United International University
Department of Computer Science and Engineering

CSE 4495: Software Testing and Quality Assurance
Final Examination : Fall 2022
Total Marks: 40 Time: 2 hours

Any examinee found adopting unfair means will be expelled from the trimester / program as per UIU disciplinary rules.

Answer all the questions. Numbers to the right of the questions denote their marks.

1 Unit Testing and Build Scripts(18 marks)

1. Consider the following class diagrams for a simplified simulation of UIU UCAM website. The three diagrams below represent the student, course and Section of a course. Your Job is to write unit test cases for the methods of this class according to the given specifications.

Figure 1: Class diagrams for Ques-1

(a) First, you need to test the *AddCourse* method from student class. A student should be able to add any course if he/she has completed the prerequisite of that course. *AddCourse* inserts the course to the student's running_courses list. You can check that by calling the *IsCourseRunning* method which returns a Boolean value based on whether a course is added or not.So, you need to write three unit tests to check the following three contexts. $[3 + 4 + 3 = 10]$

- Case 1: Check if the *AddCourse* works properly when a course with no perquisite is passed. The *IsCourseRunnning* method is expected to return true after adding the course.
- Case 2: Check the same method for a course with a prerequisite. Set a prerequisite by the *addPrerequisite* method and add the same course to the students completed course list by calling *addToCompleted* method. This should satisfy the prerequisite requirement and *Aaddcourse* is expected to work properly like test case 1
- Case 3: Now run test case 2 without adding the prerequisite course to the student's completed courses. The system should throw *RequirementInvalid.class* error with message *"You cannot add a course without completing its perquisite first."*

Now devise executable test cases for all of the above specifications for this method in the *JUnit* notation.

(b) Now you need to test the *selectSection* method.This method has one parameter which is a Section object. The section class has a fixed *capacity* and if a new student is enrolled into the section he/she is added to the *students* list, increasing the total student number by one. If the total student number and capacity is equal a new student cannot enroll to that section. This can be achieved by comparing the values returned by *getCapacity* and *getTotalStudentNumber* methods. Now test the method according to the following specifications- [3+2=5]

- Case 1: Assume that the methods of Section class has not been written yet. So you need to use object mocking to create a mock Section which has capacity full. To emulate this you will have to set properties for *getTotalStudentNumber* and *getCapacity* methods of you section object. Then if a student is enrolled to this section the system is expected to throw *RequirementInvalid.class* error with message *"Section capacity full, Enrollment cancelled."*
- Case 2: Now write a similar test case where section capacity is not full. The *selectSection* method should not throw any error in this case.

(c) The following code snippet is taken from a build script which is written to clean the build directory. Explain which files will be deleted and which files will be excluded from deletion after executing this code. [3]

```
<target name = "clean" description = "Clean output directories">
    <delete>
        <fileset dir = "${build.dir}">
            <include name = "**/*.class"/>
            <include name = "**/*.out"/>
            <exclude name = "**/gitignore/**"/>
        </fileset>
    </delete>
</target>
```

2 Model Based Testing (12 marks)

2. (a) Explain why achieving transition and state coverage is not enough to judge test case adequacy. Also introduce different metrics of path coverage and how they solve the problems with other coverage metrics. [3]

(b) UIU cafeteria has recently implemented an mobile app to control and organize lunch time traffic. From now on, during the lunch hours to buy something students will first have to raise an e-ticket from the app. Each e-ticket has a serial number. After raising a ticket the customer will either choose the daily set menu or choose to create his own meal.If the customer goes for the set menu he/she can immediately checkout with the food and collect their food by paying for it on cafe stall no. 1. On the other hand if a customer chooses to create his/her own dish, they will be redirected to the active menu page where one can add or remove items to customize their own meal. After they finish customizing they will be inserted into a queue until their ticket number is announced on the app. Once notified the customer can then collect their meal by completing payment from stall no.2. You must design a state machine that simulates a customers journey through the mentioned app. [3]

(c) Consider the following classes in a simple video streaming service named *TenFlix*. After signing up a user can purchase a package of their choice and enjoy the shows. All the methods of the class *TenflixUser* are described.

