HARVEY MUDD COLLEGE

A Qualitative Analysis of Code Comprehension and Debugging Strategies

Andrew Carter and Beryl Egerter

Abstract

We performed a qualitative analysis on a study consisting of four interviews of college students with varying amounts of experience beyond basic programming. We asked the students to perform code comprehension or debugging on four code samples with varying levels of difficulty. We are interested in the different strategies the students used for exploring unfamiliar code and have qualitatively analyzed how well the strategies worked on our code samples.

Strategies we saw included students delving into the details of a function or obtaining a high level summary of a function from the name. Some students even changed strategies depending on the given task. We believe evaluating these strategies is important because the strategies can affect the success or failure of programmers trying to maintain, change, or interact with code written by others. These situations often occur in industry. Our study shows that programmers may change strategy based on problem type and complexity, of which future research may need to be careful about.

Introduction

While many studies have looked at code comprehension strategies[1] and many studies have looked at debugging methodology[2], we were interested in how a student's strategy changes between these tasks. We performed a study on code comprehension involving evaluation or debugging on different code samples. Evaluation is figuring out what a piece of code will output, while debugging is looking through a piece of code to identify the cause of a specific problem. For this poster we have qualitatively analyzed one student's response to an evaluation question and a debugging question. We found that while the student could have followed the execution of the program in both code samples, while evaluating, he followed the execution, but while debugging, he searched top down for where the problem could occur.

Methods

We designed our data collection to capture the code comprehension of a student looking at pieces of paper containing code in the programming language Python. A camera was set up to capture the hands of the student interacting with pen and paper as well as audio of anything they said out loud. At times the student was prompted by the interviewers to provide more information.

Task 1

In this evaluation question, we asked the student to familiarize themselves with the python code in Figure 1, and what they thought the code did.

```
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()
```

Figure 1. The Python code given for Task 1. The program prints out “40” when run.

Interview 1

1:55 : Alright so, its got a main, so thats gonna start, its going to print whatever the result of
2:00 : function 3 on 1, 2, 3, 4, some array. So lets see,
2:05 : Function 3 takes a list and returns whatever function 2 does called with list
2:10 : and some argument 4. Function 2 returns function 1
2:15 : with the same two arguments already passed to it, and
2:20 : function 4 the result of, which doesn't have any
2:25 : implied arguments, thats interesting. Umm,
2:30 :
2:35 : thats odd, lets see, so its calling functi-, ooh,
2:40 : its calling function 1. There we go. Uhh, yes, so its calling function 1 with a list a number, the array
2:45 : and 4, and the function 4 as sort of a multiplier. A function to apply.
2:50 : Alright, so function 1 is doing the actual work here. Umm, see, it starts with some accumulator 0,
2:55 : iterates across the uh items in the list, list,
3:00 : and plus equals that function 4 applied...

Figure 2. The transcript of what the student said for Task 1.

As shown in Figure 2, the student quickly identifies that main is being called and traces through the execution to func2. At around 2:20 the student identifies that func4 is used in func2, but is confused because no arguments are being passed to the function. He is further confused because func4 does not have default arguments (note that even if func4 did have default arguments the syntax presented would still not have resulted in calling func4 in the body of func2). At 2:30 he stops talking for 5 seconds as he looks at func1, he identifies that the argument “f” is being called as a function, which is passed func4 from func2.

Task 2

In this debugging question, we asked, "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."

```
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
```

Figure 3. The Python code given for Task 2. The program fails to check if a column is full.

Interview 2

Discussion

The student uses two different strategies to solve the problem in each section. While there are differences between the code used in each problem such as one being much longer than the other, and one including a class as opposed to a number of functions, they both have pieces of code that execute. In the first problem, when the student is told to figure out what the code does, he follows the thread of execution. By following the thread of execution, when he encounters the function being passed as an argument, he is able to figure out what is occurring by following the execution. It is possible that had the student worked from top to bottom he might not have realized the connection between passing func4 and calling f, and thus taken much longer to understand what the code is doing. In the other code sample, when the student is first told the overall function of the code and a bug to fix, the student started from the top of the file and moved down until finding a likely area that could contain the bug. It would have been possible to start where the program starts executing and trace the thread of execution to where the pieces drop. In fact, by looking through the code from the top and stopping at the drop function, the student doesn't actually know if this function is ever used. The student is assuming the author of the program wrote it well. Unlike the first code sample, if the student had followed the execution of the program, it would have likely taken him much longer to look at the code where the bug could be fixed.

Future Work

In future work, we hope to explore strategies used by more students and how it impacts their ability and confidence in overcoming unfamiliar constructs or semantics. For instance, one of the students we interviewed switched strategies between two evaluation questions based on the perceived complexity of the code. Another started all of the questions by looking at the code from the top down, but had varying degrees of success depending on the problem. A couple of students read through the entire code base for the debugging problems, and got bogged down in trying to understand complicated functions, to the point of forgetting the initial debugging problem. We would like to explore the advantages and disadvantages of familiarizing oneself with the code base to varying degrees such as: a detailed understanding of each function, a high level understanding of the whole program, or determining where the most likely location of the bug is without regard to anything else.

References

[1] M.-A. Storey, Theories, methods and tools in program comprehension: past, present and future, (2005), 181 – 191.

[2] McCauley, Renée et al. (2008) 'Debugging: a review of the literature from an educational perspective',Computer Science Education,18:2,67 — 92