Reconstructing design thinking and learning through code snapshots and clinical interviews

Authors Blinded For
Review

ABSTRACT

CCS Concepts

•Social and professional topics → Computing education;

Keywords

ACM proceedings; LaTeX; text tagging

1. INTRODUCTION

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 [5, 10, 13]. 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 [13], the development of compile-time detectors to catch common student errors [14], and the proposal of formative assessment models to predict student success [5, 15]. 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 [6].

What's common to these threads of research is how they mine the data. In most applications of code snapshot[1] research, the aim is to average across events and sessions to arrive at a characteristic measure of a student's behavior. Jadud [5], 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.[2] Rodrigo and colleagues extend the idea of error quotient and introduce similar measures such as students' "compilation profiles" [10], "frustration profiles" [9], and "error profiles" [15]. Because these measures can be collected in real-time as students code, they offer the potential to identify at-risk students and respond with early interventions.

But, for all the data these methods collect their focus is primarily what Jadud [5] 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[3], 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

[1]What I call "code snapshot" research may be more formally called "online protocol" research [5, 10]. "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*.

[2]A useful way of thinking about Jadud's [5] error quotient is that the computation believes in second chances. It doesn't penalize students for making errors; it penalizes students if they don't fix a given error before compiling the same piece of code *again*.

[3]In a screenplay, a slug line gives information about where and when a scene takes place. For example: ■`INT. FORTRESS OF SOLITUDE - DAY.`■

ICER 2016 Melbourne, Australia

DOI: 10.475/123_4

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 :

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. CONTEXT AND METHODS

2.1 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.[4] 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.

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

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).

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.

2.2 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.[5] I interviewed Rebecca five times, and in typical interviews I split 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.

3. ANALYZING ONLY THE CODE REBECCA TURNED IN

In this phase of analysis, I restrict data to only the final version of `one_stop_flight.c` Rebecca submitted as part of her project. This restriction is important because of the data that gets left out. We're forced to see Rebecca's design work the way her instructor did when he graded it: as a final product. The submitted code contains little—if any—evidence of design iteration. And, we have almost no access to streams of activity (inscriptional, gestural, verbal, and otherwise material) that would help us understand Rebecca's early stage design [8]. Indeed, other than the code itself, the only artifacts that actually carry Rebecca's voice in this phase of analysis are the comments she places in her code. Put another way: restricting our scope to just code—and only the final submitted code at that—denies the analyst access to the channels of "talk, embodied action, and inscription" involved in design work [3, p. 179].

3.1 Rebecca's file-scanning solution is hard to read and has high time-complexity

[5]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.

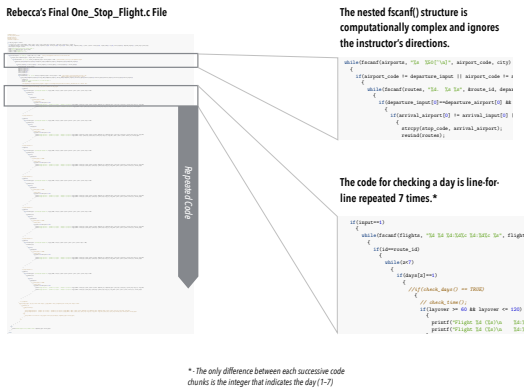


Figure 1: The entirety of Rebecca's `one_stop_flight.c` source code at the time her project was submitted for grading. Our final-code-only analysis focuses on two design features of this code: its multiply-nested `fscanf()` structure for handling flight data; and the seven-fold line-for-line repetition of a single block of code for checking days of the week.

After declaring variables and opening the three provided text files (flights, routes, and airports)[6], Rebecca's one-stop flight code enters a series of conditionally-nested `fscanf()` commands. Any suitable solution for this project would need extract text patterns from source files, which `fscanf()` does. So, *that* Rebecca uses `fscanf()` in her code is not itself surprising. 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 (or sometimes repeatedly open the same file multiple times) and scan lines 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 (line 20)
2. While scanning a given line containing an airport code/city pair, 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 com-

[6]See Appendix A for examples of each of the three file types.

```
while(fscanf(airports, "%s  %50[^\n]", airport_code, city)
  {
    if(airport_code != departure_input || airport_code != a
      {
        while(fscanf(routes, "%d.  %s %s", &route_id, depar
          {
            if(departure_input[0]==departure_airport[0] &&
              {
                if(arrival_airport[0] != arrival_input[0]
                  {
                    strcpy(stop_code, arrival_airport);
                    rewind(routes);
```

Figure 2: A portion of Rebecca's nested `fscanf()` statements

plex. The multiply-nested blocks can make it difficult for a human reader (or grader) 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 dimensions of complexity in Rebecca's program—what computer scientists would call the "Big-O" characterization of her program [1]. For every nesting of a scan loop (there are 4 in her submitted code) 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. But, Rebecca's code also has a particular kind of elegance.

