CSCE 740 - Practice Final
Name:
Student ID Number:

Question 1—7 Points.

The following short questions are worth 1 or 2 points each.

2 Points each: (more than one answer may be correct, pick all that apply).

1. Which of the following make sense as classes (rather than objects) in a class diagram?
   a. Homework Assignment
   b. Manton Matthews
   c. Group 5's assignment 5
   d. Person

2. Which of the following coverage criteria **<u>always</u>** requires more test cases than the others?
   a. Statement Coverage
   b. Branch Coverage
   c. Path Coverage
   d. None of the above

1 Point:

3. Requirements-based test cases help the writer clarify the requirements.
   a. True
   b. False

4. In UML, a Class describes an Object.
   a. True
   b. False

5. The use of global variables generally increases coupling.
   a. True
   b. False

Question 2—6 Points.

Describe the key difference between black-box testing and white-box testing.

Suggested Solution:

Black-box testing involves testing the functionality of a software component without knowing the details of its internal logic - you do not know what it inside its source code, what methods are called, or what objects exist at runtime. White-box testing involves testing the independent logic paths with full implementation knowledge (you can see the code) - however, you do not have full knowledge of the intended functionality (white box tests cannot look for unimplemented code).

Question 3—8 Points.

Mention two fundamental characteristics of software that makes software engineering different than other engineering disciplines. Please elaborate briefly on each characteristic as to why it makes software engineering different.
(Alternatively, if you do not agree with the premise of the question, argue briefly that there is no difference between software engineering and other engineering disciplines.)

Suggested Solution:

Here any of the issues we discussed in class will work.
If anyone decided to argue the alternative, we will deal with that if the situation occurs.

- *Intangibility*
    - *We can't visualize software. Thus, it is hard to see problems early, and hard to judge progress.*
- *"Software" is not one thing*
    - *A programming language can be used to build software for almost any imaginable purpose. Software engineers are responsible for a wider variety of products than, say, bridge engineers.*
    - *The skills needed to design accounting software differ from those needed for a pacemaker.*

Question 4 – 9 Points.

When we discuss software testing, we refer to Faults and Failures. Please briefly describe what a Fault is and what a Failure is. Make sure to point out the difference between a Fault and a Failure.

Suggested Solution:

A Fault is a problem with the implementation. It is something that is missing, extra, or erroneous.
A Failure is an incorrect execution of the software; we get an output we did not expect.
A Failure is the manifestation of a Fault, if the execution executes the Fault and the corrupted state propagates to the output, we can observe it as a Failure.

Question 5—10 Points.

Are path coverage and exhaustive testing the same thing? Motivate your answer.

Suggested Solution

No. Path coverage "only" requires that every path is exercised; it does not require that every input is tested. One can provide path coverage without testing every instance of the inputs that would take you down that path. Thus, problems with divide-by-zero and null-pointer-dereferencing might not be caught.

Question 6—14 (3 + 9 + 2) Points.

You are developing a train scheduling tool for a rail network, where - for each station - a list of arriving trains is tracked (using a train ID that is a string of three characters and four single-digit integers). Each day, a new schedule is initialized and the previous day's schedule is deleted. Additionally, a list is kept of valid train IDs.

The data structure containing train records contains the following independently testable features:

- void insertInSchedule(station, trainID)
- Boolean existsInSchedule(station, trainID)
- void deleteFromSchedule(station, trainID)

Part 1:
For the system, you receive the following requirement:
"We can't have a train arrive at a station more than once."

Revise this requirement so that it is testable.

Part 2:
Given the obvious meaning of the above methods, develop test cases using input domain partitioning. You can define your test cases as input/output pairs. For example, to test insert(station, trainID), one test case could be:
<u>Input:</u> station with empty container, valid trainID
<u>Output:</u> trainID in container

Note - Do not go overboard with test cases, 4-6 test cases per method is adequate

Part 3:
Identify a test case from the above that could be used to verify your revised requirement from Part 1.

Suggested Solution
Part 1 - Looking for concept of the train arriving once per day. That is, there should only be one entry per train in the list.

