Final Paper

Andrew Carter and Beryl Egerter

April 22, 2013

1 Abstract

2 introduction

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. Mosemann and Wiedenbeck found no interaction between program and navigation method. However, they suggested that with more diverse problems and 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 similar results, and so we decided to focus on discovering the location of a bug rather than fixing 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 occurred in interview ACBE3 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 interviews, Mayrhauser and Vans noticed that often program comprehension 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 and provided results 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 subjects 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

5 Data Analysis Methods

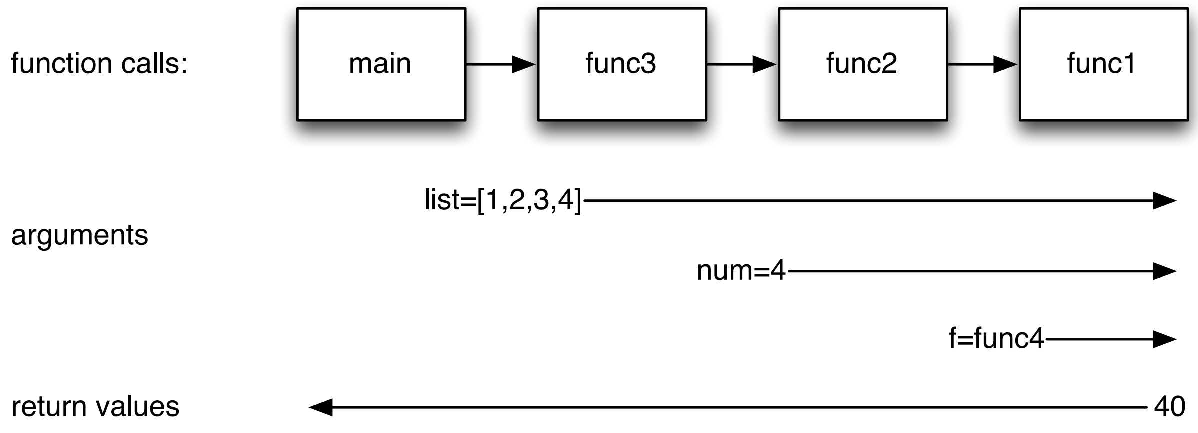

Figure 1: Flow of Arguments and Return Values in Q1

6 Problem Descriptions

6.1 Q1

6.1.1 Summary

Interview Question 1 (Q1) asks the interviewee to evaluate a short program. The program passes a function as an argument, a concept they 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.

6.1.2 Copy of Problem

The problem can be found in the Appendix attached below as Interview Question 1.

6.1.3 Solution Description

One (popular) way to determine the solution is to realize the only piece of code that get executed at the top level is the call to main on 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 that call is equivalent to calling `func1([1,2,3,4],4,func4)`. Note that func4 is being passed, thus when "f" is called in func1 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. When we unwind we just hit return statements until main, when we print 40. This process can be seen in Figure 1.

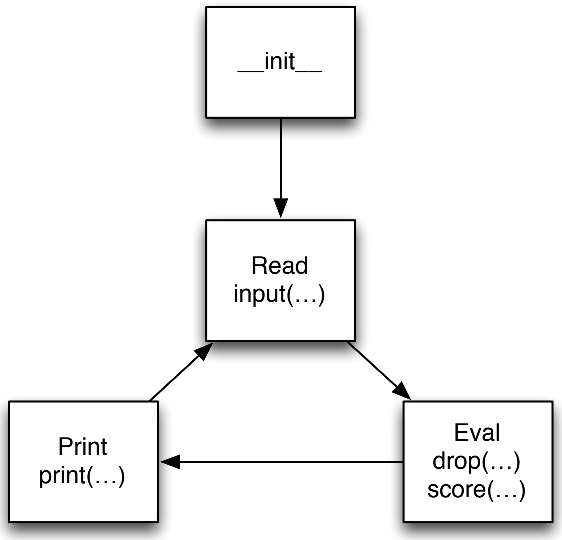

Figure 2: Read-Eval-Print Loop and the important functions used in each part of the loop in Q4

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.

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 create 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 itself, and then prints a representation of itself. In Figure 2 we can see the structure of this loop and the previous initialization as well as the important function that correspond to each part of the loop.

At line 55 we see that the program asks for input, the read part of the loop. Then, it calls board.drop(player,c). 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 where the bug needs to be fixed by.