3.2 Rebecca's file-scanning solution has elegant constant space complexity

While Rebecca's design isn't optimized for speed, it uses drastically less memory than do other solutions (including the instructor's official solution). What Rebecca could have done (and what I'll discuss in a later section) is read the entire contents of all data files into RAM. I'll call that approach an *in-memory* solution, because all of the flight information is loaded into memory. Instead, her solution stores only a handful of lines (4 at most) in memory at a time, leaving the rest *on-disk*. An analogy helps clarify the difference.

If we think of the input files like a giant grocery list, an in-memory solution would be like having to put each list item into your cart before you go to check out. The longer the list gets, the bigger cart you'd need to collect every item on the list before checking out. Crucially, everything has to go into the cart before it can be purchased. If the grocery list very long, you may need multiple carts. If the list gets absurdly long you may even exhaust all carts in the store and still find yourself with an unfinished list.

Rebecca's *on-disk* solution would be like restricting yourself to a single, small grocery basket, but taking as many trips as you need to get all the itmes from the store shelves to the checkout conveyor belt. She can only ferry just a few groceries each go-around. But no matter how long the list gets, she'll never need more than a single handbasket to get all her items to checkout. She just takes more trips.

The reason for comparing an in-memory to an on-disk solution is that design work always operates under constraints. Sometimes (say, in cloud computing applications) RAM is cheap, and a solution that loads all data into RAM may be optimal. But, there are also applied situations (biomedical implants, space technology) where memory is expensive and possibly not even upgradeable. In those contexts, a constant-space solution like Rebecca's might be ideal, because designing for the long-term means assuming a computer we launch into space now and can't ever touch again for years or decades.

But, while we know the structure of Rebecca's design, we know almost nothing about its context. Our analytical method—examining only the code in front of us—forecloses possibilities of recovering those details. We lack access to activity history: we don't know *how* Rebecca ended up structuring her code this way, nor do we know *why*. And, we lack access to conceptual information: we can't know from just this code whether Rebecca knew or understood the kind of complexity and performance trade-off she made. We also can't know with certainty how she felt about the consequences of the decision. We know only that her final submitted design used multiply-nested scanning loops.

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

For each of the four projects during the semester, the instructor gave students what he called a "design brief." Each design brief outlined the problem to be solved as well as any constraints imposed on students' solutions. For example, in the flights project, students' programs had to reject candidate multi-stop trip routes if the layover time would be too short (under 30 minutes) or too long (more than two hours). But, in addition to what I might call *user-centered constraints* (viz., reject multi-stop trips that would have grueling layovers), the instructor also directed students on *implementation details*: ways their program should work at a technical level that would be invisible to the user.

Specifically, the design brief discusses how to handle reading in data from the files provided for the project:

> 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). And, to fulfill that need as stated, a student's code would have to:

- Create arrays by declaring them as variables
- Store data from the files in array entries using variable assignment
- Access the arrays to fetch relevant flight data

Rebecca creates no arrays. Instead, her code attempts to accomplish the same task that an array would, but she doesn't use a global data structure at all.[7] A consequence

[7]That's not entirely true. Technically, variables including `route_id` and `flight_number` are globally-acessible within the scope of the `one_stop_flight()` function. But, those variables are integers. There's no way the variables Rebecca declares could store all the data required in-memory.