Part 2 - Something along the following lines. Each method has two explicit inputs - the train schedule for a station and the ID of a train. There is also an implicit input - the list of valid train IDs - to consider. This suggests input partitions.

For the station list - an empty list, a list containing the train ID already, a list not containing the train ID. (as no maximum bound is suggested, a full list is not a good *partition). Other partitions are possible, but the above are essential to demonstrate that the function works.*

For the train ID - it can be valid, invalid (not on the master train list), or malformed (not following the stated format).

Insert	*ID in station / valid ID*	*no change*
	ID not in station / valid ID	*ID in container*
	ID in station / invalid or malformed ID	*Error or no change*
	ID not in station / invalid or malformed ID	*Error or no change*
	empty list / valid ID	*ID in container*
	empty list / invalid or malformed ID	*Error or no change*
Exists	*ID in station / valid ID*	*True*
	ID not in station / valid ID	*False*
	ID in station / invalid or malformed ID	*Error (or false)*
	ID not in station / invalid or malformed ID	*Error (or false)*
	empty list / valid ID	*False*
	empty list / invalid or malformed ID	*Error (or false)*
Delete	*ID in station / valid ID*	*ID no longer in list*
	ID not in station / valid ID	*no change (or error)*
	ID in station / invalid or malformed ID	*no change (or error)*
	ID not in station / invalid or malformed ID	*no change (or error)*
	empty list for station/ valid ID	*no change (or error)*
	empty list for station/ invalid or malformed ID	*no change (or error)*

Part 3 - Very first test for insert. Try to insert a train when it is already in the list. It should not be added a second time.

Question 7—16 (3+3+3+3+4) Points.

For the following function,

a. Draw the program flow graph for the program.

b. Develop test input that will provide statement coverage. (Input output pairs will be fine.)

c. Develop test input that will provide branch coverage.

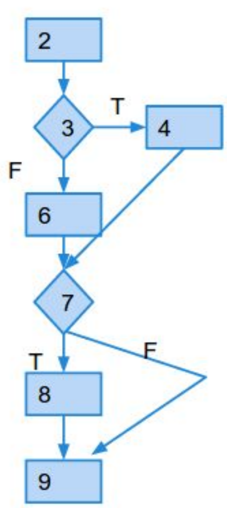
d. Develop test input that will provide full-path coverage.

e. Modify the program to introduce a fault so that you can demonstrate that even achieving full path coverage will not guarantee that we will reveal all faults. Please explain how this fault is missed in your example.

```
int findMax(int a, int b, int c)
{
    int temp;
    if (a>b)
        temp=a;
    else
        temp=b;

    if (c>temp)
        temp = c;
    return temp;
}
```

Suggested Solution




```
1. int findMax(int a, int b, int c) {
2.   int temp;
3.   if (a>b)
4.        temp=a;
5.   else
6.        temp=b;
7.   if (c>temp)
8.        temp = c;
9.   return temp;
10. }
```

a)

b) (3, 2, 4); (2, 3, 4)

c) *(3, 2, 4); (3, 4, 1)*
d) *(4, 2, 5); (4, 2, 1); (2, 3, 4); (2, 3, 1)*
e) *If we have (a>b+1) in the first condition as opposed to (a>b), the tests in part D will not reveal this flaw. Only a boundary value test will.*

