# Introduction

I sat, the night before our next interview, poring over Rebecca’s code. Her module for finding all the one-stop flights between two airports absolutely baffled me. It had loops within loops within loops. By my count her program went 15 levels deep in scope, and that was only the lines I could see at the top of the function on my screen; I hadn’t even begun scrolling down into the deepest bowels of her one-stop flight code.

In fairness, it wasn’t just the Russian doll quality of Rebecca’s that shocked me. In my nearly two semesters of ethnographically studying students in Intermediate Programming for Engineers, I had gotten used to certain cosmetic features of students’ programming code, chief among which were lines of code that plowed far off the rightward edge of my screen, often declaring variables or specifying Booleans without any of the visual punctuation expert programmers employ to keep their code readable. But, it also wasn’t at all unusual in my study to see a single top-level procedure dozens of lines long, popping deeply into and ultimately out of a series of nested rabbit holes as it executed. No, what struck me about Rebecca’s code wasn’t just that it was labyrinthine. What struck me was that I couldn’t even understand what it was doing or why. A dozen questions swirled in my mind, but by far the loudest was “is there any way she can even *explain* to me how this code works?”

? In the case presented here, I use data on the form and evolution of one student’s software course project to outline two mutually dependent concerns in computer science education. The first concern, core to the disciplines of computer science and software engineering, is about how students learn and deploy conceptual ideas in computer programming as design solutions to manage the intellectual complexities inherent in solving problems with software. The second concern, crucial to refinement of a science of learning, is how we construct tools and build theory to account for students’ growth and development as designers of software.

In what follows, I relate the case study of Rebecca, a student taking a second-semester course in programming for engineers. I first stake out relevant strands of literature for contextualizing my theoretical and methodological aims of studying students’ software development in context. Then, I describe the study design, its participants, and my role as ethnographer /interviewer. I move to data and analysis for the core of my case, which revolves around a particular piece of code Rebecca created as a module in a multi-week “Flights Database” project for her programming course. Finally, I conclude with a synthetic discussion, drawing together current aims from computer science and engineering education and exploring what it means to study and support students learning to design code.

# Literature Review

If the question undergirding this study is about how programming code reflects students’ knowledge of design, the search for an answer then necessarily involves appeals to different, contextually relevant bodies of knowledge. Any account of the processes by which novice students might become expert software designers needs a far-reaching and flexible vocabulary to describe the entities and phenomena involved. Accordingly, I borrow ideas from three bodies of literature. *Software engineering* literature seeks, among other things, to document and systematically improve expert programmer performance. *Conceptual and epistemological* framing literature fuses linguistics, sociology, and cognitive science to address most broadly students’ sense of what kinds of knowledge “counts” when engaging in certain activities. Finally, specific directions in *computer science education* advance new methods and techniques that help us understand students’ difficulties when learning to program.

## Software engineering, code reuse, and cognitive bias

## Framing, accountable disciplinary knowledge, and cognitive artifacts

## Studying the novice programmer

In recent years, researchers in the field of computer science education have developed software tools for capturing snapshots of students’ programming code.[[1]](#footnote-1) The argument for such work has, at least in part, been that capturing students’ code allows us to better understand how novices program. We can, for instance, use the data to detect the most common compile-time errors students tend to make (Spacco, Pugh, Ayewah, & Hovemeyer, 2006). We can also use compile-time snapshots of students’ code to help us create predictive indices of student performance. For example, Jadud (2006) developed the notion of an “error quotient” as a way of quantifying how reliably — in an aggregate sense — a novice was able to subsequently fix an error on their next compilation.

Jadud (Jadud, 2006) found the error quotient correlated negatively with students’ final exam scores. That result makes sense: students who seem unable to quickly and reliably fix code problems as they arise are at the very least the ones we expect might score less well on larger summative programming assessments. Jadud (Jadud, 2006) acknowledged, however, that the negative correlation (R^2 = 0.25) between error quotient and final exam score was “very poor,” citing that among other factors, the snapshots that served as the basis for inference were only collected when students worked on their code in a lab on campus.

Two larger-scale concerns emerge with research that attempts to synthesize information across snapshots of students’ code. The first is the challenge of moving across scales: how do we interpret within and aggregate across snapshots of students’ code to recover information about how they’re learning to program? For any individual snapshot we know a great deal about the *structure* of the code: we can statically analyze it for compile-time errors, we can compare those errors to ones students commonly display as a group, and we can even in principle compare snapshots of how different students solved the same problem. But, we can’t know for certain what a student might have been thinking at the time the code looked like that. We also have avowed difficulty in moving across snapshots for the same student, attempting to infer why they would have ignored a compile-time error or, as in the case of Neville, why their response to a given error would take the form that it did.