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.[8]

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 and created March 19, 2012, we see the following lines:

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

his header file!!!
-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]
```

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?

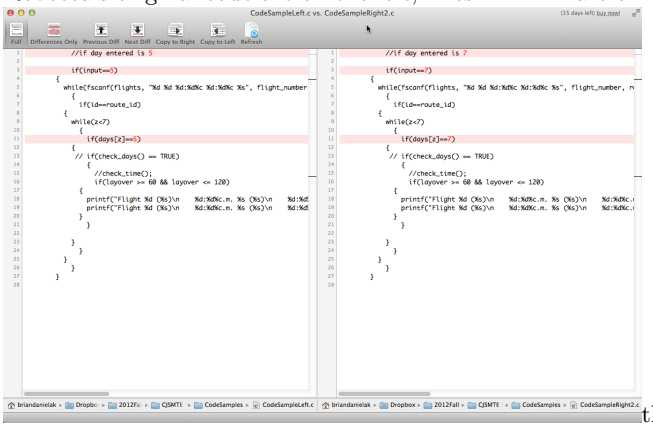
We return to this question in a later section, but first we turn our attention to another unusual feature of Rebecca's work: seven-fold repetition of code.

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

3.4 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.[9] 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.

Figure 2-1 represents a side-by-side delta-comparison of two such day-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).[10] 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 seven different blocks of code: once

[9]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.

[10]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.).

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.

4. AUGMENTING CODE SNAPSHOTS WITH INTERVIEW DATA

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

1. 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).

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

Rebecca's choices become easier to understand when we consider what she said in interviews about the code she wrote. My first opportunity to discuss that code 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)

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.

4.0.2 *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

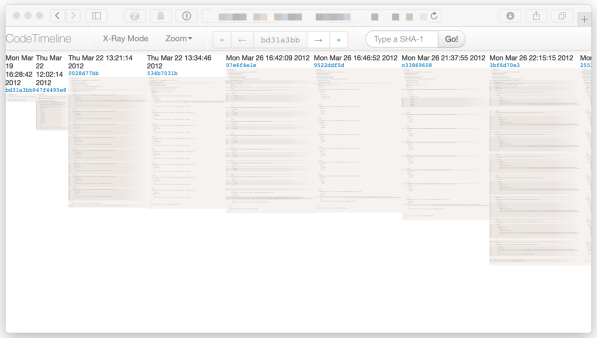


Figure 3: We developed CodeTimeline to display snapshots in a timeline. Darkened lines reveals how the seven-fold repeated code began as a very simple switch statement but grew and grew in complexity over time.

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."

5. CONCLUSION

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, 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. [7] Parson and Saunders 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 [2, 4, 11] and mathematics education [16], 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 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). [16]

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 [12] 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. [12]

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" [12]

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.

6. ACKNOWLEDGMENTS

Blinded for review.

References

[1] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to algorithms*. The MIT electrical engineering and computer science series. MIT Press ; McGraw-Hill, Cambridge, Mass. : New York, 1990.

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

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

[4] Hammer, D., Elby, A., Scherr, R. E., and Redish, E. F. Resources, framing, and transfer. In *Transfer of learning from a modern multidisciplinary perspective*, J. P. Mestre, Ed., Current perspectives on cognition, learning, and instruction. IAP, Greenwich, CT, 2005.

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

[6] Kölling, M., and Utting, I. 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* (New York, NY, USA, 2012), SIGCSE '12, ACM, pp. 323–324.

[7] Parsons, J., and Saunders, C. Cognitive heuristics in software engineering: Applying and extending anchoring and adjustment to artifact reuse. *IEEE Transactions on Software Engineering 30*, 12 (Dec. 2004), 873 – 888.

[8] Petre, M., and van der Hoek, A., Eds. *Software designers in action: a human-centric look at design work*. Chapman & Hall/CRC innovations in software engineering and software development. CRC Press, Taylor & Francis Group, Boca Raton, 2014.

[9] Rodrigo, M. M. T., Baker, R. S., Jadud, M. C., Amarra, A. C. M., Dy, T., Espejo-Lahoz, M. B. V., Lim, S. A. L., Pascua, S. A., Sugay, J. O., and Tabanao, E. S. Affective and behavioral predictors of novice programmer achievement. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (New York, NY, USA, 2009), ITiCSE '09, ACM, pp. 156–160.

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

[11] Scherr, R. E., and Hammer, D. Student Behavior and Epistemological Framing: Examples from Collaborative Active-Learning Activities in Physics. *Cognition & Instruction 27*, 2 (Apr-Jun2009 April 2009), 147–174.

[12] Schwartz, D. L., Chase, C. C., and Bransford, J. D. Resisting Overzealous Transfer: Coordinating Previously Successful Routines With Needs for New Learning. *Educational Psychologist 47*, 3 (2012), 204–214.

[13] Spacco, J., Pugh, W., Ayewah, N., and Hovemeyer, D. 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* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 669–670.

[14] Spacco, J., Strecker, J., Hovemeyer, D., and Pugh, W. 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, May 2005).

[15] Tabanao, E. S., Rodrigo, M. M. T., and Jadud, M. C. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research* (New York, NY, USA, 2011), ICER '11, ACM, pp. 85–92.

[16] Van de Sande, C. C., and Greeno, J. G. Achieving Alignment of Perspectival Framings in Problem-Solving Discourse. *Journal of the Learning Sciences 21*, 1 (2012), 1–44.

