Final Paper

Andrew Carter and Beryl Egerter

April 29, 2013

1 Abstract

We performed a qualitative analysis on a small set of interviews asking questions about code evaluation and questions about debugging. The code samples included programming concepts or syntax that was unfamiliar to the students. We saw different strategies such as following order of execution and reading code from the top of the file down. Students' strategies often changed from question to question, and some strategies appeared more suited to different questions. We believe evaluating these strategies is important because a programmer's choice of strategy can affect programmers' success or failure in maintaining, changing, or interacting with code written by others.

2 Introduction

Programmers often have to look at code they are unfamiliar with for a variety of reasons, whether they are students or professionals. In industry, programmers often work with a large code base and fixing a bug or finding code that does something in particular may not be easy or entirely understandable. In education, computer science students sometimes start programming assignments with a code base that they are expected to alter or add to, but do not necessarily have the background to understand some of it, and may be extremely confused if they cannot figure out where to change or add code. Programmers with any amount of experience need effective strategies to understand existing code depending on the code and their objective.

In our qualitative study we asked two questions where each student needed to do code evaluation and two where each student needed to do code debugging. In each of the questions we embedded a concept we did not expect our participants to be familiar with or code that we thought would be hard to understand. These included overly complicated functions, an unfamiliar programming language, or the functional programming concept of passing a function as an argument. Including questions about code evaluation and debugging was important to us since the research we have read has looked at code evaluation on its own or debugging on its own but not together with the same participant. We wanted to see how strategies changed depending on the situation.

We found that a participant's choice of strategy could help them succeed in solving the problem by being able to avoid complicated code or understand unfamiliar concepts. One participant in particular chose two different strategies for an evaluation problem and a debugging problem. The strategies helped him understand an unfamiliar concept in the evaluation problem and avoid complicated functions in the debugging problem.

The primary contribution of this work is a qualitative analysis of how different strategies helped a student overcome unknown concepts both in code evaluation and in debugging. As a small qualitative study, there are many questions that could be focused on next. In particular, we are interested in other strategies that could be used for each problem, their effectiveness, and what leads students to choose which strategy. We could also follow up on some factors that may have affected some of the decisions students made about strategy in this study such as length of code or access to resources such as Google.

3 Previous Research

A number of previously written papers are relevant to our work because they cover various studies about code debugging, evaluation, and the strategies used in either case. Our work builds on these papers by comparing debugging and evaluation strategies directly instead of treating them as separate topics. These papers also highlight some weaknesses of our study.

Mosemann and Wiedenbeck investigated code comprehension by novices with respect to navigation method (?, ?). They gave students two different programs to evaluate and a navigation method to use: sequential, data flow, or control flow. The students were evaluated by asking a series of yes/no questions about the program to assess their understanding of the code. Mosemann and Wiedenbeck found no interaction between program and navigation method. However, they suggested that with a more diverse set of programs and navigation methods there may be some interaction, but the study was not broad enough to discover them.

Fitzgerald, et al. investigated debugging methods among novices (?, ?). They confirmed that students often used a tracing strategy on a variety of different debugging problems. Their results show that the bugs in their study were relatively easy to fix, and that determining how programmers discover where a bug is located is more useful. Our study found consistent results, and so we decided to focus on how the student discovers the location of a bug rather than how the student fixes it. However, Fitzgerald, et al. note an increased reliance on online resources for debugging, something that we did not provide. Some student strategies exhibited in their study that we could not possibly replicate in our study are using println statements and tracing using a debugger.

Rajlich and Wilde talk about concepts both as code structures, such as loops or passing a function, and domain knowledge, what the programmer wants the code to do (?, ?). From a series of case studies, they argued that experts do not understand the entirety of the code they are debugging, but simply map their domain knowledge onto code concepts to narrow down the code areas they must look at. This mapping of domain knowledge to code appears to occur in interview we analyzed as the student connected functions within the code to actions taken in a connect four game. These mappings allowed the student to focus only on a single function instead of understanding the entire program.

Through qualitative analysis of interviews, Mayrhauser and Vans noticed that program comprehension often involves multiple strategies of thinking about the code (?, ?). In addition to explaining a number of strategies, how often they were used, and how they believe they work together, they performed a study of how often each of the strategies was used by a programmer looking for a section of code within a 40000 line program within two hours. While most of our interviews were far too short for our students to have the time or need to switch between different strategies, we did occasionally see strategies changing, and this paper provides information about what might occur if our problems and interviews had been longer.