- ***purchaseSubscription(package)*** – This method can be called only when there are no active plans. Invoking this deducts the package's **MonthlyCharge** from user's **CurrentBalance** and sets the user's current **ActivePlan** to the corresponding package. Also adds the package's **Duration** to the User's **currentDuration**.
- ***renewSubscription()*** – Can be invoked only when the user has an **ActivePlan**. It can be called as many times as needed each time deducing the package's **MonthlyCharge** from user's **CurrentBalance** and adding the package's **Duration** to the User's **currentDuration**.
- ***cancelSubscription()*** – Resets the existing **ActivePlan** to null and **currentDuration** to zero. No refunds are given if user cancels with **currentDuration** remaining.
- ***refillBalance(amount)*** – Can be invoked from any state. Adds the corresponding amount to User's **CurrentBalance**.
- ***expireSubscription()*** – Called when the packages' duration expires ,i.e. user's **currentDuration** is equal to zero.

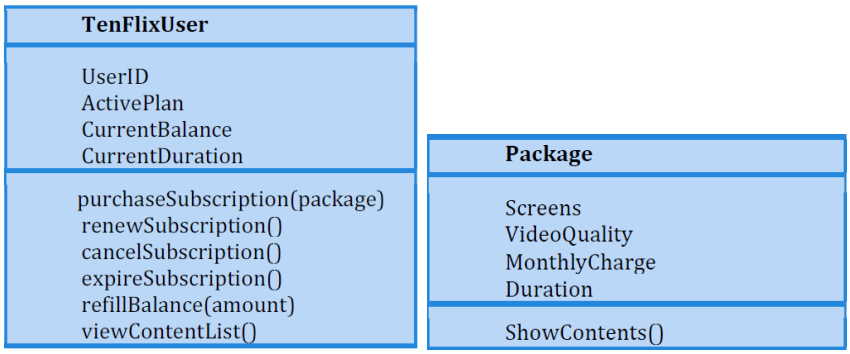

Figure 2: Class diagrams for Ques-2(b)

- ***viewContentList()*** – Displays the list of content that is accessible from this package. Can be invoked if the user has an **ActivePlan**.

Now, identify the states and design a Finite State Model for the *TenflixUser* class with all the relevant labeled transitions. [You don't need to consider viewContentList() as an event]. [4]

(d) Write a test suite (with two or more test cases) that achieves transition coverage for the finite state model you have created in Q-2(c). [2]

3 Finite State Verification (10 marks)

3. (a) Consider a simple elevator controller modeled as a finite state machine using the following state variables:
   - Door : OPEN, CLOSED – sensor input indicating state of the door
   - Elevator_Direction : UP, DOWN, IDLE – indicates the moving direction of the elevator
   - Button : UPWARD, DOWNWARD, NONE – button press (assumes at most one at a time).
   - Button_Floor : 0,1..8 – Indicates which floor's button has been called.
   - Floor : 0,1..8 – Current position of the elevator cell.

   Formulate the following informal requirements in **LTL**: [2+1+2 = 5]

   i. If elevator button is pressed(upward or downward) while elevator is idle at the same floor that the button is in, the door will open immediately.
   ii. The elevator will start moving upward if it is currently idle and called from a floor that is above it.
   iii. If the downward button from any floor is pressed while the elevator is moving upward it will eventually stop at the floor it was called from.

(b) Consider a finite state model of a simple microwave controller. State variables:
   - traffic-light: RED, YELLOW, GREEN
   - pedestrian-light: WAIT, WALK, FLASH
   - button: RESET, SET

   Formulate the following informal requirements in **CTL**: [1+1+1 = 3]

   i. if the button is set, then eventually the pedestrian light will signal walk.
   ii. If the light is red, and the button is reset, then eventually, the light will turn green.
   iii. If the pedestrian light signals wait and the traffic light is red, the pedestrian light will signal walk immediately.

(c) Here are some implementation details on how the traffic _light variable will change in the next state - [2]
   - If the light is red and the button is reset the light will turn green in the next state.
   - When the button is set and the light is green, in the next step the light turns either green or yellow randomly.
   - When the light is green or red and the button state is unchanged , the light remains same.
   - When the light is yellow and the button state is unchanged , the light turns red or yellow randomly.
   - In default case the light is red.

   Write NuSMV code to capture the changes only in traffic light variable according to the mentioned details.