ICER 2016 Submission 114

Update authors

Submission information updates are disabled.
For all questions related to processing your submission you should contact the conference organizers. Click here to see information about this conference.
All **reviews sent to you** can be found at the bottom of this page.

Paper 114	
Title:	Reconstructing design thinking and learning through code snapshots and clinical interviews
Paper	
Track:	Research papers
Author keywords:	code snapshot systems knowledge analysis clinical interviews design thinking
EasyChair keyphrases:	flight number (670), departure airport (460), arrival airport (410), stop code (350), travel day (250), check day (250), stop flight (235), check time (230), stop city (190), hour stop (190), airport code (170), letter stop (160), d min stop (126), departure city (110), arrival city (110), rebecca code (90), basic programming (80), code snapshot (70), student code (70), d hour stop (63), flight database project (63), departure input airport (63), stop flight code (63), fantasy football (60), arrival time (60), flight database (60), departure time (60), rebecca design (50), arrival code (50), computer science education (47)
Abstract:	This paper combines two existing research methods—code snapshots and clinical interviews — to understand how a student's design and and learning evolve in a university introductory programming course for engineers. Historically, clinical interviews have proven very useful for understanding the structure and dynamics of student knowledge (Sherin, 2013; Smith, diSessa, & Roschelle, 1993). But, in computing education the constraints of interviewing make it hard to resolve the fine-grain changes students may make to code over time. On the other hand, code-snapshotting systems (Blikstein et al., 2014; Jadud, 2006; Spacco et al., 2006) are very well-suited to resolve those kinds of changes, but they have historically been used to look between students for groupwise effects and correlations, rather than within students to trace the evolution of code as a design artifact. The case study we present uses a knowledge-in-pieces perspective (diSessa, 2002, 2006) to analyze more than 900 compile-time snapshots and more than 5 hours of clinical interview data and reconstruct a student's design thinking over the trajectory of a multi-week programming project. We argue that augmenting snapshots with interview data offers substantial explanatory power for understanding how a student's design choices reflect their design thinking and learning.
Time:	Apr 16, 04:55 GMT

Authors

first name	last name	email	country	organization	Web page	corresponding?
Brian	Danielak	briandaniela.k+easychairconference@gmail.com	United States	Michigan State University		✔

William	Doane	wdoane@ida.org	United States	Institute for Defense Analyses		

Reviews

Review 3	
Overall evaluation:	**4**: (Borderline, lean to accept)
Summary of paper:	The authors describe a case study (of one student) where they are trying to develop accounts of students' programming activity that explain the form and evolution of their code on a design project. They use ethnographic observation clinical interviewing, and code snapshot analysis to examine a student's design choices on a coding project.
Discussion of related work:	**4**: (covers key related work; its relationship to submission is described, but could be extended further)
Theoretical basis for the paper:	**4**: (theoretical basis obvious, with some citations and argument for how it is applied in the research)
Use of theory:	The authors cite various works that are related to their study. They motivate their work by explaining the current state of the world in the context of code snapshot review, and talk about the pros and cons of using this kind of data to interpret information about users and their needs.
Research methodology:	**2**: (questionable choice of research approach and methods)
Exposition of research methods:	**3**: (data collected and analyzed, but some aspects unclearly described)
Discussion of results and conclusions:	**3**: (plausible interpretation of findings)
Methodology and empirical basis:	Overall, it is very clear what the authors did in this study – they analyzed the work Rebecca did for one of her projects using code snapshots and augmenting that information with interviews. The data is rich and well described. One concern the reviewer has is that this is a case study of *one* individual, so it is unclear how what we learned from Rebecca can generalize. However, the contribution of the paper is partly the method in which this study was conducted and demonstrating how it can be beneficial to use different sources of information to triangulate on interpretations and conclusions about a learners' coding process and issues they may have run into. Another concern is the (lack of) description of how Rebecca was chosen to be the subject. According to the authors, there were 3 other students who participated in the study. What happened to their data? Why exactly was Rebecca chosen and not the others? One major concern is whether or not Rebecca is representative of other students. This leads to the question of whether the method described in the paper to analyze Rebecca's work and interview her would work with other students?