In previous research, some studies covered code evaluation and debugging strategies in a narrow range of programs, and concluded that the individual problem did not have an effect on strategy. The researchers also discussed some weaknesses of our study. For instance, our students only had access to the code on paper, limiting more interactive debugging and web searches. Additionally, our code samples were relatively short compared to what might be found in industry; longer code samples might have caused students to display different behavior.

4 Data Collection Methods

In this study we conducted four interviews where two interviewers interacted with a single student. Four students were interviewed in total. For each interview, a student was given four different problems. Each interview took between 40 minutes and 1 hour to complete. After the students felt comfortable with the answer to their problem, the interviewers asked the student questions about the problem to gather more information on how they solved the problem, and how thoroughly they understood various sections of the provided code. For each interview, the student's actions and voice were recorded, and then transcribed and analyzed afterwards.

Though the participants all had taken basic programming courses, there were a variety of backgrounds. None of the students had completed the Programming Languages class, where topics such as functional programming are covered in depth. However, all of the students had been exposed to Python syntax and list, as well as the map function, in Python. In this paper we focused on one participant who also had tutoring experience for a course in program development and data structures.

The four problems including two evaluation problems and two debugging problems. For the evaluation problems, the student was instructed to evaluate the Python code given to them. For the debugging problems, the student was informed of the unwanted (buggy) behavior, although for the first debugging problem it was asked whether the behavior occurred (which it did), and for the second the student was told that the behavior occurred and was asked to find a solution. The students were encouraged to talk out loud and informed that they could write on the paper. Finally, the interviewers answered questions that could be reasonably inferred from a Google search, such as, "is __init__ a constructor?" However, questions specific to the code were not answered, and when students felt confidant in their solution we accepted it independent of correctness. If it became clear that a student was not making progress, hints were supplied, however this did not happen in any of the interviews we analyzed.

5 Data Analysis Methods

To analyze the data we made a content log for each interview. The content log was a written record of when the student started answering each question, and timestamps of significant events so that we could easily find interesting portions of the video. We also kept records of what we thought was interesting about the interview.

When planning the interview questions, we had already thought about the strategies students could use to solve the questions. For instance we laid out the functions in Question 1 such that we could differentiate between the student evaluating in execution order, following the numbers of the function names, starting with simple functions, and going from top down or bottom up. We noticed students switching strategies between problems in our interviews, which we then decided to analyze.

We transcribed several sections of the interview, which was too much data to present in this paper. The sections of transcripts presented in this paper best illustrate our findings. In particular, the student we present most clearly shows his confusion, and resolves it in the most clear manner. This allows us to look at the effects of a given strategy on the problem the student used it on.

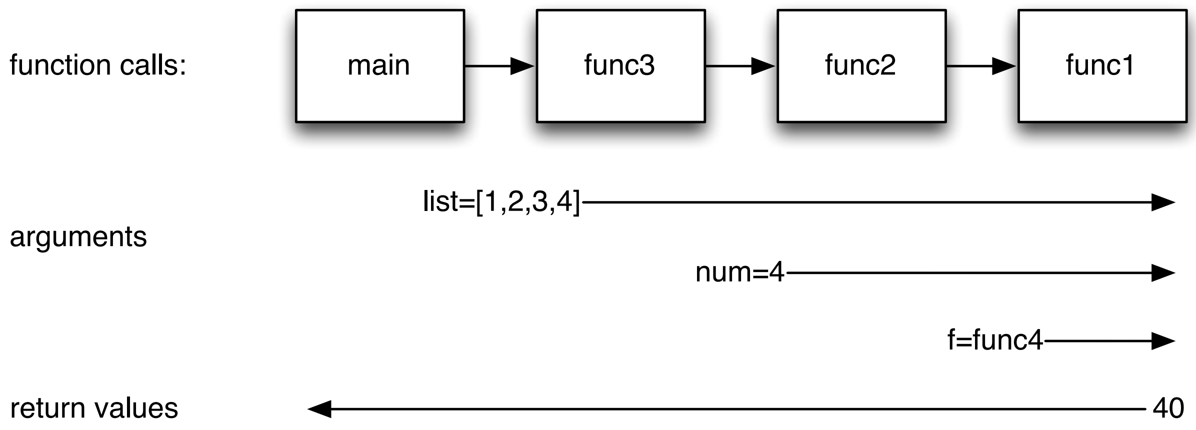

Figure 1: Flow of Arguments and Return Values in Question 1

6 Problem Descriptions

6.1 Question 1

6.1.1 Summary

Interview Question 1 asks the interviewee to evaluate a short program. The program passes a function as an argument, a concept the students might not be familiar with, but is a key concept of functional programming. The program is 18 lines long and takes up less than a page when printed. The problem can be found in the Appendix as Interview Question 1.

