CSCI 3901

Software Development Concepts

DALHOUSIE UNIVERSITY

Faculty of Computer Science

Lab 1: “DEBUGGING”

Kishan Kahodariya B00864907

Objective

Working Independently, debug a few programs and list the defects in programs. Get familiar with how your debugger works before you are faced with complex code.

HfxDonairExpress Problem

Cause of the defects

- When type of food is selected - Donair and anyone of the size is selected, instead of getting base prices from list of Donair's base prices, code was fetching prices from pizza's base price list because break statement was missing after case 0 in the switch statement.
- While adding toppings, user was able to enter the toppings but once added the price was not reflecting in the total price of the pizza. The cause of this was, the price of the toppings weren't added to base price, it was replacing the current value of the price variable. The expression (price=1.00), (price=0.99) and (price=0.75) were replace by (price **+**=1.00), (price**+**=0.99) and (price**+**=0.75) respectively by adding "+" before equal sign.
- In the expression which evaluates discount, only the discount percentage was getting deducted from the total price. The correction evaluation would be as follows –

  price = price –(price*(coupon/100))

Approach that let you locate the defect

- By setting up concurrent debugging breakpoint and using debugger to track value of the price variable.

Appropriate amount of time taken to debug

- It took me around 2 hours to figure out the problem and pin point to the exact location in the code.

Ackermann Problem

Cause of the defects

- The problem with this particular problem was the conditions which were setup i.e. if-else conditions and the output which is thrown if the condition satisfies.
- First of all, I was not aware with the workings of a Ackermann's algorithm. So I studied it and draw the fundamentals conditions of Ackermann's algorithm as follow:
    - If (m==0) then A(m , n) = n+1
    - If (m!=0) and (n==0) then A(m , n) = A(m-1 , n)
    - If (m!=0) and (n!=0) then A(m , n) = A(m-1 , A(m , n-1))
- Based on the above conditions, I made necessary changes to line 21, 25 and 27 as follows:
    - (n==0) to ((n==0) && (m!=0)) **(line 21)**
    - Else {} condition to ((n!=0) && (m!=0)) **(line 25)**
    - ackermann(m-1 , ackermann(m , n)) to ackermann(m-1,ackermann(m , n-1)) **(line 27**)

Approach that let you locate the defect

- By first understanding the logic behind the Ackermann algorithms and noting the fundamental conditions.
- After that, it was just comparing the conditions with conditions written in the code and make the necessary correction to that condition.

Appropriate amount of time taken to debug

- It took me around 45 hours to understand the code and pin point the exact location in the code where the problem lies.
- Note that while comparing the conditions, it seemed necessary to create a separate condition when (m!=0 && n!=0) instead of considering it in a generalized else statement.

Linked List Problem

Cause of the defects

- So far I have been able to identified that defects lies in copy() method as well as append() method which needs necessary corrections.

Approach that let you locate the defect

- By setting up concurrent debugging breakpoint and using debugger.
- Also I noticed an anomaly that while using debugger, when the list2 is initialized in main() method with list1.copy() method, the memory address of the elements in list2 was similar to that of elements of list1 which seems strange because if copy() method creates a whole new list then this addresses much be different.

Appropriate amount of time taken to debug

- It took me more than 4 to 5 hours trying to debug this code but still at end I was not able to pin point the exact defect with the code and make the necessary corrections.

(Note: Linked List Code that I have uploaded is not the fully corrected version. I have made change to the append() method which I thought would solve the problem but no success**)**

Analysis

Strategy / Strategies for debugging

- Run your program until you have identified the defected variable which does not retain values as you expected.
- Then set debug breakpoint before operations which involves that variables and check the change in value of that variable.
- After this evaluation, set breakpoint before operation which doesn't involve that defected variable, to check if these operations in any way affects the desired outcome of the suspected variable.
- On a different note, using "System.out.println" to printout value of a variable does help at certain extent because you can track its value while running the code without setting breakpoints but it's not advisable as it's not a viable approach.

What makes them effective?

- Putting breakpoint just at point where the variable is possibly affected saves lots of time in debugging and help in keeping track and pin point the problem in a specific operation.
- At start of my debugging process, I start by printing the variable used in the code before any main operation to check if it is getting value as planned.
- By doing so, it allows me to identify approximate locations in code where should the debugging breakpoints be placed.

For which conditions, will your strategies be effective or ineffective?

- Setting several breakpoints throughout your code will be effective in debugging when your code involves one of the following conditions-
    - recursive functions
    - Value of a variable is changing often
    - Methods are called by reference variable

    - Keeping track of important private variables
    - Keep track of a variable which is used in a recursive function

How can a debugger support your strategies?

- Debugger works like a notebook which keeps track of every operation and its result along with other variables throughout your code.
- While using breakpoint strategy, debuggers allow you to keep track of your suspected variable from the start of the code using interface which can also showcase the stack of operation which are still to be processed.
- Depending on the debugger, it can also showcase ‘id’ of a variable, which is used to store that variable in the memory.
- Along with that, some debuggers are also capable of displaying amount of bytes i.e. memory every variable is utilizing in the system.