Contribution and relevance to the international computing education research field:	**3**: (minor contribution or contribution is bound to a local context, perhaps with the promise of more to come)
Significance of contributions/results	The contribution of this paper is not the generalizability of the analysis of one student's data, but the process in which the authors conducted the analysis, which included pseudo-ethnographic participation in the students' course, code snapshot review, and interviews. This was an interesting analysis that went deep into one student's actions in creating a project for her class, and the authors' attempt in creating/interpreting meaning from her actions.

:	The reviewer is still somewhat concerned about the case study only following one person's work on one specific project. Also, the reviewer would have liked to have seen more discussion and suggestions about insights that the authors had while taking their approach in analyzing Rebecca's data, and more information about future directions that this can take, especially for wider scale use (is this kind of analysis even scalable?).
Writing and expression:	**5**: (exemplary writing that enhances the quality of the paper) The reviewer would classify what the authors claim as "ethnography" as pseudo-ethnography. The paper is well written and easy to follow. The reviewer applauds the authors for taking the time to edit and iterate on the paper – it really shows in the quality of the writing.
Suggestions regarding the writing or other comments:	There were a few places where text was misaligned. Possibly due to Latex formatting issues. There were some orphan lines (e.g., "... it's much harder to answer questions such as:"). Next to Figure 2-1, there is a "the" floating to the right of the image. T". This also happens in "...from a previous semester's project. The figure that appears between Figure 2 and 3 should have its own Figure number and caption.

Review 2

Overall evaluation:	**4**: (Borderline, lean to accept)
Summary of paper:	The author argues that there are research questions that cannot be answered by mining code snapshot data, and backs up that assertion by showing how an approach involving subject interviews illuminated code-design issues that would not have been apparent by looking at snapshots of code alone.
Discussion of related work:	**4**: (covers key related work; its relationship to submission is described, but could be extended further)
Theoretical basis for the paper:	**3**: (there is a theory there, but its relevance to the research is vague)
Use of theory:	This section of the review form is not a good match for the paper. The author makes sufficient references to the literature on mining of code snapshots to make the point that there are questions the approach cannot answer. It is then shown how one such design-related question *can* be answered by supplementing code snapshots with more traditional ethnographically oriented methods. Thus, there isn't really a *theory* being used here, but the point is being made via demonstration.
Research methodology:	**3**: (research approach and methods well-suited for the research questions/hypotheses)
Exposition of research methods:	**4**: (data collected and analyzed; methods clear and thoroughly described)
Discussion of results and conclusions:	**3**: (plausible interpretation of findings)
Methodology and empirical basis:	The evaluation rubric once again fails to adequately capture this paper. The author's ethnographic exploration is rigorous and well informed, and it is nicely supplemented with information from code snapshots. In that sense, the research methodology is well suited to the research question being asked. (Namely, "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?") On the other hand, it's not surprising that there are research questions that cannot be adequately answered via code snapshots alone. (Or, that including code snapshots with other avenues of inquiry can yield valuable insights.)
Contribution and	

relevance to the international computing education research field:	**3**: (minor contribution or contribution is bound to a local context, perhaps with the promise of more to come)
Significance of contributions/results:	As mentioned above, the author does a nice job of pointing out how code snapshots can work in conjunction with other avenues of inquiry to answer research questions that cannot be answered via snapshots alone -- in particular, questions about student design processes. But the subject interviews alone would have revealed much of the story of Rebecca's project design, so it doesn't seem like supplementing the interviews with code snapshots added much value. The paper would have much greater impact if it found a way to reduce the time and expense of ethnographic approaches by combining them with the automation and scale that comes with snapshot mining.
Writing and expression:	**4**: (well written and expressed)
Suggestions regarding the writing or other comments:	I'm sympathetic to the argument that much is lost in the dash to "big data" analysis approaches of student work traces. I think the paper would be stronger if you highlighted ways in which code snapshots could improve more traditional research approaches, or in which the data-mining approaches could make it easier to do the ethnographic work. The figures are illegible.