6.1.2 Solution Description

One (popular) way to determine the solution is to realize the only piece of code that gets executed at the top level is the call to main on line 18. Then following the execution of the program, main prints out the result of `func3([1,2,3,4])`. The functions func3 and func2 just add arguments, so the original call to main is equivalent to calling `func1([1,2,3,4],4,func4)`. Note that func4 is being passed, thus when the parameter "f" is called in func1 on line 9 it is really calling the function func4. Continuing the trace in func1, acc starts at 0 then adds `func4(i,4)` for each "i" in the list. Since func4 is just multiplication, this results in summing 4 times the elements of the list, which comes out to 40. From there the program just repeatedly returns the 40, until it returns to main when we print 40. This process can be seen in Figure 1.

6.2 Q4

6.2.1 Summary

Interview Question 4 (Q4) asks the interviewee to locate and fix a bug in a program that they are told plays connect four. The bug is that when pieces are dropped into the board, the program lets pieces be dropped in even if the column is full. The program is 70 lines long and covers two pages when printed.

6.2.2 Copy of Problem

The problem can be found in the Appendix attached below as Interview Question 4.

6.2.3 Solution Description

The following solution uses execution order to come to the correct conclusion. Other methods are possible.

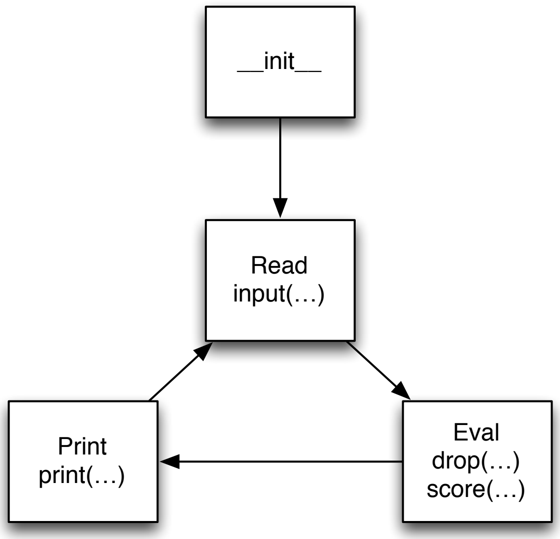

Figure 2: Read-Eval-Print Loop and the important functions used in each part of the loop in Q4

To find the correct solution we start at line 49 where the code starts executing. Some variables are set up to be used later. Here a Board is initialized. It calls the __init__ function (line 3) which creates a list of lists stored as the self.board variable. The initial lists are empty. It also stores the height and width of the board in class variables.

Here the program starts a Read-Eval-Print loop. A Read-Eval-Print loop is a structure that takes in input, evaluates how that changes the stored data, and then prints a representation of the data. Figure 2 shows the structure of this loop and the previous initialization as well as the function calls that correspond to each part of the loop.

At line 55, the program asks for input, the read part of the loop. Then, it calls board.drop(player,c) which adds a piece to the column that the user has specified through the input. In the drop function (line 8), we can see that if the column selected is part of the board, the piece is appended to that column in the list of lists. This is analogous to dropping a piece into the board, meaning this is a logical place where the bug could be fixed.

Since we check that the column is valid, we can also check that the column is not full by changing the drop function to include this line between lines 9 and 10: if len(self.board[column]) < self.height:. This code will make sure that pieces will not be dropped in full columns.

Other possible solutions that cause the drop function to not be called if the column is full would also be acceptable.

7 Analysis

7.1 Overview of Claims & Introduction to Analysis

We will show that a student used different strategies to decide which order to look at functions between two problems. For an evaluation problem, the student evaluated the code in the order it would have been executed. For a debugging problem, the student started looking at the code at the top of the paper and continued down until reaching a likely candidate for the location of the bug. For this student, each strategy appear to help him understand unfamiliar constructs, which then allows him to solve the problem. We include two partial transcripts from a debugging and an evaluation problem to back these claims. This student has taken a couple of computer science classes at Harvey Mudd College, including Principles and Practices of Computer Science, Data Structures and Program Development, and has tutored extensively for the Data Structures and Program Development class.