A second large-scale concern has to do with the unit of analysis: treating code as student work. A noted limitation with some of the first studies of the BlueJ Java development environment was that researchers could only capture the code students wrote while they were in lab. We know from recent research that students program in many environments outside of just lab (Fincher, Tenenberg, & Robins, 2011). Moreover, we know that these environments can have both emotional affordances for students—as in Fincher et al’s (Fincher et al., 2011) description of a student who kept their secondary school awards in their room as a reminder of their achievements—and that for some students the *majority* of time spent “programming” could be outside of the lab. Another way to formulate this second concern is to say that if seemingly external factors like the environment can matter, that if they can matter a *great deal* for the frame of mind from which students approach programming, then have we sufficiently specified such microdynamics such that our large-scale analyses of students’ code snapshots will produce meaningful results? Or, to put it more simply: snapshots can tell us in exhaustive detail what students’ code looks like, but are they providing a sufficient account of *why* it looks the way it does, and why it evolves the way it does over time?

One attempt to move toward explanation comes from Rodrigo and colleagues, who have tried to use snapshot data to understand student affect (Rodrigo, Baker, et al., 2009; Rodrigo & Baker, 2009; Rodrigo, Tabanao, Lahoz, & Jadud, 2009; Tabanao, Rodrigo, & Jadud, 2011). And, while the title of Rodrigo and Baker (Rodrigo & Baker, 2009) suggests a “coarse-grained” approach, the methods used actually amount to detailed time-series sampling: observing students facial expressions at 20-second intervals and coding their concurrent behavior at the time. Such sampling can tell us, certainly with minute-by-minute resolution, the externally observable features of a students’ programming activity. Notably, however, the researchers worked explicitly to *distance* themselves from what they observed, even taking care to make observations “surreptitiously” (Rodrigo & Baker, 2009, p. 76) and in a way that made it difficult if impossible for any given student to know whether they were being observed. Moreover, and perhaps crucially for this discussion, Rodrigo and Baker (Rodrigo & Baker, 2009) remind readers that “the purpose of this study is to detect frustration in an aggregate fashion across all labs and not to detect frustration as it happens, action-by-action” (Rodrigo & Baker, 2009, p. 77).

The above clarification helps us put such research in context: the aim of Rodrigo and Baker (Rodrigo & Baker, 2009) was to determine whether, when, and how much frustration presented itself in a given classroom. Their aim was not, however, to delve deeply into the source of the frustration. They were also not focused on how students experienced what might be externally observed as boredom or frustration. Are two students who externally evince frustration really experiencing the same emotion, and for the same reason? I contend that while the answers to those questions were understandably outside the scope of Rodrigo and Baker’s (Rodrigo & Baker, 2009) investigation, they are answers that nonetheless matter a great deal for applying a science of learning to the deep questions of how students come to learn programming and develop expertise in it.

# Discussion

There are several features of Rebecca’s code that would seem unusual, complicated, or otherwise inefficient to professional programmers. Moreover, there are features of her code that expressly ignore directives given in the assignment—though, in Rebecca’s defense, the project assignment certainly wasn’t sparing in the number of express directives it dictated to students. In this section, I focus on two features of Rebecca’s code: the stateless scanning patterns she uses to read in the file and the repeated chunks of code she uses to handle differing user input for days. For each feature, I try to detail what makes it a striking design choice. I also draw support from background literature to explain why experts might regard such choices as inferior or suboptimal. What I aim to show, however, is that while Rebecca’s design choices depart from expert practice, it’s consequential that they retain an unproblematic character in Rebecca’s discussion of them. These are design choices that might make a team of professionals livid, but that’s not quite Rebecca’s response to them. Furthermore, though Rebecca might not know it, these are choices that actually would be sensible or (in some cases) optimal for building certain kinds of software systems in particularly constrained environments.

---

It’s enlightening to compare Rebecca’s sentiments to the preface of Abelson and Sussman’s classic introductory computer science text *The Structure and Interpretation of Computer Programs*:

Our design of this introductory computer-science subject reflects two major concerns. First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. Second, we believe that the essential material to be addressed by a subject at this level is not the syntax of particular programming- language constructs, nor clever algorithms for computing particular functions efficiently, nor even the mathematical analysis of algorithms and the foundations of computing, but rather the techniques used to control the intellectual complexity of large software systems. {http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-7.html#%\_chap\_Temp\_4}

At face value, Rebecca has {Abelson and Sussman’s} idea precisely backward: Rebecca’s program-writing priorities are to get her code to properly execute, and only incidentally (if there is time) to write it for people to read. Rebecca’s flipped priorities are even more consequential in light of {Abelson and Sussman’s} second concern: managing the intellectual complexity of software systems.