Review 4

Overall evaluation:	**3**: (Borderline, lean to reject)
Summary of paper:	META-REVIEW: ICER meta-reviews are entered when appropriate to provide additional context for interpreting paper reviews, to capture program committee deliberations, and to direct authors' attention to the most salient points. Meta-reviews are not intended as an independent review, but rather account for all aspects of the program decision process. The overall evaluation score in this review reflects this final judgement. NOTE: Individual element scores on a meta-review are all given values of 1 and are meaningless placeholders due to limitations of EasyChair. Meta-reviewer comments are provided in the last field of the review form.
Discussion of related work:	**1**: (no discussion of related work)
Theoretical basis for the paper:	**1**: (no obvious theory being applied)
Use of theory:	.
Research methodology:	**1**: (research approach and methods inappropriate for research objectives)
Exposition of research methods:	**1**: (no empirical data collected)
Discussion of results and conclusions:	**1**: (unjustifiable interpretation of findings)
Methodology and empirical basis:	.
Contribution and relevance to the international computing education research field:	**1**: (contributes little or nothing to computing education research)

Significance of contributions/results:	.
Writing and expression:	**1**: (extremely poorly written; hard to understand)
Suggestions regarding the writing or other comments:	META-REVIEW: We value the research direction of this paper. Examining some of the underlying rationale for programming decisions by novices through an in-depth analysis of program snapshots and clinical interviews appears to be a fruitful path to take, and one that will add to the existing discourse in CS Ed. There was support from all of the reviewers for acceptance, though it was borderline in all cases. As a result of its resting on the border, the paper generated considerable discussion among meta-reviewers. We had a few key concerns that tipped us into rejecting this paper, concerns that we encourage you to address in the future. Your research question is about method, with the suggestion that including the interview data will, in general, be illuminating. As a result, the paper undermines itself, because there is no research question in CS Ed concerning (say) novice programming, or programmer cognition that is informed by your methodological choice and its associated analysis. This thus makes your choice of the case that is discussed in depth to appear ad-hoc, i.e. we did not know “why this case, rather than any other?”. In addition, there is no reference to prior research related to discourse about relevant matters in CS Ed. This, along with such things as explaining big-O, lead the meta-reviewers to believe that this paper was not targeted to the ICER audience. In sum, we would like to encourage you to continue this important line of work. But rather than leaping to a methodological argument, we think that you can directly inform a research question of interest to the ICER community. We think that you would benefit by participating in the ICER Works-in-Progress workshop, where the other participants in the workshop can help provide links to the existing discourse. By altering your argument to address the concerns above, we think you could make important contributions to the CSER literature in the future.

Review 1

Overall evaluation:	**4**: (Borderline, lean to accept)
Summary of paper:	This paper dives deep into an ethnography of a single student working on a single programming, using log data and interviews. The richness of the story is fantastic and reveals the kind of story that usually is known only by a student herself and/or her TAs.
Discussion of related work:	**4**: (covers key related work; its relationship to submission is described, but could be extended further)
Theoretical basis for the paper:	**1**: (no obvious theory being applied)
Use of theory:	This is an ethnographic study with grounded observations. There's no need for much theory here.
Research methodology:	**3**: (research approach and methods well-suited for the research questions/hypotheses)
Exposition of research methods:	**4**: (data collected and analyzed; methods clear and thoroughly described)
Discussion of results and conclusions:	**5**: (incisive interpretation of findings and limitations)
Methodology and empirical basis:	Everything is gone into such incredible detail, utilizing the depth of information from the log data along with interviews provides awesome insight into what the student "Rebecca" was thinking when she worked on the assignment.
Contribution and relevance to the	

international computing education research field:	**3**: (minor contribution or contribution is bound to a local context, perhaps with the promise of more to come)