7.2 Q1: Evaluation

The analysis begins from when the student is handed the first problem,

```
def main():
    print(func3([1,2,3,4]))
...
main()
```

The student begins by identifying the call to main.

T1 Alright so, its got a main, so thats gonna start. [...]

The student starts at the main function, it is the very first thing he identifies. He continues evaluating the code in execution order. This student's evaluating strategy clearly is identify where execution starts and then continue in execution order.

The student continues in execution order until he hits func2,

```
def func2(list, num):
    return func1(list, num, func4)
```

The student tries to evaluate func4 instead of passing it to func1.

T2 Function 2 returns function 1 with the same two arguments already passed to it, and function 4, the result of -

He immediately jumps to "the result of," even there are no parenthesis that are required for function calls. Clearly he is unfamiliar with passing functions as arguments in Python.

The student tries to resolve his confusion by looking at func4

```
def func4(a, b):
    return a * b
```

He tries to find ways to call func4, without any arguments,

T3 Which doesn't have any implied arguments, thats interesting. Umm, thats odd [pause]

In this case he is working off of the hypothesis that if you have default arguments to a function, then you can call the function parenthesis. However, the student notes that there are no default arguments (he refers to them as implied arguments). Note, that even though the program does not call func4 from func2, from the student's perspective, because he believes func4 needs to be called, func4 was next in execution order.

He continues onto func1, even though he is confused about how func4 works in func2.

```
def func1(list, num, f):
    acc = 0
    for i in list:
        acc += f(i, num)
    return acc
```

He notes that the `f` in func1 is the same as func4 in func2.

T4 lets see, so its calling functi-,ooh, it calling function 1.
T5 There we go. Uhh, yes, so its calling function with a list a number, the array and 4,
T6 and the function 4 as sort of a multiplier.
T7 A function to apply.

The student is able to figure that func4 is being passed as an argument. He note that function 4 is being used as a multiplier, and the succinctly describes it as a function to apply. He clearly now understands that functions can be passed as arguments, and the semantics of doing so. The student drew this conclusion by continuing with execution order, and matching up the f in function 1 with the func4 in function 2. It appears that following execution order helped him figure this out, he might not have been able to make that connection if he had not looked at function 1 while being confused about the need for func4. However since he was already primed to call a function he was able to identify that the argument f got called.

7.3 Q4: Debugging

In this problem, the student debugged by reading from the top of the file down, but stopping upon encountering a good location to insert a solution. In the process, he was confused about a data member in the code, but was able to recover by continuing with his chosen strategy.

The student began the problem by looking at the top of the given code:

```
#!/bin/env python3

class Board(object):
  def __init__(self, width=7, height=6):
    self.board = [[] for i in range(width)]
    self.width = 7
    self.height= 6
```

He relates the board class to the game described in the problem statement:

T1 Alright so we have a board class which is gonna presumably represent the connect 4 board [...]

He is able to relate the code concept of a class to the domain concept of a game of connect four.

After talking about the other data members, the student mentions multiple possibilities for the data structure of this data member:

```
    self.board = [[] for i in range(width)]
```

He said:

T2 it's going to create an array of arrays or a list of lists depending on what you call it in python [...]
T3 i don't know if, if lists in python are dynamically allocated or not [...]
T4 so i wonder what this is doing. I guess it would be creating - an array with at least six - er, seven secondary arrays in it [...]

This demonstrates a lack of experience with Python, which confuses him and does not allow him to map the data member to the physical representation of a connect four board.

After he finished describing the constructor, he moved down the code to the next function, drop:

```
  def drop(self, player, column):
    if column < len(self.board):
      self.board[column].append(player)
      return True
    return False
```

Based on the code in this function, the student figured out what the board data structure looks like:

T5 so drop, i guess that is where you drop a piece in - if column less than self.width um self.board.append player

T6 oh ok so its doing [starts drawing on paper] like a - array of arrays essentially and then it just pops on a color [...] as they happen [...]

The strategy he is using is allowing him to keep gaining clarity on what the code is doing.

Finally, he identifies this function as a point where the bug could be fixed, and does not go through any other code in the file. He is confident hat he understands the solution to the problem even though he has only looked at a small portion of the provided code.