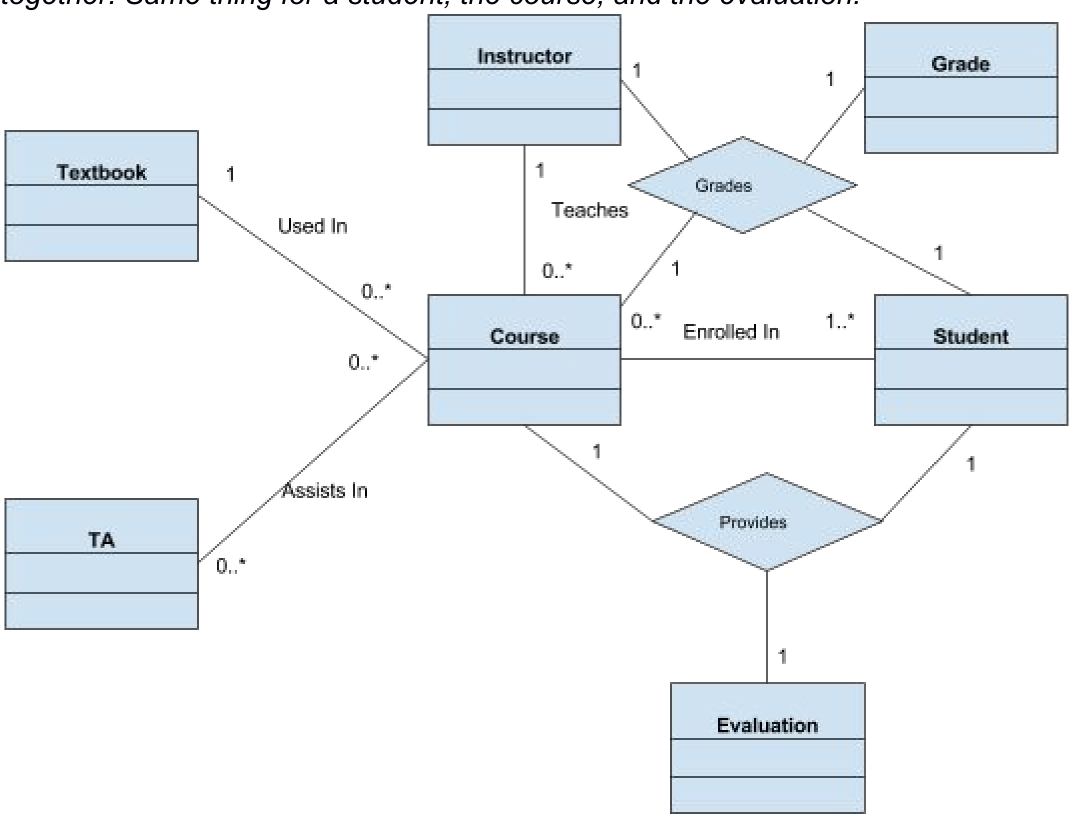
Question 8 – 12 Points.

Students at the University of South Carolina can be enrolled in more than one class at the time. There is also an option to not be enrolled in any classes (under special circumstances such as completion of all requirements except your PhD dissertation defense). Naturally, we do not offer classes with no students at all.
To equitably allocate teaching effort, there is one instructor assigned to each class (there is no co-teaching). Some instructors might not teach any class (buyout for research for example). Each class uses a textbook (a book that—incidentally—can be used in other classes also).
Depending on class size, there are TAs assisting in the class. A small class gets no TAs, a large class might get several TAs.
When all is done in the class, the instructor assigns the student a grade for the course. In return, each student must fill out a course evaluation form for the course.

Draw a **class diagram** for the description above. Make sure to show attributes, multiplicities, association names, data attributes, and aggregations/compositions, where appropriate. You may omit operations.

