* differently toward the question. Furthermore, I take the following as evidence of such an orientation:

- A change in sentence focus from the other to the self
- Introspection into the source of a feeling
- Acknowledgment of personal responsibility for feeling those feelings

I think we as analysts can acknowledge that after therapy, our participant frames the event of being asked, "how does that make you feel?" differently.

OK. So, you're saying that if we look at the change in how someone answers the question "how does that make you feel?" we can find evidence of the superstructure—a frame—that directs their sense of appropriate ways to answer that question?
Yes.

So how does this get back to your claims about Lionel?
I think it's tempting to take this position that, "it's inevitable/totally expected that he would just verbalize the top-level description of his procedure! I mean, what else could he *possibly* do?"

I mean, yeah. Kind of.
Right, but I think actually that assumption isn't in line with a classic frame analysis approach. Even if the assumption is right, frame analysis changes the explanation

from “it’s inevitable” to “hmm, I guess almost everyone I’ve ever seen brings or enacts the same structure of expectations.”

So, my argument is, if you ask someone to describe to you how they’re thinking about their program, they’re gonna do that. What else ***would*** *they do?*
Right. And I don’t think we disagree that they’ll describe their thinking. I think where we disagree is that everyone will describe in the same way. I’m saying *if* they describe their program to you, *that* they’re able to express it, what they choose to verbalize, what they choose to leave out, how their gestures accompany the explanations, and their ultimate criteria for having satisfied the task of describing for themselves are all driven underneath (or from above?) by structures of expectations. And even if *one thousand* out of one thousand people do the same kind of thing Lionel does, that doesn’t undermine the idea that people have and bring structures of expectations to a task. What I think it does is show that one thousand out of one thousand people share the same sets of expectations about what it is that’s going on when they’re asked to verbalize their thinking on a programming protocol.

So, what’s the point of this whole section on Lionel’s verbal description, then?
I think the point is that when I ask Lionel to verbalize his thinking, he outlines through talk what the “main concept” of his program is. I focus on *that* he decides to describe it out in words, *how* he describes it, *what* he leaves out, *how* he repairs, and how all of that reflects structures of expectations about the activity and what he’s being asked to do. Ultimately, the argument is that Lionel’s choice to take me up on that bid and how he does so reflects—to me, the analyst—that he views *verbal description of a high-level program* as a legitimate activity and part of his approach to programming.