7.4 Relations Between Evaluation and Debugging

The student used two different strategies in the debugging and evaluation question which helped him move past an unfamiliar concept each time. Despite using different strategies, each strategy was clearly an efficient method to solve each problem. For example, in other interviews, if the student did not move directly from function 2 to function 1 in Q1, it took much longer to connect function 4 passed as an argument as the applied function in function 1. In Q4, some other strategies we have seen have led the student to get stuck in other locations as they try to relate the code to the physical model. Thus, being able to solve a problem efficiently requires the student to choose an effective strategy after only being given a little information about the problem.

7.5 Summary

We have shown that a student used different strategies to decide which order to look at functions between two problems. For the evaluation problem, the student evaluated the code in the order it would have been executed. In the process, he was initially stumped by the code passing a function as an argument, but realized what was happening by continuing to follow the path of execution. For the debugging problem, the student started looking at the code at the top of the paper and continued down until reaching a likely candidate for the location of the bug. As he debugged, he was initially unsure about the data structure used as the board data member, but as he continued down, he was able to find code that supported one of the possibilities he had enumerated earlier. For this student, each strategy appeared to help him understand unfamiliar constructs, allowing him to solve the problems.

8 Conclusions

9 Future Work

A Full Code Samples

A.1 Evaluation Interview Question

Code given to participants did not include line numbers.
Code given to ACBE1, ACBE2, and ACBE3 was in color. Code given to ACBE4 was in grayscale.
Verbal prompt was given before handing code to participant:
"For this question we would like you to familiarize yourself with some Python code. Please explain to us what you think this code does."

```
def func2(list, num):
  return func1(list, num, func4)

def func4(a, b):
  return a * b

def func1(list, num, f):
  acc = 0
  for i in list:
      acc += f(i, num)
  return acc

def main():
  print(func3([1,2,3,4]))

def func3(list):
  return func2(list, 4)

main()
```

A.2 Debugging Interview Question

Code given to participant did not contain line numbers.
Code given to ACBE1, ACBE2, and ACBE3 was in color. Code given to ACBE4 was in grayscale. Code given to ACBE1 and ACBE2 had double equals signs that were joined together. Code given to ACBE3 and ACBE4 had spaces between the equals signs.
Verbal prompt was given before handing code to participant:
"For this question we would like to have you look at some code in Python. This is the scenario: You acquired a connect 4 program from a friend. However, the friend has warned you that you can put too many pieces in a column. Determine a possible fix for this bug so that you can enjoy your connect 4 program."

```python
#!/bin/env python3

class Board(object):
  def __init__(self, width=7, height=6):
    self.board = [[] for i in range(width)]
    self.width = 7
    self.height= 6

  def drop(self, player, column):
    if column < len(self.board):
      self.board[column].append(player)
      return True
    return False

  def __str__(self):
    result = ""
    for r in reversed(range(self.height)):
      result += "|"
      for c in range(self.width):
        if r < len(self.board[c]):
          result += self.board[c][r]
        else:
          result += " "
        result += "|"
      result += "\n"
    result += "-" * (2 * self.width + 1)
    return result

  def full(self):
    return all(len(col) >= self.height for col in self.board)

  def score(self, player):
    for c in range(self.height):
      for r in range(len(self.board[c])):
        p = self.board[c][r]
        for dc,dr in ((0,1),(1,0),(1,1),(1,-1)):
          for i in range(1,4):
            nc = c + i*dc
            nr = c + i*dr
            if nc < 0 or self.width <= nc:
              break
            if nr < 0 or len(self.board[nc]) <= nr:
              break
```

```
                if self.board[nc][nr] != p:
                    break
            else:
                return 1 if p == player else -1
        return 0

other = {'X' : 'O', 'O' : 'X'}
player = 'X'
board = Board()

while True:
    try:
        c = int(input("%s > " % player))
    except TypeError:
        continue
    if not board.drop(player,c):
        continue
    print(board)
    if board.score(player):
        print("Player %s Wins!!!" % player)
    elif board.full():
        print("Tie")
    else:
        player = other[player]
        continue
    board = Board()
    player = 'X'
    print(board)
```

B Full Transcripts of ACBE3, Q1 and Q4

S indicates the student and I the interviewer for the problem (In Q4, I is Andrew Carter).