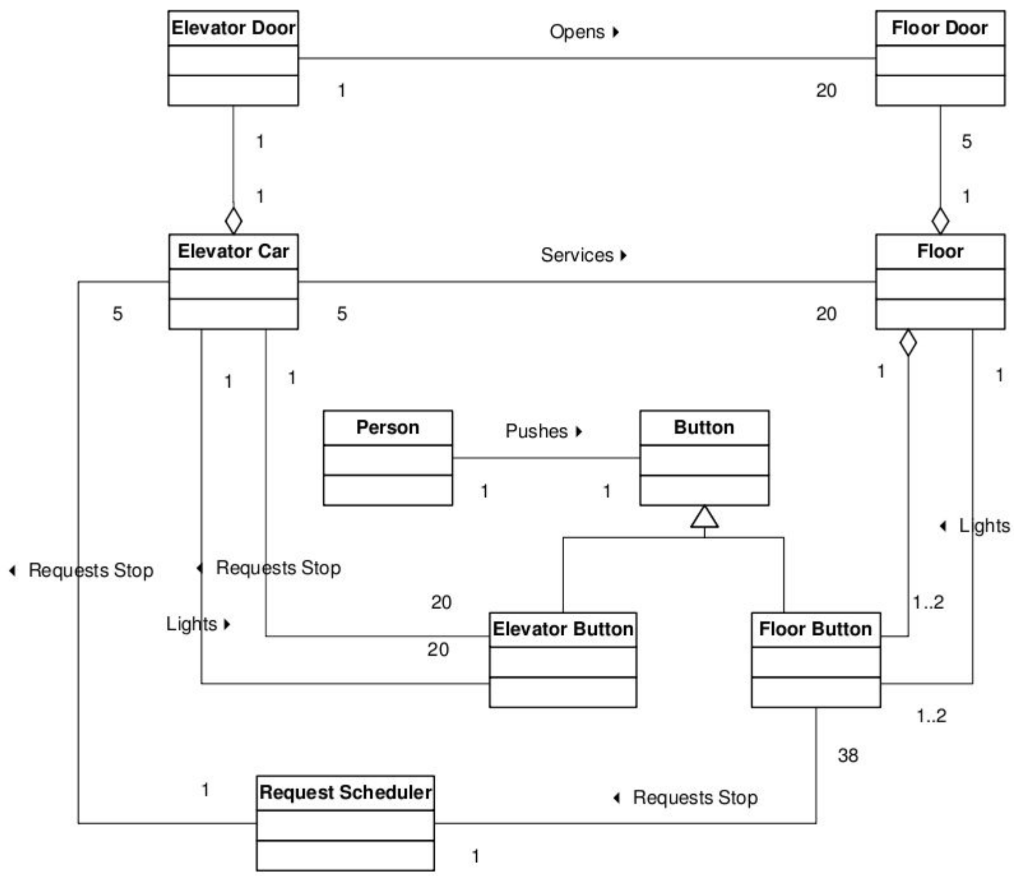
Suggested Solution

This constitutes one solution. Note that a grade, a student, and a course ought to be tied together. Same thing for a student, the course, and the evaluation.



Question 9—10 Points.

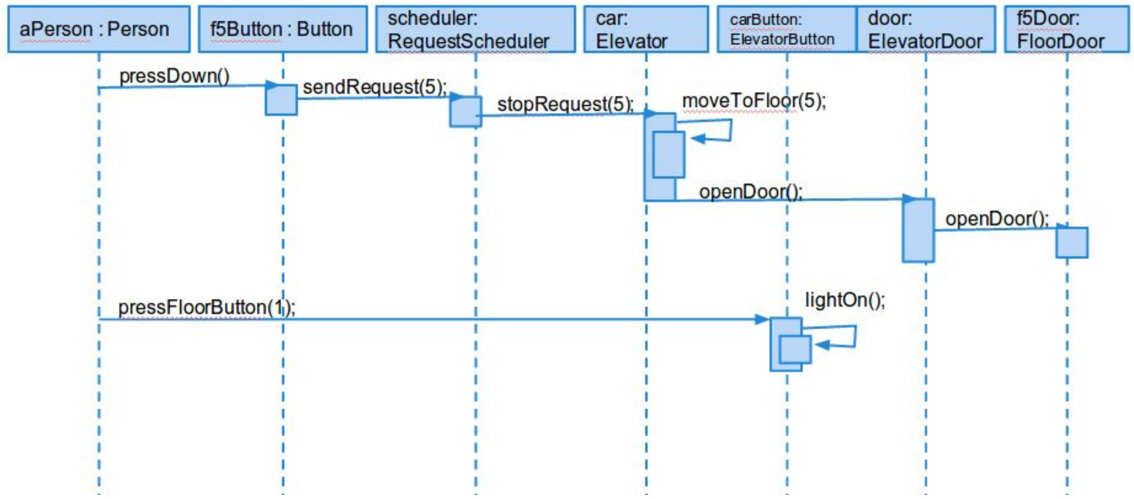
Based on the class diagram below, please draw possible sequence diagrams for the two high-level scenarios below. Note that the operations are not yet defined for the classes. We are drawing these sequence diagrams to help us discover what operations will be needed for each class. Thus, the sequence diagrams will have to contain a little bit more detail then the high-level scenarios we captured when we discussed the use-cases with the customer.