Significance of contributions/results:	I like the depth of the understanding of how the student solved her assignment, but OMG, the amount of effort required to get this kind of story cannot scale. Even just getting a classroom of students observed and analyzed might require an army of ethnographically-trained graduate students! I agree with the author's frustration with prior research because of its focus on the quantitative aspects of their data despite having the potential to discover rich experiences. But, there's got to be an incremental step here. Can you tell part of this story through automated mining of the data stream or does it require a human touch? This would make a great PhD dissertation if you've got any students available to work on it. :)
Writing and expression:	**5**: (exemplary writing that enhances the quality of the paper)
Suggestions regarding the writing or other comments:	Everything was super clear except for the screenshots and code shots. Those are way way too small and are impossible to read.

Sample review process

The template below is provided to help guide reviewers and authors during the reviewing process. You may also find it helpful to review the tips for authors and reviewers.

NOTE: Actual reviews should be completed using the EasyChair system. Reviewers will receive a link via email when papers have been assigned.

Review Template

- 1. Overall evaluation
    - 6: I strongly support accepting this paper
    - 5: I would argue for accepting this paper
    - 4: Borderline, lean to accept
    - 3: Borderline, lean to reject
    - 2: I would argue to reject this paper
    - 1: I strongly recommend rejecting this paper
- 2. Reviewer's confidence
    - 3: (high)
    - 2: (medium)
    - 1: (low)
- 3. Summary of Paper (*)

  Provide a 1-2 sentence summary of the work in your own words. We use this to verify that reviews are entered for the correct submission.
- 4. Discussion of related work
    - 5: all relevant work discussed and cited, and

relationship to submission clearly and thoroughly described
    - 4: covers key related work; its relationship to submission is described, but could be extended further
    - 3: some references missing, or relationship to submission not clearly described
    - 2: several important or key reference(s) missing, and relationship of references to submission not apparent
    - 1: no discussion of related work
- 5a. Theoretical basis for the paper
    - 5: clear and strong theoretical basis, well documented with citations and clearly applied in the research
    - 4: theoretical basis obvious, with some citations and argument for how it is applied in the research
    - 3: there is a theory there, but its relevance to the research is vague
    - 2: maybe there's a theory there, but it is vague and has no clear relevance to the research
    - 1: no obvious theory being applied
- 5b. Use of Theory (*)
  Discuss the appropriateness and quality of the theoretical framework for addressing the explored research topic.
- 6a. Research methodology
    - 3: research approach and methods well-suited for the research questions/hypotheses
    - 2: questionable choice of research approach and methods
    - 1: research approach and methods inappropriate for research objectives
- 6b. Exposition of research methods

4: data collected and analyzed; methods clear and thoroughly described.
3: data collected and analyzed, but some aspects unclearly described.

2: data collected and analyzed, but unclear or inadequate description.

1: no empirical data collected

- 6c. Discussion of results and conclusions
    - 5: incisive interpretation of findings and limitations
    - 4: good interpretation of findings; limitations considered
    - 3: plausible interpretation of findings
    - 2: questionable interpretation of findings
    - 1: unjustifiable interpretation of findings
- 6d. Methodology and Empirical Basis (*)
  Discuss the appropriateness and quality of the chosen methodology. Evaluate the way authors have applied the research, interpreted their findings and drawn conclusions. Comment on whether the findings have been examined in the context of related work and the limitations of the research.
- 7a. Contribution and its relevance to the international computing education research field
    - 5: a major and significant contribution to the field that explicitly presents results in a manner directly applicable to international research contexts
    - 4: a clear contribution to the field
    - 3: minor contribution or contribution is bound to a local context, perhaps with the promise of more to come
    - 2: no obvious contribution, but the promise of future value
    - 1: contributes little or nothing to computing

education research

- 7b. Significance of Contributions/Results (*)
  Make a case for the importance of this finding for our community (or indicate your views if you believe it not so important). We hopefully have a number of high-quality papers, and your input on what is important for our community matters.
- 8a. Writing and expression
    - 5: exemplary writing that enhances the quality of the paper
    - 4: well written and expressed
    - 3: not well written, but could probably be made acceptable
    - 2: very poorly written; unlikely that it can be improved enough
    - 1: extremely poorly written; hard to understand
- 8b. Suggestions regarding the writing or other comments
  If you would like to draw particular aspects of the writing to the authors' attention and/or make any further recommendations, please do so here. If you have any further references to recommend, please add them here.