**Background**

Complete the following exercises. Your answers must be submitted via Canvas.

Note that for each code section, you may assume that the in-code documentation is accurate and reflects the both the correct approach and desired design for the piece of software.

**Section 1: Equivalence Class & Edge Condition Testing**

An online store sells customized flags and has a custom-built front-end to order flags from. The store uses a program where the customer inputs the specifics for a flag order. The program then calculates the total cost of the order. The following input is required:

Length of the pole: 0 - 9 ft, inclusive and specifiable to the half foot

Should the pole be weighted? Yes or No

The pattern on the flag: Solid Background Color, Tricolor, Gradient

The preferred fabric: Cotton or Nylon

The number of flags to be purchased as an integer: 1 – 100, inclusive

Each nylon flag costs $20 plus $2 per foot over five feet. Each cotton flag costs $30 plus $3 per foot over five feet. A weighted pole adds $10 to the total price. A discount of 10% is applied to orders of 25 flags or more.

A $1.50 fee for shipping and handling FOR EACH FLAG is applied to all orders.

Coloring will be handled in a separate system so you do not have to worry about the constituent colors of the flag. All flags in the same order must be of the same type; this means that the 10% discount must be incurred for a single flag type and that you do not need to worry about aggregating multiple flag types into one order.

1. Identify and explain the various equivalence classes by input parameter. Be sure to include a brief explanation as to why the equivalence class exists. This could be a natural categorization based off of the data or it could be a split due to other parameters and may represent valid or invalid data.

Length of Pole:

* 0-9ft (inclusive and specifiable to the half foot)
  + Class covers the valid range of lengths for the pole allowing for precise measurements

Weighted Pole:

* Yes
* No
  + Class covers the decision to add weight to the pole which would increase the price.

Flag Pattern:

* Solid background
* Tricolor
* Gradient
  + Since the different patterns have different costs, this class allows for different pattern options

Fabric:

* Cotton
* Nylon
* This class exists to categorize the different fabrics which can be used in the system to link the cost to the fabric

Number of flags purchased:

* Valid quantities
  + Range of integers from 1-100
* Invalid quantities
  + Negative values, values greater than 100, non- integer inputs
* The number of flags ordered impacts the total cost of the order and the discounts that are applied to each order. This equivalence class is to make sure that these values are valid and don’t go into the negative or above 100 per order

Combination of parameters:

* Valid Combinations
  + Discount of 10% on orders of 25 or more
  + Nylon flags cost $20 plus $2 per foot over 5 feet
  + Cotton flags cost $30 plus $3 per foot over 5 feet
* Invalid combinations
* Having valid combinations of parameters make sure that the system can process orders correctly with different combinations of parameters. Invalid combinations help identify errors in user input or system behavior

**Section 2: Decisions Tree Testing**

The following is a partial screen shot of a new student record search on Harrisburg University’s MyHU:



Assuming that the user has been authenticated, they can enter either of the following in the search box:

* 1. The ID of a student, which consists of 7 or 8 digits, or
  2. The last name or full name of a student.

The result of the search operation can be one of the following:

1. If a unique match is found, the record of the matching student shall be displayed
2. If multiple matches are found, a list of full names of all matching students, in ascending alphabetical order of their last names then first name, shall be displayed for selection.
   1. If no match is found, an appropriate message shall be displayed to indicate so.
   2. If the search operation cannot be successfully completed for any reason, an appropriate message shall be displayed.
3. Create a decision table black box testing plan to adequately test this search function.

Conditions:

Input Method

# of Matches

Output

Test Case 1:

Input - Student ID

# of Matches – 1

Output - The search attempt should return the record of the matching student

Test Case 2:

Input - Last Name

# of Matches - 1

Output - The search attempt should return a record of the matching student

Test Case 3:

Input – Full Name

# of Matches - 1

Output - The search attempt should return the record of the matching student

Test Case 4:

Input – Full Name

# of Matches - >1

Output - The search attempt should return a list of full names of all matching students in ascending alphabetical order of their last names then first name.

Test Case 5:

Input – Last Name

# of Matches - >1

Output - The search attempt should return a list of full names of all matching students in ascending alphabetical order of their last names then first name.

Test Case 6:

Input – Student ID

# of Matches - 0

Output - The search attempt should return an appropriate error message to indicate that no matches were found

Test Case 7:

Input – Last Name

# of Matches - 0

Output - The search attempt should return an appropriate error message to indicate that no matches were found

Test Case 8:

Input – Full Name

# of Matches – 0

Output - The search attempt should return an appropriate error message to indicate that no matches were found

Test Case 9:

Input – Not Found/Invalid Input

# of Matches – N/A

Output – The search attempt should return an appropriate error message to indicate that the search could not be completed because of invalid/missing input

Test Case 10:

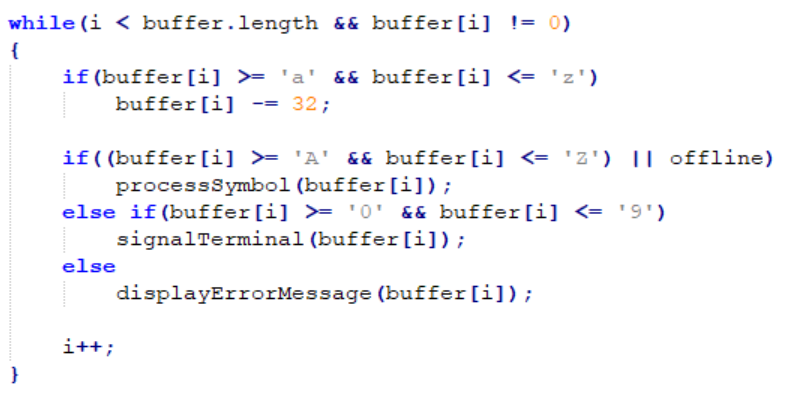
Input – Student ID

# of Matches - >1

Output - The search attempt should return a list of full names of all matching students in ascending alphabetical order of their last names then first name.

**Section 3: White Box Testing**

Consider the following code segment:



1. Write the test cases for white box testing plan based on:
   1. Branch(decision) coverage (5 points)

Test 1:

While loop is true and buffer[i] >= ‘a’ && buffer[i] <= ‘z’

Test 4:

While loop is true, first if statement is true, and buffer[i] >= ’A’ && buffer[i] <= ‘Z’ || offline == True

Test 8:

While loop is true, first if statement is true, second if statement is false, and buffer[i] >= ‘0’ && buffer[i] <= ‘9’

Test 8:

While loop is true, first if statement is true, second if statement is false, and buffer[i] < ‘0’ && buffer[i] > ‘9’

* 1. Condition coverage (5 points)

Test 1:

I > buffer.length && buffer[i] != 0

Test 2:

I< buffer.length && buffer == 0

Test 3:

While loop is true and buffer[i] < ‘a’

Test 4:

While loop is true and buffer[i] > ‘z’

Test 5:

While loop is true, first if statement is true, and buffer[i] < ‘A’ && buffer[i] <= ‘Z’ || offline == False

Test 6:

While loop is true, first if statement is true, and buffer[i] >=’A’ && buffer[i] > ‘Z’ || offline == False

Test 7:

While loop is true, first if statement is true, and buffer[i] >=’A’ && buffer[i] > ‘Z’ || offline == True

Test 8:

While loop is true, first if statement is true, and buffer[i] < ‘A’ && buffer[i] <= ‘Z’ || offline == True

Test 9:

While loop is true, first if statement is true, second if statement is false, and buffer[i] < ‘0’ && buffer[i] <= ‘9’

Test 10:

While loop is true, first if statement is true, second if statement is false, and buffer[i] >= ‘0’ && buffer[i] > ‘9’

Test 11:

While loop is true, first if statement is true, and buffer[i] >= ‘A’ && buffer[i] <= ‘Z’ || offline == True

Test 12:

While loop is true, first if statement is true, and buffer[i] >= ‘A’ && buffer[i] <= ‘Z’ || offline == False

* 1. Compound condition coverage (10 points)

Test 1:

I > buffer.length && buffer[i] != 0

Test 2:

I< buffer.length && buffer == 0

Test 3:

I < buffer.length && buffer[i] != 0 and buffer[i] < ‘a’ && buffer[i] <= ‘z’

Test 4:

I < buffer.length && buffer[i] != 0 and buffer[i] >= ‘a’ && buffer[i] > ‘z’

Test 5:

While loop is true, first if statement is true, and buffer[i] < ‘A’ && buffer[i] <= ‘Z’ || offline == False

Test 6:

While loop is true, first if statement is true, and buffer[i] >=’A’ && buffer[i] > ‘Z’ || offline == False

Test 7:

While loop is true, first if statement is true, and buffer[i] >=’A’ && buffer[i] > ‘Z’ || offline == True

Test 8:

While loop is true, first if statement is true, and buffer[i] < ‘A’ && buffer[i] <= ‘Z’ || offline == True

Test 9:

While loop is true, first if statement is true, second if statement is false, and buffer[i] < ‘0’ && buffer[i] <= ‘9’

Test 10:

While loop is true, first if statement is true, second if statement is false, and buffer[i] >= ‘0’ && buffer[i] > ‘9’

Test 11:

While loop is true, first if statement is true, and buffer[i] >= ‘A’ && buffer[i] <= ‘Z’ || offline == True

Test 12:

While loop is true, first if statement is true, and buffer[i] >= ‘A’ && buffer[i] <= ‘Z’ || offline == False

1. Given your testing plans and understanding that this system is not responsible for anything to do with physical safety, which of the above test plans do you think is sufficient to test our code and why?

I believe that the most appropriate test plan would be the conditional testing since there are a lot of conditions in this block of code and having tests for each one would allow for better testing and debugging capabilities. Branch coverage would also work because this code isn’t too complicated in its function so as long as its hitting each branch with the appropriate parameters, then that would also work.