B.1 Q1: Evaluation Question

[1:55 Begin]

S	[1:55] :	Alright umm, so, its got a main, so thats gonna start ummm, its going to print whatever the result of
S	[2:00] :	function 3 on 1, 2, 3, 4, some array. So lets see,
S	[2:05] :	Function 3 takes a list and returns whatever function 2 does called with list
S	[2:10] :	and some argument 4. Function 2 returns function 1
S	[2:15] :	with the same two arguments already passed to it, and
S	[2:20] :	function 4 the result of, which doesn't have any
S	[2:25] :	implied arguments, thats interesting. Umm,
S	[2:30] :	[pause]
S	[2:35] :	thats odd, lets see, so its calling functi-, ooh,
S	[2:40] :	its calling function 1. There we go. Uhh, yes, so its calling function 1 with a list a number, the array
S	[2:45] :	and 4, and the function 4 as sort of a multiplier. A function to apply.
S	[2:50] :	Alright, so function 1 is doing the actual work here. Umm, see, it starts with some accumulator 0,
S	[2:55] :	iterates across the uh items in the list, list,
S	[3:00] :	and plus equals that function 4 applied
S	[3:05] :	to i being the item from the list and that number 4 that was included in function 3.
S	[3:10] :	So its going to essentially sum the list multiplied by 4,
S	[3:15] :	it would appear, and print that sum out.
S	[3:20] :	Yeah, its gonna take, each element 1 2 3 4 and multiply it by 4
S	[3:25] :	add that to 0 and then return the accumulator
S	[3:30] :	back up the steps. So function 1 2 3 yeah.

[3:35 End]

B.2 Q4: Debugging Question

[18:55 Begin]

S	[18:55] :	Alright so we have a board class which is gonna presumably represent the connect 4 board
S	[19:00] :	- um - the constructor automatically sets the width to seven
S	[19:05] :	oh in this case it forces constraints um
S	[19:10] :	like arguments essentially - and width is always going to be equal to seven, the height equal to six
S	[19:15] :	um - and its going to create an array of arrays
S	[19:20] :	or a list of lists depending on what you call it in python
S	[19:25] :	- um - then it appears to not force
S	[19:30] :	the height to be six. For i in range width -
S	[19:35] :	yeah that could be
S	[19:40] :	cause of problems - um - i don't know if,
S	[19:45] :	if lists in python are dynamically allocated or not. I don't think -
S	[19:50] :	yeah i think they are, you can keep adding to them can't you -
I		Yes.
S	[19:55] :	Yeah - so i wonder what this is doing.
S	[20:00] :	I guess it would be creating ... an array with at least six - er,
S	[20:05] :	seven secondary arrays in it. Um -
S	[20:10] :	and you don't necessarily [gestures] specify the height - i guess
S	[20:15] :	that doesn't matter you just need to check along the way - um - alright so drop
S	[20:20] :	I guess that is where you drop a piece in. If column less than self dot width
S	[20:25] :	um self dot board dot append
S	[20:30] :	player. Oh ok so its doing
S	[20:35] :	[starts drawing on paper] like a - array of arrays
S	[20:40] :	essentially and then it just pops
S	[20:45] :	on a color - i don't know are they black and red? I think they are.
S	[20:50] :	so sorta pops on a color as they happen and that way -
S	[20:55] :	oh i guess no append is going on the end isn't it. Um-
S	[21:00] :	so this is where you would need to do the check [points at drop function with pen]
S	[21:05] :	if column less than self dot width then append it
S	[21:10] :	- um -
S	[21:15] :	You could do - you could add to that if statement? Let's see
S	[21:20] :	and [writing on paper] um
S	[21:25] :	column dot size, is that a thing?
S	[21:30] :	Oh no its going to be board at column is what it's going to be. Board
S	[21:35] :	at column. Some sort of size operator -
I	[21:40] :	It's len
S		Ok, len. Um -
S	[21:45] :	less than. I have -[inaudible]
S	[21:50] :	and board
S	[21:55] :	column dot len
S	[22:00] :	less than - height is six so it's gotta be
S	[22:05] :	Oh we can just do less than self dot height.
S	[22:10] :	-gotta keep good encapsulation. Um - yeah.
S	[22:15] :	So this check would go right here [points at if in drop function] and that would keep you
S	[22:20] :	from - otherwise it would return false
S	[22:25] :	and not allow you to drop if you exceeded the height . Um - bounds
S	[22:30] :	That would work.

[22:30 End]