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:, to make sure that pieces will not be dropped.

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. The student whom we have transcribed 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.

8 Q1

He begins by identifying the call to main.

"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,

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

"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

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

"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.

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

lets see, so its calling functi-,ooh, it calling function 1. There we go. Uhh, yes, so its calling function with a list a and the function 4 as sort of a multiplier. 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.

8.1 Play-by-Play

8.1.1 Q1

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

The student is going to start with the main function, which gets called at the bottom of the script.

Omitted is the transcript of the student continuing to follow the execution until the time he reaches function 2.

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

The student is evaluating function 2. He notes that there is function 4, and he tries to determine what the resulting value should be.

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

The student is looking for implied arguments to function 4.

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

The student continues to function 1 and notices how the argument is being used. He then deduces the correct semantics for the function call in function 2.

8.1.2 Q4

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

The student relates the board class to the game described in the problem statement.

2 it's going to create an array of arrays or a list of lists depending on what you call it in python [...]
3 i don't know if, if lists in python are dynamically allocated or not [...]
4 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 [...]

The student mentions multiple possibilities for the data structure of the board member.

5 so drop, i guess that is where you drop a piece in - if column less than self.width um self.board.append player
6 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 student figures out what the board data structure looks like and how it changes.

7 so this is where you would need to do the check [points at drop function with pen]

The student says the drop function is where the bug could be fixed.

8.2 Interpretation

All line numbers in the following paragraphs refer to line numbers in the transcript of the corresponding problem.

The student is evaluating in execution order in Q1. The student is unfamiliar with the semantics of passing a function as an argument in Python. Initially, as demonstrated on line 3 of Q1, he tries to call the function which led to confusion since there were no arguments. On line 4 of Q1, he looks for implied arguments, further evidence that he is trying to call this function. However, by continuing in execution order to function 1,he is able to notice how the argument he had tried to call earlier is now being used as a function to apply, as stated in line 9 of Q1.

The student is debugging by reading from top down, stopping upon encountering a likely solution, in Q4. Upon reaching the creation of the board data member, the student's lack of experience with Python causes him to be confused about how the physical representation of a connect four board maps onto this data member. As he continues reading top down, the Python list functions used in the drop function help him determine which of the possibilities (previously enumerated on line 3 of Q4) were true for the board data member. This helped him map the physical representation onto the data member. This knowledge helps him fix the bug which had been given to him in relation to the physical connect four board.

The student used two different strategies in the debugging and evaluation question. These strategies helped him move past an unfamiliar concept each time. 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.

8.3 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.

9 Conclusions

10 Future Work

A Full Code Samples

A.1 Interview Question 1

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."

```python
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 Interview Question 4

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

B.1 Q1

[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

[18:54 Begin] S [18:54]: Alright so we have a board class which is gonna presumably represent the connect 4 board - um - S [19:00]: the constructor automatically sets the width to seven oh in this case it forces constraints um 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 or a list of lists depending on what you call it in python - um - S [19:28]: then it appears to not force the height to be six S [19:32]: for i in range width - yeah that could be cause of problems - um S [19:42]: i don't know if, if lists in python are dynamically allocated or not S [19:49]: i don't think yeah i think they are, you can keep adding to them can't you - I [19:54]: Yes S [19:55]: Yeah - 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 S [20:09]: um ... and you don't necessarily [gestures] specify the height ... i guess 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.width um self.board.append player S [20:32]: oh ok so its doing [starts drawing on paper] like a ... array of arrays essentially and then it just pops on a color - i don't know are they black and red? I think they are... so sorta pops on a color as they happen and that way ... oh i guess no append is going on the end isn't it ...um S [21:01]: so this is where you would need to do the check [points at drop function with pen] S [21:04]: if column less than self.width then append it um S [21:15]: You could do - you could add to that if statement? let's see ... and [writing on paper] um S [21:25]: column.size, is that a thing? oh no its going to be board at column is what it's going to be board at column S [21:38]: some sort of size operator I: It's len S: Ok, len S: [21:45]: um ... less than I have ...[mumbles] ... and board column . len less than ... height is six so it's gotta be - oh we can just do less than self.height S [22:10]: ...gotta keep good encapsulation...um ... yeah so this check would go right here [points at if in drop function] S [22:20]: and that would keep you from ... otherwise it would return false and not allow you to drop if you exceeded the height ... um ... bounds ... that would work. [22:30 End]