There is a subtle yet consequential difference between viewing code as “cosmetically readable” and understanding a particular program as helping manage aspects of complexity. {Abelson and Sussman} offer a striking example when they discuss the `map` feature of the Scheme programming language.[[2]](#footnote-2)

Map is an important construct, not only because it captures a common pattern, but because it establishes a higher level of abstraction in dealing with lists. In the original definition of scale-list, the recursive structure of the program draws attention to the element-by-element processing of the list. Defining scale-list in terms of map suppresses that level of detail and emphasizes that scaling transforms a list of elements to a list of results. The difference between the two definitions is not that the computer is performing a different process (it isn't) but that we think about the process differently. In effect, map helps establish an abstraction barrier that isolates the implementation of procedures that transform lists from the details of how the elements of the list are extracted and combined. Like the barriers shown in figure [2.1](http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html#%_fig_2.1), this abstraction gives us the flexibility to change the low-level details of how sequences are implemented, while preserving the conceptual framework of operations that transform sequences to sequences. {http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-15.html#%\_sec\_2.2.1}

In a sense, {Abelson and Sussman} contend that programming a computer is not about specifying a process, but rather about formalizing *how we think* about processes.

That conclusion brings us fully back to Rebecca’s comment that her repeated code chunk made sense, and became something she could actually work with. Rebecca’s description of the code chunk she repeated echoes that of a child in Seymour Papert’s seminal book on children and the LOGO programming environment:

Our aim is to subdivide the program into natural parts so that we can debug programs for each part separately….Robert, a seventh grader, expressed his conversion to this style of programming by exclaiming: “See, all my procedures are mind-sized bites.” (Papert, 1980, pp. 102–103)

Papert describes students’ adoption of this kind of programming approach as one of “conversion,” tied in part to the utility it affords for tasks like debugging. Indeed, Papert contrasts Robert’s work in structured programming with that of Keith, who writes out each individual step for his program to draw a stick figure, rather than using hierarchy to decompose them. Papert notes:

In Keith’s long, featureless set of instructions it is hard to see and trap a bug. By working with small parts, however, bugs can be confined and more easily trapped, figured out….Once these “subprocedures” have been written and tested, it is extremely easy to write the “superprocedure” to draw the stick figure itself. (Papert, 1980, p. 102)

What’s particularly striking is that Rebecca was aware of what made the fantasy football setup advantageous: a single line of the file contained all the relevant information about the data unit of interest: in this case, a player. Because C has `fscanf()`—a command that lets a program read in an entire line of a file at once rather neatly—there was a consequently neat pairing of design constraints and programming logic: a single `fscanf()` statement could handle reading in all the information needed to make judgments about a player. The flights database presented a more complicated design context, since she’d have to coordinate multiple files. One thing Rebecca *could* have done was recognize the advantage of the old assignment—the “one file” that unified all the information—and recreate a similar one-ness using an abstract structure to represent something like a flight.

{Insert graphic that makes a neat comparison between the “single hypothetical line” of a fantasy football file and an abstract “flight” structure that would clearly have fields for airport code, route number, flight day/time information, full airport name, and route ID}

In other words, one potential way Rebecca could have *transferred* what she learned from the fantasy football context would have been to recognize how useful it can be to have data objects that collect all the query-able fields (airport code, day/time information, etc.). In the case of fantasy football, that bundling together of the relevant fields was a feature of the input data. In the airport project, then, one way to achieve a similar unification would have been to create a unit of abstraction—called flight—that could be built from information contained in the three separate files and would embody the same elemental one-ness that characterized the lines of the fantasy football input file. In fact, it seems reasonable to conclude the instructor *wanted* the students to adopt such an approach, given that the topics of abstract structures and memory management—both of which are essential to creating new units of abstraction—were covered in the first two lectures after the flights database project was handed out.

1. I try, where possible, to provide clearer descriptions of computer science content and particular research techniques for those readers who may be less familiar with their technical specifics. [↑](#footnote-ref-1)
2. Map provides a way in Scheme of expressing that a single-argument procedure should be applied to each and every element of a corresponding list. As a simple example, map makes it easy to define procedures like “re-scale each number in this list by multiplying it by a constant factor of my choosing.” In the above quotation, the authors are explaining that map is at the core of the scale-list function. They use map to define the relationship between the input scaling factor, the list, and the operation to be applied to the list (multiplication by the scaling factor). [↑](#footnote-ref-2)