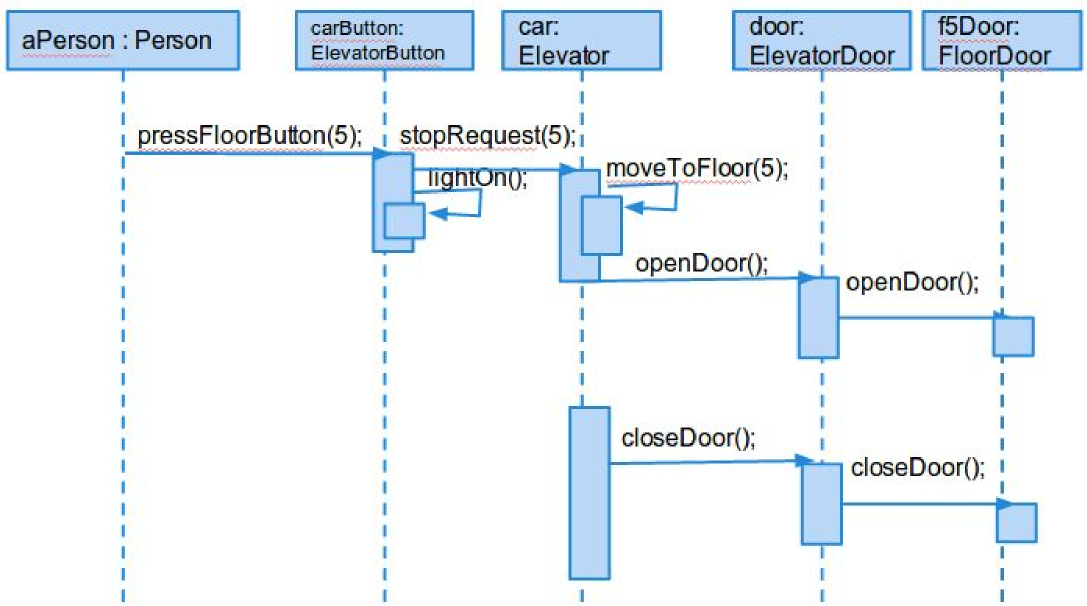


Scenario 1 (Requesting a Ride Down):
A person approaches the elevator on the fifth floor. She wants to go down so she presses the "down" button next to the elevators. She waits until an elevator arrives and the doors open. She enters the elevator and presses the elevator button for the ground floor (floor 1). The light next to the button for the first floor is lit.

Scenario 2 (Getting Off at a Floor):
A person is standing in the elevator with the door closed. The person pushes the elevator button for floor 5(and there are no other requests). The elevator stops at the fifth floor, opens the doors, and the person steps out. The elevator doors close.

Suggested Solution:





Some variation OK as long as they make it clear that they have understood the fundamentals of a sequence diagram.

Question 10—8 Points.

You are developing software that will simulate and execute finite state machines.
A state machine consists of states and transitions. One state is special and designated to be the initial state (this is where we always start). Besides this, the initial state is just like all other states.
The transitions have transition conditions associated with them. A transition condition consists of a trigger event, a guarding condition, and a possibly empty set of actions (actions are events generated as a result of taking the transition).

Draw a **class diagram** for the description above. Make sure to show attributes, multiplicities, association names, data attributes, and aggregations/compositions, where appropriate. You may omit operations.

Suggested Solution
This constitutes one solution. Make sure you read the problem description carefully. Here I would like to see TriggerEvent and Action both inherit from a parent, as it is mentioned that actions are events. TransitionCondition and GuardingCondition can also inherit from a common parent, as both are conditions.